CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Correct System Design
University of Oldenburg

Carl von Ossietzky Universität Oldenburg
Department für Informatik - Abteilung Correct System Design

Diplomstudiengang Informatik

Diplomarbeit

Runtime-Checking von JML-Spezifikationen mit Jass

Vorgelegt von: Martin Schnaidt

Betreuender Gutachter: Prof. Dr. Ernst-Rüdiger Olderog

Zweiter Gutachter: Dipl. inf. André Platzer

Oldenburg, den 28.02.2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Mit der rapide anwachsenden Leistungsfähigkeit von Hardwaresystemen und der stetig zunehmenden Vernetzung und Multimedialität unseres Alltags hat in den letzten Jahren auch der Umfang und die Komplexität von Softwaresystemen immer mehr zugenommen. Damit wird es auch für Entwickler schwieriger, das Verhalten der von ihnen konzipierten und implementierten Systeme zu überschauen und im Detail nachzuvollziehen. Auswirkungen dieser Schwierigkeiten zeigen sich zum Beispiel in den Problemen bei der Einführung der Systeme zur Erfassung der LKW-Maut auf deutschen Autobahnen im Jahr 2003 und in den letztjährigen Softwareproblemen bei der Bundesagentur für Arbeit, bei denen durch einen Fehler in der Berechnung der Krankenkassenbeiträge für Langzeitarbeitslose insgesamt ein finanzieller Schaden von geschätzten 50 Millionen Euro entstand.

Ein Ansatz, dieser Problematik zu begegnen, ist das *Design by Contract* zur Spezifikation des Verhaltens von Softwarekomponenten, dem durch die Arbeit von Bertrand Meyer [Mey92] ein theoretisch fundierter Ansatz zur Verfügung steht. Die grundlegende Idee ist dabei folgende: Zwischen einer Softwarekomponente und einem potenziellen Aufrufer, genannt *Client*, wird ein *Vertrag* (engl.: *contract*) geschlossen, der die Rahmenbedinungen eines möglichen Aufrufs festlegt. Dazu gehören einerseits die Bedingungen, die der Client erfüllen muss, damit er die Komponente aufrufen darf, und andererseits die Garantien, die die aufgerufene Instanz für ihre Ausführung gibt. Für die Programmiersprache Java, die insbesondere durch ihre Anwendungsmöglichkeiten im Internet in den letzten Jahren eine große Verbreitung erfahren hat, steht durch die *Java Modeling Language*, kurz *JML*, eine Spezifikationssprache zur Verfügung, die den Ansatz des Design by Contract konsequent umsetzt. JML wurde an der Iowa State University in den USA unter der Leitung von Gary T. Leavens entwickelt und ist inzwischen eine weit verbreitete und akzeptierte Modellierungssprache. Mit dem *JML Runtime Assertion Checking Compiler* steht zudem ein Werkzeug zur Verfügung, das JML-Spezifikationen in ausführbaren Java-Quellcode übersetzt, der das definierte Verhalten zur Laufzeit auf Einhaltung überprüft.

Detlef Bartetzko entwickelte in seiner Diplomarbeit [Bar99], die auf den Arbeiten von Clemens Fischer und Dieter Meemken [Mee97] aufbaut, den Precompiler *Jass*. *Jass* verfolgt dabei ebenfalls die Idee des Design by Contract, um das Verhalten von Methoden und Klassen in Java zur Laufzeit zu beschreiben. Dazu können in der gleichnamigen Spezifikationssprache Aussagen über das gewünschte Verhalten getroffen werden, die in der Form spezieller Kommentare direkt im Quellcode formuliert werden. Durch das Übersetzen dieser Kommentare in Java-Quellcode lässt sich das spezifizierte Verhalten zur Laufzeit auf Einhaltung überprüfen.

1.1 Aufgaben und Ziele

Bei einigen Fallstudien im Rahmen des Projekts „ForMoos“ am Department für Informatik der Universität Oldenburg hat sich herausgestellt, dass der JML Runtime Assertion Checking Compiler (kurz: JML RACC) insbesondere bei der Verwendung von sogenannten Modellvariablen in Interfaces Probleme bei der Übersetzung hat, die sich unter anderem in zur Laufzeit scheinbar verletzten Invarianten äußert. Zudem bietet die aktuelle Version 5.3 des JML RACC noch keine Unterstützung der in Java 1.5 [BGJS05] neu hinzugekommenen Sprachelemente. Gerade diese Neuerungen wie beispielsweise die generischen Definitionen wurden aber von aktiven Entwicklern der Java Community lange erwartet und können wesentlich dazu beitragen, Implementierungen elegant und typsicher zu gestalten.
Aufgrund der bereits großen Akzeptanz und den umfassenden Möglichkeiten, die diese Spezifikationssprache bietet, ist es Ziel dieser Diplomarbeit, eine signifikante Teilsprache von JML zu identifizieren und als Eingabesprache in das Runtime-Checking-Werkzeug *Jass* zu integrieren. Dabei soll speziell für Spezifikationen in Interfaces eine alternative Übersetzungsstrategie entwickelt werden, um die angesprochenen Probleme zu vermeiden. Ein weiterer wichtiger Aspekt ist die vollständige Unterstützung von Java 1.5, um Entwicklern die Möglichkeit zu bieten, JML-Spezifikationen auch in Kombination mit den sprachlichen Neuerungen von Java verwenden zu können.

Die erste Aufgabe besteht darin, eine signifikante Teilsprache von JML als spätere Eingabesprache für *Jass* zu entwickeln. Der Begriff der Signifikanz bedeutet in diesem Zusammenhang, dass die wesentlichen Eigenschaften und Möglichkeiten von JML ohne größere Einschränkungen auch für die Teilsprache erhalten bleiben. Im Kern betrifft diese Forderung insbesondere die grundlegenden Ideen des Design by Contract; d.h. die zu entwickelnde Sprache muss es Benutzern ebenfalls erlauben, Verträge zwischen Softwarekomponenten zu definieren.
Ein weiterer Schritt besteht in der Integration dieser Teilsprache von JML als neue Eingabesprache in den Precompiler *Jass*. Dabei soll, so weit dies möglich ist, auf die bereits bestehende Architektur zurückgegriffen und diese im Bedarfsfall erweitert werden. Um diese Integration zu leisten, müssen unter anderem eine codeanalysierende Komponente, die Eingabedokumente auf syntaktische und

semantische Korrektheit hin untersucht, und eine codegenerierende Komponente, die die gegebenen Spezifikationen in diese prüfenden Quellcode übersetzt, entwickelt werden. Um die Probleme des JML RACC zu vermeiden, soll speziell für Spezifikationen in Interfaces eine alternative Übersetzungsstrategie entwickelt und in der codegenerierenden Komponente implementiert werden.
Abschließend bleibt noch die Aufgabe, die Wirksamkeit insbesondere der alternativen Übersetzungsstrategie nachzuweisen. Dazu sollen anhand einer Fallstudie die Arbeitsweisen von *Jass* und des JML RACC vergleichend untersucht werden.

1.2 Aufbau der Arbeit

Im nächsten Kapitel dieser Arbeit werden kurz die für das weitere Verständnis benötigten Grundlagen vorgestellt. Diese umfassen zunächst eine Einführung in die wichtigsten Sprachelemente der Spezifikationssprache JML und die Vorstellung deren syntaktischen und semantischen Aufbaus. Daran schließt sich eine vergleichende Vorstellung der bisherigen Eingabesprache von *Jass* an.
Das dritte Kapitel beschreibt die Konzeption der Integration von JML in *jass*. Dazu werden in einem ersten Abschnitt die verwendeten Technologien kurz vorgestellt; außerdem wird auf die Neuerung in Java 1.5 gegenüber den vorherigen Versionen eingegangen. Anschließend wird die Grammatik der signifikanten Teilsprache von JML, die als neue Eingabesprache für *Jass* fungieren soll, entwickelt und vorgestellt. Den Abschluss der Konzeption bildet die Beschreibung der Architektur des Precompilers und die Erläuterung der benötigten Übersetzungsmuster für die codegenerierende Komponente.
Kapitel 4 wird im ersten Teil einige Implementierungsdetails vorstellen. Dabei wird insbesondere auf die Realisierung der Übersetzungsmuster für Spezifikationen in Interfaces und Modellvariablen eingegangen, wobei die konzeptionellen Unterschiede zum JML Runtime Assertion Checking Compiler herausgestellt werden. Im zweiten Teil wird der Übersetzungsvorgang anhand eines einfachen Beispiels exemplarisch vorgestellt. Dabei wird auch auf Besonderheiten, die während der Übersetzung zu beachten sind, eingegangen.
Im fünften Kapitel wird anhand einer kurzen Fallstudie *Jass* und der JML RACC vergleichend untersucht. Als Vergleichskriterien sollen dabei der generierte Java-Code und die Laufzeit der übersetzten Programme verwendet werden.
Den Abschluß dieser Arbeit bildet die Darstellung der gewonnenen Ergebnisse und ein Ausblick auf wünschenswerte Funktionalitäten und Verbesserungen, die im Rahmen dieser Diplomarbeit nicht realisiert werden konnten.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige grundlegende und für das spätere Verständnis notwendige Technologien und Konzepte vorgestellt. Zunächst erfolgt eine kurze Einführung in die *Java Modeling Language* (kurz: JML) in der die wichtigsten Aspekte dieser Spezifikationssprache angesprochen werden. Daran schließt sich ein Abschnitt über das Runtime-Checking Werkzeug Jass an, das im Rahmen dieser Diplomarbeit erweitert wurde und somit als Implementierungsgrundlage diente.

2.1 JML - Die Java Modeling Language

2.1.1 Einführung

Die *Java Modeling Language* wurde als eine formale, verhaltensorientierte Schnittstellenspezifikationssprache[1] an der Iowa State University unter der Leitung von Gary T. Leavens entwickelt. Als solche dient sie dazu, einerseits die Schnittstellen von Klassen und Interfaces in Java zu spezifizieren, und andererseits Aussagen über das gewünschte Verhalten während der Laufzeit zu machen. JML verfolgt dabei einen Ansatz, der als „Design by Contract“, kurz DBC, bekannt ist und insbesondere in [Mey92] vorgestellt und diskutiert wird.
Die Idee hinter DBC ist folgende: Bevor Methoden einer Klasse aufgerufen werden, schließen die aufrufende Instanz (der sogenannte Client) und die aufgerufene Instanz (die Klasse) einen „Vertrag“ miteinander ab. In diesem „Vertrag“ wird festgelegt:

- welche Bedingungen erfüllt sein müssen, damit der Client die Methode aufrufen darf,

[1] Gebräuchlicher ist der englische Ausdruck „behavioral interface specification language“, kurz *BISL*.

- welche Garantien die Klasse für den Zustand nach dem Aufruf gibt.

Die Verwendung von Vorbedingungen, die der Client erfüllen muss, und Nachbedingungen, die die Klasse garantiert, beruht auf der Arbeit von C.A.R. Hoare [Hoa69] und insbesondere auf den sogenannten „Hoareschen Tripeln“. Ein allgemeines Hoaresches Tripel ist gegeben in der Form

$$\{P\}\, S\, \{Q\}\,.$$

Dabei sind P und Q logische Formeln, die *Vorbedingung* bzw. *Nachbedingung* genannt werden, und S ein *Programm*, das meist wie etwa in [AO94] in der Form einer Programmiersprache mit einer formalen Semantik gegeben ist. Die vereinfachte Bedeutung dieses Ausdrucks ist folgende: Ist P vor der Ausführung von S erfüllt und terminiert die Berechnung von S, so ist nach der Ausführung Q erfüllt.
Aus Sicht der Java Modeling Language ist weniger die Erfülltheit von Vor- und Nachbedingungen im Sinne einer formalen Korrektheitsaussage nach Hoare von Interesse als vielmehr die Einhaltung der zwischen Client und Klasse geschlossenen Verträge. In diesem Sinne dient JML nicht dazu, die Korrektheit eines Programms nachzuweisen, wie Hoare das etwa in [Hoa69] verfolgte, sondern wird dazu verwendet, Aussagen im Sinne des Design by Contract über das Verhalten von Klassen und Methoden in der Form von bestimmten Spezifikationen zu definieren.

Das folgende einführende Beispiel soll einen Eindruck davon geben, wie eine Spezifikation der mehrfach angesprochenen Verträge zwischen Client und Klasse erfolgen kann. Gegeben sei eine einfache Java-Methode mit einer Fließkommazahl x als Argument, die eine Näherung der positiven Quadratwurzel von x berechnen und zurückliefern soll:

```
public float squareRoot(float x) {
  // berechne Näherung der positiven Quadratwurzel von x
}
```

Die genaue Art der Berechnung ist an dieser Stelle nicht relevant und sollte im Idealfall beliebig austauschbar sein. Eine für die Berechnung von reellen Quadratwurzeln wichtige Bedingung ist die Nicht-Negativität des Radikanten, da sonst keine (nichtkomplexe) Lösung existiert. Die umgangssprachlich beschriebenen Anforderungen an den Umgang mit dieser Methode sind also:

1. Die übergebene Fließkommazahl x soll nicht negativ sein, da sonst keine (im Sinne der reellen Zahlen) sinnvolle Berechnung möglich ist.

2. Das Ergebnis der Berechnung soll ungefähr der positiven Quadratwurzel des Arguments entsprechen.

JML-Spezifikationen werden in der Form von erweiterten ein- oder mehrzeiligen Annotationen angegeben, die sich von den gewöhnlichen Java-Kommentaren durch die Verwendung des einleitenden Symbols @ unterscheiden und, ähnlich wie formale Kommmentare, bereits *vor* der Methodendeklaration stehen können[2].
Zunächst wird die Nicht-Negativität des Arguments als *Vorbedingung* spezifiziert. In JML werden Vorbedingungen durch das Schlüsselwort *requires* eingeleitet:

```
//@ requires x >= 0.0;
public float squareRoot(float x) {
  // berechne Näherung der positiven Quadratwurzel von x
}
```

Aus der Sicht eines DBC-Vertrags ist damit ein Client, der diese Methode aufrufen will, zur Einhaltung der Bedingung *x >= 0.0* verpflichtet. Liegt zur Laufzeit eine Verletzung der Vorbedingung vor, so ist die aufgerufene Methode dann ihrerseits von den Pflichten des Vertrags entbunden; insbesondere sind in diesem Fall also keine Aussagen über das weitere Verhalten der Berechnung gegeben.

Abgeschlossen wird diese Spezifikation durch das Hinzufügen einer *Nachbedingung*, die durch das Schlüsselwort *ensures* eingeleitet wird:

```
//@ requires x >= 0.0;
//@ ensures -eps <= x - (\result * \result) <= eps;
public float squareRoot(float x) {
  // berechne Näherung der positiven Quadratwurzel von x
}
```

Durch *ensures* eingeleitete Nachbedingungen müssen nur dann erfüllt sein, wenn die Berechnung normal, d.h. nicht durch Werfen einer Exception, terminiert[3]. Die Bedingung *-eps <= x - (\result * \result) <= eps* ist äquivalent dazu, dass das Ergebnis der Berechnung ungefähr der Quadratwurzel von *x* entspricht. *eps* steht dabei für eine beliebige Toleranzgrenze, die in der die Methode umfassenden Klasse festgelegt sein muss. `\result` ist ein weiteres JML-Schlüsselwort, mit dem sich der Rückgabewert einer Methode referenzieren lässt. Eine ausführlichere Beschreibung der verschiedenen JML-spezifischen Sprachkonstrukte findet sich in Abschnitt 2.1.3.
Die Methode verpflichtet sich durch diese Nachbedingung dazu, eine Näherung der Quadratwurzel eines übergebenen Arguments zu berechnen, falls die Berechnung nicht durch einen Fehler abbricht. Damit ist die Spezifikation dieser Methode abgeschlossen. Der Vertrag zwischen Client und Klasse garantiert nun zur Laufzeit folgendes Verhalten: Wenn die Methode, aufgerufen mit einem nicht-negativen Argument, normal terminiert, dann ist das Ergebnis der Berechnung eine innerhalb einer festen Toleranzgrenze liegende Näherung der Quadratwurzel des Arguments.

[2] Weniger gebräuchlich, aber für das Verständnis unter Umständen einfacher ist es, Spezifikationen zwischen Methodenheader und -rumpf zu schreiben.

[3] Siehe dazu auch: Abschnitt 2.1.5

Es ist zu bemerken, dass die Nachbedingung *\ensures false;* im Unterschied zu der Korrektheitsformel

$$\{P\}\, S\, \{false\}$$

nicht notwendigerweise bedeutet, dass eine Berechnung divergieren muss, da die Ausführung der entsprechenden Methode auch durch das Werfen einer Exception beendet werden kann.

Dieses Beispiel zeigt, dass ein einfaches Verhalten mittels JML auch schnell und unkompliziert spezifiziert werden kann. Für einen erfahrenen Entwickler ist es bereits nach kurzer Einarbeitungszeit möglich, sinnvolle Spezifikationen zu schreiben, da die Syntax der JML-Ausdrücke sich stark an die Java-Syntax anlehnt. Zudem erlaubt die Prüfung des Vertrags zwischen Client und Klasse die Ermittlung eines „Schuldigen“ im Falle einer Vertragsverletzung: Wurde beispielsweise eine Vorbedingung verletzt, so liegt der Fehler beim Client, da er für die Einhaltung dieser Bedingung zuständig war; bei der Verletzung einer Nachbedingung muss umgekehrt die Klasse die „Schuld“ auf sich nehmen. Dies kann vor allem bei komplizierten und umfangreichen Systemen die Fehlersuche erheblich vereinfachen und beschleunigen.

JML erlaubt mittels eines umfangreichens Sortiments aus Konstrukten und Ausdrücken ein detailliertes Beschreiben des Verhaltens einer Klasse oder eines Interfaces. Diese Konstrukte und Ausdrücke werden in den folgenden Abschnitten genauer vorgestellt und erläutert.

2.1.2 Invarianten und Constraints

Neben der Spezifikation von Vor- und Nachbedingungen für die Ausführung von Methoden bietet JML auch die Möglichkeit, *Invarianten* und sogenannte *History Constraints* zu definieren. Beide spezifizieren Eigenschaften, die in einem oder mehreren der sogenannten *sichtbaren Zustände* der Programmausführung gelten müssen.

Defintion 1 (Sichtbare Zustände):

Als die *sichtbaren Zustände* einer Programmausführung sind nach [CCC+06, Abschnitt 8.2] definiert:

- das Ende eines Konstruktorenaufrufs,
- das Ende einer statischen Initialisierung,

- Beginn und Ende einer nicht-statischen Initialisierung,
- der Beginn eines Finalizer-Aufrufs[4], sowie
- Beginn und Ende eines Methodenaufrufs

Für die spätere Argumentation über die Semantik von JML-Ausdrücken spielen zwei sichtbare Zustände eine besondere Rolle:

Definition 2 (Pre-State und Post-State):

Der Zustand zu Beginn einer Methodenausführung, d.h. dem Zeitpunkt, nachdem die Methode aufgerufen wurde und die notwendigen Änderungen des Laufzeitstapels stattgefunden haben, aber noch bevor die erste Anweisung des Methodenrumpfs ausgeführt wurde, wird als *Pre-State* bezeichnet. Analog nennt man den Zustand am Ende einer Methodenausführung unmittelbar vor dem Rücksprung im Kontrollfluss den *Post-State*.

Zu bemerken ist, dass ein Post-State auf zwei unterschiedliche Arten erreicht werden kann: einerseits durch normale Terminierung der Methodenausführung, und andererseits durch Erreichen eines Ausnahmezustand, der in Java durch das Werfen einer Exception angezeigt wird.

Invarianten

Mittels *Invarianten* lassen sich Eigenschaften definieren, die während der gesamten Laufzeit eines Programms unverändert erfüllt sein müssen. Angewendet auf die sichtbaren Zustände der Programmausführung bedeutet dies, dass Invarianten in den meisten, nicht jedoch allen sichtbaren Zuständen gelten müssen. Diese Einschränkung wird erklärt durch die Unterscheidung zwischen statischen und instanzgebunden Invarianten, die später in diesem Abschnitt erläutert werden.

Die Syntax der Invarianten ist in erweiterter Backus-Naur-Form (kurz: EBNF) gegeben:

Invariante ::= (`invariant` | `invariant_redundantly`) *Prädikat* `;`

Ein *Prädikat* kann ein beliebiger, um JML-Ausdrücke[5] erweiterter boolescher Java-Ausdruck sein. Die Definition von Invarianten kann an beliebiger Stelle in

[4] In Java wird der Speicher von nicht mehr referenzierten Objekten in regelmäßigen Abständen durch die sogenannte „garbage collection“ freigegeben. Um eventuell noch belegte Ressourcen wie Netzwerkverbindungen ebenfalls freigeben zu können, kann eine Instanzmethode namens „finalize“ implementiert werden, die auch als „Finalizer“ bezeichnet wird. Details dazu finden sich insbesondere in [BGJS05].

[5] Die speziellen JML-Ausdrücke werden in Abschnitt 2.1.3 behandelt.

einer Klasse, ausgenommen innerhalb von Methoden und deren Spezifikationen, erfolgen. Das Schlüsselwort `invariant_redundantly` leitet eine redundante Invariantendefinition ein. Redundanz in Spezifikationen wird in Abschnitt 2.1.7 näher erläutert.

Das folgende Beispiel zeigt eine typische Invariante:

Beispiel 2.1 (Invariante)

```
//@ invariant i > 0;
```

Dabei soll `i` ein Feld des primitiven Typs `int` referenzieren. ■

In JML erfolgt analog zu statischen Methoden und Instanzmethoden in Java eine Unterscheidung zwischen *statischen Invarianten* und *Instanzinvarianten*. Statische und instanzgebundene Invarianten schließen sich gegenseitig aus; eine Invariante ist also entweder statisch oder instanzgebunden, nie jedoch beides. Werden Invarianten in Klassen definiert, so sind sie standardmäßig Instanzinvarianten; werden sie in Interfaces definiert, so werden sie als statische Invarianten behandelt. Um auch in Interfaces instanzgebundene und in Klassen statische Invarianten deklarieren zu können, ist eine Qualifizierung der Invariantendefinition möglich:

Qualifizierte_Invariante ::= (`static` | `instance`) *Invariante*

Die Verwendung der Schlüsselworte `static` und `instance` ist dabei selbst erklärend.

Der semantische Unterschied zwischen statischen und instanzgebundenen Invarianten liegt in den sichtbaren Zuständen der Programmausführung, in denen sie gelten müssen[6]. Statische Invarianten müssen gelten:

- wenn die statische Initialisierung einer Klasse abgeschlossen ist,
- zu Beginn und Ende der Ausführung eines Konstruktors,
- zu Beginn und Ende der Ausführung einer statischen Methode,
- zu Beginn und Ende der Ausführung einer nicht-statischen Methode.

[6] Siehe dazu auch [CCC+06, Abschnitt 8.2]

Instanzinvarianten müssen gelten:

- am Ende der Ausführung eines Konstruktors,
- zu Beginn und Ende der Ausführung einer Instanzmethode, die kein Finalizer ist.

Es sei an dieser Stelle noch einmal darauf hingewiesen, dass das Ende einer Methodenausführung auch durch das Werfen einer Exception erreicht werden kann; damit müssen Invarianten also auch in einem solchen Fall erfüllt sein. Dadurch soll sichergestellt werden, dass auch nach dem Auftreten eines Fehlers noch sinnvolle Aussagen über das Verhalten von späteren Methodenaufrufen gemacht werden können.

Invarianten können ebenfalls mit Zugriffsmodifikatoren[7] versehen werden:

Modifizierte_Invariante ::= *Modifikator Qualifizierte_Invariante*
Modifikator ::= (`public`|`protected`|`private`)

Der Zugriffsmodifikator regelt, auf welche Felder und Methoden einer Klasse in der Definition der Invarianten zurückgegriffen werden darf; er bestimmt jedoch nicht, welche Methoden die Invariante einhalten müssen. So darf eine als `private` gekennzeichnete Invariante nur auf private Felder und Methoden zurückgreifen; sie muss jedoch auch in den anwendbaren sichtbaren Zuständen einer `public`-deklarierten Methode erfüllt sein. Details zu Invarianten und Modifikatoren finden sich in [CCC+06, Abschnitt 8.2].

History Constraints

History Constraints sind den Invarianten sehr ähnlich. Im Unterschied zu Invarianten definieren sie jedoch Beziehungen zwischen dem momentanen Zustand der Programmausführung und einem früheren Zustand. Ein typischer Constraint ist im folgenden Beispiel gegeben:

Beispiel 2.2 (History Constraint)

```
//@ constraint list.size() >= \old(list.size());
```

Dabei bezieht sich `list` auf ein in der entsprechenden Klasse definiertes und instanziiertes Objekt eines beliebigen Subtyps von `java.util.AbstractCollection`. Das hier erstmals auftauchende `\old`-Konstrukt kann dazu verwendet werden, den Zustand eines Ausdrucks im

[7] Gebräuchlicher ist das englische Pendant *access modifiers*.

Pre-State der Methodenausführung zu referenzieren[8]. ■

Die Syntax der Constraints ist wie folgt gegeben:

Constraint ::= *Constraint_Schlüsselwort Prädikat* [`for` *Constraint_Liste*]
Constraint_Schlüsselwort ::= (`constraint` | `constraint_redundantly`)
Constraint_Liste ::= *Methodenname* [`,` *Methodenname*] | `\everything`

Ein weiterer Unterschied in der Verwendung von Constraints im Vergleich zu Invarianten ist die optionale Angabe einer Liste von Methoden, für die ein History Constraint gelten muss. Dadurch lässt sich der Geltungsbereich von Constraints auf die tatsächlich benötigten oder gewünschten Methoden einschränken. Die Angabe des Schlüsselwortes `\everything` wird bei Auslassung einer Methodenliste standardmäßig angenommen und bezieht alle Methoden in die Pflicht mit ein, den Constraint einzuhalten.

Bei den History Constraints unterscheidet man, ebenso wie bei den Invarianten, zwischen statisch und instanzgebunden:

Qualifizierter_Constraint ::= (`static` | `instance`) *Constraint*

Die Semantik von statischen und instanzgebundenen Constraints unterscheidet sich dabei aber von den entsprechend qualifizierten Invarianten. Nach [CCC+06, Abschnitt 8.3] *respektiert* eine Methode einen Constraint, falls Pre- und Post-State der Methodenausführung in der vom Constraint definierten Beziehung stehen. Dazu muss das angegebene *Prädikat* im Post-State erfüllt sein, wobei eventuell verwendete `\old`-Ausdrücke im Pre-State evaluiert wurden. Für das Beispiel 2.2 bedeutet dies, dass die Größe von `list` im Post-State nicht kleiner sein darf als sie im Pre-State war.
Statische History Constraints müssen respektiert werden von statischen Methoden, Instanzmethoden und Konstruktoren einer Klasse. Im Gegensatz dazu müssen Instanzconstraints nur von Instanzmethoden eingehalten werden. Diese Einschränkung erklärt sich dadurch, dass in Java aus einem statischen Kontext heraus nicht auf instanzgebundene Konstrukte wie Felder oder Methoden zugegriffen werden darf.

Zu bemerken ist, dass die Forderung, ein Konstruktor müsse einen Constraint respektieren, insofern problematisch ist, als die erste Anweisung eines Konstruktors eine Superreferenz sein kann, die nach [BGJS05, Abschnitt 8.8] Vorrang vor allen anderen Anweisungen hat. In solchen Fällen ist die Prüfung eines statischen Constraints streng genommen nicht sinnvoll möglich, da der Pre-State der Methodenausführung nach Aufrufen der Superreferenz bereits verlassen wurde und

[8] Details zu Ausdrücken, die `\old` verwenden, finden sich im Abschnitt 2.1.3

eine Auswertung der benötigten Ausdrücke erst unmittelbar nach dem Rücksprung in den Konstruktorenkörper möglich ist. Eine Möglichkeit, mit diesem Problem umzugehen, ist in [CCC+06, Abschnitt 8.3] gegeben. Dort wird vorgeschlagen, die benötigten Auswertungen nach der eigentlichen Übersetzung einer Klasse durch eine Bytecodemanipulation einzufügen. Dieser Ansatz sei jedoch nur der Vollständigkeit halber erwähnt und soll aus Gründen des Umfangs dieser Arbeit nicht weiter verfolgt werden.

Für die Zugriffsmodifikatoren gelten für History Constraints dieselben Regeln wie für die Invarianten:

Modifizierter_Constraint ::= *Modifikator Qualifizierter_Constraint*
Modifikator ::= (`public` | `protected` | `private`)

Auch hier bestimmen diese Modifikatoren lediglich, auf welche Felder und Methoden zugriffen werden darf, nicht jedoch, welche Methoden den Constraint respektieren müssen. Details dazu können in [CCC+06, Abschnitt 8.3] nachgelesen werden.

2.1.3 JML-spezifische Ausdrücke und Operatoren

In den vorangegangen Abschnitten fanden bereits mehrere JML-spezifische Konstrukte wie etwa `\result`- oder `\old`-Ausdrücke Erwähnung. Die wichtigsten Ausdrücke und Operatoren, die nicht zum ursprünglichen Sprachumfang von Java gehören, werden im Folgenden dargestellt und erläutert.

`\result`-Ausdrücke

Die Syntax eines `\result`-Ausdrucks ist wie folgt:

*result*_Ausdruck ::= `\result`

`\result`-Ausdrücke werden dazu verwendet, in `ensures`-Ausdrücken den Rückgabewert einer nicht als `void` deklarierten Methode zur Laufzeit zu referenzieren. Der Typ eines `\result`-Ausdrucks entspricht dabei dem Rückgabetyp der entsprechenden Methode. In Spezifikationen für Methoden ohne Rückgabewert und Konstruktoren ist die Verwendung nicht erlaubt.

\old-Ausdrücke

Die Syntax der \old-Ausdrücke ist die folgende:

*\old*_Ausdruck ::= \old (*Spezifikationsaudruck* [, IDENTIFIER])
| \pre (*Spezifikationsaudruck*)

\old-Ausdrücke können sich auf beliebige Spezifikationsausdrücke[9] beziehen. Die Semantik eines Ausdrucks der Form

```
\old(E);
```

entspricht der Semantik des Ausdrucks E im Pre-State der Methodenausführung. Die Verwendung eines zusätzlichen IDENTIFIERS, der ein zuvor definiertes Label refenzieren muss, bewirkt, dass der angegebene Spezifikationsausdruck nicht im Pre-State, sondern nach Erreichen des festgelegten Labels ausgewertet wird.
\old-Ausdrücke können sowohl für die Definition von Nachbedingungen als auch in History Constraints verwendet werden, wenn sie kein methodeninternes Label referenzieren. Ein Ausdruck der Form

```
\old(E, Label);
```

darf hingegen nur für Spezifikationen innerhalb der Methode[10], in der das Label gültig ist, benutzt werden.

Quantifizierte Ausdrücke

JML umfasst neben den logischen Existenz- und Universalquantoren noch eine Reihe spezieller Quantoren, die für Aussagen über einen bestimmten numerischen Bereich verwendet werden können. Letztere sind für diese Arbeit jedoch nicht relevant; Details zu diesen numerischen Quantoren finden sich insbesondere in [CCC+06, Abschnitt 11.4].

Die Syntax der quantifizierten Ausdrücke ist wie folgt:

Quantifizierter_Ausdruck ::= (*Quantifizierung Quantifizierte_Variablen* ;
[[*Prädikat*] ;]
Spezifikationsausdruck)
Quantifizierung ::= \forall | \exists
Quantifizierte_Variablen ::= *Typ Variablendeklarator* (, *Variablendeklarator*)*

[9] Spezifikationsausdrücke umfassen sowohl die einfachen Java-Ausdrücke als auch die zusätzlichen JML-spezifischen Ausdrücke. Genaueres dazu findet sich in der Gesamtgrammatik im Anhang.

[10] Spezifikationen innerhalb einer Methode umfassen insbesondere die Schleifenspezifikationen, die später in diesem Abschnitt vorgestellt werden.

`\forall`- und `\exists`-Ausdrücke können dazu verwendet werden, universelle bzw. existenzielle Aussagen über einen Bereich von Werten oder über die Existenz eines bestimmten Wertes für die spezifizierten Variablen zu machen. Nach [CCC+06, Abschnitt 11.4] wird in JML stets über alle *potenziellen* Werte des Typs der angegebenen Variablen quantifiziert. Bei Quantifizierungen über nicht-primitive Referenztypen kann dies auch `null`-Referenzen und noch nicht erzeugte bzw. bereits gelöschte Objekte umfassen[11]. Um diesen umfassenden Wertebereich einzuschränken, können durch Angabe des optionalen *Prädikats* nicht erwünschte Werte ausgeschlossen werden.

Folgendes Beispiel verdeutlicht die Verwendung von quantifizierten Ausdrücken:

Beispiel 2.3 (Quantifizierung)

```
(\forall int i;
         i >= 0 && i < list.size();
         list.get(i) != null)
```

Hier wird die Aussage getroffen, dass alle Elemente eines Objekts `list` von `null` verschieden sind. Das Prädikat `i >= 0 && i < list.size()` definiert dabei den Wertebereich der Variablen, für den die Aussage `list.get(i) != null` erfüllt sein muss. Eine ausgelassene Spezifikation des Wertebereichs könnte in diesem Beispiel leicht zu Fehlern wie zum Beispiel Nullpointerreferenzierungen oder Arraygrößenverletzungen führen. Um diese Problematik zu vermeiden, sollte in der Regel ein einschränkendes Prädikat definiert werden. ■

Aus der Sicht der Laufzeitprüfung von Spezifikationen ist zu bemerken, dass die Angabe eines den Wertebereich einer Quantifizierung einschränkenden Prädikats die Prüfung der Quantifizierung erst ermöglicht, da zur Laufzeit natürlich nur über in der Java Virtual Machine existierende Objekte Aussagen getroffen werden können. Außerdem kann durch ein solches Prädikat die Effizienz der Auswertung einer Quantifizierung etwa über einen primitiven, numerischen Datentyp wie die Integerzahlen erheblich gesteigert werden.

`\type`- und `\typeof`-Ausdrücke

`\type`- und `\typeof`-Ausdrücke erlauben Aussagen über Typen in Java. Ihre Syntax ist wie folgt:

*type*_Ausdruck ::= `\type` (*Typ*)
*typeof*_Ausdruck ::= `\typeof` (*Spezifikationsausdruck*)

[11] Zum Zeitpunkt der Fertigstellung dieser Arbeit herrschte über die semantischen Details der Quantifizierung noch keine abschließende Einigkeit.

Ein Ausdruck der Form

```
\type(T)
```

bedeutet für einen Referenztyp `T` dasselbe wie `T.class` bzw. äquivalent dazu `T.getClass()`; für einen primitiven Typ `T` bedeutet obiger Ausdruck dasselbe wie `T.TYPE`. Ein `\type`-Ausdruck entspricht also einer Objektreferenz vom Typ `java.lang.Class`, mittels der sich weitergehende Aussagen über den Typ `T` formulieren lassen. Details zu Klassenliteralen der Form `T.class` und `T.TYPE` finden sich insbesondere in [BGJS05, Abschnitt 15.8].

Ein `\typeof`-Ausdruck der Form

```
\typeof(E),
```

wobei `E` ein von `null` verschiedener, um JML-Ausdrücke erweiterter Java-Ausdruck sein kann, liefert den dynamischen Laufzeittyp des Ausdrucks `E` zurück. Für einen Ausdruck `E` des Laufzeittyps `T` sind `\type(T)` und `\typeof(E)` auf semantischer Ebene identisch. Für den Sonderfall einer `null`-Referenz ist der entsprechende Ausdruck `\typeof(null)` nicht definiert.

Der Subtypoperator <:

Um spezifische Aussagen bezüglich der Typisierung von Objekten und Klassen zur Laufzeit machen zu können, wurde in JML ein weiterer relationaler Operator eingeführt, der *Subtypoperator* `<:`. Ein binärer Ausdruck der Form

```
LHS_Ausdruck <: RHS_Ausdruck
```

wird dabei als Boolescher Ausdruck betrachtet, der genau dann zu `true` evaluiert, wenn der Typ von `LHS_Ausdruck` ein Subtyp des Typs von `RHS_Ausdruck` ist. Das folgende Beispiel erläutert die Verwendung des Subtypoperators:

Beispiel 2.4 (Subtypoperator)

Gegeben sei eine von `null` verschiedene Objektinstanz `exc` des Typs `java.lang.NullPointerException`. Dann sind folgende Aussagen wahr:

```
\typeof(exc) <: \type(java.lang.Exception)
exc.class <: \type(java.lang.Object)
\typeof(exc) <: java.lang.NullPointerException.getClass()
```

Insbesondere ist der Operator `<:` also reflexiv, wie sich in der letzten Zeile

zeigt.

■

Boolesche Operatoren

Um umfangreiche Spezifikationen leichter lesbar zu machen, wurden in JML einige Boolesche Operatoren eingeführt, die nicht zum Standardrepertoire von Java gehören. Zu den wichtigsten gehören die *Äquivalenz-*, *Inäquivalenz-* und *Implikationsoperatoren*, die nachfolgend vorgestellt werden.

Der Äquivalenzoperator `<==>` drückt die logische Äquivalenz zweier Boolescher Ausdrücke aus und ist bis auf die geringere Priorität gleichbedeutend zum Java-Operator `==`. Ein Ausdruck der Form

```
\old(list.size()) == 0 <==> list.size() == 0
```

bedeutet also dasselbe wie der geklammerte Ausdruck

```
(\old(list.size()) == 0) == (list.size() == 0),
```

ist aber intuitiver. Gleiches gilt für den Inäquivalenzoperator `<=!=>` und den Java-Operator `!=`.

Die Implikationsoperatoren `==>` und `<==` agieren ebenfalls als binäre Boolesche Operatoren und drücken logische Implikationen aus. Ein Ausdruck der Form `P ==> Q` ist äquivalent zu `Q <== P`. Beide sind als Abkürzung zu verstehen für den Standard-Java-Ausdruck `!P || Q`. Dabei ist zu beachten, dass zur Laufzeit durch die Semantik von Java nach [BGJS05, Abschnitt 15.2] eine Auswertung von `!P` zu `true` zur Folge hat, dass der Ausdruck `Q` nicht mehr evaluiert wird.

Schleifenspezifikationen

Schleifenspezifikationen umfassen *Invarianten* und *Varianten*. Die Syntax ist wie folgt:

Annotierte_Schleife ::= (*Schleifeninvariante*)*
(*Schleifenvariante*)*
Schleife
Schleifeninvariante ::= *Schleifeninvarianten_Schlüsselwort Prädikat* `;`
Schleifeninvarianten_Schlüsselwort ::= `maintaining` | `loop_invariant`
| `maintaining_redundantly`
| `loop_invariant_redundantly`
Schleifenvariante ::= *Schleifenvarianten_Schlüsselwort Spezifikationsausdruck* `;`

Schleifenvarianten_Schlüsselwort ::= `decreasing` | `decreases`
| `decreasing_redundantly`
| `decreases_redundantly`

Schleifeninvarianten werden dazu verwendet, Aussagen über die partielle Korrektheit[12] einer Schleife zu machen. Eine Schleife der Form

```
//@ loop_invariant I;
while (B) {
  // Ausführung des Schleifenrumpfes
}
```

beinhaltet also die Aussage, dass die Invariante `I`

- vor der Ausführung der Schleife gilt,
- bei jeder Iteration des Schleifenrumpfes erhalten bleibt,
- und nach der Ausführung der Schleife gilt[13].

Schleifenvarianten dienen in Ergänzung zu den Schleifeninvarianten der Frage nach der Terminierung einer Schleifenausführung. Eine Schleife der Form

```
//@ decreasing D;
while (B) {
  // Ausführung des Schleifenrumpfes
}
```

beinhaltet einen Ausdruck `D` vom Typ `int` oder `long`, der

- vor der Ausführung der Schleife einen beliebigen Anfangswert größer gleich Null hat,
- vor jedem Iterationsschritt des Schleifenrumpfes größer gleich Null ist,
- in jeder Iteration echt verringert wird.

Damit lässt sich die Terminierung einer so spezifizierten Schleife nachweisen[14].

[12] Insbesondere wird also keine Aussage über die Terminierung der Schleifenausführung gemacht.

[13] Nach Verlassen der Schleife ist insbesondere der Java-Ausdruck `I && !B` erfüllt, womit die Gesamtaussage der Hoareschen Korrektheitsformel { I } **while (B)** { **...** } { I $\wedge \neg$ B } gleich kommt.

[14] Details zur partiellen und totalen Korrektheit sowie zu Schleifeninvarianten und -varianten finden sich insbesondere in [AO94].

2.1.4 Modellvariablen und Initiallies

Modellvariablen

Um in Spezifikationen weitergehende Aussagen über den momentanen Zustand oder zukünftige Zustandsänderungen der Programmausführung machen zu können, besteht in JML die Möglichkeit, *Modellvariablen* einzuführen. Modellvariablen sind spezifikationsinterne Variablen, die einen Zustand oder einen Teil der gewünschten Zustandsbeschreibung repräsentieren können. Sie haben einige entscheidende Vorteile gegenüber anderen, auf den ersten Blick etwas einfacheren Ansätzen[15]:

- Modellvariablen erlauben eine strikte Trennung zwischen Spezifikation und Implementierung.
- Einfache Hilfsvariablen sind in der Regel nicht vor externen Modifikationen sicher, da Java keine Definition von nicht-konstanten, schreibgeschützten Feldern erlaubt.
- In Interfaces ist die Verwendung von Hilfsvariablen nicht bzw. nur sehr eingeschränkt möglich, da hier keine Deklarationen von nicht-konstanten Variablen möglich ist.
- Die Verwendung von Hilfsmethoden, die zur Evaluierung eines Zustands herangezogen werden könnten, wird mit zunehmender Spezifikationsgröße unübersichtlicher als die Verwendung einfacher Felder.
- Sowohl Hilfsvariablen als auch -methoden erlauben keine transparente[16] Spezifikationen, da sie als gewöhnlicher Programmcode stets mit ausgeführt bzw. verwaltet werden, während Modellvariablen beim Übersetzen optional ausgelassen werden können[17].

Mittels Modellvariablen lassen sich also klar gegliederte und transparente Spezifikationen schreiben, die sich, falls gewünscht, bei der Übersetzung des Quellcodes ignorieren lassen.

Die Einführung einer Modellvariable ist in JML durch Verwendung des Modifikators *model* in der Definition eines Feldes möglich:

```
//@ private model int size;
```

[15] Eine detaillierte Diskussion dieser Ansätze findet sich in [CELS04].

[16] Der Begriff der Transparenz bezieht sich hier auf die klare Trennung zwischen Spezifikationsanteilen und „normalem“ Programmcode.

[17] Einschließlich aller Spezifikationskonstrukte, die diese unter Umständen verwenden.

Die so definierte Modellvariable soll dann in beliebigen Spezifikationen für eine gewisse Aussage über den Zustand der Programmausführung stehen. Es fehlt also noch die Bestimmung eines Ausdrucks, der die Semantik der Variablen festlegt. Dazu ist das *represents*-Konstrukt vorgesehen:

```
//@ represents size <- list.size();
```

Dabei soll `list` ein in der entsprechenden Klasse definiertes und instanziiertes Objekt eines beliebigen Subtyps von `java.util.AbstractCollection` sein. Die linke Seite einer *represents*-Deklaration muss eine Referenz auf eine Modellvariable sein; der Ausdruck auf der rechten Seiten wird *Abstraktionsfunktion* genannt und muss bezüglich der Variablen typgerecht sein. Diese Definition legt fest, dass die Semantik der Modellvariablen *size* zu jedem sichtbaren Zeitpunkt der Programmausführung durch die Semantik ihrer Abstraktionsfunktion gegeben ist. Die Abstraktionsfunktion kann dabei ein beliebiger, jedoch typgerechter Java- oder JML-Ausdruck sein.

Die Syntax der *represents*-Ausdrücke ist wie folgt:

Represents ::= *Represents_Schlüsselwort Variablenreferenz Abstraktion* **;**
Represents_Schlüsselwort ::= (`represents` | `represents_redundantly`)
Abstraktion ::= *Funktionale_Abstraktion* | *Relationale_Abstraktion*
Funktionale_Abstraktion ::= (<− | =) *Ausdruck*
Relationale_Abstraktion ::= `\such_that` *Prädikat*

Neben der bereits vorgestellten *funktionalen Abstraktion* bietet JML auch die Möglichkeit, *relationale Abstraktionen* anzugeben. Diese verwenden das Schlüsselwort `\such_that` und erlauben die Angabe eines beliebigen Prädikats als Abstraktionsrelation. In dieser Form entfällt die Forderung nach Zuweisungkompatibilität zum Typ der Modellvariablen. Die Semantik der relationalen Abstraktion ist die folgende: Zu jedem sichtbaren Zeitpunkt der Programmausführung muss die Modellvariable so belegt sein, dass sie das angegebene Prädikat erfüllt; der genaue Wert des Feldes ist nicht von Interesse.

Es sei an dieser Stelle darauf hingewiesen, dass es aus der Sicht der Laufzeitprüfung von Spezifikationen zwar möglich ist, Modellvariablen mittels relationaler Abstraktion zu spezifizieren, diese jedoch aufgrund der hohen Komplexität der Darstellung und Auswertung von Relationen zur Laufzeit in der Regel nicht berücksichtig werden. Im weiteren Verlauf dieser Arbeit werden aus diesem Grund nur die Abstraktionsfunktionen Beachtung finden.

Analog zu Java ist es möglich, statische und instanzgebundene Modellvariablen zu definieren. Eine statische Modellvariable muss durch einen ebenfalls statischen *represents*-Ausdruck spezifiziert werden; gleiches gilt für instanzgebundene Variablen. Dabei ist zu beachten, dass in der Abstraktionsfunktion eines

statischen *represents*-Ausdrucks nur statische Felder und Methoden einer Klasse referenziert werden dürfen. Eine weitere Einschränkung der Verwendung statischer Modellvariablen ist die Forderung, dass die Semantik des Feldes in derselben Klasse bzw. demselben Interface festgelegt werden muss, in der die Variable definiert wurde. Die Definition der Semantik instanzgebundener Modellvariablen kann im Gegensatz dazu auch durch Subtypen der Klasse oder des Interfaces [18] erfolgen.

Zu bemerken ist, dass Abstraktionsfunktionen in *represents*-Ausdrücken rekursiv definiert sein können, d.h. sie können die Modellvariable auf der linken Seite referenzieren. Dabei liegt Verantwortung der Wohldefiniertheit solcher Ausdrücke beim Entwickler.

Initiallies

Initiallies können dazu verwendet werden, Aussagen über den initialen Zustand einer Klasseninstanz zu machen, der sich aus den Initialwerten der in der entsprechenden Klasse definierten Felder ableitet. Dies schließt insbesondere auch Modellvariablen mit ein.

Gegeben sei folgendes Beispiel, das den Initialzustand der bereits im vorherigen Abschnitt definierten Modellvariablen *size* spezifiziert:

Beispiel 2.5 (Initially)

```
//@ initially size == 0;
```

Die Semantik dieses Ausdruck ist folgende: Im ersten sichtbaren Zustand nach der Initialisierung eines Objekts der Klasse, in der der *initially*-Ausdruck definiert ist, besitzt die Modellvariable `size` den Initialwert `0`.

■

Die Syntax der *initially*-Ausdrücke ist wie folgt:

Initially ::= `initially` *Prädikat* **`;`**

Es ist ebenfalls möglich, statische Initiallies zu definieren:

Qualifizierter_Initially ::= (`static` | `instance`) *Initially*

Statische *initially*-Ausdrücke müssen im Gegensatz zu instanzgebundenen Initiallies nicht für jedes neu instanziierte Objekt eines Typs gelten, sondern nur einmalig

[18] Siehe dazu auch Abschnitt 2.1.8

im ersten sichtbaren Zustand nach der Initialisierung der entsprechenden Klasse durch die Java Virtual Machine. Details zur Initialisierung von Typen finden sich in [BGJS05, Abschnitt 12.4].

2.1.5 Methodenspezifikationen

Methodenspezifikationen machen den Kern einer jeden JML-Spezifikation aus und erlauben es, detaillierte Aussage über das erwartete Verhalten von Methoden während der Programmausführung zu machen. Neben den bereits im einführenden Beispiel erwähnten Vor- und Nachbedingungen, die standardmäßig zum Umfang von BISLs[19] gehören, bietet JML einige besondere Möglichkeiten:

- Die Unterscheidung zwischen normaler Terminierung von Methodenausführungen und Terminierung durch Werfen einer Exception erlaubt eine genaue Beschreibung des Verhaltens von Methoden zur Laufzeit.
- Zugriffsmodifikatoren für Spezifikationen ermöglichen eine hierarchische Gestaltung der Spezifikationen.
- Die Definition von Rahmenbedingungen wie das Einschränken von Feldern, die durch eine Methode manipuliert werden dürfen, erlaubt eine weitere Verfeinerung von Spezifikationen.
- Durch die Verwendung von Redundanz können wichtige Folgen von Methodenspezifikationen herausgestellt werden.

Syntax der Methodenspezifikationen

Im Folgenden wird die Syntax der Methodenspezifikationen in mehreren Schritten vorgestellt. Dabei wird aus Gründen der Einfachheit zumeist nur von Spezifikationen für Methoden in Klassen die Rede sein, da die Definition von Methodenspezifikationen in Interfaces analog erfolgt. Der Aufbau des folgenden Abschnitt orientiert sich an der hierarchischen Organisation der Methodenspezifikationen und insbesondere an [CCC+06, Abschnitt 9].

Methodenspezifikation ::= *Spezifikation* | *Erweiterte_Spezifikation*
Erweiterte_Spezifikation ::= `also` *Spezifikation*
Spezifikation ::= *Spezifikationsfallsequenz* [*Redundante_Spezifikation*]
| *Redundante_Spezifikation*
Spezifikationsfallsequenz ::= *Spezifikationsfall* [`also` *Spezifikationsfall*]

[19] BISL: kurz für engl. „behavioral interface specification language“. Siehe dazu auch Abschnitt 2.1.1.

Eine Methodenspezifikation besteht zunächst im Wesentlichen aus einer durch `also` getrennte Folge von Spezifikationsfällen. Dabei muss die Spezifikation einer Methode in einer Klasse genau dann mit `also` beginnen, falls in mindestens einem Supertyp der Klasse bereits eine Spezifikation für die entsprechende Methode existiert. Redundante Spezifikationen nehmen eine Sonderrolle ein, da sie es erlauben, auf wichtige Konsequenzen von Methodenspezifikationen hinzuweisen[20].

Die Spezifikationsfälle in Methodenspezifikationen haben die folgende Form:

Spezifikationsfall ::= *Lightweight_Spezifikation* | *Heavyweight_Spezifikation*
| *Modellprogramm* | *Code_Contract_Spezifikation*

Da Modellprogramme ein eher selten benutztes Konstrukt sind, das für diese Diplomarbeit keine Rolle spielt, sei an dieser Stelle auf [CCC+06, Abschnitt 14] verwiesen. Ebenfalls nicht besprochen werden in dieser Arbeit die Code-Contract-Spezifikation. Auch hier sei auf [CCC+06, Abschnitt 15.2] als weiterführende Literatur verwiesen.

Lightweight- und Heavyweightspezifikationen

Die Syntax der *Lightweightspezifikationen* ist folgende:

Lightweight_Spezifikation ::= *Generische_Spezifikation*
Generische_Spezifikation ::= [*Spezifikationsvariablen*]
Spezifikationsheader
[*Generischer_Spezifikationskörper*]
|
[*Spezifikationsvariablen*]
Generischer_Spezifikationskörper
Generischer_Spezifikationskörper ::= *Einfacher_Spezifikationskörper*
| *Eingebettete_Spezifikation*
Eingebettete_Spezifikation ::= { | *Generische_Spezifikationssequenz* | }
Generische_Spezifikationssequenz ::= *Generische_Spezifikation*
(`also` *Generische_Spezifikation*)*
Spezifikationsheader ::= (`requires`-*Ausdruck*)+
Einfacher_Spezifikationskörper ::= (*Einfache_Spezifikation*)+
Einfache_Spezifikation ::= `diverges`-*Ausdruck* | `assignable`-*Ausdruck*
| `captures`-*Ausdruck* | `when`-*Ausdruck*
| `working-space`-*Ausdruck* | `duration`-*Ausdruck*
| `ensures`-*Ausdruck* | `signals`-*Ausdruck*
| `signals-only`-*Ausdruck*

[20] Redundanz wird in Abschnitt 2.1.7 genauer besprochen.

In Lightweightspezifikationen können zunächst einige *Spezifikationsvariablen* deklariert werden[21]. Daran kann sich ein *Spezifikationsheader* anschließen, der aus einer Folge von `requires`-Ausdrücken besteht, die die Vorbedingung der Methodenspezifikation ausmachen. Abschließend können im *generischen Spezifikationskörper* die Nachbedingung der Methodenspezifikation sowie Rahmenbedingungen, die zur Laufzeit eingehalten werden müssen, definiert werden. Im Rahmen dieser Arbeit werden nur die für die Spezifikation von Nachbedingungen relevanten Ausdrücke genauer vorgestellt werden; Details zu den Rahmenbedingungen finden sich insbesondere in [CCC+06, Abschnitt 9.9].
Eingebette Spezifikationen lassen sich auf einfache Weise in mehrere durch `also` getrennte, nicht eingebettete Spezifkationen umwandeln. Dies wird im Abschnitt „Desugaring von Spezifikationen“ später in diesem Kapitel dargestellt.

Die Semantik einer Lightweightspezifikation der Form

```
requires P;
ensures Q;
signals (Exception) S;
```

ist wie folgt: Wird die Methode in einem Zustand aufgerufen, in dem die Vorbedingung P und alle anwendbaren Invarianten erfüllt sind, dann

- terminiert die Berechnung in einem Zustand, der Q erfüllt, oder
- terminiert die Berechnung durch das Werfen einer Exception in einem Zustand, der S erfüllt, oder
- terminiert die Ausführung der Java Virtual Machine durch das Werfen einer nicht behandelbaren Exception vom Typ `java.lang.Error`.

Außerdem sind in jedem sichtbaren Zustand der Ausführung alle anwendbaren Invarianten und Constraints erfüllt, und alle Rahmenbedingungen wurden eingehalten.

Ein typisches Beispiel einer Lightweightspezifikation ist etwa mit der bereits vorgestellten Methode zur Berechnung einer Quadratwurzel gegeben:

Beispiel 2.6 (Lightweightspezifikation)

```
//@ requires x >= 0.0;
//@ ensures -eps <= x - (\result * \result) <= eps;
public float squareRoot(float x) {
  // berechne Näherung der positiven Quadratwurzel von x
}
```

[21] Details zu Definition und Semantik von Spezifikationsvariablen finden sich in Abschnit 2.1.3

Der Spezifikationsheader besteht hier aus genau einem `requires`-Ausdruck, der Spezifikationskörper aus genau einem `ensures`-Ausdruck. Spezifikationsvariablen werden nicht benötigt.

■

Eine Lightweightspezifikation hat per Definition stets dieselbe Sichtbarkeit wie die Methode, deren Verhalten sie spezifiziert. Um hierarchisch abgestufte Spezifikationen unterschiedlicher Sichtbarkeit schreiben zu können, bietet JML die Möglichkeit, *Heavyweightspezifikationen* zu definieren. Dabei gibt es drei verschiedene Arten von Heavyweightspezifikationen:

- *Behavior*-Spezifikationen
- *Normal-Behavior*-Spezifikationen
- *Exceptional-Behavior*-Spezifikationen

Behavior-Spezifikationen

Die Behavior-Spezifikation ist die allgemeinste Form der Methodenspezifikation. Normal- und Exceptional-Behavior-Spezifikationen sowie Lightweightspezifikationen lassen sich durch den sogenannten „Desugaring"-Prozess in Behavior-Spezifikationen der gleichen Semantik überführen[22].

Die Syntax der Behavior-Spezifikationen ist folgende:

Behavior-Spezifikation ::= [*Sichtbarkeit*]
Behavior_Schlüsselwort
Generische_Spezifikation
Behavior_Schlüsselwort ::= `behavior` | `behaviour`
Sichtbarkeit ::= `public` | `protected` | `private`

Der syntaktische Unterschied zu Lightweightspezifikationen besteht also lediglich aus der Voranstellung eines der äquivalenten Schlüsselworte `behavior` oder `behaviour` und der Möglichkeit, eine explizite Sichtbarkeit der Spezifikation zu definieren. Semantische Unterschiede finden sich in den impliziten Standardwerten für Konstrukte wie zum Beispiel `ensures`- oder `signals`-Ausdrücken, die in eigenen Abschnitten besprochen werden.

Ein anschauliches Beispiel einer Behavior-Spezifikation erhält man durch Umschreiben der bereits mehrfach benutzten Spezifikation einer Methode zur Berechnung der Quadratwurzel eines Arguments:

[22] Siehe dazu auch den Unterabschnitt „Desugaring von Spezifikationen" in Abschnitt 2.1.5.

Beispiel 2.7 (Behaviorspezifikation)

```
/*@ behavior
  @   requires x >= 0.0;
  @   ensures -eps <= x - (\result * \result) <= eps;
  @*/
  public float squareRoot(float x) {
    // berechne Näherung der Quadratwurzel von x
  }
```

Es sei darauf hingewiesen, dass sich die hier angegebene Behavior-Spezifikation aufgrund der unterschiedlichen impliziten Standardwerte semantisch von der scheinbar äquivalenten Lightweightspezifikation in Beispiel 2.6 unterscheidet. Details dazu finden sich in den entsprechenden Abschnitten der verschiedenen JML-Ausdrücke für Vor- und Nachbedingungen. ■

Normal-Behavior-Spezifikationen

Eine Normal-Behavior-Spezifikation kann dazu verwendet werden, das „normale“ Verhalten von Methoden zur Laufzeit zu spezifizieren. Dabei bedeutet „normales“ Verhalten, dass die Ausführung einer Methode weder divergiert noch durch das Werfen einer Exception terminiert. Normal-Behavior-Spezifikationen sind als syntaktische Abkürzungen für Behavior-Spezifikationen derselben Semantik zu verstehen[23].

Die Syntax der Normal-Behavior-Spezifikationen ist wie folgt:

Normal_Behavior_Spezifikation ::= [*Sichtbarkeit*]
Normal_Behavior_Schlüsselwort
*Generische_Spezifikation**
Normal_Behavior_Schlüsselwort ::= `normal_behavior` | `normal_behaviour`

Dabei ist zu bemerken, dass im generischen Spezifikationskörper von Normal-Behavior-Spezifikationen die Verwendung von `signals`- und `signals-only`-Ausdrücken untersagt ist, da gerade die normale Terminierung modelliert werden soll.

Als Beispiel sei wieder auf die übliche Methode zurückgegriffen:

[23] Genaueres dazu im Abschnitt „Desugaring von Spezifikationen“.

Beispiel 2.8 (Normal-Behavior-Spezifikation)

```
/*@ normal_behavior
  @    requires x >= 0.0;
  @    ensures -eps <= x - (\result * \result) <= eps;
  @*/
   public float squareRoot(float x) {
     // berechne Näherung der positiven Quadratwurzel von x
   }
```

Im Unterschied zu der vorangegangenen Behavior-Spezifikation lässt diese Normal-Behavior-Spezifikation keine Terminierung durch das Werfen einer Exception mehr zu.

■

Exceptional-Behavior-Spezifikationen

So wie Normal-Behavior-Spezifikationen zur Modellierung des normalen Laufzeitverhaltens von Methoden verwendet werden können, kann mittels Exceptional-Behavior-Spezifikationen das Verhalten von Methoden im Falle einer Terminierung durch Werfen einer Exception genauer definiert werden. Allerdings ist auch hierbei die Divergenz von Berechnungen ausgeschlossen. Auch Exceptional-Behavior-Spezifikationen sind als syntaktische Abkürzung für Behavior-Spezifikationen der gleichen Semantik zu verstehen.

Die Syntax der Exceptional-Behavior-Spezifikationen ist wie folgt:

Exceptional_Behavior_Spezifikation ::= [*Sichtbarkeit*]
Exceptional_Behavior_Schlüsselwort
*Generische_Spezifikation***

Exceptional_Behavior_Schlüsselwort ::= `exceptional_behavior`
| `exceptional_behaviour`

Abweichend von der Syntax des gewöhnlichen generischen Spezifikationskörpers dürfen in Exceptional-Behavior-Spezifikationen keine `ensures`-Ausdrücke verwendet werden.

Zur Anschauung wird die Spezifikation der Beispielmethode um folgende Forderung erweitert: Wenn das übergebene Argument negativ sein sollte, so wird dies durch das Werfen einer `NegativeArgumentException` signalisiert. Realisiert wird dies durch die Verknüpfung einer Normal- und einer Exceptional-Behavior-Spezifikation:

Beispiel 2.9 (Exceptional-Behavior-Spezifikation)

```
/*@ normal_behavior
  @    requires x >= 0.0;
  @    ensures -eps <= x - (\result * \result) <= eps;
  @ also
  @ exceptional_behavior
  @    requires x < 0.0;
  @    signals (NegativeArgumentException) true;
  @*/
  public float squareRoot(float x)
      throws NegativeArgumentException {
    // signalisiere Negativität des Arguments
    // berechne Näherung der positiven Quadratwurzel von x
  }
```

■

Da sich die normale Terminierung und die Terminierung durch das Werfen einer Exception wechselseitig ausschließen, ist es wichtig, auf die *Konsistenz* der Vorbedingungen zu achten. Konsistenz bedeutet in diesem Kontext, dass sich die Vorbedingungen von verknüpften Normal- und Exceptional-Behavior-Spezifikationen ebenfalls gegenseitig ausschließen müssen und nicht zur selben Zeit erfüllbar sein dürfen.

Spezifikationsvariablen

Spezifikationsvariablen sind lokale Variablen, die nur innerhalb der direkt nachfolgenden Spezifikation gültig sind. Sie dürfen nicht mit Modellvariablen verwechselt werden, deren Sichtbarkeit wie bei gewöhnlichen Java-Feldern vom jeweiligen Modifikator abhängt.

Die Syntax der Spezifikationsvariablendeklaration ist folgende:

Spezifikationsvariablen ::= *Forall_Variablen* [*Old_Variablen*]
| *Old_Variablen*

Es wird unterschieden zwischen *Forall*-Variablen und *Old*-Variablen. Mittels *Forall*-Variablen können universal quantifizierte Spezifikationsaussagen gemacht werden, die genau wie die *forall*-Ausdrücke für alle möglichen Werte der quantifizierten Variablen gelten müssen. *Old*-Variablen hingegen können als Abkürzung für *old*-Ausdrücke verwendet werden[24].

[24] *forall*- und *old*-Ausdrücke werden im Abschnitt 2.1.3 genauer erläutert.

Die Syntax der *Forall*-Variablen ist wie folgt:

Forall_Variablen ::= (`forall` *Quantifizierte_Variable* `;`)+
Quantifizierte_Variable ::= *Typ Variablendeklarator* (`,` *Variablendeklarator*)*
Variablendeklarator ::= `IDENTIFIER` (`[]`)*

Die Syntax der *Old*-Variablen ist wie folgt:

Old_Variablen ::= (`old` *Spezifikations_Variable* `;`)+
Spezifikations_Variable ::= *Variablendeklarator* [= *Variableninitialisierung*]

Die Semantik der Spezifikationsvariablen entspricht der Semantik der jeweiligen JML-Ausdrücke, die im Abschnitt 2.1.3 erläutert werden.

`requires`-Ausdrücke

`requires`-Ausdrücke werden im Spezifikationsheader dazu verwendet, eine Vorbedingung zu definieren, die im Pre-State der Methodenausführung erfüllt sein muss. Die Syntax ist die folgende:

requires_Ausdruck ::= *requires_Schlüsselwort Prädikat_Oder_Nicht* `;`
requires_Schlüsselwort ::= `requires` | `pre`
| `requires_redundantly` | `pre_redundantly`
Prädikat_Oder_Nicht ::= *Prädikat* | `\not_specified`

Wird anstelle eines Prädikats das Schlüsselwort `\not_specified` verwendet, so kann dies als beliebiger, je nach Sichtweise der Spezifikation unterschiedlicher Boolescher Wert aufgefasst werden. Es steht damit unter anderem also verschiedenen Runtime-Checking-Werkzeugen auch eine jeweils eigene Interpretation frei; der JML Runtime Assertion Checking Compiler etwa interpretiert ein `\not_specified` stets als Booleschen Ausdruck `true`.

Eine Methodenspezifikation ohne explizit definierte Vorbedingung wird erweitert um

- einen impliziten `requires`-Ausdruck der Form `requires \not_specified;`, falls es sich um eine Lightweightspezifikation handelt,
- einen impliziten `requires`-Ausdruck der Form `requires true;`, falls es sich um eine Heavyweightspezifikation handelt.

Im Spezifikationsheader selbst können beliebig viele `requires`-Ausdrücke verwendet werden. Die Semantik einer Spezifikation der Form

```
requires P1;
requires P2;
...
requires Pn;
```

entspricht dabei der konjunktiven Verknüpfung der einzelnen Prädikate:

```
requires P1 && P2 && ... && Pn;
```

ensures-Ausdrücke

`ensures`-Ausdrücke spezifizieren das Verhalten bei einer normalen Terminierung der Methodenausführung. Die Syntax ist wie folgt:

ensures_Ausdruck ::= *ensures_Schlüsselwort Prädikat_Oder_Nicht* `;`
ensures_Schlüsselwort ::= `ensures` | `post`
| `ensures_redundantly`
| `post_redundantly`

Im Prädikat eines `ensures`-Ausdruck können beliebige `\old`-Ausdrücke sowie das `\result`-Konstrukt verwendet werden, um auf den Zustand von Variablen im Pre-State oder auf den Rückgabewert einer nicht `void`-deklarierten Methode zurückzugreifen. Dabei gilt ein Ausdruck `ensures Q;` als erfüllt, wenn die Implikation $\backslash old(P) \Rightarrow Q$ erfüllt ist, wobei `P` das Prädikat der Vorbedingung ist. War also die Vorbedingung im Pre-State der Methodenausführung erfüllt, dann muss die spezifizierte Nachbedingung im Post-State gelten.

Werden in der Spezifikation von Nachbedingungen Methodenparameter verwendet, so werden diese mit den Werten des Pre-States interpretiert. Ein Parameter `p` ist im Prädikat eines `ensures`- oder auch `signals`-Ausdrucks also implizit in ein `\old`-Konstrukt der Form `\old(p)` eingebettet.

Eine Methodenspezifikation ohne explizit definierte normale Nachbedingung wird erweitert um

- einen impliziten `ensures`-Ausdruck der Form `ensures \not_specified;`, falls es sich um eine Lightweightspezifikation handelt,
- einen impliziten `ensures`-Ausdruck der Form `ensures true;`, falls es sich um eine Heavyweightspezifikation handelt.

In einem Spezifikationskörper können beliebig viele `ensures`-Ausdrücke verwendet werden. Die Semantik einer Spezifikation der Form

```
ensures Q1;
ensures Q2;
...
ensures Qn;
```

entspricht - wie bei den Vorbedingungen - der konjunktiven Verknüpfung der einzelnen Prädikate:

```
ensures Q1 && Q2 && ... && Qn;
```

`signals`-Ausdrücke

`signals`-Ausdrücke können in Spezifikationen dazu verwendet werden, das Verhalten einer Methode bei Terminierung durch das Werfen einer Exception festzulegen. Die Syntax ist die folgende:

signals_Ausdruck ::= *signals_Schlüsselwort*
(*Exception_Typ*)
[`IDENTIFIER`]
[*Prädikat_Oder_Nicht*] **;**
signals_Schlüsselwort ::= `signals` | `exsures`
| `signals_redundantly`
| `exsures_redundantly`

Der angegebene Typ der signalisierten Exception muss ein Subtyp von `java.lang.Exception` sein; nicht erlaubt sind Subtypen von `java.lang.Error`, da das Werfen solcher Fehler nach [BGJS05, Abschnitt 11.2] eine schwerwiegende Ausnahme signalisiert, die nicht behandelt werden darf. Außerdem muss er ein Sub- oder Supertyp einer der im `throws`-Ausdruck der Methode angegebenen Exceptions sein. Der optionale `IDENTIFIER` wird im direkt nachfolgenden Prädikat als gebundene Variable betrachtet, die die geworfene Exception referenziert und dazu verwendet werden kann, genauere Aussagen über die geworfene Exception zu machen. Wird kein Prädikat definiert, so ist dies als impliziter Ausdruck `true` zu verstehen.

Ein `signals`-Ausdruck der Form

```
signals (E e) S;
```

ist äquivalent zu einem verallgemeinerten `signals`-Ausdruck der Form

```
signals (java.lang.Exception e)
            (e instanceof E) ==> S;
```

Die im Prädikat des obigen Ausdrucks verwendete logische Implikation `(e instanceof E) ==> S;` ist ein JML-spezifischer Ausdruck und wurde in Abschnitt 2.1.3 bereits erklärt. Analog zu den normalen Vorbedingungen ist ein `signals`-Ausdruck der obigen Form erfüllt, falls die Implikation *\old(P) ⇒ ((e instanceof E) ⇒ S)* erfüllt ist.

Eine Methodenspezifikation ohne explizit definierte Nachbedingung für die Terminierung durch Werfen einer Exception wird erweitert um

- einen impliziten `signals`-Ausdruck der Form `signals \not_specified;`, falls es sich um eine Lightweightspezifikation handelt,
- einen impliziten `signals`-Ausdruck der Form `signals (Exception) true;`, falls es sich um eine Heavyweightspezifikation handelt.

Folgen von `signals`-Ausdrücken in einer Methodenspezifikationen lassen sich nach der bereits vorgestellten Verallgemeinerung wie folgt vereinfachen: Aus

```
signals (E1 e) P1;
signals (E2 e) P2;
...
signals (En e) Pn;
```

wird

```
signals (java.lang.Exception e)
            (e instanceof E1) ==> P1
         && (e instanceof E2) ==> P2
         && ...
         && (e instanceof En) ==> Pn;
```

Dabei ist Folgendes zu bemerken: Ist ein Typ E einer zur Laufzeit geworfenen Exception ein Subtyp *mehrerer* angegebener Typen E1, E2,..., En, dann folgt aus obiger Darstellung, dass auch alle entsprechenden Prädikate P1, P2,...,Pn erfüllt sein müssen.

`signals_only`-Ausdrücke

`signals_only`-Ausdrücke können als Abkürzung für `signals`-Ausdrücke verstanden werden, die spezifizieren, welche Exceptions zur Laufzeit einer Methodenausführung geworfen werden dürfen. Damit legen `signals_only`-Ausdrücke

implizit fest, welche Exceptions *nicht* geworfen werden dürfen. Die Syntax ist wie folgt:

signals_only_Ausdruck ::= *signals_only_Schlüsselwort Exception_Typ_Liste* `;`
| *signals_only_Schlüsselwort* `\nothing ;`
signals_only_Schlüsselwort ::= `signals_only`
| `signals_only_redundantly`
Exception_Typ_Liste ::= *Exception_Typ* (`,` *Exception_Typ*)*

Analog zu den `signals`-Ausdrücken dürfen in der Typliste nur Exceptions angegeben werden, die ein Subtyp von `java.lang.Exception` sind. Zudem müssen auch hier alle aufgeführten Typen Sub- oder Supertyp einer Exception sein, die im `throws`-Ausdruck der entsprechenden Methode genannt wird.

Ein `signals_only`-Ausdruck der Form

```
signals_only E1, E2,..., En;
```

ist äquivalent zu folgendem `signals`-Ausdruck:

```
signals (java.lang.Exception e)
          (e instanceof E1)
        || (e instanceof E2)
        || ...
        || (e instanceof En);
```

Wird in einer Spezifikation kein explizitier `signals_only`-Ausdruck definiert, dann werden sowohl Light- als auch Heavyweightpezifikationen um einen impliziten `signals_only`-Ausdruck erweitert, der alle im `throws`-Ausdruck spezifizierten Exceptions, die ein Subtyp von java.lang.Exception sind, umfasst. Dies wird an folgendem Beispiel anschaulich:

Beispiel 2.10 (`signals_only`)

```
//@ requires P;
//@ ensures Q;
public void getArgument(int index)
     throws NullPointerException, VirtualMachineError {
     // liefert das Argument einer Datenstruktur
     // am angegebenen Index zurück
}
```

Dies beinhaltet implizit den `signals_only`-Ausdruck

```
signals_only NullPointerException;
```

Die zweite aufgeführte Exception ist kein Subtyp von `java.lang.Exception` und wird deshalb nicht in die Typliste des Ausdrucks übernommen. ■

Die Verwendung mehrerer `signals_only`-Ausdrücke in einer Spezifikation kann im Gegensatz zu den bisher besprochenen Ausdrücken unter Umständen missverstanden werden. Die semantische Besonderheit mehrerer `signals_only`-Ausdrücke zeigt sich im Folgenden:

```
signals_only E1, E2;
signals_only E3, E4;                (1)
```

ist nicht etwa äquivalent zu

```
signals_only E1, E2, E3, E4;        (2)
```

wie nach den bisher vorgestellten Spezifikationsausdrücken zu vermuten gewesen wäre, sondern zu

```
signals (Exception e)
            (e instanceof E1)
         || (e instanceof E2);
signals (Exception e)
            (e instanceof E3)
         || (e instanceof E4);      (3)
```

Dies wiederum lässt sich umformen zu

```
signals (Exception e)
            (   (e instanceof E1)
             || (e instanceof E2)
            )
         &&
            (   (e instanceof E3)
             || (e instanceof E4)
            );                          (4)
```

Die Semantik von (1) ist also folgende: Tritt zur Laufzeit der Methodenausführung eine Exception auf, so muss sie vom Typ E1 oder E2 *und* vom Typ E3 oder E4 sein, nicht jedoch nur von einem der angegebenen Typen, wie dies in (2) spezifiziert ist.

Desugaring von Spezifikationen

Es wurde bereits mehrfach angemerkt, dass sich Normal- und Exceptional-Behavior-Spezifikationen sowie Lightweightspezifikationen als syntaktische Abkürzungen für Behavior-Spezifikationen mit der gleichen Semantik auffassen lassen. In der englischsprachigen Literatur hat sich dafür der Begriff „syntactic sugar“ geprägt, wovon sich der Ausdruck des „Desugaring“ als Auflösung

dieser syntaktischen Abkürzungen ableitet. Durch das Desugaring können JML-Spezifikationen in einheitliche Behaviorspezifikationen derselben Semantik umformuliert werden. Dieser Vorgang ist insbesondere für Runtime-Checking-Werkzeuge wichtig, da sich hierdurch standardisierte Übersetzungsmuster für die verschiedenen JML-Spezifikationen angegeben lassen.

Im nachfolgenden Abschnitt finden nur die Desugaring-Regeln der für diese Arbeit wichtigen JML-Ausdrücke Beachtung; das Desugaring der Rahmenbedingungen und Code-Contract-Spezifikationen finden sich detailliert in [CCC+06] und [RL00] beschrieben.

Regel 1: Desugaring von Lightweightspezifikationen

Eine Lightweightspezifikation der Form

```
requires P;
ensures Q;
signals (Exception) S;
```

wird durch das Desugaring umgewandelt in eine Behavior-Spezifikation der Form

```
behavior
  requires P;
  ensures Q;
  signals (Exception) S;
```

Nicht explizit definierte JML-Ausdrücke werden dabei mit den jeweiligen impliziten Standardwerten eingeführt. ■

Ähnlich verhält sich das Desugaring für Normal-Behavior-Spezifikationen.

Regel 2: Desugaring von Normal-Behavior-Spezifikationen

Eine Normal-Behavior-Spezifikation der Form

```
normal_behavior
  requires P;
  ensures Q;
```

wird durch das Desugaring umgewandelt in eine Behavior-Spezifikation der Form

```
behavior
  requires P;
  ensures Q;
  signals (Exception) false;
```

Hierbei wird zunächst ein `signals`-Ausdruck mit dem Prädikat `false` eingeführt, der die normale Terminierung der Methodenausführung modelliert und gleichzeitig eine Terminierung durch Werfen einer Exception ausschließt. Der zusätzliche `signals_only`-Ausdruck wird für Normal-Behavior-Spezifikationen in der allgemeinsten, die Spezifikation nicht einschränkenden Form eingeführt. ■

Für die Exceptional-Behavior-Spezifikationen ergibt sich Folgendes:

Regel 3: Desugaring von Exceptional-Behavior-Spezifikationen

Eine Exceptional-Behavior-Spezifikation der Form

```
exceptional_behavior
  requires P;
  signals (Exception) S;
```

wird durch das Desugaring umgewandelt in eine Behavior-Spezifikation der Form

```
behavior
  requires P;
  ensures false;
  signals (Exception) S;
```

Analog zu den Normal-Behavior-Spezifikationen wird hier ein zusätzlicher `ensures`-Ausdruck mit dem Prädikat `false` eingeführt, der die normale Terminierung ausschließt. ■

In einem ähnlichen Prozess lassen sich eingebettete Spezifikationen zu einer durch `also` getrennten Folge von Behavior-Spezifikationen umwandeln.

Regel 4: Desugaring von eingebetteten Spezifikationen

Aus einer eingebetteten Spezifikation der Form

```
behavior
  requires P;
  {|  ensures Q1;
      signals (E1) S1;
    also
      ...
    also
      ensures Qn;
      signals (En) Sn;
  |}
```

wird durch das Desugaring eine Folge von Behavior-Spezifikationen der Form

```
  behavior
    requires P;
    ensures Q1;
    signals (E1) S1;
also
  ...
also
  behavior
    requires P;
    ensures Qn;
    signals (En) Sn;
```

■

Durch `also` getrennte Folgen von Spezifikationsfällen spielen insbesondere bei der Vererbung und der Verfeinerung eine wichtige Rolle[25]. Die Semantik solcher Spezifikationen ergibt sich durch die Verknüpfung der Semantiken der einzelnen Teilspezifikationen. Gegeben sei also eine Spezifikation der Form

```
  behavior
    requires P1;
    ensures Q1;
also
  ...
also
  behavior
    requires Pn;
    ensures Qn;
```

Die Ermittlung der Semantik dieser Spezifikation geschieht nach der folgenden Regel:

Regel 5: Semantik nach dem Desugaring von Spezifikationen

1. Die Vorbedingungen werden durch disjunktives Verknüpfen abgeschwächt: $P_{gesamt} := \bigvee_{i=1}^{n} P_i$. Die Menge der gültigen Zustände wird also insgesamt vergrößert.

2. Die Nachbedingungen werden durch konjunktives Verknüpfen verschärft: $Q_{gesamt} := \bigwedge_{i=1}^{n} \backslash old(P_i) \Rightarrow Q_i$. Die Menge der gültigen Zustände wird dadurch eingeschränkt. Dabei spielt es keine Rolle, ob Aussagen über normale Terminierung oder Terminierung durch Werfen einer Exception gemacht werden, d.h. `ensures`- und `signals`-Ausdrücke werden analog behandelt.

■

[25] Siehe dazu Abschnitt 2.1.8.

2.1.6 JML-spezifische Modifikatoren

Neben den üblichen Java-Modifikatoren für Sichtbarkeit und Zugreifbarkeit wurden in JML mehrere Modifikatoren speziell für Spezifikationsausdrücke eingeführt. Im Folgenden werden die für das weitere Verständnis wichtigsten kurz vorgestellt; ausführlich diskutiert werden JML-Modifikatoren und deren Verwendung insbesondere in [CCC+06].

`spec_public` und `spec_protected`

Um JML-Konstrukten für Spezifikationszwecke eine differenziertere Sichtbarkeit zu verleihen, als dies mit den Java-Modifikatoren möglich wäre, können die beiden Modifikatoren `spec_public` und `spec_protected` verwendet werden. Beide erlauben es, die Sichtbarkeit zum Beispiel einer Methodenspezifikation gegenüber der entsprechenden Methode zu erweitern. Es ist dabei darauf zu achten, dass die Sichtbarkeit nur vergrößert werden kann, d.h.

- Spezifikationen für `private` oder `package-visible` deklarierte Methoden dürfen sowohl mittels `spec_protected` als auch mittels `spec_public` modifiziert werden,
- Spezifikationen für `protected` deklarierte Methoden dürfen durch `spec_public` modifiziert werden, und
- Spezifikationen für `public` deklarierte Methoden können nicht weiter modifiziert werden.

`pure`

Eine Modifikation einer Klasse oder Methode durch das Schlüsselwort `pure` bedeutet, dass die Ausführung dieses Konstrukts keine Nebeneffekte mit sich bringt. `pure` kann nur auf Klassen, Konstruktoren und Methoden angewendet werden; dabei bedeutet die Modifikation einer Klasse, dass implizit alle Konstruktoren und Methoden `pure` deklariert sind. Soll in Spezifikationen auf Methoden zurückgegriffen werden, um Aussagen über das Verhalten einer Komponenten zu machen, so muss die entsprechend referenzierte Methode als `pure` deklariert worden sein, um die Nebeneffektfreiheit von Spezifikationsprüfungen garantieren zu können.

`model`

Neben den bereits vorgestellten Modellvariablen lassen sich auch Klassen, Konstruktoren und Methoden mittels `model` modifizieren. Die so erhaltenen Ausdrücke

können dazu verwendet werden, Modellprogramme zu spezifizieren, die im Rahmen dieser Arbeit jedoch nicht besprochen werden.

`ghost`

Felder können mittels des Modifikators `ghost` ähnlich wie Modellvariablen als spezifikationsintern deklariert werden und dürfen nur innerhalb von Spezifikationen verwendet werden. Im Gegensatz zu Modellvariablen werden sie nicht durch eine Abstraktionsfunktion beschrieben, sondern besitzen genau wie gewöhnliche Felder einen Initialwert, der durch spezielle `set`-Ausdrücke modifiziert werden kann. Für diese Arbeit spielen `ghost`-Felder nur eine untergeordnete Rolle, weshalb an dieser Stelle auf [CCC+06, Abschnitt 2.2 und 12.4] verwiesen sei.

`helper`

Mittels `helper` deklarierte Methoden sind von der Pflicht, Invarianten und History Constraints einzuhalten, ausgenommen. Sie müssen allerdings noch immer eventuell geerbte oder lokal definierte Vor- und Nachbedingungen beachten.

2.1.7 Redundanz

Redundanz wird in JML dazu verwendet, auf wichtige Konsequenzen von Spezifikationen hinzuweisen. Neben den bereits vorgestellten Schlüsselworten mit dem Suffix `redundantly` lassen sich auf zwei weitere Arten redundante Aussagen treffen:

- durch *redundante Implikationen*,
- durch *redundante Beispiele*.

Redundante Beispiele können ganze Methodenspezifikationen umfassen, die ein Verhalten einer Methode zur Laufzeit beschreiben, das sich aus der eigentlichen, nicht-redundanten Spezifikation ableiten lässt. Solche Beispiele können sehr umfangreich und detailliert sein; aus der Sicht von Runtime-Checking-Werkzeugen haben sie jedoch keine Bedeutung. Aus diesem Grund werden redundante Beispiele in dieser Arbeit nicht weiter besprochen; als weiterführende Literatur sei hier auf [CCC+06, Abschnitt 13] verwiesen.

Redundante Implikation haben die folgende Syntax:

Redundante_Implikation ::= `implies_that` *Spezifikationsfallsequenz*

Eingeleitet durch das Schlüsselwort `implies_that` dienen redundante Implikationen ebenfalls dazu, auf Konsequenzen einer Methodenspezifikation hinzuweisen. Gegeben sei folgendes Beispiel:

Beispiel 2.11 (Redundante Implikation)

```
/*@ requires x >= 0.0;
  @ ensures -eps <= x - (\result * \result) <= eps;
  @
  @ implies_that
  @ ensures (x == 0) ==> (eps >= (\result * \result) >= -eps);
 */
public float squareRoot(float x) {
  // berechne Näherung der positiven Quadratwurzel von x
}
```

Die hier angegebene redundante Zusicherung folgt für `eps` > 0, wovon hier ausgegangen werden kann, direkt aus der Nachbedingung der Methodenspezifikation. ■

2.1.8 Vererbung und Verfeinerung

Analog zur *Vererbung* von Methoden und Feldern werden in JML spezifikationsrelevante Komponenten wie Modellvariablen und natürlich Methodenspezifikationen vererbt. Dabei hängt die Vererbung jedoch nicht in gleichem Maße von den Zugriffsmodifikatoren ab, wie dies in Java der Fall ist. *Verfeinerung* ist ein der Vererbung ähnliches Konzept, das insbesondere bei der Erstellung von hierarchisch organisierten Spezifikationen und der Verwendung separater, vom Java-Quellcode unabhängiger Spezifikationsdateien eine wichtige Rolle spielt.
Im Folgenden werden zunächst die Vererbungseigenschaften der bereits vorgestellten JML-Ausdrücke und der Methodenspezifikationen dargelegt und erläutert. Daran schließt sich ein Abschnitt über die Grundlagen der Verfeinerung von Spezifikationen an. Als weiterführende Literatur sei auf [CCC+06, Abschnitt 8 und 16] verwiesen.

Vererbung von Invarianten und Constraints

Jede Methode, die nicht durch das Schlüsselwort `helper` modifiziert wurde, muss sämtliche anwendbaren Invarianten und History Constraints der eigenen Klasse, der Superklasse und aller implementierten Interfaces einhalten. Daher werden sowohl statische als auch instanzgebundene Invarianten und Constraints unabhängig von ihrem jeweiligen Zugriffsmodifikator[26] vererbt.

[26] Zugriffsmodifikatoren von Invarianten und Constraints dienen, wie bereits angesprochen, lediglich der Regelung, auf welche Felder und Methoden diese zugreifen dürfen.

Vererbung von Modellvariablen und Initiallies

Die Vererbung von Modellvariablen folgt den gleichen Regeln wie die Vererbung von Feldern in Java; sie ist also vom definierten Zugriffsmodifikator der Modellvariablen abhängig. Initiallies müssen hingegen nicht vererbt werden, da nach [BGJS05, Abschnitt 12.4] eine Instanziierung einer Klasse stets die Instanziierung ihrer Superklasse nach sich zieht und die in `initially`-Ausdrücken definierten Bedingungen dadurch implizit geprüft werden.

Vererbung von Methodenspezifikationen

Methodenspezifikationen können durch Verwendung von Zugriffsmodifikatoren hierarchisch organisiert werden. Ein Anliegen einer solchen Organisation ist es, Spezifikationen für verschiedene „Interessenten“ zu schreiben, zum Beispiel

- eine `private` deklarierte Spezifikation für die internen Vorgaben, die für eine Kommunikation mit externen Komponenten keine Rolle spielen, und
- eine `public` deklarierte Spezifikation für den Design-by-Contract-Vertrag mit einem möglichen Client.

Die Vererbung von Methodenspezifikationen hängt damit wie in Java von ihrer Sichtbarkeit ab:

- `private` deklarierte Spezifikationen werden nicht vererbt und sind nicht extern zugreifbar,
- `protected` deklarierte Spezifikationen werden vererbt, sind jedoch nur von Instanzen der deklarierenden Klasse oder einer Subklasse heraus sichtbar,
- `package-visible` deklarierte Spezifikationen werden vererbt, sind jedoch nur von Instanzen der deklarierenden Klasse oder einer Klasse aus dem selben Package heraus sichtbar,
- `public` deklarierte Spezifikationen werden vererbt und sind generell sichtbar.

Verfeinerung von Spezifikationen

Eine Verfeinerung (engl.: refinement) einer Spezifikation hat zum Ziel, Verhalten und Struktur etwa einer Implementierung noch detaillierter zu beschreiben, als dies in der entsprechenden Schnittstellenbeschreibung der Fall war. Die Verfeinerung ähnelt dabei sehr der Vererbung von Spezifikationen, da eine Klasse auch für

überschriebene Methoden[27] einen eigenen Beitrag zur Methodenspezifikation leisten kann, der die geerbte Spezifikation im obigen Sinne verfeinert.
Im Gegensatz zur Vererbung kann die Verfeinerung auch dazu verwendet werden, Java-Quellcode und JML-Spezifikationen in separaten Dokumenten zu verwalten. Auf diese Art kann zunächst eine quellcode-unabhängige Spezifikation eines Systems erstellt werden, das dann in einem weiteren Schritt zu einem späteren Zeitpunkt implementiert werden kann, ohne dabei die Spezifikationsdateien zu beeinträchtigen.

Angezeigt wird eine Verfeinerung durch die Referenzierung einer externen Spezifikationsdatei direkt nach der Packagedeklaration einer Java-Compilation-Unit:

CompilationUnit ::= (*Formaler_Kommentar*)*
[*Package_Deklaration*]
[*Refinement_Deklaration*]
(*Import_Deklaration*)*
(*Typ_Deklaration*)*
Refinement_Deklaration ::= (`refines` | `refine`) *String_Literal* **;**

Zur Veranschaulichung sei folgendes Beispiel gegeben:

Beispiel 2.12 (Verfeinerung)

```
package my.examples;
//@ refines my.specs.MySpecification;

public class MyClass {
  // Klassendeklaration
}
```

Die in der referenzierten, externen Datei gegebenen Spezifikationen werden in der verfeinernden Klasse wie lokal definierte Beschreibungen behandelt. Im Unterschied zur Vererbung werden dabei natürlich auch `private` deklarierte Konstrukte berücksichtigt.

■

Die externen Spezifikationsdateien werden ebenfalls in der Form von um JML-Spezifikationen angereicherten Java-Quellcode geschrieben. Im Unterschied zu gewöhnlichem Quellcode dürfen in diesen Dateien jedoch keine Methoden- oder Konstruktorenkörper definiert werden.

Ein wichtiger Begriff im Zusammenhang mit Verfeinerung ist die sogenannte

[27] Überschriebene (engl.: *overridden*) Methoden sind natürlich ebenfalls von einer Superklasse oder einem implementierten Interface geerbt.

Verfeinerungskette (engl.: *refinement-chain*). Verfeinerungsketten entstehen durch mehrere hintereinander geschaltete Spezifikationen, die selbst andere Spezifikationen verfeinern. Deutlich wird dies an einem Beispiel.

Beispiel 2.13 (Verfeinerungskette)

Gegeben sei folgende Klasse:

```
package my.examples;
//@ refines my.specs.VectorSpec;

public class VectorImpl {
  // Implementierung eines Vector-Objekts
}
```

`my.examples.VectorImpl` verfeinert, angezeigt durch einen `refines`-Ausdruck, folgende Spezifikationsklasse:

```
package my.specs;
//@ refines my.specs.ListSpec;

public class VectorSpec {
  // Spezifikation eines Vector-Objekts
}
```

`my.specs.VectorSpec` verfeinert wiederum `my.specs.ListSpec`:

```
package my.specs;

public class ListSpec {
  // Spezifikation eines List-Objekts
}
```

`my.specs.ListSpec` stellt das Ende der Verfeinerungskette dar, da sie selbst keine weitere Spezifikation verfeinert und somit die *am wenigsten verfeinerte* Spezifikation darstellt. Am Anfang der Kette steht die *am meisten verfeinerte* Klasse `my.examples.VectorImpl`. Die Spezifikation eines solchen `VectorImpl`-Objekts setzt sich also aus der Verknüpfung[28] aller Spezifikationen in der Verfeinerungskette zusammen.

■

Die Verwendung solcher Verfeinerungsketten ist insbesondere dann sinnvoll,

- wenn die Spezifikationen und der eigentliche Quellcode getrennt gehalten werden sollen,

[28] Die Art der Verknüpfung ergibt sich dabei aus den Vererbungsregeln der jeweiligen Konstrukte.

- wenn Subklassen von Klassen mit externen Spezifikationen implementiert werden, die selbst wiederum Spezifikationsanteile beitragen,
- wenn Spezifikationen unterschiedlicher Sichtbarkeit in separaten Dateien definiert werden sollen.

2.2 Jass - Java with Assertions

2.2.1 Einführung

Jass wurde von Detlef Bartetzko während seiner Diplomarbeit [Bar99] im Jahre 1999 mit dem Ziel entwickelt, die Idee des Design by Constract in Java durch Laufzeitprüfungen von Zusicherungen zu unterstützen. Dies sollte durch Erweiterung des Java-Sourcecodes um Verträge erreicht werden, die in der momentanen Implementierung von Jass in der gleichnamigen Spezifaktionssprache modelliert werden können. Die grundlegende Idee beruht dabei unter anderem auf den Arbeiten von Clemens Fischer und Dieter Meemken, der in [Mee97] bereits ein Runtime-Checking-Werkzeug für Java entwickelte.
Verträge werden in Jass in der Form von erweiterten Java-Kommentaren definiert und nach der Verarbeitung durch den Jass-Precompiler in Java-Sourcecode umgewandelt, der die Einhaltung der Verträge zur Laufzeit überprüft. Die grundlegenden Ansätze bei JML und bei Jass überschneiden sich in vielen Punkten; allerdings ist die Sprache Jass im Gegensatz zu JML direkt für die spätere Laufzeitprüfung ausgelegt. Weitere konzeptionelle Unterschiede zeigen sich in Umfang und Syntax der jeweiligen Spezifikationssprachen. Nachdem JML bereits im vorherigen Kapitel ausführlich besprochen wurde, soll nun die Spezifikationssprache Jass kurz vergleichend vorgestellt werden.

2.2.2 Klasseninvariante

Die *Klasseninvariante* dient analog zu ihrem JML-Äquivalent der Spezifikation von Eigenschaften, die in allen sichtbaren Zuständen[29] der Programmausführung erfüllt sein müssen. Die gewünschten Eigenschaften werden als Prädikat definiert, das in Jass ein beliebiger Boolescher Ausdruck ist, der neben den normalen Java-Konstrukten auch quantifizierte Ausdrücke enthalten kann. Quantifizierte Ausdrücke werden in Abschnitt 2.2.5 genauer besprochen. Folgendes Beispiel zeigt die Ähnlichkeit zu den JML-Invarianten:

Beispiel 2.14 (Invariante in Jass)

```
/** invariant i > 0; **/
```

■

Invarianten müssen ebenfalls von allen Subtypen einer Klasse erfüllt werden, um deren Spezifikation zu verfeinern[30]. Damit sind die Invarianten in Jass denen in

[29] In [Bar99] ist nicht von *sichtbaren* Zuständen die Rede, sondern von *stabilen* Zuständen. Da diese jedoch im Wesentlichen die selben sind, wird im Weiteren aus Konsistenzgründen von sichtbaren Zuständen die Rede sein.

[30] Vererbung und Verfeinerung werden in Abschnitt 2.2.6 besprochen.

JML - abgesehen von der Unterscheidung zwischen statisch und instanzgebunden - sowohl syntaktisch als auch semantisch gegenüber gleichwertig.

2.2.3 Zusicherungen für Methoden

Jass bietet die Möglichkeit, Vor- und Nachbedingungen für Methodenausführungen zu definieren. Vorbedingungen werden durch das Schlüsselwort `require` eingeleitet und stehen syntaktisch vor der ersten Anweisung des Methodenrumpfes:

Beispiel 2.15 (Vorbedingung in Jass)

```
public float squareRoot(float x) {
  /** require x >= 0.0 **/
  // berechne Näherung der positiven Quadratwurzel von x
}
```

■

Nachbedingungen werden durch das Schlüsselwort `ensure` eingeleitet und stehen syntaktisch nach der letzten Anweisung des Methodenrumpfes:

Beispiel 2.16 (Nachbedingung in Jass)

```
public float squareRoot(float x) {
  /** require x >= 0.0; **/
  // berechne Näherung der positiven Quadratwurzel von x
  /** ensure -eps <= x - (Result * Result) <= eps; **/
}
```

■

Die in der Nachbedingung eingeführte Variable `Result` ist eine Jass-spezifische Variable, mittels der sich der Rückgabewert einer Methode referenzieren lässt. Außerdem kann in Nachbedingungen auch die spezielle Variable `Old` verwendet werden, die im Unterschied zu den `\old`-Ausdrücken in JML eine Kopie der gesamten Klasseninstanz vor der Ausführung der ersten Anweisung des Methodenrumpfes referenziert.

Damit lassen sich wie in JML-Lightweightspezifikationen beliebige Vorbedingungen und Nachbedingungen für die normale Terminierung der Methodenausführung definieren. Nicht möglich ist es dagegen, das Verhalten bei Terminierung durch Werfen einer Exception zu spezifizieren oder mittels Zugriffsmodifikatoren hierarchisch gestaffelte Spezifikationsanteile zu definieren.

Zu bemerken ist an dieser Stelle, dass es ein großer Nachteil der syntaktischen Positionierung der Vor- und Nachbedingungen in Jass ist, dass in Interfaces

keine Spezifikationen definiert werden können, da dort nur abstrakte Methoden ohne Methodenrumpf erlaubt sind. Dadurch entstehen insbesondere bei der Verfeinerung von Spezifikationen Probleme, die sich auch in der Unmöglichkeit, externe Spezifikationen zu schreiben, begründen. In Abschnitt 2.2.6 wird auf diese Schwierigkeiten noch eingegangen werden.

2.2.4 Zusicherungen für Schleifen

Analog zu den entsprechenden JML-Konstrukten bietet auch Jass die Möglichkeit, Invarianten und Varianten für Schleifen zu definieren. Schleifeninvarianten spezifizieren Eigenschaften, die durch die Ausführung des Schleifenrumpfes nicht verändert werden[31]. In Jass werden sie wie folgt definiert:

Beispiel 2.16 (Schleifeninvariante in Jass)

```
while(B) {
  /** invariant I **/
  // Ausführung des Schleifenrumpfes
}
```

■

Entsprechend können Schleifenvarianten spezifiziert werden, die Aussagen über die Terminierung einer Schleifenausführung machen:

Beispiel 2.17 (Schleifenvariante in Jass)

```
while(B) {
  /** variant D **/
  // Ausführung des Schleifenrumpfes
}
```

■

Dabei ist der Typ des Variantenausdrucks `D` in Jass auf den primitiven Typ `int` beschränkt.

2.2.5 Quantifizierte Ausdrücke

Quantoren können wie in JML dazu verwendet werden, quantifizierte Aussagen in Form von `forall`- und `exists`-Ausdrücken zu spezifizieren. Dabei sollte auch hier im Regelfall ein einschränkender Wertebereich angegegeben werden, um potenzielle Fehlerquellen zu minimieren.

[31] Siehe dazu auch den entsprechenden Unterabschnitt über Schleifenspezifikationen in Abschnitt 2.1.3.

Das folgende Beispiel dient der Erläuterung der Verwendung quantifizierter Ausdrücke:

Beispiel 2.18 (Universalquantor in Jass)

```
forall i : { 0 .. list.size() } # list.get(i) != null
```

Der hier definierte Ausdruck hat dieselbe Semantik wie der im entsprechenden Beispiel in Abschnitt 2.1.3 definierte JML-`\forall`-Ausdruck. ■

Analog zu Quantifizierungen durch `forall` lassen sich auch durch `exists` quantifizierte Ausdrücke erstellen:

Beispiel 2.19 (Existenzquantor in Jass)

```
exists i : { 0 .. list.size() } # list.get(i) != null
```

Die Semantik dieses Ausdrucks besagt, dass es mindestens einen Index `i` im Wertebereich `0 .. list.size()` geben muss, so dass der entsprechende Eintrag im Objekt `list` ungleich `null` ist. ■

2.2.6 Vererbung und Verfeinerung

Das Konzept der Vererbung und der Verfeinerung in der Spezifikationssprache Jass unterscheidet sich deutlich von dem in JML, da in Jass ein Klasse nicht automatisch die Spezifikationen ihres Supertyps erbt. Soll eine Klasse die Spezifikationen ihres Supertyps *verfeinern*, so muss für die überschriebenen Methoden explizit eine schwächere Vorbedingung bzw. eine stärkere Nachbedingung definiert werden. Durch die Implementierung des speziellen Interfaces `jass.runtime.Refinement` kann dann nach [Bar99, Abschnitt 5] eine Verfeinerungsrelation zwischen zwei Klassen angezeigt werden. Dieses Interface besitzt selbst weder abstrakte Methoden noch statische Felder, sondern dient lediglich dazu, auf die Verfeinerung der Superklasse hinzuweisen:

Beispiel 2.20 (Verfeinerung in Jass)

```
package my.examples;

public class MyClass
            implements jass.runtime.Refinement
            extends    my.examples.MySuperClass {
    // Klassendeklaration und Spezifikationen
}
```

Im Beispiel wird durch den zusätzlichen `implements`-Ausdruck angezeigt, dass die Klasse `MyClass` ihre Superklasse `MySuperClass` verfeinert. *Jass* generiert dann Quellcode, der die Einhaltung der Verfeinerungsrelation zur Laufzeit prüft. ■

Details zu Vererbung und Verfeinerung in Jass sind in [Bar99, Abschnitt 5] ausführlich dargestellt.

2.2.7 Vergleich zu JML

Abschließend soll noch eine vergleichende Übersicht über die verschiedenen Konstrukte in JML und in Jass gegeben werden. Tabelle 2.1 stellt anschaulich die in dieser Arbeit besprochenen Sprachkonstrukte mitsamt der relevanten Schlüsselworte in JML bzw. Jass dar. Dabei ist zu bemerken, dass JML über eine Vielzahl weiterer Sprachkonstrukte verfügt, die in dieser Arbeit aber aus Gründen des Umfangs keine Beachtung finden. Eine detaillierte Beschreibung ihrer Syntax und Semantik findet sich insbesondere in [CCC+06], auf das an dieser Stelle der Vollständigkeit halber verwiesen sei. Details zur Spezifikationssprache Jass finden sich der Diplomarbeit von Detlef Bartetzko [Bar99].

Übersicht

Sprachkonstrukt	**JML**	**Jass**
Invariante	`invariant`	`invariant`
History Constraint	`constraint`	-
Modellvariablen	`model`, `represents`	-
Spezifikationsvariablen	`forall`, `old`	-
Initiallies	`initially`	-
Vorbedingung	`requires`	`require`
normale Terminierung	`ensures`	`ensure`
Terminierung durch Exception	`signals`, `signals_only`	-
Zugriffsmodifikatoren für Methodenspezifikationen	`private`, `protected` `public`	-
Spezielle Modifikatoren	`spec_public`, `spec_protected`, `pure`, `model`, `helper`	-
Referenzierung von Rückgabewerten	`\result`	`Result`
Referenzierung von vorangangenen Zuständen	`\old`	`Old`
Quantifizierung	`\forall`, `\exists`	`forall`, `exists`
Spezielle Operatoren	`<:`, `<==>`, `<=!=>`, `==>`, `<==`	-
Schleifenspezifikationen	`loop_invariant`, `decreases`	`invariant`, `variant`
Vererbung und Verfeinerung von Spezifikationen	*Verfeinerungsketten, umfassende Vererbung*	*Verfeinerung zwischen Klassen*

Tabelle 2.1: Vergleichende Übersicht der wichtigsten Sprachkonstrukte von JML und Jass

Kapitel 3

Entwurf

Nachdem im vorherigen Kapitel die grundlegenden Konzepte der Spezifikationssprache JML ausführlich erläutert und mit der bisherigen Eingabesprache Jass des gleichnamigen Precompilers verglichen worden sind, widmet sich dieses Kapitel zunächst einer kurzen Einführung in zwei wichtige Technologien: der Programmiersprache Java und dem Java Compiler Compiler. Daran anschließen wird sich die Entwicklung von *JMLjass*, einer signifikanten Teilsprache von JML, die als neue Eingabesprache für den Jass-Precompiler dienen soll. Ein weiterer Abschnitt widmet sich dem Aufbau und den Analysephasen des neuen Precompilers. Abschließend folgt eine ausführliche Beschreibung der Übersetzungsmuster der Sprachkonstrukte von JMLjass, die die Grundlage für die Umwandlung von Spezifikationen in Java-Quellcode, der diese zur Laufzeit prüft, darstellen.

3.1 Verwendete Technologien

In diesem Abschnitt werden zwei grundlegende Technologien vorgestellt, die für die Implementierung des in dieser Arbeit konzipierten Precompilers von großer Bedeutung sind: die Programmiersprache Java in der aktuellen Version *Java 2 Platform Standard Edition 5.0* (kurz: *Java 1.5*), und der *Java Compiler Compiler* (kurz *JavaCC*).

3.1.1 Die Programmiersprache Java

Java ist eine relativ neue objekt-orientierte Programmiersprache, die in der ursprünglichen Version 1995 von der Firma Sun Microsystem, Inc. unter der Leitung von James Gosling entwickelt wurde. Durch die einfache Erlernbarkeit, die Plattformunabhängigkeit und die insbesondere daraus resultierenden, umfangreichen Anwendungsmöglichkeiten im Internet erfuhr Java eine große Popularität, die zu einer weiten Verbreitung und steten Weiterentwicklung dieser Sprache geführt hat.

Insbesondere in der momentan aktuellen Version Java 1.5 hat es umfassende Neuerungen gegenüber der Vorgängerversion gegeben. Neben Performanzsteigerungen durch Optimierungen in diversen Klassen und der Erweiterung vieler Bibliotheken sind die wichtigsten sprachrelevanten Änderungen nachfolgend aufgeführt:

- Einführung generischer Definitionen für Typen
- Verbesserte `for`-Schleife
- Autoboxing / Unboxing primitiver Typen
- Typsichere Aufzählungstypen

Generische Typdefinitionen

Generische Typdefinitionen gehören zu den interessantesten und von Entwicklern unter anderem der Java Communitiy am meisten eingeforderten sprachlichen Neuerungen in Java 1.5. Sie erlauben es, Datenstrukturen wie etwa Vektoren oder Listen, die prinzipiell für beliebige Typen nutzbar sind, schon bei der Implementierung mit geringem Aufwand typsicher zu gestalten. Dazu sind generische Typen mit einem oder mehreren *generischen Parametern* versehen, die durch konkrete Parameter ersetzt werden können. Formal lässt sich dies wie folgt darstellen:

Generischer_Typ ::= *Typ* < *Formale_Parameter* >
Formale_Parameter ::= *Parameter* (, *Parameter*)*

Formale Parameter müssen in der Implementierung durch einen konkreten Typ ersetzt werden, wie das folgende Beispiel zeigt:

Beispiel 3.1 (Generische Allokation)

```
Vector<String> vectorOfString = new Vector<String>();
```

■

Erlaubt sind dabei neben den üblichen Java-Typen auch generische Typen selbst erlaubt, d.h. es können Schachtelungen der folgenden Form enstehen:

Beispiel 3.2 (Schachtelung generischer Typen)

```
Vector<Vector<String>> vectorOfGenericVector
                     = new Vector<Vector<String>>();
```

■

Generische Typen können *generische Methoden* definieren, die ebenfalls von den konkreten Werten der generischen Typparameter abhängig sind. Oftmals betrifft diese Abhängigkeit den Rückgabewert einer generischen Methode:

Beispiel 3.3 (Generische Methoden)

```
public <T> T get(int index) {
  // Methodenrumpf
}
```

Der hier in spitzen Klammern gegebene formale Parameter `<T>` zeigt dem Compiler die Definition einer generischen Methode an, deren Rückgabewert vom konkreten Typ, der bei der Instanziierung der entsprechenden generischen Klasse für `T` eingesetzt wird, abhängig ist.

■

Die im ersten Beispiel erzeugte Instanz `vectorOfString` der Klasse `java.util.Vector` erhält den zusätzlichen Typparameter `String`, der den generischen Parameter zur Übersetzungszeit bindet. Ein solcher Vektor darf dadurch nur Objekte vom Typ `String` enthalten und verwalten; dafür entfällt das Typecasting, das in vorherigen Versionen von Java etwa beim Durchlaufen eines Vektors notwendig war:

Beispiel 3.4 (Vereinfachtes Typecasting)

Neben

```
/** Diese Methode liefert den String
    am angegebenen Index zurück. */
public String getString(int index) {

  return (String)myVector.get(i);
}
```

ist nun auch die einfachere Version

```
/** Diese Methode liefert den String
    am angegebenen Index zurück. */
public String getString(int index) {

  return myVector.get(i);
}
```

typkorrekt definiert.

■

Insbesondere bei umfangreichen Implementierungen macht sich diese Ersparnis deutlich bemerkbar, zumal durch die Konkretisierung des erlaubten Typs auch

mögliche Typisierungsfehler bereits zur Übersetzungszeit entdeckt werden.

Der Rückgabewert der generischen Methode `get` im Beispiel 3.3 ist ebenfalls vom konkreten Typ abhängig. So liefert ein Aufruf dieser Methode für den Typ `Vector<String>` ein `String`-Objekt zurück; für einen Vektor vom Typ `Vector<Integer>` liefert `get` dagegen ein `Integer`-Objekt zurück. Wird in einer Implementierung kein konkreter Typ angegeben, so wird standardmäßig der Typ `java.lang.Object` angenommen.

Zu bemerken ist, dass der Begriff des *generischen Typs* in diesem Zusammenhang leicht missverständlich ist, da solche Typen nur aus der Sicht des Entwicklers generisch sind. Zur Laufzeit sind diese Typen durch die Bindung der generischen Parameter auf einen bestimmten Typ festgelegt und agieren nicht, wie sich vermuten ließe, als beliebig austauschbare Typen.

Erweiterte `for`-Schleife

In Java 1.5 wurde die `for`-Schleife syntaktisch erweitert, um die Iteration über Collections und Arrays zu vereinfachen und die Fehleranfälligkeit der Iterationsschritte insbesondere bei geschachtelten Schleifen zu verringern. Die sogenannte `for-each`-Schleife lässt sich am besten an einem Beispiel[1] erläutern:

Beispiel 3.5 (`for-each`-Schleife, 1)

```
/** Diese Methode bricht alle TimerTasks
    in der übergebenen Collection ab. */
public void cancelAll(Collection<TimerTask> c) {

  for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
    i.next().cancel();
}
```

Der Iterator `i` wird insgesamt dreimal referenziert; abzüglich der Deklaration bleiben damit noch zwei potenziell fehleranfällige Referenzen. Durch die Verwendung der *for-each*-Schleife vereinfacht sich diese Methode wesentlich:

```
/** Diese Methode bricht alle TimerTasks
    in der übergebenen Collection ab. */
public void cancelAll(Collection<TimerTask> c) {

  for (TimerTask t : c)
    t.cancel();
}
```

[1] Die Beispiele zur erweiterten `for`-Schleife stammen aus dem Java Language Guide, http://java.sun.com/j2se/1.5.0/docs/guide/language.

Der Ausdruck `for (Timertask t : c)` wird dabei gelesen als „for each TimerTask in c“; daher auch der Name der erweiterten `for`-Schleife. Dabei ist zu bemerken, dass die `for-each`-Schleife insbesondere auf die Verwendung in Kombination mit generischen Typen ausgelegt ist.

■

Auch die Iteration über Arrays lässt sich durch die `for-each`-Schleife eleganter und einfacher gestalten, wie folgendes Beispiel belegt:

Beispiel 3.6 (`for-each`-Schleife, 2)

```
// Returns the sum of the elements of a
int sum(int[] a) {
  int result = 0;

  for (int i : a)
      result += i;

  return result;
}
```

■

Autoboxing und -unboxing primitiver Typen

In den bisherigen Versionen von Java war die Verwendung primitiver Typen in Kombination mit dynamischen Datenstrukturen wie etwa Listen oder Vektoren relativ aufwändig, da diese nur Objektreferenzen verwalten können. Um also zum Beispiel eine dynamische Liste von `int`-Werten zu implementieren, musste bisher der Umweg über sogenannte Wrapperklassen, die nach [BGJS05, Abschnitt 5.1.7] als nicht-primitive Repräsentation von primitiven Datentypen verstanden werden können, gegangen werden:

Beispiel 3.7 (Wrapping)

```
// eine dynamische Liste
ArrayList list = new ArrayList();

// Beispielwert
int value = 5;

intList.add(new Integer(value));
```

■

Das „Einpacken“ eines primitiven Datentypen in die entsprechenden Wrapperklasse wird in [BGJS05, Abschnitt 5.1] als *Boxing* bezeichnet; der umgekehrte Vorgang wird *Unboxing* genannt. Das in Java 1.5 eingeführte *Autoboxing* befreit den

Entwickler von der Pflicht, das Boxing bzw. Unboxing selbst zu implementieren. Statt des obigen Beispiels ist nun Folgendes möglich:

Beispiel 3.8 (Autoboxing)

```
// eine dynamische Liste
ArrayList list = new ArrayList();

// Beispielwert
int value = 5;

list.add(value);
```

■

Das Autounboxing, d.h. das automatische Rückgewinnen eines primitiven Typs aus der nicht-primitiven Wrapperklasse funktioniert analog zum Boxing und wird hier nicht weiter ausgeführt.

Zusammenfassend lässt sich feststellen, das auch das Autoboxing, wie schon die generischen Typen und die `for-each`-Schleife, einerseits den Quellcode vereinfacht und elegantere Implementierungen ermöglicht; andererseits hilft diese Funktionalität ebenfalls, überflüssige Typfehler zu vermeiden.

Obwohl primitive Datentypen und deren entsprechende Wrapperklassen in Java 1.5 nahezu austauschbar verwendbar sind, sollte nicht vergessen werden, dass sie eigentlich unterschiedlichen Zwecken dienen. Insbesondere bei performanzsensitiven Anwendungen sollte darauf geachtet werden, möglichst wenig Gebrauch von Wrapperklassen und der Autoboxingfunktionalität zu machen.

Typsichere Aufzählungstypen

Eine weitere interessante Neuerung in Java 1.5 ist die Möglichkeit, *typsichere Aufzählungstypen* verwenden zu können. Bisherige Aufzählungen hatten meist die folgende Gestalt:

Beispiel 3.9 (Aufzählung)

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
//...
public static final int SUNDAY = 6;
```

■

Eine solche Repräsentation von Wochentagen durch `int`-Werte hat mehrere Probleme:

- Die Aufzählung bietet keine Typsicherheit, da an jeder Stelle, an der ein Wochentag erwartet wird, ein beliebiger `int`-Wert übergeben werden kann.
- Die numerische Darstellung von Wochentagen ist für textuelle Ausgaben wenig informativ.
- Um Überschneidungen mit anderen Enumerationen zu vermeiden, müsste ein gemeinsamer Präfix wie etwa `TAG_` verwendet werden, da `int`-Konstanten über keinen eigenen Namensraum verfügen, der Kollisionen verhindern könnte.
- Aufzählungen wie oben sind nur mit einigem Aufwand nachträglich zu ändern.

Typsichere Aufzählungstypen bieten einen einfachen Mechanismus, diese Probleme zu vermeiden, wie folgendes Beispiel aus [Sun04] zeigt:

Beispiel 3.10 (Typsichere Aufzählung)

```
public enum Day {MONDAY, TUESDAY, ... , SUNDAY}
```

■

Der mit dem Schlüsselwort `enum` eingeführte Auzählungstyp ist nach [Sun04] leicht erweiterbar, typsicher und einer Java-Klasse gleichwertig. Kollisionen mit anderen Aufzählungen sind aufgrund dieser Gleichwertigkeit ebenfalls ausgeschlossen.

Java 1.5 liefert zu den Aufzählungstypen einen einfachen Mechanismus, effizient mit diesen zu arbeiten:

Beispiel 3.11 (EnumSet)

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
  System.out.println(d);
```

■

Die hier benutzte Klasse `EnumSet` kann dazu verwendet werden, über einen bestimmten Teilbereich einer Aufzählung zu iterieren. Die Referenzierung einzelner Aufzählungselemente erfolgt dabei in einer Form, der der Referenzierung statischer Konstanten gleicht. Weitere Fähigkeiten dieser Klasse und der typsicheren Aufzählungstypen finden sich detailliert in [BGJS05, Abschnitt 8.9] und werden an dieser Stelle nicht besprochen.

3.1.2 Der Java Compiler Compiler

Der Java Compiler Compiler (kurz: JavaCC) ist ein weit verbreiteter Parsergenerator, der als OpenSource-Projekt unter der Leitung von Sreenivasa Viswanadha [Vis05] entwickelt wurde. Er kann dazu verwendet werden, aus Grammatikspezifikationen einen Java-basierten Parser zu generieren, der Dokumente auf die Einhaltung der gegebenen Grammatik überprüfen kann. In Kombination mit einem Tool namens *JJTree*, das im Lieferumfang des JavaCC enthalten ist, lassen sich auch Parser generieren, die zur Übersetzungszeit einen Ableitungsbaum zu einem gegebenen Dokument erstellen können.
Beide Werkzeuge wurden in dieser Diplomarbeit dazu verwendet, aus der Grammatik einer Teilsprache von JML, die im nächsten Abschnitt vorgestellt wird, einen Parser zu erzeugen, der um JML-Annotationen erweiterten Java-Quellcode analysieren und einen entsprechenden Ableitungsbaum erstellen kann. Im Folgenden wird zunächst der Funktionsumfang des JavaCC und des Werkzeugs JJTree vorgestellt. Anschließend werden die für diese Arbeit notwendigen Grundlagen der Funktionsweise der durch diese Tools erstellten Parser erläutert.

Funktionsumfang des JavaCC

Der JavaCC ermöglicht es Entwicklern, lexikalische Festlegungen wie zum Beispiel reguläre Ausdrücke und Schlüsselworte sowie grammatikalische Spezifikationen in einem einzelnen Dokument definieren zu können. Dies hat einerseits den Vorteil, dass Grammatiken leichter lesbar sind, da die Terminale in regulären Ausdrücken innerhalb von Grammatikspezifikationen verwendet werden können, und erleichtert andererseits aus dem selben Grund die Wartung und Erweiterung einer Grammatik.
Eine weitere wichtige Funktionalität ist die Möglichkeit, erweiterte Backus-Naur-Formen für die Definition von Grammatiken verwenden zu können, was insbesondere die Lesbarkeit von Grammatiken wesentlich erleichtert und sich fehlervermeidend auswirkt.

Standardmäßig erzeugt JavaCC aus einer gegebenen Grammatik einen LL(1)-Parser, d.h. einen Parser, der

- eine Folge von Eingabesymbolen von **l**inks nach rechts abarbeitet,
- eine **L**inksableitung erstellt,
- einen Lookahead von **1** hat[2].

[2]Ein Parser mit einem Lookahead von n betrachtet stets n Eingabesymbole im Voraus. Dabei ist zu bemerken, dass die Effizienz eines Parsers in großem Maße nach [Vis05] und [ASU99] von seinem Lookahead abhängig ist.

Nach [Vis05] kann an es manchen Entscheidungspunkten während der Ableitung einer Eingabe jedoch erforderlich sein, mehr als nur ein einzelnes Token im Voraus zu betrachten. Daher bietet der JavaCC die Möglichkeit, den Lookahead eines Parser lokal zu verändern. Dies kann auf zwei Arten geschehen:

1. Durch syntaktischen Lookahead,
2. durch semantischen Lookahead.

Syntaktischer Lookahead wird durch die Angabe eines syntaktischen Sprachkonstrukts definiert, das an einem Entscheidungspunkt abzuleiten versucht werden soll. Gelingt die Ableitung, so entscheidet sich der Parser für die so bevorzugte Alternative. Ein Misslingen der im syntaktischen Lookahead spezifizierten Ableitung führt zu ihrer Verwerfung und der des entsprechenden Pfades im Syntaxbaum.
Im Gegensatz dazu erfolgt die Definition eines semantischen Lookaheads durch die Angabe eines beliebigen Prädikats, das über die Auswahl der folgenden Teilableitung entscheidet. Meist beziehen sich solche Prädikate auf eine bevorzugte Folge von Eingabesymbolen, die über die weiteren Ableitungsschritte entscheidet.

Durch die Verwendung lokaler Lookaheads lässt sich ein Parser erzeugen, der sich global gesehen wie ein LL(1)-Parser verhält und nur an unumgänglichen Entscheidungspunkten mit einem größeren Lookahead arbeiten muss. Dies wirkt sich insgesamt positiv auf die Effizienz des gesamten Ableitungsvorgangs aus.

Anschauliche Beispiele zu syntaktischem bzw. sematischem Lookahead finden sich [Vis05], worauf an dieser Stelle als verwiesen sei.

Funktionsumfang von JJTree

JJTree ist ein Präprozessor, der in eine JavaCC-Grammatik bestimmte Aktionen einfügt, die dazu dienen, während eines Parsedurchgangs einen durch Softwarewerkzeuge manipulier- und traversierbaren Ableitungsbaum zu erstellen. Dabei erzeugt JJTree standardmäßig Quellcode zur Erstellung von Ableitungsknoten für jedes Terminalsymbol. Es ist allerdings möglich, die Erzeugung solcher Knoten an bestimmte Bedingungen zu knüpfen. Dabei ist folgende Unterscheidung zu berücksichtigen:

- *Endliche Knoten* werden immer erzeugt, und zwar mit einer fest vorgegebenen Anzahl von Kinderknoten.
- *Bedingte Knoten* werden nur dann erzeugt, wenn ein vorgegebenes Prädikat erfüllt ist.

Die Syntax dieser Bedingungen wird insbesondere in [Vis05] beschrieben und soll an dieser Stelle nicht weiter besprochen werden.

Arbeitsweise der generierten Parser

Die LL(k)-Parser[3], die durch die Verwendung von JJTree und JavaCC generiert werden, arbeiten nach dem *top-down*-Prinzip, d.h. die Ableitungsschritte beginnen an der Wurzel des späteren Ableitungsbaums und werden durch Linksableitungen fortgesetzt, bis die Blätter erreicht werden. Um diese Arbeitsweise an einem Beispiel zu illustrieren, sei folgende Grammatik gegeben:

JML_Code ::= *Invariante* | *Initially*
Invariante ::= (`invariant` | `invariant_redundantly`) *Prädikat* **;**
Initially ::= `initially` *Prädikat* **;**
Prädikat ::= *Name* (> | <) *Wert*

Der Einfachheit halber wird hier ein sehr eingeschränktes Prädikat verwendet, das aus einer einfachen Relation zwischen einer numerischen Variablen, gegeben durch den *Namen*, und einer oberen oder unteren Schranke, gegeben durch den *Wert*, bestehen kann. Als Namen seien alle gültigen Java-Namen zugelassen, als Schranke alle numerischen Werte, die für den Typ der referenzierten Variablen zugelassen sind.

Der Parser soll eine Ableitung für folgende Eingabe finden:

```
invariant x > 0;
```

Der Ablauf der Generierung der Ableitung und des zugehörigen Baumes ist nachfolgend in einzelnen Schritten dargestellt. Dabei sind in jedem Schritt der gerade betrachtete Knoten und das gelesene Eingabesymbol durch einen Pfeil markiert.

[3] JavaCC generiert standardmäßig LL(1)-Parser, da diese am effizientesten arbeiten. Es ist jedoch auch möglich, einen beliebig großen globalen Lookahead zu verwenden, obwohl dies zu erheblichen Performanzeinbußen führen kann.

Beispielhafte Ableitung

(1)	Ableitungsbaum	→ *JML_Code*
	Eingabe	`invariant x > 0 ;` ↑
(2)	Ableitungsbaum	*JML_Code* | → *Invariante*
	Eingabe	`invariant x > 0 ;` ↑
(3)	Ableitungsbaum	*JML_Code* | *Invariante* | → `invariant` *Prädikat* `;`
	Eingabe	`invariant x > 0 ;` ↑
(4)	Ableitungsbaum	*JML_Code* | *Invariante* | `invariant` → *Prädikat* `;`
	Eingabe	`invariant x > 0 ;` ↑
(5)	Ableitungsbaum	*JML_Code* | *Invariante* | `invariant` *Prädikat* `;` | → *Name* (> | <) *Wert*
	Eingabe	`invariant x > 0 ;` ↑

Tabelle 3.1: Ableitungsschritte 1 bis 5

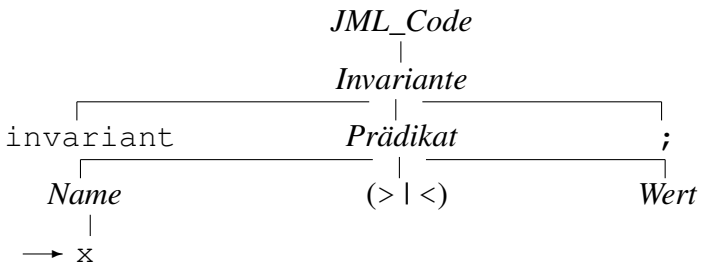
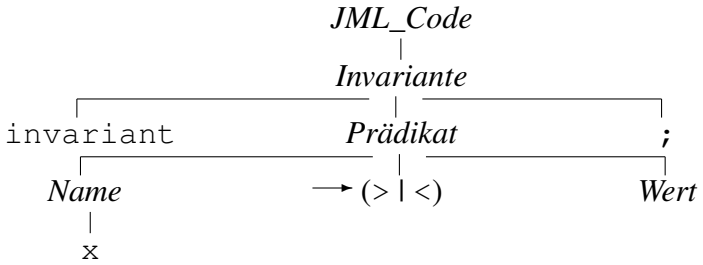
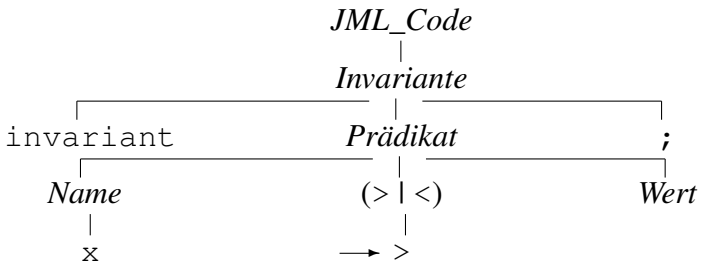
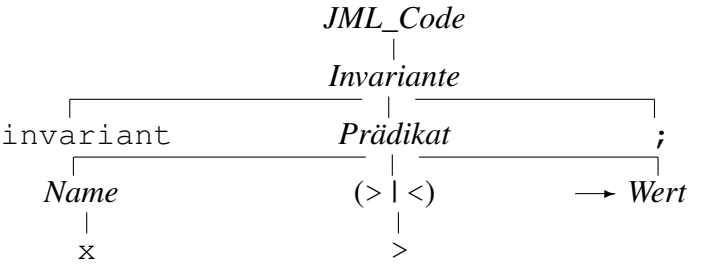
(6)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`
(7)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`
(8)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`
(9)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`

Tabelle 3.2: Ableitungsschritte 6 bis 9

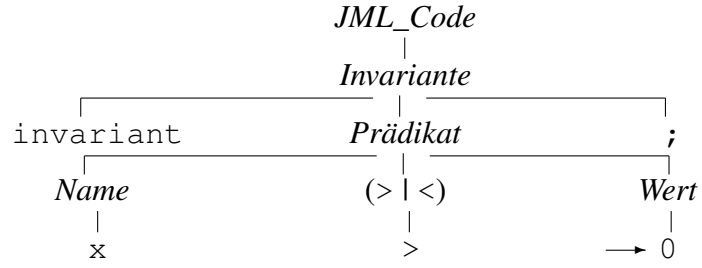
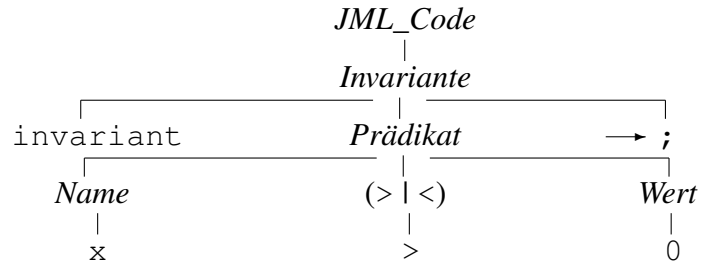
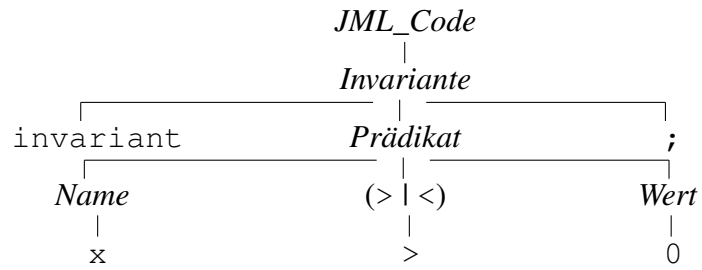
(10)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`
(11)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`
(12)	Ableitungsbaum	
	Eingabe	`invariant x > 0 ;`

Tabelle 3.3: Ableitungsschritte 10 bis 12

Während einer Ableitung wird ein Eingabesymbol *konsumiert*, wenn im Ableitungsbaum ein Terminal erreicht wird, auf das das Eingabesymbol passt. Dies bedeutet, dass der Zeiger, der im Eingabestrom das aktuelle Symbol referenziert, um eine Stelle nach rechts gerückt wird. Im Beispiel geschieht dies in den Schritten (3), (6), (8), (10) und (11). Die Ableitung ist vollständig, wenn das letzte Eingabesymbol konsumiert wurde.

Die durch JJTree eingefügten Aktionen, die den manipulierbaren Ableitungsbaum erzeugen, bauen diesen im Gegensatz zur *top-down*-Strategie des Parsers *bottom-up* auf. Dazu wird zur Laufzeit ein Stack verwendet, auf dem die bereits

besuchten (aber noch nicht vollständig abgeleiteten) Knoten zwischengespeichert werden. Gelangt der Ableitungsprozess bei einem Blatt an, so wird der oberste Knoten vom Stack geholt, dem Blatt als Elternknoten zugeteilt, und wieder auf den Stack gelegt, falls er noch nicht vollständig abgeleitet wurde. Ist auch der oberste Knoten des Stacks vollständig abgeleitet, so wiederholt sich dieser Prozess mit dem nächsten Knoten des Stacks so lange, bis ein noch unvollständiger Knoten oder die Wurzel erreicht wird.
Im gegebenen Beispiel wird etwa in Schritt (4) der Knoten *Prädikat* auf den Stack gelegt, bis er in Schritt (11) vollständig abgeleitet ist und dem Elternknoten *Invariante* zugeordnet wird.

3.2 Die Teilsprache JMLjass

Eines der Hauptziele dieser Diplomarbeit ist es, eine signifikante Teilsprache von JML zu entwickeln und als Eingabesprache für das Runtime-Checking-Werkzeug Jass zu implementieren. Diese Sprache soll aus naheliegenden Gründen *JMLjass* genannt und in diesem Abschnitt vorgestellt werden.
Zunächst werden die einzelnen Sprachkonstrukte von JMLjass einschließlich ihres syntaktischen Aufbaus[4] und ihrer Semantik erläutert; daran anschließen wird sich ein kurzer Abschnitt über diejenigen Teile von JML, die nicht in JMLjass übernommen wurden. Aus Gründen der Übersichtlichtkeit und des Umfangs dieser Arbeit findet sich die komplette Grammatik von JMLjass im Anhang A; in den folgenden Unterabschnitten wird daher nur auf die für das Verständis wichtigsten Konstrukte eingegangen.

3.2.1 Sprachumfang von JMLjass

JMLjass wurde als eine *signifikante Teilsprache*[5] von JML entwickelt, die trotz der Einschränkungen, denen sie unterworfen ist, einem Entwickler die Möglichkeit gibt, sinnvolle und umfassende Spezifikationen zu schreiben. Dazu muss sie die grundlegenden Funktionalitäten, die JML auszeichnen, bereitstellen können und insbesondere die Idee des Design by Contract aufrechterhalten. Als grundlegend werden dabei erachtet:

- Sprachkonstrukte, die für Spezifikationen im Sinne der partiellen bzw. totalen Korrektheit nach [AO94] von Programmen benötigt werden, wie zum Beispiel Vor- und Nachbedingungen und Schleifenspezifikationen.
- Modellvariablen, die ein hohes Abstraktionsniveau und eine Entkopplung von Spezifikation und Implementierung ermöglichen.
- Die Möglichkeit, Spezifikationen hierarchisch zu gestalten und in vom Quellcode getrennten Dokumenten zu verwalten.
- Die Vererbung von Spezifikationen.
- Die Verfeinerung von Spezifikationen.

Sowohl JML als auch JMLjass sind echte Erweiterungen der Programmiersprache Java, wodurch die Spezifikationsausdrücke grammatikalisch in die Syntax von Java

[4] Die Syntax einiger Konstrukte ist aufgrund der Einbettung von JMLjass in die Programmiersprache Java außerordentlich umfangreich und in Teilen eher unübersichtlich als hilfreich, weshalb hier nur eine eingeschränkte Syntax angegeben wird.

[5] Der Begriff der Signifikanz bedeutet in diesem Kontext, dass der Umfang einer Teilsprache groß genug sein muss, um die grundlegenden Ideen einer Spezifikationssprache erhalten zu können.

eingebettet sein müssen. Grundlage der JMLjass-Grammatik war in diesem Fall die Java 1.5-Grammatik für den JavaCC, die von Sreenivasa Vishwanada [Vis05] bereitgestellt wurde.

Ausgehend von diesen Überlegungen werden im Folgenden die für das weitere Verständnis wichtigen grammatikalischen Regeln von JMLjass vorgestellt; die vollständige Grammatik ist, wie bereits erwähnt, im Anhang A aufgeführt, die zugehörigen Terminalsymbole im Anhang B.

Compilation Units

Compilation Units bilden den äußeren Rahmen eines jeden Java-Programms. Sie werden in einem einzelnen Dokument definiert und können mehrere Klassen umfassen, wobei die äußerste bzw. zuerst auftretende Klasse *Top-Level Klasse* (engl.: top-level class) genannt wird. Die Syntax der Compilation Unit ist wie folgt:

CompilationUnit ::= (<FORMAL_COMMENT>)*
(PackageDeclaration)?
(RefinePrefix)?
(ImportDeclaration)*
(TypeDeclaration)*
<EOF>

Die Wurzel eines Ableitungsbaums eines JMLjass-Dokuments ist stets eine Compilation Unit. Sie umfasst neben den in Java üblichen Konstrukten wie Package- und Typdeklarationen auch den Refinementpräfix, der eine Verfeinerung einer externen Spezifikation anzeigt.

Refinement

Der *Refinementpräfix* kann dazu verwendet werden, externe Spezifikationen zu referenzieren und damit anzuzeigen, dass die Top-Level Klasse einer Compilation Unit diese verfeinert. Die Syntax ist die folgende:

RefinePrefix ::= (`refine` | `refines`) <STRING_LITERAL> `;`

Da die Semantik von Verfeinerungen von Spezifikationen bereits in Abschnitt 2.1.8 diskutiert wurden, soll hier nicht noch einmal auf sie eingegangen werden.

Importdeklarationen

Importdeklarationen ermöglichen es, in einer Java-Klasse Methoden und Felder externer Klassen zu verwenden. Ihre Syntax ist die folgende:

ImportDeclaration ::= [`model`] `import` [`static`] Name() [`. *`] `;`

Zu bemerken ist dabei, dass durch das Schlüsselwort `model` ein Modellimport definiert wird, der ausschließlich zu Spezifikationszwecken verwendet werden kann. Die Referenzierung einer durch einen Modellimport eingeführten Klasse außerhalb JML-Spezifikationen ist nicht erlaubt und führt während der Übersetzung zu Referenzierungsfehlern.

Typdeklarationen

Die bekanntesten und am häufigsten verwendeten *Typdeklarationen* sind die Deklarationen einer Klasse oder eines Interfaces. In Java 1.5 neu hinzugekommen sind Annotationen und typsichere Aufzählungstypen:

TypeDeclaration ::= `;`
| ClassOrInterfaceDeclaration
| EnumDeclaration
| AnnotationTypeDeclaration

Aufzählungstypen wurden bereits in Abschnitt 3.1.1 diskutiert; Details zu Annotationen finden sich insbesondere in [BGJS05].

Klassen- und Interfacedeklarationen

Klassen- bzw. *Interfacedeklarationen* bilden die Grundlage der objektorientierten Modellierung und erlauben in JMLjass neben den üblichen Java-Konstrukten die Definition einiger spezieller Spezifikationskonstrukte. Sie setzen wie folgt zusammen:

ClassOrInterfaceDeclaration ::= (<FORMAL_COMMENT>)*
Modifiers
(`class` | `interface`)
<IDENTIFIER>
(TypeParameters)?
(ExtendsList)?
(ImplementsList)?
ClassOrInterfaceBody

Typparameter

Um in Java 1.5 generische Typen zu definieren, können Klassen- und Interfaces mit einer Liste formaler *Typparameter* in spitzen Klammern versehen werden:

TypeParameters ::= < TypeParameter (, TypeParameter)* >
TypeParameter ::= <IDENTIFIER> (TypeBound)?
TypeBound ::= `extends` ClassOrInterfaceType (`&` ClassOrInterfaceType)*

Ein einzelner Typparameter besteht aus einem Identifier und einer *Typbegrenzung*, die sich aus einer Liste von Klassen oder Interfaces zusammensetzt und die Menge der zulässigen Parameter einschränkt. Wird zur Laufzeit eine Instanz einer solchen Klasse erzeugt, so muss ein konkreter Typ übergeben werden, der den formalen Parameter ersetzt. Dabei muss der übergebene Typ ein Subtyp aller in der Typbegrenzung angegebenen Typen sein.

JMLjass-spezifische Modifikatoren

In JMLjass wurden folgende *Modifikatoren* von JML übernommen:

```
JMLModifier ::= spec_public
              | spec_protected
              | model
              | ghost
              | pure
              | instance
              | helper
              | non_null
```

Neben den bereits in Abschnitt 2.1.6 besprochen Modifikatoren sind hier noch folgende zu erwähnen:

- Das Schlüsselwort `instance` kann benutzt werden, um Felder auch in Interfaces als instanzgebunden zu definieren.
- Eine Modifikation eines Feldes mittels `non_null` legt fest, dass ein solches Feld in keinem sichtbaren Zustand einer Nullreferenz entspricht.

Klassen- und Interfacekörper

Klassen und Interfaces können eine Vielzahl von Deklarationen umfassen, die den sogenannten *Körper* ausmachen. Die Syntax des Körpers ist wie folgt:

```
ClassOrInterfaceBody ::= { ( ClassOrInterfaceBodyDeclaration )* }
ClassOrInterfaceBodyDeclaration ::= Initializer
                                  | JMLDeclaration
                                  | MemberDeclaration
```

```
                                    | ;
MemberDeclaration ::= ClassOrInterfaceDeclaration
                    | EnumDeclaration
                    | ConstructorDeclaration
                    | FieldDeclaration
                    | MethodDeclaration
```

JML-spezifische Deklarationen

Neben inneren Klassen, Methoden und Feldern kann ein Klassenkörper auch spezielle *JML-Deklarationen* beinhalten:

```
JMLDeclaration ::= Modifiers
                   ( Invariant
                   | HistoryConstraint
                   | RepresentsDeclaration
                   | InitiallyClause )
```

Invarianten und Constraints definieren Eigenschaften, die in (fast) allen sichtbaren Zuständen einer Programmausführung gelten müssen. Representsdeklarationen definieren Abstraktionsfunktionen für Modellvariablen; Initiallies legen Eigenschaften für Modellvariablen fest, die unmittelbar nach der Initialisierung einer Klasse in der Java Virtual Machine gelten müssen. Die Semantik dieser Ausdrücke wurde bereits in den Abschnitten 2.1.2 und 2.1.4 besprochen.

Methodendeklarationen

Methoden- und *Konstruktorendeklarationen* spielen für eine BISL[6] wie JML eine zentrale Rolle, da sie die Schnittstelle einer Klasse oder eines Interfaces darstellen. Die Syntax der Methodendeklarationen ist wie folgt:

```
MethodDeclaration ::= ( <FORMAL_COMMENT> )*
                      Modifiers
                      ( TypeParameters )?
                      ( method )?
                      ResultType
                      MethodDeclarator
                      ( throws NameList )?
                      ( MethodSpecification )?
                      ( Block | ; )
                      |
```

[6] Abkürzung für engl.: „behavioral interface specification language“

```
                    ( <FORMAL_COMMENT> )*
                    MethodSpecification
                    Modifiers
                    ( TypeParameters )?
                    ( method )?
                    ResultType
                    MethodDeclarator
                    ( throws NameList )?
                    ( Block | ; )
MethodDeclarator ::= <IDENTIFIER>
                     FormalParameters
                     ( [ ] )*
```

Konstruktoren werden auf ähnliche Weise deklariert wie Methoden, allerdings dürfen sie im Unterschied zu diesen nicht abstrakt definiert werden. Externe Spezifikationen sind von dieser Regelung natürlich ausgenommen.
Analog zu generischen Klassen können auch für Methoden formale Typparameter definiert werden, die zur Laufzeit durch den bei der Instanziierung der umgebenden Klasse übergebenen konkreten Typ ersetzt werden[7].

Methodenspezifikationen

Ein Kernstück von JMLjass sind die *Methodenspezifikationen*, die in erweiterten Kommentaren vor dem Methodenkörper deklariert werden können:

```
MethodSpecification ::= Specification | ExtendingSpecification
ExtendingSpecification ::= also Specification
Specification ::= SpecificationCaseSequence ( RedundantSpecification )?
                | RedundantSpecification
SpecificationCaseSequence ::= SpecificationCase ( also SpecificationCase )*
SpecificationCase ::= LightweightSpecificationCase
                    | HeavyweightSpecificationCase
                    | CodeContractSpecification
```

Methodenspezifikationen setzen sich im Wesentlichen aus einer durch `also` getrennten Folge von Spezifikationsfällen zusammen, die wiederum aus Lightweight- oder Heavyweightspezifikationen bestehen können. Redundante Spezifikationen sind in JMLjass syntaktisch erlaubt, werden aber auf semantischer Ebene nicht verarbeitet. Dies stellt jedoch keine Einschränkung der wesentlichen Funktionalität von JML dar, da redundante Spezifikationen zwar auf Konsequenzen nicht-redundanter Spezifikationen hinweisen, selbst aber keinen wesentlichen Beitrag zur Definition

[7] Siehe dazu Abschnitt 3.1.1.

des Verhaltens einer Methode leisten[8].
Auch die Code-Contract Spezifikationen werden syntaktisch erlaubt, um bereits erstellte Spezifikationen verarbeiten zu können. Die dort definierbaren Ausdrücke spezifizieren Rahmenbedingungen, die bei der Ausführung einer Methode eingehalten werden müssen, wie zum Beispiel welche Methoden aufgerufen (`callable`-Ausdrücke) oder auf welche Felder zugegriffen werden darf (`accessible`-Ausdrücke)[9]. Eine semantische Verarbeitung findet aber auch hier nicht statt.

Lightweightspezifikationen

Mittels *Lightweightspezifikationen* lassen sich schnell einfache Vor- und Nachbedingungen für Methoden definieren:

```
LightweightSpecificationCase ::= GenericSpecificationCase
GenericSpecificationCase ::= ( SpecificationVariableDeclarations )?
                             SpecificationHeader
                             ( GenericSpecificationBody )?
                             |
                             ( SpecificationVariableDeclarations )?
                             GenericSpecificationBody
GenericSpecificationBody ::= SimpleSpecificationBody
                              |
                              { | GenericSpecificationCaseSequence | }
GenericSpecificationCaseSequence ::= GenericSpecificationCase
                                       ( also GenericSpecificationCase )*
SpecificationHeader ::= RequiresClause ( RequiresClause )*
SimpleSpecificationBody ::= SimpleSpecificationBodyClause
                            ( SimpleSpecificationBodyClause )*
SimpleSpecificationBodyClause ::= DivergesClause
                                  | AssignableClause
                                  | CapturesClause
                                  | EnsuresClause
                                  | SignalsOnlyClause
                                  | SignalsClause
```

Die Semantik gleicht derjenigen der Lightweightspezifikation in JML, mit einer Ausnahme: Alle Konstrukte, die zur Laufzeit eine Daten- oder Kontrollflussanalyse benötigen, werden zwar syntaktisch akzeptiert, semantisch aber nicht verarbeitet. Betroffen sind hierbei die `assignable`- und `captures`-Ausdrücke, die zur Spezifikation von Rahmenbedingungen genutzt werden könnten. Diese Einschränkung begründet sich durch den Aufwand, der zur Laufzeit betrieben werden muss, um

[8] Siehe dazu auch Abschnitt 2.1.7.
[9] Code-Contract Spezifikationen und alle relevanten Ausdrücke werden in [CCC+06] besprochen.

vollständige Daten- oder Kontrollflussanylsen zu garantieren, da er den Umfang dieser Arbeit überschreiten würde[10].

Heavyweightspezifikationen

Die *Heavyweightspezifikationen* erweitern die Lightweightspezifikationen mit einem Zugriffsmodifikator, der die Sichtbarkeit der Spezifikation gegenüber der Methode erweitern oder einschränken kann:

HeavyweightSpecificationCase ::= BehaviourSpecificationCase
| ExceptionalBehaviourSpecificationCase
| NormalBehaviourSpecificationCase
BehaviourSpecificationCase ::= (Privacy)? `behavior` GenericSpecificationCase
Privacy ::= `public` | `protected` | `private`
NormalBehaviourSpecificationCase ::= (Privacy)?
`normal_behavior`
GenericSpecificationCase
ExceptionalBehaviourSpecificationCase ::= (Privacy)?
`exceptional_behavior`
GenericSpecificationCase

Auch für Heavyweightspezifikationen gilt natürlich die Einschränkung, dass Ausdrücke, die eine Daten- oder Kontrollflussanalyse benötigen, semantisch nicht verarbeitet werden.

Vor- und Nachbedingungen

Vor- und *Nachbedingungen* spielen eine wichtige Rolle bei der Frage nach der Korrektheit eines Programms. Aus diesem Grund sind in JMLjass die üblichen Vor- und Nachbedingungen (sowohl für die normale Terminierung als auch für die Terminierung durch Werfen einer Exception) erlaubt:

RequiresClause ::= `requires` PredicateOrNot `;`
PredicateOrNot ::= Predicate | `\not_specified`
EnsuresClause ::= `ensures` PredicateOrNot `;`
SignalsClause ::= `signals (` ReferenceType (<IDENTIFIER>)? `)` (PredicateOrNot)? `;`
SignalsOnlyClause ::= `signals_only` ReferenceType (`,` ReferenceType)* `;`
| `signals_only` `\nothing` `;`

[10] Siehe dazu auch Abschnitt 4.

`requires`-Ausdrücke werden im Spezifikationsheader definiert und müssen als Vorbedingung im Pre-State der Methodenausführung erfüllt sein. `ensures`-Ausdrücke spezifizieren Nachbedingungen, die im Falle der normalen Terminierung der Programmausführung gelten müssen; `signals`- und `signals_only`-Ausdrücke müssen dagegen bei Terminierung durch Werfen einer Exception erfüllt sein. Ausgenommen sind `diverges`-Ausdrücke, die Nachbedingungen definieren, welche bei Divergenz[11] einer Methodenausführung erfüllt sein müssen, da hierbei wiederum eine Kontrollflussanalyse notwendig wäre[12].

Modellvariablen

Modellvariablen spielen in JML-Spezifikationen eine wichtige Rolle, da sie ein hohes Abstraktionsniveau und die Loslösung von Spezifikation und Implementierung ermöglichen.Die Modifikation eines Feldes mit dem Schlüsselwort `model` definiert es als Modellvariable, und mittels eines `represents`-Ausdrucks wird eine Abstraktionsfunktion spezifiziert, die die Lücke zwischen abstrakten Variablen und konkreten, von der Implementierung abhängigen Werten schließt:

RepresentsDeclaration ::= `represents` StoreReferenceExpression
AbstrFuncOrAssign SpecificationExpression **;**
|
`represents` StoreReferenceExpression
`\such_that` Predicate **;**
AbstrFuncOrAssign ::= <− | =

Schleifenspezifikationen

Auch in JMLjass ist möglich, *Schleifeninvarianten* und *-varianten* innerhalb von Methodenkörpern zu verwenden:

PossiblyAnnotatedLoop ::= (LoopInvariant)* (VariantFunction)* LoopStatement
LoopStatement ::= WhileStatement
| DoStatement
| ForStatement
LoopInvariant ::= (`maintaining` | `loop_invariant`) Predicate **;**
VariantFunction ::= (`decreasing` | `decreases`) SpecificationExpression **;**

[11] Abweichend vom üblichen Sprachgebrauch müssen `diverges`-Ausdrücke auch in Deadlock-Situationen erfüllt sein. Siehe dazu auch [CCC+06, Abschnitt 9.9].

[12] Ohne eine umfassende Kontrollflussanalyse ist es nicht möglich, etwa eine Divergenz durch Endlosschleifen nachzuweisen.

Die Invariante definiert ein beliebiges Prädikat, das zu jedem relevanten Zeitpunkt[13] der Methodenausführung erfüllt sein muss; die Variante hingegen dient dem Nachweis, dass die Ausführung der Schleife nach einer endlichen Anzahl von Iterationen terminiert. Die Kombination von Schleifeninvarianten und Variantenfunktionen in einem Methodenkörper erlaubt die Überprüfung folgender Korrektheitsaussage: Die Invariante ist vor der ersten Ausführung des Schleifenrumpfes erfüllt und bleibt bei jeder Iteration erhalten und die Ausführung der Schleife terminiert nach einer endlichen Anzahl von Iterationen in einem Zustand, der ebenfalls die Invariante erfüllt.

`\result` und `\old`

Um in Spezifikationen Referenzierungen des Rückgabewertes einer Methode zu zulassen, erlaubt JMLjass die Verwendung des `\result`-Konstrukts:

ResultExpression ::= `\result`

Dabei sei daran erinnert, dass dieser Ausdruck aus offentsichtlichen Gründen nur in Prädikaten normaler Nachbedingungen verwendet werden darf.

`\old`-Ausdrücke erlauben es, beliebige Statements im Pre-State einer Methodenausführung auszuwerten und zu späteren Zeitpunkten darauf zurückzugreifen:

OldExpression ::= `\old (` SpecificationExpression (`,` <IDENTIFIER>)? `)`
| `\pre (` SpecificationExpression `)`

Die Verwendung von `\old`-Ausdrücken ist auf Nachbedingungen beliebiger Art und History Constraints beschränkt.

Quantoren

JMLjass ermöglicht die Quantifizierung von Ausdrücken durch den universellen `\forall`- und den existenziellen `\exists`-Quantor:

SpecificationQuantifiedExpression ::= `(` Quantifier
QuantifiedVariableDeclarations `;`
(Predicate `;`)?
SpecificationExpression `)`
Quantifier ::= `\forall` | `\exists`
QuantifiedVariableDeclarations ::= TypeSpec QuantifiedVariableDeclarator
(`,` QuantifiedVariableDeclarator)*
QuantifiedVariableDeclarator ::= <IDENTIFIER> (`[ ]`)*

[13] Siehe dazu im Abschnitt 2.1.3 den Unterabschnitt „Schleifenspezifikationen“.

Die Quantifizierung findet über einen bestimmten Wertebereich statt, der vom Typ der quantifizierten Variablen abhängt. Dieser Wertebereich kann durch die Angabe eines optionalen Prädikats eingeschränkt werden; ist dies nicht der Fall, so wird über einen größtmöglichen Bereich quantifiziert. Die Bestimmung des jeweiligen Wertebereichs wird in Abschnitt 3.4 in diesem Kapitel besprochen.

3.2.2 Bemerkungen

JML ist eine außerordentlich umfangreiche Spezifikationssprache, die neben den bisher aufgeführten Sprachkonstrukten noch eine Vielzahl weiterer Spezifikationskonstrukte anbietet. Um JMLjass genauer von JML abzugrenzen, werden diese im folgenden Abschnitt kurz aufgeführt. Ausführliche Beschreibungen der jeweiligen Konstrukte finden sich in den entsprechenden Abschnitten in [CCC+06].

Erweiterte Typspezifikationen

Neben Invarianten und History Constraints erlaubt JML auch die Definition folgender Ausdrücke:

- *Axiome* spezifizieren Prädikate, die von einem Theorembeweiser als `wahr` angenommen werden sollen. Zur Kombination von Theorembeweisern und JML-Spezifikationen sei [CCC+06] als weiterführende Lektüre empfohlen.
- *Readable If*-Ausdrücke schränken die Zugreifbarkeit von Feldern durch ein Prädikat ein, das unmittelbar vor dem *Lesen* des Feldes erfüllt sein muss.
- *Writable If*-Ausdrücke schränken die Zugreifbarkeit von Feldern durch ein Prädikat ein, das unmittelbar vor dem *Schreiben* des Feldes erfüllt sein muss.

Erweiterte Methodenspezifikationen

Methodenspezifikationen können in JML neben Vor- und Nachbedingungen weitere Ausdrücke enthalten, die insbesondere zur Definition von Rahmenbedingungen genutzt werden:

- *When*-Ausdrücke erlauben die Einbeziehung einiger Aspekte paralleler Verarbeitung in Spezifikationen, indem sie - ähnlich wie ein Semaphor - den Zugriff auf eine Ressource nur unter bestimmten Bedingungen erlaubt.
- *Assignable*-Ausdrücke definieren eine Rahmenbedingung für Methodenspezifikationen, indem sie die Menge der zugreifbaren Felder und Methoden während einer Methodenausführung einschränken.

- *Working Space*-Ausdrücke geben eine Obergrenze für den während einer Methodenausführung benötigten Arbeitsspeicher an.

- *Duration*-Ausdrücke geben eine Obergrenze für die während einer Methodenausführung benötigte Rechenzeit an.

Datengruppen

Datengruppen können in Rahmenbedingungen wie *assignable*-Ausdrücken dazu verwendet werden, Mengen von Feldern, die in einer festgelegten Beziehung zueinander stehen, unter einem Namen zusammenzufassen und zugreifbar zu machen. Außerdem erlaubt die Verwendung von Datengruppen das Verstecken von Implentierungsdetails und hilft so, das Abstraktionsniveau zu erhöhen. Details zu Datengruppen finden sich in [CCC+06, Abschnitt 9].

Spezielle Annotationen

JML stellt eine Reihe spezieller Annotationen zur Verfügung, die in manchen Spezifikationen hilfreich sein können:

- *Assume*-Statements gleichen den `assert`-Ausdrücken in Java, indem sie eine Eigenschaft spezifizieren, die zu einem bestimmten Zeitpunkt der Programmausführung erfüllt sein muss.

- *Set*-Statements werden dazu verwendet, den Wert von als `ghost` deklarierten Feldern zu verändern.

- *Unreachable*-Statements definieren einen Punkt im Kontrollfluss, der zur Laufzeit eines Programms nicht erreicht werden soll.

- *Debug*-Statements können innerhalb von JML-Annotationen definiert werden und erleichtern die Fehlersuche in Spezifikationen.

Modellprogramme

Modellprogramme sind im Wesentlichen umfangreiche Spezifikationen, die das Verhalten einer Implementierung in Teilen vorwegnehmen soll. Dazu kann ein Modellprogramm zum Beispiel einen Zustand und bestimmte Methodenaufrufe definieren, die von der Implementierung nachvollzogen werden müssen, wenn sich die Programmausführung in einem Zustand befindet, der die spezifizierten Bedingungen erfüllt. Da Modellprogramme sehr umfassend sein können, sei an dieser Stelle auf [CCC+06, Abschnitt 14] verwiesen.

Code-Contract Spezifikationen

Code-Contract Spezifikationen definieren Rahmenbedingungen, die zur Laufzeit einer Methodenausführung eingehalten werden müssen. Dazu stehen die folgenden Ausdrücke zur Verfügung:

- *Accessible*-Ausdrücke definieren, auf welche Felder und Methoden während einer Ausführung zugegriffen werden darf.
- *Callable*-Ausdrücke definieren, welche Methoden während einer Ausführung aufgerufen werden dürfen.

MultiJava Erweiterungen

MultiJava ist eine Spracherweiterung der Programmiersprache Java, die *offene Klassen* und *symmetrisches, mehrfaches Dispatchen* unterstützt. Offene Klassen erlauben es, einer Java-Klasse neue Methoden hinzuzufügen, ohne den Quellcode selbst verändern oder kennen zu müssen. Durch das symmetrische, mehrfache Dispatchen können Methodenaufrufe abhängig von den *Laufzeittypen* einer beliebigen Teilmenge der Methodenargumente aufgelöst werden; im Gegensatz dazu steht das von Java verwendete einfache Dispatchen, bei dem die Auflösung eines Methodenaufrufs von den statischen Übersetzungszeittypen der Methodenargumente abhängig ist. Details zu MultiJava finden sich insbesondere in der Master's Thesis von Curtis C. Clifton [Cli01].

JML stellt mehrere spezielle Sprachkonstrukte zur Verfügung, um mit diesen Neuerungen adäquat umgehen zu können. An dieser Stelle sollen diese nicht besprochen werden; es sei aber auf [CCC+06, Abschnitt 17] als weiterführende Literatur verwiesen.

3.3 Architektur

Im vorherigen Abschnitt wurde die Spezifikationssprache JMLjass als signifikante Teilsprache von JML vorgestellt. Ein Ziel dieser Diplomarbeit ist es, JMLjass als Eingabesprache für das Runtime-Checking Werkzeug Jass zu implementieren. Im Folgenden wird daher die Architektur des Precompilers, der JMLjass-Spezifikationen in ausführbaren Java-Quellcode umwandeln soll, vorgestellt. Dazu gehören neben dem grundlegenden Aufbau auch die einzelnen Analysephasen der Übersetzung und natürlich die Übersetzungsmuster der JMLjass-Sprachkonstrukte.

3.3.1 Anforderungen

Der zu entwickelnde Precompiler muss mehreren Anforderungen gerecht werden, um die Übersetzung der JMLjass-Sprachkonstrukte in ausführbaren Quellcode zuverlässig und korrekt ausführen zu können:

- Die Spezifikationen müssen auf *Typkorrektheit* geprüft werden, um eventuelle Typfehler aufzudecken und um keinen fehlerhaften Code zu produzieren.
- Die für die Verwendung einiger Sprachkonstrukten wie etwa `\result`-Ausdrücken diskutierten Einschränkungen und Nebenbedingungen müssen eingehalten werden.
- Die im Abschnitt 3.4 diskutierten Einschränkungen für die Namensgebung von Methoden und Feldern müssen eingehalten werden, um Namenskonflikte zu vermeiden.
- Der Precompiler sollte Code generieren, der Verletzungen von Spezifikationen durch die Ausgabe aussagekräftiger Fehlermeldungen anzeigt.

Die hier getroffenen Überlegungen fließen als grundlegende Designentscheidungen in den Aufbau des Precompilers mit ein.

3.3.2 Aufbau des Precompilers

Der Weg von JMLjass-Annotationen bis hin zum ausführbaren Java-Bytecode führt über zwei Teilstationen, die in der folgenden Abbildung dargestellt sind:

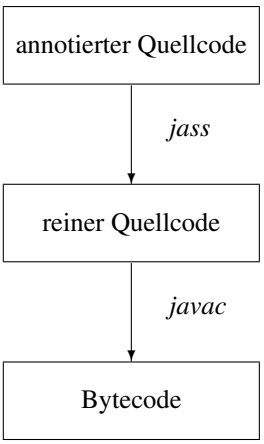


Abbildung 3.1: Übersetzung von annotiertem Quellcode in Java-Bytecode.

Der vom JMLjass-Precompiler *jass* übersetzte annotierte Java-Quellcode kann durch den Java-eigenen Compiler *javac* in Bytecode übersetzt werden, der in einer Java Virtual Machine ausführbar ist. Um dies zu leisten, verarbeitet *jass* ein Dokument in vier aufeinanderfolgenden Phasen:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Semantische Analyse
4. Zieltexterzeugung

Für jede dieser Phasen existiert eine spezialisierte Komponente des Precompilers, die für die jeweiligen verarbeitenden Aktionen zuständig ist:

1. Lexikalische Analyse: *Tokenmanager*
2. Syntaktische Analyse: *Parser*
3. Semantische Analyse: *ReflectVisitor*
4. Zieltexterzeugung: *OutputVisitor*

Anschaulich wird die prinzipielle Zusammenarbeit dieser Komponenten in folgender Abbildung:

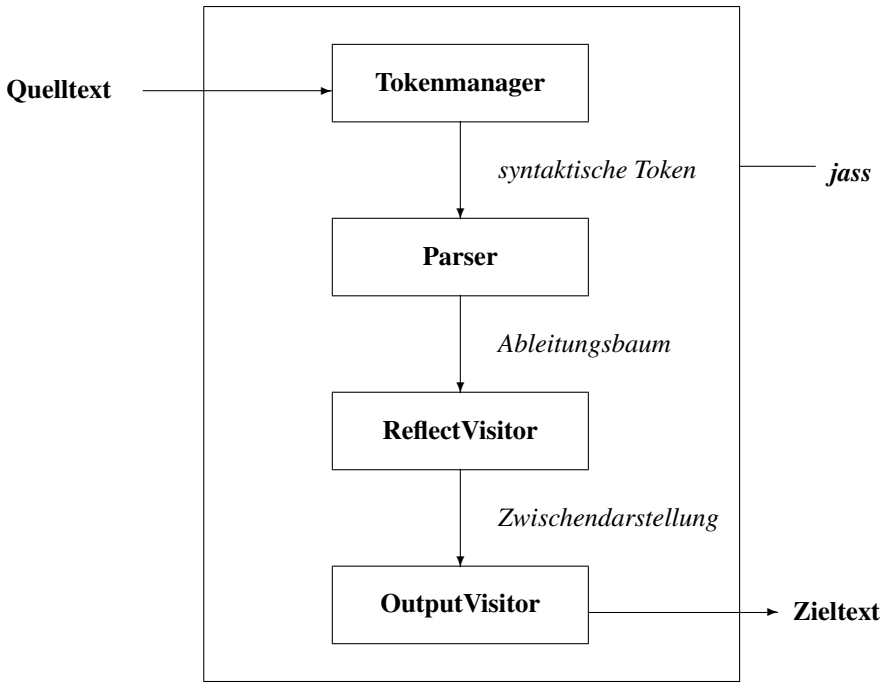


Abbildung 3.2: Übersicht über die Komponenten von *jass* und ihre Aufgaben.

3.3.3 Phasen der Übersetzung

In Abbildung 3.2 ist die Zusammenarbeit der Komponenten des Precompilers *jass* dargestellt. Jede dieser Komponenten erhält eine bestimmte *Eingabe* und erzeugt durch die Bearbeitung der ihr zugewiesenen Teilaufgabe des gesamten Übersetzungsvorgangs eine *Ausgabe*, die der jeweils nachfolgenden Komponente wiederum als Eingabe dient. Im Folgenden werden die einzelnen Komponenten und ihre jeweilige Funktionalität näher vorgestellt.

Lexikalische Analyse

Aufgabe der lexikalischen Analyse ist es, einen Quelltext (in diesem Fall einen annotierten Quellcode) zeichenweise zu untersuchen und die dabei gelesenen Symbole zu sogenannten *Token* zusammenzusetzen. Ein solches Token entspricht dabei einem Terminalsymbol der Grammatik des Eingabedokuments. So besteht etwa die hier zeichenweise dargestellte Definition einer Invarianten

```
i n v a r i a n t i > 0 ;
```

aus insgesamt fünf Tokens, die bei der Analyse von links nach rechts zu folgendem Ausdruck zusammengesetzt werden:

```
invariant i > 0;
```

In *jass* übernimmt der sogenannte *Tokenmanager* die lexikalische Analyse. Der Tokenmanager ist ein Teil des durch JavaCC generierten Parsers und arbeitet nach dem Prinzip eines endlichen Automaten: Beginnend in einem *Default* genannten Startzustand akzeptiert er eine Folge von Eingabesymbolen und durchläuft dabei eine Reihe von Transitionen, bis er in einem akzeptierenden Zustand angelangt. Die Aneinanderreihung der bis dahin gelesenen Symbole ergibt ein Token, das dann an den Parser weitergereicht wird[14]. Wie endliche Automaten kann auch der Tokenmanager in einen Zustand gelangen, in dem das nächste Eingabezeichen nicht akzeptiert wird und die gelesenen Symbole somit keinem Terminal der Grammatik entsprechen. In diesem Fall wird eine Fehlermeldung an den Benutzer ausgegeben, da es sich nicht um einen im Sinne der repräsentierten Grammtik gültigen Quelltext handeln kann.

Der für *jass* entwickelte Tokenmanager ähnelt dabei eher einem Kellerautomaten als einem einfachen endlichen Automaten, da er neben einer internen Zustandrepräsentation auch über einen kleinen kellerförmigen Speicher verfügt, der Informationen über vorangegangene Zustände enthalten kann. Dies ist unter anderem notwendig, um Spezifikationen, die in der Form von erweiterten Kommentaren gegeben sind, und eigentliche Java-Kommentare auseinander halten zu können.

Es ist zu bemerken, dass die lexikalische und die syntaktische Analyse streng genommen nicht nacheinander ausgeführt werden, sondern parallel. Dabei erzeugt der Tokenmanager nicht etwa eine abgeschlossene Folge von Tokens, die dem Parser dann als vollständige Eingabe zur Verfügung steht, sondern liefert nur auf Anfrage das jeweils nächste Token des Eingabestroms:

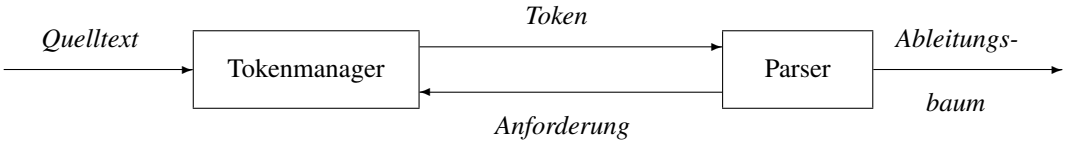


Abbildung 3.3: Zusammenspiel von Tokenmanager und Parser

Syntaktische Analyse

Die syntaktische Analyse dient dazu, Eingaben in Form eines Tokenstroms gegen eine gegebene Grammatik syntaktisch zu überprüfen. Dabei wird ein Ableitungsbaum erzeugt, dessen Blätter aus den gefundenen Terminalsymbolen bestehen. *Jass* verwendet zur syntaktischen Analyse einen aus der JMLjass-Grammatik durch JavaCC generierten LL(1)-Parser. Die Funktionsweise dieser Parser wurde bereits im Abschnitt 3.1.2 besprochen und soll an dieser Stelle nicht weiter betrachtet

[14] Siehe dazu Abbilding 3.2.

werden.

Während des Ableitungsprozesses können syntaktische Fehler auftreten, die der Parser dem Benutzer signalisieren muss. Dabei muss die *Fehlerbehandlung* des Parsers folgende Ziele verfolgen:

- Fehler müssen präzise und aussagekräftig gemeldet werden.
- Die Performanz der Verarbeitung fehlerfreier Quelltexte darf durch die Fehlerbehandlung nicht wesentlich beeinträchtigt werden.
- Der Parser sollte sich von auftretenden Fehlern erholen, um in einem Durchlauf mehrere Fehler melden zu können.

Insbesondere die *Fehlererholung* stellt eine Herausforderung beim Entwurf eines guten Parsers dar, da sie bei umfangreichen Grammatiken schnell sehr komplex werden kann. Dabei wird in [ASU99] zwischen verschiedenen *Recovery-Strategien* unterschieden:

- Panische Recovery
- Konstrukt-orientierte Recovery
- Fehlerproduktionen
- Globale Korrekturen

Die *panische Fehlererholung* ist die einfachste und am leichtesten zu implementierende Recovery-Strategie. Dabei überspringt der Parser im Falle eines Fehlers so lange Eingabesymbole, bis ein sogenanntes *synchronisierendes* Symbol auftritt. Synchronisierende Symbole signalisieren den Abschluss eines ganzen - und in diesem Falle fehlerhaften - Sprachkonstrukts; in Java sind dies etwa das Semikolon `;` am Ende eines Ausdrucks oder die schließende geschweifte Klammer `}`, die einen ganzen Block beendet. Nach dem Lesen eines entsprechenden Symbols setzt der Parser die Ableitung normal fort, um eventuell weitere Fehler finden zu können.

Bei der *konstrukt-orientierten Recovery* kann der Parser versuchen, bekannte Fehler durch das Einsetzen passender Sprachkonstrukte zu beheben. Häufige Fehler sind etwa die Verwechslung von Komma und Semikolon oder unvollständige Klammerungen. Beim Desing des Parsers muss dazu anhand von Fehlerheuristiken entschieden werden, welche Ersetzung der Parser vornehmen soll. Dabei ist zu bemerken, dass ein ungeschicktes Einsetzen von Symbolen unter Umständen zu Endlosschleifen führen kann.

Durch das Einfügen von *Fehlerproduktionen* in die Grammatik kann ein Parser auf einfache Weise häufig auftretende Fehler erkennen und ohne Überspringen

oder Ersetzen von Eingabesymbolen die Arbeit fortsetzen. Auch hier erleichtern Fehlerheuristiken die Designentscheidung, welche Fehlerarten auf diese Weise behandelt werden sollen.

Das Konzept der *globalen Korrekturen* beschreibt den Versuch, durch minimale Änderungen an der fehlerhaften Eingabe zu einem gültigen Ableitungsbaum zu kommen. Da die Ermittlung eines Programms *minimalen Abstands* nach [ASU99] sehr rechenzeit- und speicherintensiv sein kann, wird dieser Ansatz in der Praxis kaum verfolgt.

Der für *jass* entwickelte Parser verfügt aus Gründen des Umfangs dieser Arbeit nur in Ansätzen über eine panische Fehlererholung. An dieser Stelle sei daher auf das abschließende Kapitel 6 verwiesen, das sich unter anderem mit Verbesserungsmöglichkeiten des entwickelten Precompilers beschäftigt.

Semantische Analyse

Aufgabe der semantischen Analyse ist es, während der Verarbeitung eines Quelltextes durch einen Compiler nicht-syntaktische Programmierfehler aufzudecken und die semantischen Nebenbedingungen einer Programmiersprache auf Einhaltung zu überprüfen. Es müssen also

- Ausdrücke auf ihre Typisierung hin überprüft werden.
- Zuweisungen auf ihre Wohldefiniertheit[15] untersucht werden.
- Die Einhaltung von Eindeutigkeitsbedingungen zum Beispiel bei der Namensvergabe für Variablen sicher gestellt werden.

In *jass* übernimmt der sogenannte *ReflectVisitor* die Aufgaben der semantischen Analysephase. Dabei erhält er als Eingabe den vom Parser generierten Ableitungsbaum und traversiert diesen von der der Wurzel ausgehend. Während der Traversierung werden an jedem *besuchten* Knoten - abhängig von dessen Art - unterschiedliche Aktionen ausgeführt:

- Prüfung der oben aufgeführten semantischen Eigenschaften des repräsentierten Teilausdrucks.
- Erzeugung einer internen Repräsentation von Teilausdrücken.
- Zusammensetzung bereits besuchter Teilausdrücke und Vereinfachung des entstandenen Konstrukts.
- Erzeugung der Zwischendarstellung in Form bestimmter Klasseninstanzen[16],

[15] Zur Wohldefiniertheit gehören insbesondere die Existenz der linken und rechten Seite der Zuweisung und eine korrekte Typisierung.

[16] Siehe dazu auch Kapitel 4.

der den Ableitungsbaum auf eine konkretere Ebene abbildet.

Abbildung 3.4 verdeutlicht noch einmal das Zusammenspiel zwischen Parser und ReflectVisitor:

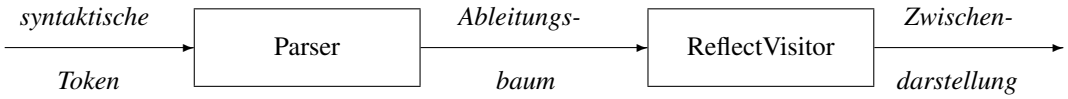


Abbildung 3.4: Zusammenspiel von Parser und ReflectVisitor

Zieltexterzeugung

In der letzten Phase eines Übersetzungsvorgangs wird schließlich der Zieltext erzeugt. Für den Precompiler *jass* bedeutet dies, dass aus den JMLjass-Annotationen reiner Java-Quellcode generiert wird, der das spezifizierte Verhalten zur Laufzeit überprüft. Dazu dient der *OutputVisitor*, der als Eingabe einerseits die vom ReflectVisitor erzeugte Zwischendarstellung verwendet und andererseits auf den Ableitungsbaum zurückgreift, um die Struktur des ursprünglichen Dokuments zu rekonstruieren. Dabei spielen die im nächsten Abschnitt erläuterten *Übersetzungsmuster* eine zentrale Rolle.

Abbildung 3.5 stellt das Zusammenspiel zwischen Parser, ReflectVisitor und OutputVisitor dar:

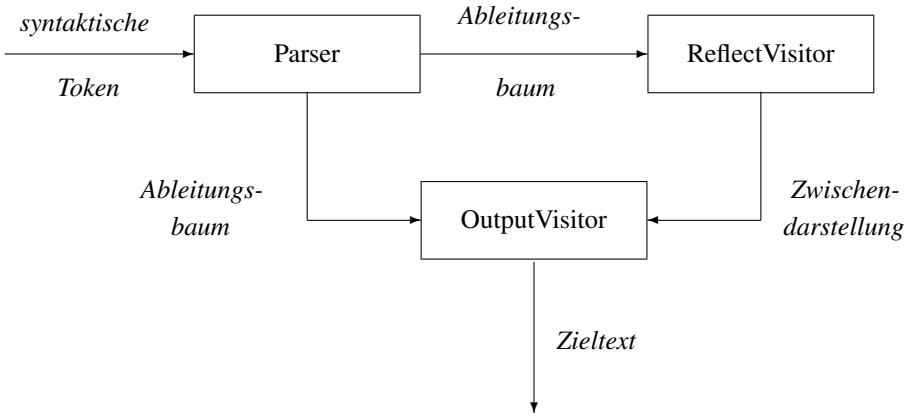


Abbildung 3.5: Zusammenspiel von Parser, ReflectVisitor und OutputVisitor

Abschließend gibt Abbildung 3.6 noch einmal detailliert die Zusammenarbeit aller Komponenten von *jass* wider:

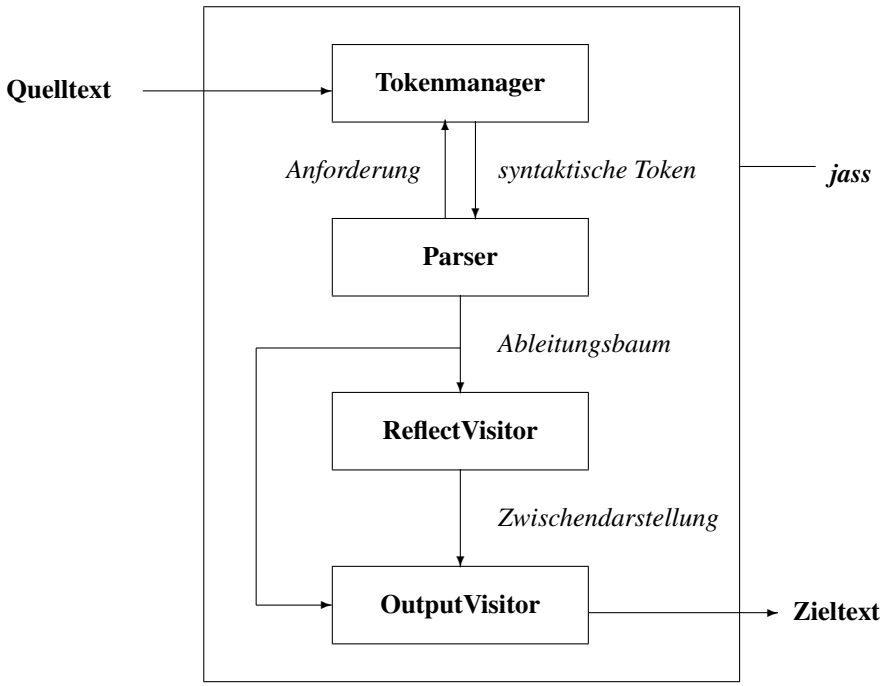


Abbildung 3.6: Detaillierte Übersicht über die Zusammenarbeit der Komponenten von *jass*.

3.4 Übersetzungsmuster der JMLjass-Konstrukte

Es ist die Aufgabe des Jass-Precompilers *jass*, aus JMLjass-Spezifikationen reinen Java-Quellcode zu erzeugen, der die spezifizierten Eigenschaften zur Laufzeit einer Klasse auf Einhaltung überprüft. Im letzten Abschnitt wurden die dazu notwendigen Einzelschritte von der lexikalischen Analyse bis zur Zieltexterzeugung detailliert erläutert. Für die letzte Phase der Übersetzung spielen die *Übersetzungsmuster* der JMLjass-Sprachkonstrukte eine entscheidende Rolle, da sie vorgeben, wie aus einer Spezifikation der entsprechende Java-Code generiert wird. Der folgende Abschnitt beschäftigt sich daher mit den Übersetzungsmustern der verschiedenen Sprachelemente von JMLjass.

Im Folgenden wird nur von Spezifikationen in Klassen die Rede sein, da es für die Übersetzungsmuster von Methodenspezifikationen im Wesentlichen keine Rolle spielt, ob die Spezifikationen in Klassen oder in Interfaces definiert wurden. Eine besondere Schwierigkeit beim Umgang mit Interfaces stellt jedoch die Einschränkung dar, dass in Interfaces selbst nur abstrakte Methoden angegeben werden können. Um dies zu umgehen, generiert *jass* eine innere Klasse, genannt *Surrogatklasse*, für die sich dann die Methoden zur Prüfung der spezifizierten Eigenschaften durch die im Folgenden beschriebenen Übersetzungmuster erstellen lassen. Der Umgang mit Interfaces und die Struktur von Surrogatklassen wird in Kapitel 4 besprochen und soll an dieser Stelle nicht weiter betrachtet werden.

Für die im Folgenden beschriebenen Übersetzungsmuster spielt die *Sternchenform* von Ausdrücken eine wichtige Rolle:

Definition 3.1 (Sternchenform von Ausdrücken):

Aus einem beliebigen, innerhalb einer JMLjass-Spezifikation verwendbaren Ausdruck `P` entsteht durch Ersetzen aller spezifikationsinterner Teilausdrücke durch Aufrufe der Methoden, die durch Anwenden der relevanten Übersetzungsmuster auf das Gesamtdokument enstanden sind, der Ausdruck `P*`. Dabei ist `P*` ein im Sinne der Grammatik von Java 1.5 gültiger Ausdruck.

Enthält das Prädikat einer Invarianten etwa eine Referenz auf eine Modellvariable, so muss diese durch den Aufruf der entsprechenden evaluierenden Methode ersetzt werden.

3.4.1 Invarianten und Constraints

Invarianten spezifizieren ein Prädikat, das in jedem sichtbaren Zustand der Methodenausführung erfüllt sein muss. Um dies zur Laufzeit zu überprüfen, erzeugt *jass* für die Invarianten eine Methode, die die entsprechenden Prädikate überprüft und

eine mögliche Verletzung durch das Werfen einer Exception anzeigt:

Definition 3.2 (Übersetzungmuster für Invarianten):

Eine Liste von Invarianten der Form

```
invariant P1;
...
invariant Pn;
```

wird zur Laufzeit durch folgende Methode geprüft:

```
/** Methode zur Prüfung von Invarianten. */
public boolean jml_check$instance_invariants$() {

  // Prüfe geerbte Invarianten
  super.jml_check$instance_invariants$();

  // Prüfe lokale Invarianten
  if (!P1* || ... || !Pn*)
       throw new JMLInvariantException(/* message */);

  return true;
}
```

Dabei muss zwischen statischen und instanzgebundenen Invarianten unterschieden werden, die in jeweils einer eigenen statischen bzw. instanzgebundenen Methode geprüft werden. Der Präfix `jml_check` ist für von *jass* generierte Methoden reserviert und weist darauf hin, dass nachfolgend die Einhaltung eines Spezifikationskonstrukts, in diesem Fall der instanzgebundenen Invarianten, geprüft wird. Eventuell geerbte Invarianten werden durch Aufrufen einer entsprechenden Methode in der Superklasse geprüft. Dazu wird an dieser Stelle der Mechanismus der `super`-Referenzierung verwendet. Werden Spezifikationsanteile von Interfaces geerbt, so muss anstelle einer `super`-Referenz eine sogenannte *Surrogatvariable* zur Referenzierung der prüfenden Methode verwendet werden. Surrogatvariablen werden, wie die Surrogklassen auch, in Kapitel 4 besprochen.

History Constraints werden zur Laufzeit durch eine analog erzeugte Methode geprüft:

Definition 3.3 (Übersetzungsmuster für History Constraints):

Eine Liste von Constraints der Form

```
constraint P1;
...
constraint Pn;
```

wird zur Laufzeit durch folgende Methode geprüft:

```
/** Methode zur Prüfung von Constraints. */
public boolean jml_check$instance_constraints$() {

  // Prüfe geerbte Constraints
  super.jml_check$instance_constraints$();

  // Prüfe lokale Constraints
  if (!P1* || ... || !Pn*)
       throw new JMLConstraintException(/* message */);

  return true;
}
```

Die für Invarianten und Constraints generierten Methoden werden während der Programmausführung zu den sich durch ihre Semantik ergebenden sichtbaren Zeitpunkten aufgerufen[17].

3.4.2 Modellvariablen und Initiallies

Eine Modellvariable ist vollständig definiert durch

- ein mittels `model` modifiziertes Feld,
- einer durch einen `represents`-Ausdruck festgelegten Abstraktionsfunktion.

Der Wert einer Modellvariablen in einem sichtbaren Zustand ergibt sich zur Laufzeit durch die Auswertung ihrer Abstraktionsfunktion. Diese erfolgt in einer von *jass* erzeugten Methode, deren Rückgabetyp dem Typ der Modellvariablen entsprechen muss. Ausgehend von einer Modellvariable `var` vom Typ `T` und ihrer Abstraktionsfunktion `func`, die in einer Klasse `C` spezifiziert werden, ergibt sich folgendes Übersetzungsmuster:

Definition 3.4 (Übersetzungsmuster von Modellvariablen):

Eine Modellvariable und die zugehörige Abstraktionsfunktion der Form

```
private model T var;
represents var <- func;
```

wird zur Laufzeit von einer Methode gleicher Sichtbarkeit ausgewertet:

[17] Die Semantik von Invarianten und Constraints wurde bereits in Abschnitt 2.1.2 besprochen.

```
/** Methode zur Auswertung einer Modellvariablen. */
private T jml_evaluate$model_field$var() {

  return func*;
}
```

Der Präfix `jml_evaluate` ist für von *jass* generierte Methoden reserviert und weist darauf hin, dass nachfolgend ein Sprachkonstrukt, in diesem Fall eine Modellvariable, ausgewertet wird; die Modifikation der evaluierenden Methode entspricht dabei derjenigen der Modellvariable.

Während der Übersetzung des annotierten Quellcodes werden alle Referenzierungen einer Modellvariable durch Aufrufe der auswertenden Methode ersetzt. Dadurch muss für Modellvariablen zur Laufzeit kein Speicher vorgehalten werden; gleichzeitig garantiert diese Art der Auswertung stets die größtmögliche Aktualität der repräsentierten Größe. Werden Modellvariablen in Interfaces definiert, so wird die zugehörige Abstraktionsfunktion für nicht-statische Modellvariablen erst in einer implementierenden Klasse spezifiziert. Dieser häufig auftretende Fall wird später im Abschnitt 4.2 besprochen.

Für Initiallies werden analog zu Invarianten und Constraints Methoden generiert, die deren Einhaltung überprüfen:

Definition 3.5 (Übersetzungsmuster von Initiallies):

Eine Liste von Initiallies der Form

```
initially P1;
...
initially Pn;
```

wird zur Laufzeit durch folgende Methode geprüft:

```
/** Methode zur Prüfung der Initiallies. */
public boolean jml_check$initiallies$() {

  // Prüfe geerbte Initiallies
  super.jml_check$instance_invariants$();

  // Prüfe lokale Initiallies
  if (!P1* || ... || !Pn*)
       throw new JMLInitiallyException(/* message */);

  return true;
}
```

Dabei müssen natürlich auch in den Prädikaten P1,...,Pn Referenzierungen von Modellvariablen durch Aufrufe der evaluierenden Methoden ersetzt werden.

3.4.3 Methodenspezifikationen

Methodenspezifikationen tragen wesentlich zur Beschreibung des Verhaltens von Methodenausführungen zur Laufzeit bei. Bei ihrer Übersetzung sind mehrere Aspekte zu beachten:

- Methodenspezifikationen können unterschiedliche Sichtbarkeiten aufweisen.
- Bei der Prüfung von Nachbedingungen muss zwischen normaler Terminierung und Terminierung durch Werfen einer Exception unterschieden werden.
- Die Vererbung und Verfeinerung von Spezifikationen muss mit in Betracht gezogen werden.

Die hier getroffenen Überlegungen fließen in die Entwicklung der entsprechenden Übersetzungsmuster mit ein, die im Folgenden vorgestellt werden.

Vorbedingungen

Vorbedingungen werden durch `requires`-Ausdrücke definiert, deren Prädikate im Pre-State der Methodenausführung erfüllt sein müssen:

```
requires P;
```

Durch die Möglichkeit, mehrere Spezifikationsfälle in einer Folge von durch `also` getrennten Spezifikationen zu unterscheiden, setzt sich die Vorbedingung einer Methodenausführung auf semantischer Ebene als Disjunktion der Vorbedingungen der einzelnen Spezifikationen zusammen. Damit muss also bei n Spezifikationsfällen mit den Prädikaten `P1,...,Pn` zur Laufzeit ein Ausdruck der Form `P1 || P2 || ... || Pn` evaluiert werden. Um außerdem verschiedene Sichtbarkeiten von Spezifikationen zu berücksichtigen, werden von *jass* für die Auswertung der Vorbedingungen entsprechend modifizierte Methoden generiert.

Zur Anschauung sei eine Vorbedingung für eine Methode `m` mit Rückgabetyp `T` gegeben, die in einer Klasse `C` definiert wurde und in der allgemeinen Form einer Behavior-Spezifikation vorliegt[18]. Außerdem soll die Methode Spezifikationsanteile aus der Superklasse `S` von `C` erben:

[18] Es sei an dieser Stelle nochmals an das Desugaring von Spezifikationen erinnert, das in Abschnitt 2.1.5 besprochen wurde.

Definition 3.6 (Übersetzungsmuster von Vorbedingungen):

Eine Liste von Vorbedingungen nach dem Desugaring

```
/*@ also
  @ public behavior
  @   requires P1;
  @ also
  @ public behavior
  @   requires P2;
  @ ...
  @ also
  @ public behavior
  @   requires Pn;
  @*/
public T m() { /* Methodenkörper */ }
```

wird zur Laufzeit durch eine wie folgt aufgebaute Methode geprüft:

```
/** Methode zur Prüfung der Vorbedingungen. */
public boolean jml_check$pre_state_public$m() {

  // Hilfsfeld zur besseren Lesbarkeit.
  boolean assertionFailure = true;

  // Evaluiere geerbte Spezifikation
  boolean inherited_precond
       = super.jml_check$pre_state_public$m();

  // Evaluiere lokale Spezifikation
  boolean local_precond = P1* || P2* || ... || Pn*;


  // Prüfe Verletzung der Zusicherung
  assertionFailure = !(inherited_precond || local_precond);

  // Zeige eventuelle Verletzung an.
  if (assertionFailure)
       throw new JMLPreStateException(/* message */);

  // Liefere die lokalen Vorbedingungen zurück.
  return local_precond;
}
```

Eine Verletzung der Vorbedingungen wird also durch das Werfen einer Exception signalisiert, falls nicht mindestens eine geerbte oder lokale Vorbedingungen erfüllt ist, da sich Vorbedingungen verschiedener Spezifikationsfälle auf semantischer Ebene abschwächen[19].

[19] Siehe dazu auch Abschnitt 2.1.5.

Normale Nachbedingungen

Nachbedingungen, die durch `ensures`-Ausdrücke definiert werden, müssen im Falle normaler Terminierung im Post-State der Methodenausführung erfüllt sein:

```
ensures Q;
```

Dabei gelten Nachbedingungung als erfüllt, falls die Implikation $(\backslash old(P) \Rightarrow Q)$ zu wahr ausgewertet wird, wobei $\backslash old(P)$ das Prädikat der zugehörigen Vorbedingung einer Methodenspezifikation im Pre-State der Methodenausführung referenziert. Werden Folgen von Spezifikationsfällen für eine Methode definiert, so setzt sich die zu prüfende Nachbedingung konjunktiv aus den Implikationen der einzelnen Nachbedingungen zusammen.

Zur Anschauung sei wiederum eine Nachbedingung für eine Methode `m` mit Rückgabetyp `T` gegeben, die in einer Klasse `C` definiert wurde und in der allgemeinen Form einer Behavior-Spezifikation vorliegt. Außerdem soll die Methode Spezifikationsanteile aus der Superklasse `S` von `C` erben:

Definition 3.7 (Übersetzungsmuster von Nachbedingungen):

Eine Liste von Nachbedingungen für normale Terminierung nach dem Desugaring

```
/*@ also
  @ public behavior
  @   requires P1;
  @   ensures  Q1;
  @ also
  @ public behavior
  @   requires P2;
  @   ensures  Q2;
  @ ...
  @ also
  @ public behavior
  @   requires Pn;
  @   ensures  Qn;
  @*/
public T m() { /* Methodenkörper */ }
```

wird zur Laufzeit durch eine wie folgt aufgebaute Methode geprüft:

```
/** Methode zur Prüfung der Nachbedingungen. */
public boolean jml_check$post_state_public$m() {

  // Hilfsfeld zur besseren Lesbarkeit.
  boolean assertionFailure = true;

  // Evaluiere geerbte Spezifikation
  boolean inherited_postcond
       = super.jml_check$post_state_public$m();


  // Evaluiere lokale Spezifikation
  boolean local_postcond = (!old_P1* || Q1*)
                        && (!old_P2* || Q2*)
                        && ...
                        && (!old_Pn* || Qn*);

  // Prüfe Verletzung der Zusicherung
  assertionFailure = !(inherited_postcond
                       && local_postcond);

  // Zeige eventuelle Verletzung an.
  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  // Liefere die lokalen Nachbedingungen zurück.
  return local_postcond;
}
```

Die hier verwendeten Booleschen Variablen `old_P1*`,...,`old_Pn*` referenzieren dabei den Wert der Ausdrücke `P1`,...,`Pn` im Pre-State der Ausführung der jeweiligen Methode.

Nachbedingungen für die Terminierung durch Werfen einer Exception

Nachbedingungen, die durch `signals`-Ausdrücke definiert werden, müssen im Falle der Terminierung durch Werfen einer Exception im Post-State der Methodenausführung erfüllt sein:

```
signals (E e) Q;
```

Dabei ist E der Typ der signalisierten Exception und Q ein Prädikat, das im Post-State erfüllt sein muss. Nachbedingungung für die Terminierung durch Werfen einer Exception gelten als erfüllt, wenn die Implikation *(\old(P) ⇒ ((e instanceof E) ⇒ Q))* zu wahr ausgewertet wird[20], wobei *\old(P)* wieder das Prädikat der zugehörigen Vorbedingung einer Methodenspezifikation im Pre-State

[20] Siehe dazu Abschnitt 2.1.5.

der Methodenausführung referenziert. Werden Folgen von Spezifikationsfällen für eine Methode definiert, so setzt sich die zu prüfende Nachbedingung konjunktiv aus den Implikationen der einzelnen Nachbedingungen für Terminierung durch Werfen einer Exception zusammen.

Zur Anschauung sei wiederum eine Nachbedingung für eine Methode `m` mit Rückgabetyp `T` gegeben, die in einer Klasse `C` definiert wurde und in der allgemeinen Form einer Behavior-Spezifikation vorliegt. Außerdem soll die Methode Spezifikationsanteile aus der Superklasse `S` von `C` erben:

Definition 3.8 (Übersetzungsmuster von Nachbedingungen):

Eine Liste von Nachbedingungen für die Terminierung durch Werfen einer Exception nach dem Desugaring

```
/*@ also
  @ public behavior
  @   requires P1;
  @   signals  (E1 e) Q1;
  @ also
  @ public behavior
  @   requires P2;
  @   signals  (E2 e) Q2;
  @ ...
  @ also
  @ public behavior
  @   requires Pn;
  @   signals  (E3 e) Qn;
  @*/
public T m() { /* Methodenkörper */ }
```

wird zur Laufzeit durch eine wie folgt aufgebaute Methode geprüft:

```
/** Methode zur Prüfung der Nachbedingungen. */
public boolean
     jml_check$post_state_public$m(Exception jml_exc) {

  // Hilfsfeld zur besseren Lesbarkeit.
  boolean assertionFailure = true;

  // Evaluiere geerbte Spezifikation
  boolean inherited_postcond
       = super.jml_check$post_state_public$m(jml_exc);


  // Evaluiere lokale Spezifikation
  boolean local_postcond
      =   (!old_P1* || ((!jml_exc instanceof E1) || Q1))
       && (!old_P2* || ((!jml_exc instanceof E2) || Q2))
       && ...
       && (!old_Pn* || ((!jml_exc instanceof E3) || Qn));

  // Prüfe Verletzung der Zusicherung
  assertionFailure = !(inherited_postcond
                       && local_postcond);

  // Zeige eventuelle Verletzung an.
  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  // Liefere die lokalen Nachbedingungen zurück.
  return local_postcond;
}
```

Im Unterschied zur Prüfung der normalen Nachbedingungen wird hier eine Exception `jml_exc` als Argument übergeben, die die während der Ausführung der Methode geworfene Exception referenziert. Dazu müssen die Anweisung des Methodenkörpers von einem `try-catch`-Block umschlossen werden, der die Exception fängt und an die prüfende Methode weiterleitet. Diese Strategie wird in Kaptiel 4 detailliert besprochen und soll hier nicht weiter vertieft werden.

3.4.4 Spezielle Ausdrücke

JMLjass erlaubt die Verwendung einiger spezieller Sprachkonstrukte, die in Abschnitt 3.2.1 vorgestellt wurden. Dazu gehören die Referenzierung von Rückgabewerten, die Auswertung von Ausdrücken im Pre-State sowie die logischen Quantoren und Operatoren. Im Folgenden werden kurz die Übersetzungsmuster dieser Ausdrücke vorgestellt.

`\result` und `\old`

Die Übersetzung von `\result`- und `\old`-Ausdrücken erfolgt in beiden Fällen analog durch die Einführung eines Feldes, das den Rückgabewert einer Methode bzw. das Ergebnis der Auswertung eines Ausdrucks im Pre-State speichert. In Spezifikationsausdrücken werden dann alle `\result`- bzw. `\old`-Ausdrücke durch Referenzierungen des entsprechenden Feldes ersetzt. Details dazu können in der beispielhaften Übersetzung in Abschnitt 4.6 nachgelesen werden.

Logische Quantoren

JMLjass stellt zwei logische Quantoren für Spezifikationsausdrücke zur Verfügung: den universellen `\forall`- und den existenziellen `\exists`-Quantor. Beide Arten der Quantifizierung finden auf einem bestimmten Wertebereich statt, der vom Typ der quantifizierten Variablen abhängig ist und durch ein optionales Prädikat eingeschränkt werden kann. Die Auswertung einer Quantifizierung geschieht dabei in zwei Schritten:

1. Bestimmung des relevanten Wertebereichs
2. Auswertung der Quantifizierung durch Iteration über den gefundenen Wertebereich

Die Bestimmung des relevanten Wertebereichs erfolgt bei der Quantifizierung über nicht-numerische Typen durch eine musterbasierte statische Analyse (engl.: „pattern-based static analysis“), deren Ziel die Bestimmung einer Menge `Q` vom Typ `java.util.Collection` ist, so dass nach [Che03, Abschnitt 3.3] die folgende Äquivalenz für einen `\forall`-Ausdruck gegeben ist[21]:

$$(\text{\forall T x; E1; E2}) \equiv (\text{\forall T x; Q.contains(x); E1} \Rightarrow \text{E2})$$

Dabei ist `T` ein Java-Typ, `E1` ein optionales, den Wertebereich einschränkendes Prädikat, und `E2` der Boolesche Ausdruck, über den quantifiziert wird. Die für *jass* implementierten Muster zur Bestimmung von `Q` werden in Abschnitt 4 besprochen.

Bei Quantifizierungen über numerische Datentypen vereinfacht sich die Bestimmung des notwendigen Wertebereichs durch die natürlichen Beschränkungen, denen diese unterworfen sind; insbesondere gibt es nämlich für jeden numerischen Typ eine obere und eine untere Schranke.

[21] Der Existenzquantor wird natürlich analog behandelt.

Kapitel 4

Realisierung

Im vorherigen Kapitel wurde der Precompiler *Jass*, der JMLjass-Spezifikationen in ausführbaren Java-Quellcode übersetzt, vorgestellt und in seinen grundlegenden Funktionsprinzipien beschrieben. Dabei spielten insbesondere die abstrakten Übersetzungsmuster der Sprachelemente eine wichtige Rolle.
Dieses Kapitel wird sich zunächst mit Spezifikationen speziell in Interfaces beschäftigen und auf Besonderheiten bei deren Übersetzung hinweisen. Daran anschließen wird sich eine Diskussion über die Umsetzung von Modellvariablen sowie eine ausführliche Beschreibung der Realisierung von Vererbung und Verfeinerung von Spezifikationen. Abschließend wird dann anhand eines kurzen Beispiels ein Übersetzungsvorgang exemplarisch dargestellt und auf Besonderheiten des generierten Quellcodes hingewiesen.

4.1 Spezifikationen in Interfaces

Die Übersetzung von Spezifikationen in Interfaces erfolgt im Wesentlichen analog zur Verarbeitung von Spezifikationen in Klassen; insbesondere werden auch hier die im vorherigen Kapitel besprochenen Übersetzungsmuster angewandt. Allerdings ergeben sich aufgrund der programmiersprachlichen Besonderheiten von Interfaces gegenüber Klassen einige Probleme, die im Folgenden näher betrachtet werden sollen.

Bei der Übersetzung von Spezifikationen in Interfaces müssen mehrere Nebenbedingungen bedacht werden:

- Innerhalb Interfaces dürfen nur abstrakte Methoden definiert werden.
- Es dürfen nur konstante Felder verwendet werden.
- Es können beliebig viele andere Interfaces erweitert werden, was in die Überlegungen zur Vererbung von Spezifikationen mit einfließen muss.

Die ersten beiden Einschränkungen stellen sowohl für die Übersetzung als auch für die Laufzeitprüfung von Spezifikationen eine echte Schwierigkeit dar. Zum Einen ist ein Interface in Java stets zustandslos, während es jedoch zum Beispiel durch die Verwendung von Modellvariablen einen für die Laufzeitprüfung relevanten Zustand erhalten kann; zum Anderen können wegen der Forderung nach Abstraktheit die zur Prüfung der spezifizierten Eigenschaften benötigten Methoden nicht im Interface selbst generiert werden.
Um mit diesen Problemen umgehen zu können, erzeugt *Jass* zur Übersetzungszeit für ein Interface eine interne *Surrogatklasse*[1], die die erforderlichen Methoden zur Prüfung der Spezifikationen und eine entsprechende Zustandsrepräsentation des Interfaces zur Laufzeit zur Verfügung stellt. Die Erzeugung der benötigten Methoden und Felder erfolgt dann nach den bereits vorgestellten Übersetzungsmustern innerhalb der Surrogatklasse:

```
public interface Interface {

  /* Surrogatklasse */
  public class JMLjassSurrogate {

    // Methoden zur Laufzeitprüfung der Spezifikationen
    // und zur Darstellung des Interfacezustands
    < ... >
  }

  // Interfacedefinitionen
  < ... >
}
```

Die Instanziierung der Surrogatklasse erfolgt automatisch während der dynamischen Initialisierung eines Objekts einer Klasse, die das entsprechende Interface implementiert. Dadurch exisitert zur Laufzeit zu jeder Instanz einer Klasse jeweils eine eindeutige Instanz der Surrogatklasse aller relevanten[2] implementierten Interfaces.

Durch die Verwendung von Surrogatklassen lässt sich auch die Vererbung von Spezifikationen zwischen Klassen und Interfaces auf einfache Weise realisieren, indem eine Klasse für jedes implementierte Interface um eine Variable erweitert wird, die eine Referenz auf die Surrogatklasse des Interfaces hält. Mittels dieser Referenz wird dann zur Laufzeit bei der Prüfung eines Spezifikationselements die entsprechende Methode der Surrogatklasse aufgerufen und so die geerbten Spezifikationsanteile überprüft. Im Folgenden wird dies für die Prüfung von Invarianten dargestellt:

[1] Der Begriff Surrogatklasse leitet sich vom englischen „surrogate“ = Stellvertreter ab.

[2] Als relevant in diesem Sinne gelten alle Interfaces, von denen eine Klasse Spezifikationsanteile erbt.

Vererbungsmuster für Invarianten

```
/* Interface mit übersetzten JMLjass-Spezifikationen */
public interface MyInterface {

  /* Surrogatklasse */
  public class JMLjassSurrogate {

    // Prüfung der Invarianten in MyInterface
    public boolean jml_check$instance_invariants$() {

      // Prüfung lokaler Invarianten
      < ... >
    }

    // Methoden zur Laufzeitprüfung
    // der Spezifikationen
    < ... >
  }

  // Interfacedefinitionen
  < ... >
}

/* Klasse, die Spezifikationen erbt. */
public class MyClass implements MyInterface {

  /* Surrogatvariable */
  private MyInterface.JMLjassSurrogate var;

  {
    var = new MyInterface.JMLjassSurrogate();
  }

  // Prüfung der Invarianten in MyClass
  public boolean jml_check$instance_invariants$() {

    // Prüfung geerbter Invarianten
    var.jml_check$instance_invariants$();

    // Prüfung lokaler Invarianten
    < ... >
  }

  // Klassendefinitionen
  < ... >
}
```

Die Initialisierung der Surrogatvariablen erfolgt während der dynamischen Initialisierung der Klasse; damit erhält jede Instanz einer Klasse automatisch eine Referenz auf eine Instanz der Surrogtklasse eines implementierten Interfaces.

Die Vererbung zwischen Interfaces erfolgt prinzipiell auf die gleiche Weise, nur dass hierbei die Surrogatklasse um eine entsprechende Variable erweitert wird. Bei der Instanziierung einer Klasse werden somit implizit Instanzen der Surrogatklassen aller implementierter Interfaces und deren Supertypen erzeugt. Abbildung 4.1 zeigt eine Vererbungskette zweier Interfaces und einer implementierenden Klasse:

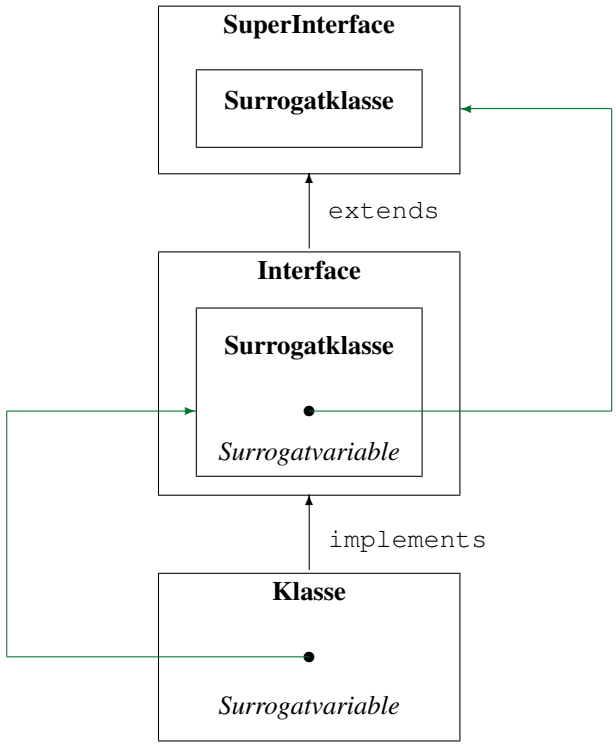


Abbildung 4.1: Schema der Vererbung von Spezifikationen mittels Surrogatvariablen

4.2 Die Behandlung von Modellvariablen

Modellvariablen sind spezifikationsinterne Variablen, die einerseits einen wichtigen Beitrag zur Abstraktion von Spezifikationen bezüglich der Implementierung einer Klasse oder eines Interfaces leisten und andererseits Aussagen über den spezifikationsbezogenen Zustand[3] von Interfaces erlauben. Insbesondere die Umsetzung der zweiten Eigenschaft beinhaltet aber einige Schwierigkeiten, deren Behandlung im Folgenden besprochen wird. Dabei beschränkt sich dieser Abschnitt auf den Fall, dass die Definition der Modellvariablen und der zugehörigen Abstraktionsfunktion in getrennten Dokumenten stattfindet, da ansonsten das in Abschnitt 3.4 vorgestellte Übersetzungsmuster ohne Einschränkung umsetzbar ist.

Der Zustands eines Interfaces ist, wie bereits angemerkt wurde, abhängig von der Belegung der in diesem definierten Spezifikationsvariablen, wobei aus der Sicht der Laufzeitprüfung insbesondere die Modellvariablen von Interesse sind. Modellvariablen werden durch die Modifikation eines Feldes mittels des Schlüsselworts `model` eingeführt; ihre konkrete Belegung in einem sichtbaren Zustand ergibt sich jedoch erst durch die Auswertung der zugehörigen Abstraktionsfunktion. Bei Modellvariablen in Interfaces entwickelt sich die Art ihrer Repräsentation meist erst aus Designentscheidungen während der Implementierung heraus, wie am folgenden Beispiel klar wird:

Beispiel 4.1 (Modellvariable in Interface)

```
// Einfaches Interface für eine Liste
public interface ListInterface<T> {

  // Modelvariable, die die Größe
  // der Liste repräsentiert
  //@ public model int size;

  //@ requires _entry != null;
  //@ ensures  size >= \old(size);
  public <T> void add(T _entry);
}
```

Die im generischen Interface `ListInterface` spezifizierte Eigenschaft, dass die Größe der Liste beim Hinzufügen eines von `null` verschiedenen Eintrags mittels der Methode `add` nicht kleiner wird, greift auf die Modellvariable `size` zurück. Da jede Klasse, die eine solche Liste realisieren soll, in der Wahl der internen Repräsentation der Liste frei ist, ergibt sich auch die Abstrakionsfunktion der Modellvariablen erst in Abhängigkeit von der konkreten Implementierung. ■

[3] Es sei nochmals daran erinnert, dass ein Interface aus Sicht von Java zustandslos ist.

Um aus der Sicht der Laufzeitprüfung von Spezifikationen in Interfaces mit Situationen, in denen die Abstraktionsfunktion einer in einem Interface definierten Modellvariablen erst in einer implementierenden Klasse spezifiziert wird, adäquat umgehen zu können, muss die Möglichkeit gegeben sein, eine Instanz einer solchen Klasse zur Laufzeit referenzieren zu können, um den aktuellen Wert einer Modellvariablen zu bestimmen. Dazu generiert *Jass* in der im vorherigen Abschnitt vorgestellten Surrogatklasse eines Interfaces eine Variable, die eine Referenz auf eine Instanz einer implementierenden Klasse speichert:

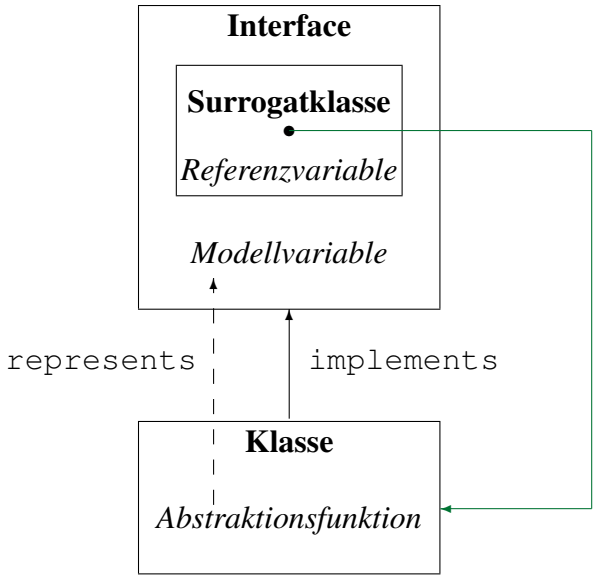


Abbildung 4.2: Referenzierung der Abstraktionsfunktion einer nicht-statischen Modellvariablen in einem Interface.

Die Initialisierung der Referenzvariablen erfolgt dabei während der dynamischen Initialisierung der Surrogatklasse, die durch die Erzeugung einer Instanz einer Klasse, die das entsprechende Interface implementiert, ausgelöst wird. Dies garantiert die frühestmögliche Verfügbarkeit der benötigten Abstraktionsfunktion. Die Referenzvariable wird ebenfalls dazu benutzt, um Methoden, die in Spezifikationen innerhalb von Interfaces benötigt werden, dynamisch aufzulösen und auszuwerten. Dadurch können innerhalb von Interfaces in Spezifikationen auch die Methoden des Interfaces selbst verwendet werden, um Aussagen über das erwartete Verhalten zur Laufzeit zu machen.

In der Implementierung von *Jass* wird dieser Ansatz durch die Erzeugung eines Konstruktors für Surrogatklassen realisiert, der eine Instanz einer implementierenden Klasse als Argument erwartet und den leeren Standardkonstruktor[4] ersetzt:

[4]In Java besitzt jede Klasse einen impliziten Konstruktor ohne Argumente, der durch die Angabe eines konkreten Konstruktors ersetzt werden kann.

```
public interface Interface {

  // Surrogatklasse
  public class JMLjassSurrogate {

    // Referenzvariable
    private Interface referenceVar;

    // Generierter Konstruktor
    public JMLjassSurrogate(Reference _instance) {

      referenceVar = _instance;
    }

    < Methoden zur Spezifikationsprüfung >
  }

  < Interfacedeklarationen >
}
```

Dadurch wird gewährleistet, dass zur Laufzeit jede Instanz einer Surrogatklasse eindeutig einer Instanz einer Klasse zugeordnet ist, die das entsprechende Interface implementiert. Wird also für die Prüfung einer Spezifikation der konkrete Wert einer Modellvariablen benötigt, so ist er über die Referenzvariable leicht zu ermitteln.

4.3 Laufzeitprüfung von Spezifikationen

In den vorherigen Abschnitten wurde die Übersetzung von JMLjass-Spezifikationen in Methoden, die das gewünschte Verhalten zur Laufzeit überprüfen sollen, detailliert besprochen. Um die Einhaltung der Spezifikationen während der Programmausführung zu untersuchen, müssen die von *Jass* generierten Methoden in den anwendbaren[5] sichtbaren Zuständen der Laufzeit aufgerufen werden. Der folgende Abschnitt wird daher die Umsetzung der Laufzeitprüfung von JMLjass-Spezifikationen genauer betrachten.

4.3.1 Wrappermethoden

Es gibt im Wesentlichen zwei unterschiedliche Ansätze, die Aufrufe der zur Prüfung von Spezifikationen generierten Methoden in den Quellcode zu integrieren:

1. Direktes Einfügen der Methodenaufrufe an den passenden Stellen im Methodenrumpf.

2. Einführen von *Wrappermethoden*, die zunächst die spezifikationsprüfenden Methoden aufrufen und dann den ursprünglichen Methodenrumpf in einer eigenen, internen Methode ausführen.

Der erste Ansatz hat dabei mehrere schwerwiegende Nachteile: Zum Einen ist die Bestimmung der Stellen, an denen die benötigten Methodenaufrufe eingefügt werden sollen, schon aufgrund der Unterscheidung zwischen normaler Terminierung und Terminierung durch Werfen einer Exception nicht ohne Weiteres möglich; zum Anderen erschwert dieses Vorgehen den Umgang mit Rückgabewerten in Nachbedingungen erheblich, da jedes `return`-Statement in jedem beliebigen Pfad der Methodenausführung als möglicher Terminierungspunkt in Betracht gezogen und die Nachbedingung an dieser Stelle auf ihre Gültigkeit hin untersucht werden müsste.

Jass verfolgt daher den Ansatz der *Wrappermethoden*, der insbesondere in [Cli01, Abschnitt 4.1 und 4.3] beschrieben wird. Dabei wird für jede Methode einer Klasse[6] zunächst der Methodenrumpf in eine interne, nur lokal sichtbare Methode ausgelagert und anschließend eine Wrappermethode derselben Signatur eingeführt, die die ursprüngliche Methode ersetzt. Im Folgenden ist dieser Vorgang für eine allgemeine Methode `m` mit dem Rückgabetyp `T` dargestellt:

[5] Die Anwendbarkeit bezieht sich in diesem Kontext auf die Art der verwendeten Spezifikationselemente und die Modifikatoren von Klassen und Methoden.

[6] Es wurde bereits in Abschnitt 4.1 darauf hingewiesen , dass die Behandlung von Klassen und Interfaces im Wesentlichen die selbe ist; daher wird im Folgenden nur die Rede von Methoden in Klassen sein.

Definition 4.1 (Wrappermethode)

Eine Methode

```
// Die ursprüngliche Methode
public T m(Object ... args) {
  < Methodenrumpf >
}
```

wird zur Übersetzungszeit ersetzt durch

```
// Die interne Methode
private T jml_internal$m$(Object ... args) {
  < Methodenrumpf >
}

// Die Wrappermethode
public T m(Object ... args) {

  < Prüfung des Pre-States >

  try {

    T jml_result = jml_internal$m$(args);

    < Prüfung des normalen Post-States >

    return jml_result;
  }
  catch (Exception jml_exc) {

    < Prüfung des Post-States nach Exception >

    throw jml_exc;
  }
}
```

Diese Vorgehensweise vermeidet die Schwächen der direkten Codeeinsetzung, da

- die Stellen, an denen die zur Laufzeitprüfung der Spezifikationen benötigten Methodenaufrufe eingefügt werden müssen, klar definiert sind.
- der Methodenrumpf unmodifiziert ausgeführt wird.
- der Rückgabewert einer Methodenausführung einfach zu referenzieren ist.
- der Umgang mit Terminierung durch Werfen einer Exception durch den umgebenden `try-catch`-Block erheblich vereinfacht wird.

Die folgende Abbildung stellt den Ablauf der geschachtelten Methodenaufrufe während der Programmausführung dar:

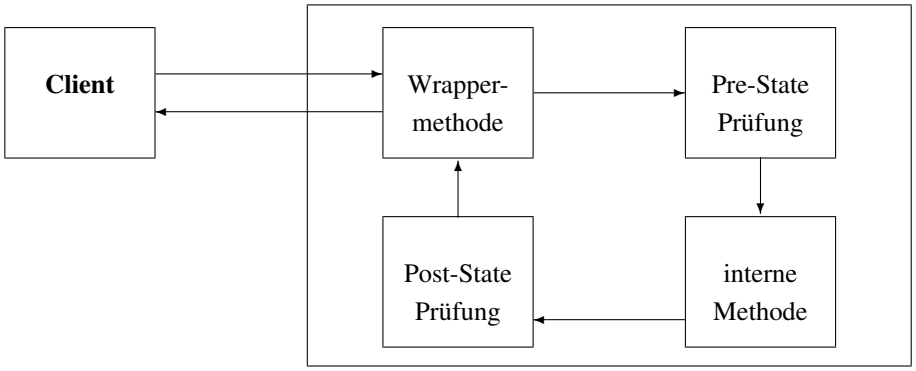


Abbildung 4.3: Schema des Kontrollflusses zur Laufzeit bei der Verwendung von Wrappermethoden und internen Hilfsmethoden.

Zur Laufzeit ruft ein Client also nicht die ursprüngliche Methode auf, sondern die entsprechende Wrappermethode, die daraufhin die Prüfung der spezifizierten Eigenschaften anstößt. Dabei übernimmt die Wrappermethode auch die Aufgaben der externen Kommunikation mit anderen Komponenten, was insbesondere das Zurückliefern des Rückgabewerts und die Weiterleitung eventuell geworfener Exceptions umfasst. Dazu werden zwei reservierte Variablen verwendet: Die Variable `jml_result` dient der Speicherung des Rückgabewerts und wird unter anderem in der Methode zur Prüfung der normalen Nachbedingung benötigt, und die Variable `jml_exc` referenziert eine eventuell gefangene Exception, die der Methode zur Prüfung der Nachbedingung als Parameter übergeben und anschließend nach außen propagiert wird[7].

Zu bemerken ist, dass eventuell definierte `\old`-Ausdrücke noch vor der Prüfung des Pre-States evaluiert und in entsprechenden Variablen gespeichert werden. Details dazu finden sich im Abschnitt 4.6 in diesem Kapitel.

4.3.2 Aktionen im Pre-State

Im Pre-State der Ausführung einer Wrappermethode, der unmittelbar vor dem Aufruf der durch den Präfix `jml_internal` gekennzeichneten internen Methode erreicht wird und im Wesentlichen dem Pre-State der ursprünglichen Methode entspricht[8], werden folgende Aktionen durchgeführt:

[7] Siehe dazu auch Abschnitt 3.4.

[8] Unterschiede zwischen dem Pre-State der Wrappermethode und dem Pre-State der ursprünglichen Methode bestehen insbesondere bezüglich des Laufzeitstacks. Aus der Sicht des durch Variablenwerte definierten Zustands einer Programmausführung sind sie jedoch identisch.

- Prüfung der statischen und instanzgebundenen Invarianten durch Aufrufen der entsprechenden Methoden.
- Prüfung der lokalen und geerbten Vorbedingungen durch Aufrufen der zugehörigen Methoden, und Initialisierung der entsprechenden Variablen, um die Ergebnisse bei der Prüfung der Nachbedingungen verwenden zu können.
- Auswertung eventueller `\old`-Ausdrücke und Initialisierung der zugehörigen Variablen.

Dabei müssen natürlich die Modifikatoren der jeweiligen Methode berücksichtigt werden: In statischen Methoden dürfen auch nur statische Konstrukte überprüft werden, und in als `helper` gekennzeichneten Methoden entfällt die Prüfung der Invarianten.
Die Verletzung einer Spezifikation im Pre-State wird dem Benutzer durch Werfen einer entsprechenden Exception signalisiert, deren Typ die Art des verletzten Spezifikationskonstrukts widerspiegelt[9].

4.3.3 Aktionen im Post-State

Bei der Betrachtung des Post-States muss zunächst zwischen normaler Terminierung und Terminierung durch Werfen einer Exception unterschieden werden. Im ersten Fall wird der Post-State unmittelbar nach dem normalen Rücksprung aus der internen Hilfsmethode erreicht, im zweiten Fall unmittelbar nach dem Fangen der Exception, die zur Beendung der Ausführung der internen Methode geführt hat. Folgende Aktionen werden im Post-State durchgeführt:

- Prüfung der statischen und instanzgebundenen Invarianten und Constraints durch Aufrufen der entsprechenden Methoden.
- Prüfung der lokalen und geerbten Nachbedingungen durch Aufrufen der zugehörigen Methoden abhängig von der Art des Terminierung.
- Zurückliefern des Ergebnisses der Berechung der internen Hilfsmethode, falls diese normal terminiert und einen von `void` verschiedenen Rückgabetyp besitzt.
- Propagieren einer eventuell gefangenen Exception, um den normalen Kontrollfluss der Programmausführung aufrecht zu erhalten.

Auch hier können die Modifikatoren wiederum die anwendbaren Spezifikationselemente einschränken. Verletzungen von Spezifikationen im Post-State werden dem Benutzer analog zum Pre-State durch Werfen einer entsprechenden Exception signalisiert, deren Typ die Art des verletzten Spezifikationskonstrukts widerspiegelt.

[9] Siehe dazu auch Abschnitt 3.4.

4.4 Musteranalyse für Quantoren

Die Übersetzung der JML-spezifischen `\forall`- und `\exists`-Quantoren erfordert, wie in Abschnitt refJMLjass:Sprachumfang bereits erwähnt, die Bestimmung eines konkreten Wertebereichs, über den quantifiziert wird. *Jass* implementiert dazu ein Verfahren, das *statische Musteranalyse* genannt wird. Dabei wird die Quantifizierung zur Übersetzungszeit nach folgenden Kriterien analysiert:

- Falls über einen primitiven, ganzzahlig-numerischen Typen oder eine äquivalente Wrapperklasse quantifiziert wird, so wird das einschänkende Prädikat auf eine Ober- bzw. Untergrenze hin untersucht. Ist kein Prädikat gegeben, so wird über den natürlichen Wertebereich des entsprechenden Typs quantifiziert.
- Falls das einschränkende Prädikat auf einer Objektinstanz des Typs `org.jmlspecs.models.JMLCollection` oder `java.util.Collection` beruht, so ergibt sich der relevante Wertebereich durch Iteration über alle in der Collection referenzierten Objektinstanzen.
- Falls der Typ der quantifizierten Variablen weder dem ersten noch dem zweiten Fall zuzuordnen ist, kann der Wertebereich nicht bestimmt werden. Dann muss die Quantifizierung so durch `true` oder `false` ersetzt werden, dass sie sich in ihrem jeweiligen Kontext neutral verhält.

Die Laufzeitprüfung des quantifizierten Ausdrucks ist wiederum vom Typ der quantifizierten Variablen abhängig:

- Falls über einen primitiven, ganzzahlig-numerischen Typen oder eine äquivalente Wrapperklasse quantifiziert wird, so wird die Quantifizierung durch eine einfache `for`-Schleife mit entsprechend initialisierten Variablen geprüft.
- Falls eine Objektinstanz eines Subtyps von `java.util.Collection` oder `org.jmlspecs.models.JMLCollection` im Prädikat identifiziert wurde, kann der Iterationsmachanismus der Klasse `java.util.Iterator` in Kombination mit `while`-Schleife zur Auswertung der Quantifizierung verwendet werden.
- Falls kein Wertebereich bestimmt werden kann, gibt `Jass` eine Warnung an den Benutzer aus und ersetzt die Quantifizierung durch einen kontextneutralen Booleschen Wert.

Zur Anschauung seien die folgende Beispiele gegeben:

Beispiel 4.2 (Numerische Quantifizierung)

Eine Quantifizierung der Form

```
\exists (int i;
         i >= 0 && i < list.size();
         list.get(i) != null);
```

wird zur Laufzeit geprüft durch

```
boolean result = false;

for (int i = 0; i < list.size(); i++) {
  result = result || (list.get(i) != null);
}
```

■

Beispiel 4.3 (Quantifizierung über eine Collection)

Im Folgenden sei `accounts` ein Feld eines Subtyps von `org.jmlspecs.models.JMLCollection`. Eine Quantifizierung der Form

```
\forall (AccountSpec a1,a2;
         accounts.has(a1) && accounts.has(a2);
         a1.equals(a2) && a1.id.equals(a2.id));
```

wird zur Laufzeit geprüft durch

```
boolean result = true;

JMLIterator iterator1 = accounts.iterator();
JMLIterator iterator2 = accounts.iterator();
AccountSpec a1, a2;

while (iterator1.hasNext()) {

  a1 = (AccountSpec)iterator1.next();

  while (iterator2.hasNext) {

    a2 = (AccountSpec)iterator1.next();
    result =   result && (a1.equals(a2)
            && a1.id.equals(a2.id));
  }

  iterator2 = accounts.iterator();
}
```

■

4.5 Unterschiede zum JML RACC

Der *JML Runtime Assertion Checking Compiler*, kurz *JML RACC*, wurde von Yoonsik Cheon in seiner Dissertation [Che03] konzipiert und implementiert. Der *JML RACC* übersetzt JML-Spezifikationen in Java-Quellcode, der die Einhaltung der spezifizierten Eigenschaften zur Laufzeit prüft und dient damit als Vorbild für *Jass*. Allerdings implementiert *Jass* mit den in diesem Kapitel vorgestellten Strategien zum Umgang mit der Laufzeitprüfung von JML-Spezifikationen eine eigene Herangehensweise an mehrere Problematiken, die im Folgenden dargestellt werden soll.

4.5.1 Behandlung von Spezifikationen in Interfaces

Die Verwendung von Surrogatklassen zur Repräsentation des Spezifikationszustandes eines Interfaces und zur Kapselung der zur Laufzeitprüfung von Spezifikationen benötigten Methoden ist der Dissertation von Yoonsik Cheon [Che03] entlehnt. Die dort entworfene Vorgehensweise unterscheidet sich aber insbesondere bezüglich der Zuordnung der Instanzen implementierender Klassen und der entsprechenden Surrogatklassen von der Implementierung in *Jass*.

Der *JML RACC* verfolgt eine *globale* Zuordnungsstrategie, bei der mittels Hashtabellen zur Laufzeit Klasseninstanzen den entsprechenden Instanzen von Surrogatklassen zugeordnet und über den Reflektionsmechanismus von Java zugreifbar gemacht werden. Eine Surrogatklasse wird erst dann instanziiert und einer Klasseninstanz zugeordnet, wenn die entsprechende Klasseninstanz das erste Mal geerbte Spezifikationsanteile prüfen muss. Notwendig wird diese Art der Zuordnung insbesondere auch durch die Unterstützung der in MultiJava realisierten und in [Che03] beschriebenen Spracherweiterung von Java wie offenen Klassen und des multiplen Dispatchen von Methodenaufrufen. Die Zuordnung von Klasseninstanzen und Surrogatklasseninstanzen erfolgt zur Laufzeit durch spezielle Methoden, die vom *JML RACC* für jede Klasse erzeugt und in [Che03, Abschnitt 6.5] detailliert besprochen werden. Zum besseren Verständnis ist in Tabelle 4.1 eine vereinfachte Darstellung gegeben:

Quelltext 4.1

```
/* HashMap, die die Surrogatklasseninstanzen verwaltet. */
public java.util.Map rac$surrogates;

/* Methode zur Auflösung einer Surrogatklasseninstanz. */
public Object rac$getSurrogate(String name) {

  if (rac$surrogates == null) {
    rac$surrogates = new java.util.HashMap();
    return null;
  }
  return rac$surrogates.get(name);
}

/* Methode zum Speichern einer Surrogatklasseninstanz. */
public void rac$setSurrogate(String name,
                             JMLSurrogate obj) {

  if (rac$surrogates == null) {
    rac$surrogates = new java.util.HashMap();
  }
  rac$surrogates.put(name, obj);
}

/* Methode, die zur Laufzeit für eine Klasseninstanz
 * die entsprechende Instanz einer Surrogatklasse
 * eines implementierten Interfaces auflöst.
 */
public static Object rac$receiver(String name,
                                  Object forObj) {

  // Auflösung der Surrogatklasseninstanz anhand des
  // Namens der Surrogatklasse (name) und der konkreten
  // Klasseninstanz selbst (forObj)
}
```

Tabelle 4.1: Verwaltung von Surrogatklasseninstanzen im *JML RACC*

Für die Auflösung der Surrogatklasseninstanzen in der Methode `rac$receiver` spielt der `Class.forName`-Mechanismus eine wichtige Rolle, was insbesondere wegen der *type erasure* in Java für den Umgang mit generischen Typen eine starke Einschränkung bedeutet. *Type erasure* und die für generische Typen daraus folgenden Konsequenzen werden in [BGJS05, Abschnitt 4.5 und 4.6] besprochen und sollen an dieser Stelle aus Gründen des Umfangs nicht weiter vertieft werden.

Jass realisiert im Gegensatz zur globalen Verwaltung des *JML RACC* eine

bereits in den vorherigen Abschnitten besprochene *lokale* Verwaltungsstrategie. Danach wird zur Laufzeit bei der Instanziierung einer Klasse für jedes Interface, von dem die Klasse Spezifikationsanteile erbt, eine Instanz der entsprechenden Surrogatklasse erzeugt und durch eine lokale Surrogatvariable referenziert. Umgekehrt erhält die Surrogatklasseninstanz durch die Referenzvariable eine Referenz auf die jeweils zugeordnete Klasseninstanz. Im abschließenden Übersetzungsbeispiel in Abschnitt 4.6 lassen sich bei Bedarf weitere Details nachvollziehen.

4.5.2 Semantische Unterschiede

JML weist einige semantische Unterschiede zur Programmiersprache Java auf. Bereits in Abschnitt 4.1 wurde etwa darauf hingewiesen, dass Interfaces in JML nicht wie in Java zustandslos sind, sondern einen durch Spezifikationsvariablen ausdrückbaren Zustand besitzen können. Eine vollständige Übersicht über die Unterschiede auf semantischer Ebene findet sich in [Cli01, Abschnitt 1.3]; an dieser Stelle seien die wichtigsten nur kurz aufgeführt:

- JML erlaubt die *multiple Vererbung* von Spezifikationen, während Java nur einfache Vererbung kennt.
- Durch die Verwendung spezieller Modifikatoren wie etwa `spec_public` können einzelne Deklarationen aus Sicht der Spezifikation eine *größere Zugreifbarkeit* erhalten als durch ihren Java-Modifikator.
- Durch die Verwendung von Java-Ausdrücken in Spezifikationen kann es zu Situationen kommen, in denen einzelne Ausdrücke aus der Sicht von JML *undefiniert* sind.

Insbesondere der letzte Punkt spielt für diese Arbeit eine wichtige Rolle. Das Problem der Undefiniertheit von Spezifikationsausdrücken lässt sich an folgendem Beispiel nachvollziehen:

Beispiel 4.4 (Undefiniertheit von Ausdrücken)

```
//@ invariant list.size() > 0 && list != null;
```

Dabei soll `list` vom Typ `java.util.Collection` sein. ■

Ist in Beispiel 4.4 nun das Feld `list` eine `null`-Referenz, so würde beim Auswerten der Invarianten nach der Semantik in Java eine `NullPointerException` geworfen werden. Um mit solchen Situationen adäquat umgehen zu können, implementiert der *JML RACC* einen Ansatz, der in [Cli01] als *lokale, kontextuelle Interpretation* bezeichnet wird. Dabei wird der kleinste, den undefinierten Teilausdruck umfassende Ausdruck in einem *positiven Kontext* als `true` und in einem

negativen Kontext als `false` interpretiert.

Beispiel 4.5 (Positiver bzw. negativer Kontext)

Für einen Booleschen Ausdruck `E` ist etwa `E` selbst ein positiver Kontext und `!(E)` ein negativer Kontext.

■

Für das Beispiel 4.4 bedeutet dies, dass beim Auftreten eines Fehlers während der Auswertung des Ausdrucks `list.size() > 0` der kleinste umfassende Ausdruck - in diesem Fall `list.size() > 0` selbst - zu `true` evaluiert wird und die Einhaltung der Invarianten damit von der Auswertung des zweiten Teilausdrucks der Konjunktion abhängt. Die generelle Strategie der kontextuellen Interpretation ist es also, undefinierte Ausdrücke so zu ersetzen, dass das den Ausdruck umgebende Spezifikationskonstrukt falsifiziert werden kann.

Dieser Ansatz bringt aber mehrere Probleme mit sich, wie das folgende Beispiel zeigt:

Beispiel 4.6

```
//@ invariant list.size() > 0 || true;
```

■

Es sei hier `list` wiederum eine `null`-Referenz. Dann ergibt die Auswertung der Invarianten insgesamt `true`, obwohl bei ihrer Auswertung ein Fehler aufgetreten ist. Die geworfene `NullPointerException` wird während der Auswertung gefangen und unabsichtlich vor dem Benutzer versteckt. Neben der Tatsache, dass so Fehler in Spezifikationen oder dem Quelltext verschleiert werden, widerspricht dieses Verhalten auch der Erwartung der Benutzer, wie Patrice Chalin in einer Befragung [Cha05] von über zweihundert aktiven Entwicklern aus Industrie und Forschung herausfand. Etwa 84% der Befragten erwarteten nach [Cha05, Abschnitt 5.1], dass in einer Situation wie in Beispiel 4.6 zur Laufzeit eine Exception geworfen wird.

Um diesen Anforderungen Rechnung zu tragen, implementiert *Jass* die „normale“ Java-Semantik für Ausdrücke auch in Spezifikationen, die neben der größeren Transparenz der Spezifikationen bezüglich auftretenden Fehlern auch den Vorteil hat, die aufwändige Bestimmung des kleinsten umfassenden Ausdrucks für einen undefinierten Teilausdruck zu umgehen, was sich positiv auf die Performanz der Laufzeitprüfungen auswirken kann.

4.6 Beispielhafte Übersetzung

Nachdem in den vorangegangenen Abschnitten umfassend auf den Übersetzungsvorgang des JMLjass-Precompilers *Jass* eingangen wurde, soll nun ein Beispiel für einen typischen Übersetzungslauf gegeben werden. Dazu sei eine generische Klasse `List` betrachtet, die eine Liste von Objekten eines beliebigen Typs `T` speichern und verwalten soll. `List` soll dabei über folgende Methoden verfügen:

- Eine Methode `add` zum Hinzufügen von Elementen.
- Eine Methode `get`, die ein Element der Liste an einem übergebenen Index zurückliefert.
- Eine Methode `size`, die aktuelle Länge der Liste zurück liefert.

Die tatsächliche Implementierung dieser Methoden ist dabei nicht von Interesse, da der Fokus dieses Abschnitts auf der Übersetzung des spezifizierten Verhaltens liegen soll. Die Einträge der Liste sollen intern in einem Array `internalList` vom Typ `T` verwaltet werden, dessen Länge - falls erforderlich - dynamisch angepasst wird:

Quelltext 1 - Die generische Klasse List

```
/**
 * Die generische Klasse List soll eine Liste von
 * Objekten eines beliebigen Typs implementieren.
 */
public class List<T> {

  // Die Liste wird intern durch ein Array dargestellt.
  private T[] internalList;

  // Modellvariable, die die Länge der Liste repräsentiert.
  //@ private model int size;

  //@ represents size <- this.size();

  // Die Liste ist zu Beginn leer.
  //@ initially size == 0;

  // Die Länge der Liste ist stets größer oder gleich Null.
  //@ invariant size >= 0;

  // Die Liste wächst stetig an.
  //@ constraint size >= \old(size);

  ...
```

Quelltext 1 - Fortsetzung

```
                                    ...
    //@ ensures \result == internalList.length;
    public /*@ pure @*/ int size() {
      < Methodenkörper >
    }

    /*@ public normal_behavior
      @   requires _entry != null;
      @   ensures  size == \old(size) + 1;
      @ also
      @ public exceptional_behavior
      @   requires _entry == null;
      @   signals  (NullPointerException npe);
      @*/
    public <T> void add(T _entry) throws NullPointerException {
      < Methodenkörper >
    }

    /*@ public normal_behavior
      @   requires _index >= 0;
      @   ensures  \result.equals(internalList[_index]);
      @ also
      @ public exceptional_behavior
      @   requires _index < 0;
      @   signals  (IllegalIndexException iie);
      @*/
    public <T> T get(int _index) throws IllegalIndexException {
      < Methodenkörper >
    }
}
```

Um das Verhalten der Klasse zur Laufzeit zu beschreiben, werden folgende Spezifikationskonstrukte verwendet:

- Eine Modellvariable `size`, die die Länge der Liste zur Laufzeit durch einen `represents`-Ausdruck abbildet.
- Ein `initially`-Ausdruck, der die Größe der Liste unmittelbar nach der Initialisierung einer neuen Liste spezifiziert.
- Eine Invariante, die die Nichtnegativität der Modellvariablen `size` garantiert,
- Ein History Constraint, der eine Liste als stetig wachsend definiert.
- Jeweils eine Methodenspezifikation für die Methoden `add`, `get` und `size`.

4.6.1 Erzeugung benötigter Variablen

In einem ersten Schritt werden alle `\old`-Ausdrücke identifiziert und entsprechend typisierte Variablen eingeführt, die später das Ergebnis der Auswertung des jeweiligen Ausdrucks speichern werden. Des Weiteren werden für sämtliche Vor- und Nachbedingungen Boolesche Arrays definiert, die insbesondere für die Prüfung der Implikationen der Nachbedingungen benötigt werden:

Quelltext 2 - Erzeugung benötigter Variablen

```
/**
 * Die generische Klasse List soll eine Liste von
 * Objekten eines beliebigen Typs implementieren.
 */
public class List<T> {

  // Die Liste wird intern durch einen Array dargestellt.
  private T[] internalList;

  // Felder, die den Wert der \old-Ausdrücke speichern.
  protected int jml_old$List$int_0;
  protected int jml_old$List$add$T$int_0

  // Felder zur Speicherung der Vorbedingungen.
  private boolean[] jml_precond$public$List$size$;
  private boolean[] jml_precond$public$List$add$T;
  private boolean[] jml_precond$public$List$get$;

  // Felder zur Speicherung der Nachbedingungen.
  private boolean[] jml_postcond$public$List$size$;
  private boolean[] jml_postcond$public$List$add$T;
  private boolean[] jml_postcond$public$List$get$;

                              ...
```

4.6.2 Modellvariablen, Invarianten und Constraints

Nachdem die benötigten Variablen eingeführt wurden, werden die Modellvariablen gemäß dem Übersetzungsmuster in Methoden überführt. Damit sind die Grundlagen für die Überprüfung der restlichen Spezifikationen gelegt. Zunächst werden Initiallies, Invarianten und History Constraints übersetzt:

Quelltext 3 - Modellvariablen, Invarianten und Constraints

```
...
// Evaluierung der Modellvariablen size.
private int jml_evaluate$model_field$List$size$() {

  return internalList.length;
}

// Prüfung der Initially-Ausdrücke.
public boolean jml_check$initially$() {

  if (!(jml_evaluate$model_field$List$size$() == 0))
       throw new JMLInitiallyException(/* message */);

  return true;
}

// Prüfung der Invarianten.
public boolean jml_check$instance_invariants$() {

  if (!(jml_evaluate$model_field$List$size$() >= 0))
       throw new JMLInvariantException(/* message */);

  return true;
}

// Prüfung der History Constraints.
public boolean jml_check$instance_constraints$() {

  if (!(jml_evaluate$model_field$List$size$()
                                    >= jml_old$List$int_0))
       throw new JMLConstraintException(/* message */);

  return true;
}

...
```

Zu bemerken ist, dass die Referenzierungen der Modellvariablen `size` in den Spezifikationen durch Aufrufe der evaluierenden Methode ersetzt wurden.

Die Überprüfung des `initially`-Ausdrucks wird durch Einfügen eines - falls nötig statischen - Initializers realisiert. Statische und instanzgebundene Initializer werden insbesondere in [BGJS05, Abschnitt 8.6 und 8.7] erläutert.

4.6.3 Prüfung von Vorbedingungen

Im vorherigen Schritt wurden die globalen, methodenunabhängigen Spezifikationen übersetzt. Als nächstes werden die Methoden zur Prüfung von Vorbedingungen generiert:

Quelltext 4 - Vorbedingungen

...

```
// Prüfung der Vorbedingung der Methode size.
public boolean[] jml_check$pre_state_public$size$() {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_precond$public$List$size$ = new boolean[1];
  jml_precond$public$List$size$[0] = true;

  assertionFailure = !assertionSuccess
                  && !jml_precond$public$List$size$[0];

  if (assertionFailure)
      throw new JMLPreStateException(/* message */);

  return jml_precond$public$List$size$;
}

// Prüfung der Vorbedingung der Methode add.
public <T> boolean[]
     jml_check$pre_state_public$add$T(T _entry) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_precond$public$List$add$T = new boolean[2];
  jml_precond$public$List$add$T[0] = (_entry != null);
  jml_precond$public$List$add$T[1] = (_entry == null);

  assertionFailure = !assertionSuccess
                  && !(jml_precond$public$List$add$T[0]
                     || jml_precond$public$List$add$T[1]);

  if (assertionFailure)
      throw new JMLPreStateException(/* message */);

  return jml_precond$public$List$add$T;
}
```

...

Quelltext 4 - Fortsetzung

```
                                    ...
// Prüfung der Vorbedingung der Methode get.
public <T> boolean[]
     jml_check$pre_state_public$get$int(int _index) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_precond$public$List$get$int = new boolean[2];
  jml_precond$public$List$get$int[0] = (_index >= 0);
  jml_precond$public$List$get$int[1] = (_index < 0);

  assertionFailure = !assertionSuccess
                  && !(jml_precond$public$List$get$int[0]
                    || jml_precond$public$List$get$int[1]);

  if (assertionFailure)
       throw new JMLPreStateException(/* message */);

  return jml_precond$public$List$get$int;
}

                                    ...
```

Die Vorbedingung jedes Spezifikationsfalls wird in einem separaten Booleschen Feld gespeichert, um während der Auswertung der Nachbedingungen auf das jeweilige Ergebnis zugreifen zu können[10].

4.6.4 Prüfung von Nachbedingungen

Bei der Generierung des Quellcodes zur Überprüfung der Nachbedingungen muss zwischen normaler Terminierung und Terminierung durch Werfen einer Exception unterschieden werden. Dazu werden für Methodenspezifikationen, die mindestens einen `signals`- oder `signals-only`-Ausdruck aufweisen, zwei separate Methoden zur Prüfung der Einhaltung der spezifizierten Bedingungen generiert. In diesen Methoden werden die jeweiligen Nachbedingungen nach dem Muster

$$\neg(\backslash old(P) \Rightarrow Q) \Leftrightarrow (\backslash old(P) \wedge \neg Q)$$

auf eine Verletzung hin überprüft.

Die folgenden Quelltexte zeigen die Nachbedingungen der Methoden `size`, `add` und `get` der Klasse `List`:

[10] Zur Erinnerung: Nachbedingungen werden in der Form \old(P) ⇒ Q ausgewertet.

Quelltext 5 - Normale Nachbedingungen von `size` und `add`

...

```
// Prüfung der Nachbedingung von size.
public boolean[]
     jml_check$post_state_public$size$(int jml_result) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_postcond$public$List$size$ = new boolean[1];
  jml_postcond$public$List$size$[0]
       = (jml_precond$public$List$size$[0]
      && !(jml_result == internalList.length));

  assertionFailure = !assertionSuccess
                   || jml_postcond$public$List$size$[0];

  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  return jml_postcond$public$List$size$;
}

// Prüfung der normalen Nachbedingung von add.
public <T> boolean[]
     jml_check$post_state_public$add$T(T _entry) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_postcond$public$List$add$T = new boolean[2];
  jml_postcond$public$List$add$T[0]
       = (jml_precond$public$List$add$T[0]
      && !(jml_evaluate$model_field$List$size$()
               == jml_old$List$add$T$int_0 + 1));
  jml_postcond$public$List$add$T[1]
       = (jml_precond$public$List$add$T[1] && !(false));

  assertionFailure = !assertionSuccess
                   || jml_postcond$public$List$add$T[0]
                   || jml_postcond$public$List$add$T[1];

  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  return jml_postcond$public$List$add$T;
}
```

...

Quelltext 6 - Nachbedingung bei Exceptions von `add`

```
...
// Prüfung der Nachbedingung bei Exceptions von add.
public <T> boolean[]
     jml_check$post_state_public$add$T(T _entry,
                                       Exception jml_exc) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_postcond$public$List$add$T = new boolean[2];
  jml_postcond$public$List$add$T[0]
       = (jml_precond$public$List$add$T[0]
       && !((jml_exc instanceof Exception
       && !(false)));
  jml_postcond$public$List$add$T[1]
       = (jml_precond$public$List$add$T[1]
       && !((jml_exc instanceof NullPointerException
       && !(true))));

  assertionFailure = !assertionSuccess
                   || jml_postcond$public$List$add$T[0]
                   || jml_postcond$public$List$add$T[1];

  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  return jml_postcond$public$List$add$T;
}

...
```

Die Signaturen der jeweils für die Methoden `add` und `get` generierten Methoden zur Prüfung der Nachbedingung bei normaler Terminierung und bei Terminierung durch Werfen einer Exception unterscheiden sich lediglich durch einen Parameter, nämlich der während der Ausführung des entsprechenden Methodenrumpfes geworfenen Exception. Diese Art der Unterscheidung ist jedoch ausreichend, da sich normale Terminierung und Terminierung durch Auftreten einer Exception wechselseitig ausschließen.

Quelltext 7 - Normale Nachbedingung von `get`

```
                                    ...
  // Prüfung der normalen Nachbedingung von get.
  public <T> boolean[]
       jml_check$post_state_public$get$int(int _index,
                                           T jml_result) {

    boolean assertionSuccess = false;
    boolean assertionFailure = true;

    jml_postcond$public$List$get$int = new boolean[2];
    jml_postcond$public$List$get$int[0]
         = (jml_precond$public$List$get$int[0]
        && !(jml_result.equals(internalList[_index])));
    jml_postcond$public$List$get$int[1]
         = (jml_precond$public$List$get$int[1] && !(false));

    assertionFailure = !assertionSuccess
                     || jml_postcond$public$List$get$int[0]
                     || jml_postcond$public$List$get$int[1];

    if (assertionFailure)
         throw new JMLPostStateException(/* message */);

    return jml_postcond$public$List$get$int;
  }

                                    ...
```

Der in Quelltext 7 gegebene Ausdruck

```
jml_postcond$public$List$get$int[1]
         = (jml_precond$public$List$get$int[1] && !(false));
```

evaluiert genau dann zu *wahr*, wenn die Vorbedingung der *Terminierung durch Werfen einer Exception* erfüllt ist. Diese auf den ersten Blick unnötige Überprüfung erfolgt jedoch aus gutem Grund: Da die hier angegebene Methode die Nachbedingung der *normalen* Terminierung überprüft, wäre eine Erfüllung des gegebenen Ausdrucks eine Verletzung des wechselseitigen Ausschlusses normaler und außergewöhnlicher Terminierung, die dem Benutzer signalisiert werden muss.

Quelltext 8 - Nachbedingung von `get`

```
...
// Prüfung der Nachbedingung von get.
public <T> boolean[]
     jml_check$post_state_public$get$int(int _index,
                                         Exception jml_exc) {

  boolean assertionSuccess = false;
  boolean assertionFailure = true;

  jml_postcond$public$List$get$int = new boolean[2];
  jml_postcond$public$List$get$int[0]
       = (jml_precond$public$List$get$int[0]
       && !((jml_exc instanceof Exception
       && !(false)));
  jml_postcond$public$List$get$int[1]
       = (jml_precond$public$List$get$int[1]
       && !((jml_exc instanceof IllegalIndexException
       && !(true))));

  assertionFailure = !assertionSuccess
                   || jml_postcond$public$List$get$int[0]
                   || jml_postcond$public$List$get$int[1];

  if (assertionFailure)
       throw new JMLPostStateException(/* message */);

  return jml_postcond$public$List$get$int;
}

...
```

4.6.5 Interne Methoden und Wrappermethoden

Um Rückgabewerte und während der Ausführung von Methodenrümpfen geworfene Exceptions möglichst einfach behandeln zu können, generiert *Jass* für jede Methode einer übersetzten Klasse eine intern verwendete Hilfsmethode, die nur den jeweiligen Methodenrumpf umfasst. Der Aufruf der Hilfsmethoden erfolgt dann durch die entsprechende Wrappermethode[11]. Im Folgenden sind die für die Methoden `size`, `add`, und `get` generierten internen Methoden und die zugehörigen Wrappermethoden aufgeführt.

[11] Siehe dazu Abschnitt 4.3.

Quelltext 9 - Hilfsmethoden für `size`

```
...
// Interne Methode, die von der entsprechenden
// Wrapper-Methode aufgerufen wird.
private int jml_internal$size$() {

  < Methodenkörper >
}

// Wrapper-Methode, die die notwendigen Überprüfungen
// vornimmt und die entsprechende interne Methode aufruft.
public int size() {

  jml_old$List$int_0
       = jml_evaluate$model_field$List$size$();

  // Prüfe Invarianten.
  jml_check$instance_invariants$();

  // Prüfe Vorbedingung und speichere Ergebnis.
  jml_precond$public$List$size$
       = jml_check$pre_state_public$size$();

  // Aufruf der internen Methode
  int jml_result = jml_internal$size$();

  // Prüfe Invarianten und Constraints.
  jml_check$instance_invariants$();
  jml_check$instance_constraints$();

  // Prüfe und speichere Nachbedingung
  jml_postcond$public$List$size$
       = jml_check$post_state_public$size$();
}

...
```

Quelltext 10 - Hilfsmethoden für `add`

```
...
// Interne Methode, die von der entsprechenden
// Wrapper-Methode aufgerufen wird.
private <T> void jml_internal$add$T(T _entry) {

  < Methodenkörper >
}

// Wrapper-Methode, die die notwendigen Überprüfungen
// vornimmt und die entsprechende interne Methode aufruft.
public <T> int add(T _entry) {

  // Evaluiere benötigten \old-Ausdruck
  jml_old$List$add$T$int_0
       = jml_evaluate$model_field$List$size$();

  // Prüfe Invarianten.
  jml_check$instance_invariants$();

  // Prüfe Vorbedingung und speichere Ergebnis.
  jml_precond$public$List$add$T
       = jml_check$pre_state_public$add$T(_entry);

  try {

    // Aufruf der internen Methode
    jml_internal$add$T(_entry);

    // Prüfe Invarianten und Constraints.
    jml_check$instance_invariants$();
    jml_check$instance_constraints$();

    jml_postcond$public$List$add$T
       = jml_check$post_state_public$List$add$T(_entry);
  }
  catch(Exception jml_exc) {
    // Prüfe Invarianten und Constraints.
    jml_check$instance_invariants$();
    jml_check$instance_constraints$();

    jml_postcond$public$List$add$T
       = jml_check$post_state_public$List$add$T(_entry,
                                                jml_exc);
  }
}

...
```

Quelltext 11 - Hilfsmethoden für `get`

```
                                    ...
  // Interne Methode, die von der entsprechenden
  // Wrapper-Methode aufgerufen wird.
  private <T> T jml_internal$get$int(int _index) {

    < Methodenkörper >
  }

  // Wrapper-Methode, die die notwendigen Überprüfungen
  // vornimmt und die entsprechende interne Methode aufruft.
  public <T> T get(int _index) {

    // Prüfe Invarianten.
    jml_check$instance_invariants$();

    // Prüfe Vorbedingung und speichere Ergebnis.
    jml_precond$public$List$get$int
         = jml_check$pre_state_public$List$get$int(_index);

    try {
      // Aufruf der internen Methode
        T jml_result = jml_internal$get$int(_index);

      // Prüfe Invarianten und Constraints.
      jml_check$instance_invariants$();
      jml_check$instance_constraints$();

      jml_postcond$public$List$add$T
         = jml_check$post_state_public$List$get$int(_index);
    }
    catch(Exception jml_exc) {
      // Prüfe Invariante und Constraints.
      jml_check$instance_invariants$();
      jml_check$instance_constraints$();

      jml_postcond$public$List$get$int
         = jml_check$post_state_public$List$get$int(_index,
                                                    jml_exc);
    }
  }
```

Damit ist die Übersetzung des Beispiels abgeschlossen.

Kapitel 5

Fallstudie

5.1 Bemerkungen

In der ursprünglichen Konzeption dieser Arbeit sollte sich dieses Kapitel mit dem Leistungsvergleich zwischen *Jass* und dem JML RACC beschäftigen. Dazu sollten die Spezifikationen der im Projekt „ForMoos“ entstandenen Fallstudie *Automatic Teller Machine* (kurz: ATM) von beiden Runtime-Checking-Werkzeugen übersetzt und anschließend sowohl bezüglich des erzeugten Quelltextes als auch hinsichtlich des Laufzeitverhaltens untersucht werden. Ziel dieses Vorgehens war es einerseits, die prinzipielle Funktionalität des in dieser Arbeit entwickelten Precompilers nachzuweisen, und andererseits zu zeigen, dass die alternativen Übersetzungsstrategien in *Jass* die Probleme des JML RACC vermeiden. Aufgrund erheblicher softwaretechnischer Schwierigkeiten, die sich insbesondere aus dem Umstand ergeben, dass für die Übersetzung der Klassen im ATM-Beispiel mehrere JML-eigene Spezifikationsklassen benötigt werden, die *Jass* in der momentanen Implementierung nicht fehlerfrei verarbeiten kann, ließ sich dies in der gegebenen Zeit jedoch nicht realisieren. Die aufgetretenen Probleme und der teilweise implementierte Ansatz, mit diesen umzugehen, werden in den Abschnitten 6.1 und 6.2 ausführlich diskutiert.

5.2 Fallbeispiel: Ringpuffer

In diesem Abschnitt soll es vorrangig darum gehen, die prinzipielle Funktionalität von *Jass* anhand eines kurzen Beispiels nachzuweisen. Um dies zu erreichen, wurde das bereits von Detlef Bartetzko in [Bar99] verwendete Beispiel eines Ringpuffers adaptiert, um einige JML-Spezifikationen erweitert und mittels *Jass* in compilierbaren Java-Quellcode übersetzt. Im Folgenden werden die zugrundeliegenden Quelltexte vorgestellt und die speziellen Schwierigkeiten bei der Übersetzung in laufzeitprüfenden Quellcode herausgestellt.

Ein Ringpuffer ist eine Datenstruktur mit einer festen Anzahl von Pufferplätzen. Über zwei Zeiger `in` und `out` lassen sich neue Einträge einfügen bzw. bereits gespeicherte Einträge auslesen. Die folgende Abbildung aus [Bar99] stellt dies anschaulich dar:

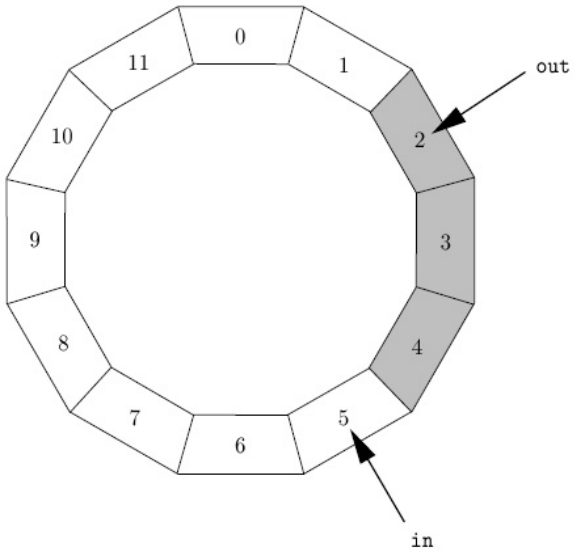


Abbildung 5.1: Ringpuffer der Größe 12. Grau unterlegte Felder stellen bereits belegte Pufferplätze dar.

Der beispielhaft implementierte Ringpuffer verfügt über die folgenden Methoden:

- `size` liefert die maximale Kapazität des Puffers zurück.
- `empty` prüft, ob der Puffer leer ist.
- `full` prüft, ob der Puffer voll ist.
- `add` fügt ein Objekt an der nächstmöglichen Stelle in den Puffer ein.
- `getNext` liefert das durch den Zeiger `out` referenzierte Objekt zurück.
- `remove` löscht das durch den Zeiger `out` referenzierte Objekt.
- `contains` prüft, ob das übergebene Objekt im Puffer gespeichert ist.

Um die Vererbung von Spezifikationen testen zu können, wurde das Interface `BufferSpec` definiert, das neben den aufgezählten Methoden auch Teile der JML-Spezifikation enthält.

Quelltext 1 BufferSpec

```
package jass.examples.jml;

/*
 * This is a buffer example.
 */
public interface BufferSpec {

  // model field representing the buffer
  //@ public instance model BufferSpec the_buffer;

  // model field representing the next index
  // where a new element should be stored
  //@ public instance model int store_index;

  // model field representing the index where
  // the last stored element should be retrieved
  //@ public instance model int extract_index;

  // buffer's size is never negative
  //@ public instance invariant the_buffer.size() >= 0;

  /*@ public instance invariant
    @      0 <= store_index - extract_index
    @   && store_index - extract_index <= the_buffer.size();
    @*/

  // buffer's elements are non_null
  /*@ public instance invariant
    @                  (\forall int i; i >= 0
    @                          && i < the_buffer.size();
    @                          the_buffer.getNext() != null);
    @*/

  // state after initialization
  /*@ public instance initially
    @                  store_index == 0 && extract_index == 0;
    @*/

  /* Returns the buffer's maximum capacity */
  //@ ensures \result == the_buffer.size();
  public /*@ pure @*/ int size();

  /* Returns if buffer is empty */
  //@ ensures \result <==> (store_index == extract_index);
  public /*@ pure @*/ boolean empty();

                                ...
```

Quelltext 2 BufferSpec (Fortsetzung)

```
                                    ...
  /*Returns true if buffer is full */
  /*@ public normal_behavior
    @ ensures \result <==>
    @         (store_index - extract_index == the_buffer.size());
    @*/
  public /*@ pure @*/ boolean full();

  /* Returns next buffered Object */
  //@ ensures \result != null;
  public /*@ pure @*/Object getNext();

  /* Adds an non-null object to buffer and signals violation
   * if buffer is full or object is null. */
  /*@ public normal_behavior
    @ requires o != null && !full();
    @ also
    @ public exceptional_behavior
    @ requires o == null || full();
    @ signals (Exception) true;
  public void add(Object o) throws Exception;

  /* Removes next buffered object */
  //@ requires !empty();
  public Object remove();

  /* Returns true if buffer contains the given element */
  //@ requires o != null;
  public boolean contains(Object o);
}
```

Bei der Definition des Verhaltens eines Ringpuffers und der Auswahl der dazu verwendeten Sprachkonstrukte wurde darauf geachtet, eine möglichst große Bandbreite von Spezifikationsfällen abzudecken. Verwendet wurden:

- Modellvariablen in Interfaces, deren Abstraktionsfunktionen erst in der implementierenden Klasse festgelegt werden, um die korrekte Zuordnung von Klasseninstanzen und Surrogatvariablen bezüglich der Auflösung der Abstraktionsfunktion zu untersuchen.
- Invarianten, die die eingeführten Modellvariablen referenzieren und in jedem sichtbaren Zustand der Programmausführung gelten müssen.
- Eine universelle Quantifizierung über einen primitiv-numerischen Datentyp, um die statische Musteranalyse zu prüfen.

- Ein `initially`-Ausdruck, der den Zustand der Pufferzeiger nach der unmittelbaren Erzeugung eines Pufferobjekts definiert.
- Mehrere Lightweightspezifikationen mit unterschiedlichen Vor- und Nachbedingungen.
- Eine Normal-Behavior- und eine Exceptional-Behavior-Spezifikation, um das Verhalten insbesondere beim Auftreten einer Ausnahmesituation zu untersuchen.
- Mehrere Methoden, die als `pure` deklariert und in Methodenspezifikationen aufgerufen werden sollen.

Das so definierte Interface wird von der Klasse `Buffer` implementiert, die damit das spezifizierte Verhalten erbt und `BufferSpec` verfeinert:

Quelltext 3 Buffer

```
package jass.examples.jml;

/*
 * This is a buffer example.
 */
public class Buffer implements BufferSpec {

  /* Pointer to next empty entry and next buffered object. */
  protected int in, out;

  // Abstraction function for model fields
  //@ represents the_buffer <- this;
  //@ represents store_index <- index;
  //@ represents extract_index <- out;

  /* Internal representation as array */
  protected Object[] buf;

  /* Creates a buffer with the given capacity  */
  //@ requires capacity > 0;
  //@ ensures the_buffer.size() == capacity;
  public Buffer (int capacity) {
    buf = new Object[capacity];
  }

  public /*@ pure @*/ int size() {
    return buf.length;
  }

                                    ...
```

Quelltext 4 Buffer (Fortsetzung)

```
                                    ...
  public /*@ pure @*/ boolean empty() {
    return in - out == 0;
  }

  public /*@ pure @*/ boolean full() {
    return in - out == buf.length;
  }

  public Object getNext() {
    return buf[out];
  }

  //@ also
  //@ ensures \old(store_index) == store_index - 1;
  public void add(Object o) {
    buf[in % buf.length] = o;
    in++;
  }

  //@ also
  //@ ensures \old(store_index) == store_index - 1;
  public Object remove() {
    Object o = buf[out % buf.length];
    out++;
    return o;
  }

  public boolean contains(Object o) {
    boolean found = false;
    for (int i = 0; i < buf.length; i++)
      if (buf[i].equals(o)) found = true;
    return found;
  }
}
```

Die Klasse `Buffer` definiert:

- Abstraktionsfunktionen für die von `BufferSpec` geerbten Modellvariablen.
- Einen Konstruktor mit einer lokalen Vor- bzw. Nachbedingung, um die Umsetzung von Spezifikationen für Konstruktoren zu prüfen.
- Mehrere lokale Methodenspezifikationen, um die Erweiterung geerbter Spezifikationen zu untersuchen.

- Mehrere `\old`-Ausdrücke in lokalen Nachbedingungen, um die korrekte Auswertung dieser Konstrukte zu prüfen.

Die Übersetzung der Spezifikationen in compilierbaren Java-Code, der die Einhaltung des definierten Verhaltens zur Laufzeit prüft, erfolgte durch *Jass* problemlos analog zum Beispiel in Abschnitt 4.6; die Übersetzung des generierten Codes in Java-Bytecode wurde durch den Java-Compiler `javac` der Firma Sun geleistet. Eine Laufzeitprüfung der übersetzten Klasse erfolgte in einer lokalen Testklasse durch Erzeugen einer Instanz der Klasse `Buffer` und Ausführen einer vorgegebenen Abfolge von Methodenaufrufen.

Abschließend ist festzuhalten, dass ein Laufzeitvergleich des einerseits durch `Jass` und `javac` und andererseits durch den JML RACC erzeugten Codes keine relevanten Unterschiede gezeigt hat. Aufgrund der geringen Größe des Beispiels war dies jedoch zu erwarten, zumal bei der Ausführung des durch den JML RACC erzeugten Codes auch noch die im Lieferumfang der aktuellen JML Distribution 5.3 enthaltenen Spezifikationen der Klassen des `java.lang`-Pakets in die Laufzeitprüfung mit einflossen.
Ein Blick auf den Umfang des generierten Codes ergibt Folgendes:

	Original	**Jass**	**JML RACC**
`Buffer`	65	1070	2493
`BufferSpec`	58	414	1855

Tabelle 5.1: Vergleich des Umfang des generierten Codes in Codezeilen

Der deutlich geringere Umfang des durch *Jass* generierten Codes ist natürlich auch auf den eingeschränkteren Sprachumfang von JMLjass gegenüber JML zurückzuführen; insbesondere bei umfangreichen Spezifikationen kann sich dieser Vorteil aber auch für die Laufzeit positiv auswirken. Weitere Ergebnisse und Schlussfolgerungen werden im anschließenden Kapitel besprochen.

Kapitel 6

Ergebnisse und Diskussion

Nachdem im vorherigen Kapitel bereits auf einige Schwierigkeiten bei der Umsetzung des projektierten Precompilers eingegangen wurde, werden diese im Folgenden noch einmal vertiefend dargestellt. Außerdem soll ein möglicher Lösungsansatz aufgezeigt werden, der den angesprochenen Problemen begegnet. Den Abschluss dieser Arbeit bildet der Ausblick auf mögliche Verbesserungen und Erweiterungen von *Jass*.

6.1 Ergebnisse

In Abschnitt 5.1 wurde schon kurz angesprochen, dass die Realisierung der ursprünglich geplanten Fallstudie *Automatic Teller Machine* aufgrund softwaretechnischer Probleme in der gegebenen Zeit nicht umsetzbar war. Die Ursache dieser Probleme ist folgende: Für die Spezifikation des Verhaltens des Geldautomaten werden mehrere Klassen aus der JML Distribution verwendet, die insbesondere eine Modellierung von Mengenstrukturen bereitstellen. Diese Klassen besitzen wiederum eigene Verhaltensspezifikationen, die auf die gesamte Bandbreite von JML-Sprachkonstrukten zurückgreifen. Aus Gründen der Kompatibilität zu bereits bestehenden JML-Spezifikationen sollte *Jass* auch diese verarbeiten können. Auf syntaktischer Ebene stellt dies kein Problem dar, da der für *Jass* implementierte Parser auf der JML-Gesamtgrammatik aufbaut und so mit beliebigen Spezifikationen umgehen kann. Auf semantischer Ebene hingegen stellt die Vielzahl der JML-Konstrukte ein erhebliches Problem dar, da sie während der semantischen Codeanalyse berücksichtigt werden müssen. Der in dieser Arbeit aus Zeitgründen nur teilweise implementierte Ansatz zur Umsetzung der geforderten Kompatibilität zu JML und der damit einhergehenden semantischen Verarbeitung der in JMLjass nicht enthaltenen Sprachelemente basiert im Wesentlichen auf zwei Überlegungen:

- Wenn *Jass* während der *syntaktischen* Analyse auf ein nicht in JMLjass unterstütztes Sprachkonstrukt trifft, so wird der von diesem Konstrukt ausgehende

Ast im Syntaxbaum nicht erzeugt. Dies ist allerdings nicht für alle Sprachelemente möglich, da an manchen Stellen ein fehlender Ast zu Folgefehlern in der Ableitung führen kann.

- Wenn *Jass* während der *semantischen* Analyse auf ein nicht durch die syntaktische Analyse ausgeschlossenes JML-Sprachkonstrukt trifft, so wird versucht, dieses Konstrukt durch einen kontextabhängigen, vom Typ des Konstrukts abhängigen Standardwert zu ersetzen.

Zur Erläuterung und Plausibilisierung dieses Ansatzes sei folgende Beispielspezifikation betrachtet:

```
/*@  public normal_behavior
  @    requires theString != null;
  @    assignable theString;
  @    ensures \fresh(\result) && \result.equals(theString);
  @*/
  public String copyString(theString);
```

Der `assignable`-Ausdruck besagt, dass während der Ausführung der Methode aus der Sicht des Aufrufenden nur auf das Feld `theString` schreibend zugegriffen wird. Da JMLjass aufgrund der fehlenden Möglichkeit zur Kontroll- und Datenflussanalyse in *Jass* dieses Konstrukt nicht unterstützt, wird es bereits während der syntaktischen Analyse durch Nichterzeugung des zugehörigen Astes des Syntaxbaums von der weiteren Verarbeitung ausgeschlossen. Dies ist ohne Weiteres möglich, da das Fehlen dieses Astes für den Elternknoten, in diesem Fall eine Methodenspezifikationen, keine kritische Einschränkung bedeutet.
Das in der Spezifikation der Nachbedingung auftauchende `\fresh`-Konstrukt besagt, dass dem übergebenen Argument im Pre-State der Methodenausführung noch kein Speicher in der Java Virtual Machine zugeordnet war. Auch dieser Ausdruck wird von JMLjass nicht unterstützt; allerdings kann an dieser Stelle nicht auf die Erzeugung des entsprechenden Astes im Syntaxbaum verzichtet werden, da der durch die Konjunktion gebildete binäre Ausdruck sonst fehlerhaft wäre. Aus diesem Grund muss das `\fresh`-Konstrukt während der semantischen Analyse *kontextneutral* ersetzt werden, d.h. es muss ein Prädikat gefunden werden, das den umgebenden binären Ausdruck nicht verfälscht. Für die Konjunktion ist dies der Wert `true`, für Disjunktionen der Wert `false`.

Das hauptsächliche Problem bei der Realisierung des beschriebenen Ansatz ist der große Sprachumfang von JML. Um mit *Jass* JML-Spezifikationen korrekt verarbeiten zu können, muss für jedes in JMLjass nicht unterstützte Sprachkonstrukt, das nicht während der syntaktischen Analyse ausgeschlossen werden kann, eine eigene Behandlung implementiert werden. Die Ersetzung des jeweiligen Konstrukts durch einen Standardwert erfordert die Analyse des entsprechenden Kontexts und die Bestimmung eines typgerechten, kontextneutralen Ausdrucks. Dies wird durch den Umstand erschwert, dass nicht nur Boolesche Ausdrücke

behandelt werden müssen, sondern Sprachkonstrukte verschiedener Typen. Die vollständige Implementierung dieses Ansatzes war aus zeitlichen Gründen in dieser Arbeit leider nicht möglich.

Abgesehen von dieser Problematik, die die geplante vergleichende Untersuchung von *Jass* und des JML RACC nicht zuließ, lässt sich festhalten, dass *Jass* mit Spezifikationen umgehen kann, die nur auf in JMLjass unterstützte Sprachelemente zurückgreifen. Sowohl die Vererbung von Spezifikationen als auch die Zuordnung von Modellvariablen in Interfaces und deren Abstraktionsfunktionen in Klassen lassen sich durch die Verwendung von Surrogat- und Referenzvariablen realisieren. Es bleibt aber zu zeigen, ob dieser Ansatz die Probleme des JML RACC mit Spezifikationen in Interfaces vermeiden kann.
Zu bemerken ist außerdem, dass *Jass* mit den in Java 1.5 eingeführten Sprachelementen sowohl syntaktisch als auch semantisch umgehen kann. Dadurch lassen sich auch Spezifikationen übersetzen, die zum Beispiel auf generische Klassen zurückgreifen. Mit dem JML Runtime Assertion Checker in der aktuellen Version 5.3 ist dies noch nicht möglich.

6.2 Ausblick

Neben der Vervollständigung des im vorherigen Abschnitts besprochenen Ansatzes zur Gewährleistung der Kompatibilität zu JML sind noch einige Erweiterungsmöglichkeiten des in dieser Arbeit entwickelten Precompilers denkbar, die im Folgenden vorgestellt werden.

6.2.1 Erweiterung zu JML

Um *Jass* für potenzielle Anwender aus Wirtschaft und Forschung interessant zu gestalten, sollte es ein langfristiges Ziel sein, den kompletten Sprachumfang von JML in die Eingabesprache zu integrieren. Ansätze zur passiven Unterstützung sind bereits teilweise gegeben. Eine aktive Übersetzung der in dieser Arbeit nicht berücksichtigten Sprachelemente erfordert aber insbesondere die Implementierung einer Kontroll- und Datenflussanalyse. Grundlegende Elemente einer statischen Datenflussanalyse sind bereits in [Bar99] beschrieben und für die Spezifikationssprache Jass realisiert. Es ist durchaus vorstellbar, Teile der diesbezüglichen Implementierung für JML zu übernehmen oder zumindest darauf aufzubauen.
Weiterhin kann *Jass* JML-Spezifikationen bereits syntaktisch verarbeiten. Die vollständige Integration von JML erfordert daher die Erweiterung der semantischen Analyse und die Anpassung der Codegenerierung an die neuen Sprachelemente. Die in dieser Arbeit entwickelten Übersetzungsschemata bleiben davon aber unberührt, da bei der Implementierung der für Analyse und Codeerzeugung benötigten Klassen und deren Beziehungen darauf geachtet wurde, diese möglichst modular

und leicht erweiterbar zu halten.

6.2.2 Verbesserung des Parsers

Bereits in Abschnitt 3.3 wurden einige Strategien zur Fehlererholung während der syntaktischen Analyse eines Eingabedokuments vorgestellt. Ziel der Fehlererholgung ist es, dem Anwender möglichst viele syntaktische Fehler in einem einzigen Verarbeitungsdurchgang aufzuzeigen. Der für *Jass* entwickelte Parser verfügt nur rudimentär über eine panische Fehlererholung, die im Eingabestrom Zeichen überspringt, bis sie zu einem der in Java synchronisierenden Zeichen `;` oder `}` gelangt. Eine Erweiterung insbesondere durch die Einführung von Fehlerproduktion wäre aber aus Sicht der Anwender wünschenswert.

6.2.3 Serialisierung von Zwischenergebnissen

Sowohl die syntaktische, vor allem aber auch die semantische Analyse von Spezifikationen sind sehr zeitaufwändig und rechenintensiv, was sich insbesondere bei umfangreichen Projekten deutlich negativ bemerkbar machen kann. Daher wäre es sinnvoll, die in Form von Klasseninstanzen vorliegenden Ergebnisse der syntaktischen und semantischen Analyse in einer persistenteren Form abspeichern zu können, um sie für spätere Übersetzungsläufe auf einfache Weise wiederverwenden zu können. Für die sogenannte *Serialisierung* von Java-Objekten, d.h. der Kodierung des Objektzustandes zum Beispiel in Form einer später auslesbaren Datei, bietet es sich in diesem Zusammenhang an, auf die *Extended Markup Language* (kurz: XML) zurückzugreifen. XML dient im Wesentlichen dazu, Daten strukturiert darzustellen und wird in heutigen Softwaresystemen insbesondere für den Austausch von Informationen über ein Netzwerk verwendet. Die Repräsentation von Objekten in XML-Dokumenten hat im Vergleich zur Darstellung in Bytecode den großen Vorteil, transparent, klar strukturiert und für Anwender leicht nachvollziehbar zu sein. Die Serialisierung von Zwischenergebnissen in XML nach Abschluss der semantischen Analyse ließe sich ohne größeren Aufwand durch die Verwendung einer externen Klassenbibliothek umsetzen. Als Beispiel sei an dieser Stelle etwa das OpenSource-Projekt *XStream* genannt, dass sich speziell mit der Serialisierung und Deserialiserung zwischen Java und XML beschäftigt.

Literaturverzeichnis

[Aus04] Calvin Austin. *J2SE 5.0 in a Nutshell.* http://java.sun.com/developer/technicalArticles/releases/index.htm (Stand: Februar 2006).

[AO94] Krzysztof R. Apt, Ernst-Rüdiger Olderog. *Programmverifikation. Sequentielle, parallele und verteilte Programme.* Springer, Berlin, 1994.

[ASU99] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann. *Compilerbau.* Oldenbourg, München, 1999.

[Bar99] Detlef Bartetzko. *Parallelität und Vererbung beim „Programmieren mit Vertrag". Weiterentwicklung von JaWa.* Diplomarbeit, Universität Oldenburg, 1999.

[BCC+04] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll. *An overview of JML tools and applications.* In: Software Tools for Technology Transfer, Springer, Hamburg, 2004.

[BGJS05] Gilad Bracha, James Gosling, Bill Joy, Guy Steele. *The Java Language Specification, Third Edition.* Addison Wesley, Boston, 2005.

[BLL+04] L. du Bousquet, J.-L. Lanet, Y. Ledru, O. Maury, C. Oriat. *A case study in JML-based software validation.* Proceedings of 19th Int. IEEE Conference on Automated Software Engineering, S. 294-297, IEEE CS Press, Linz, 2004.

[BLR99] Albert R. Baker, Gary T. Leavens, Clyde Ruby. *JML: A Notation For Detailed Design.* In: Behavioral Specifications For Business and Systems, S. 175-188, Kluwer Academic Publishers, 1999.

[BLR05] Albert L. Baker, Gary T. Leavens, Clyde Ruber. *Preliminary Design of JML: A Behavioral Interface Specification Language For Java.* Department for Computer Science, Iowa State University, 2005.

[Cli01] Curtis Charles Clifton. *MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch.* Master's Thesis, Department of Computer Science, Iowa State University, 2001.

[Che03] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. Dissertation, Iowa State University, 2003.

[Cha05] Patrice Chalin. *Reassessing JML's Logical Foundation*. Dependable Software Research Group, Concordia University, 2005.

[CCC+06] Yoonsik Cheon, Curtis Clifton, David Cok, Joseph Kiniry, Gary T. Leavens, Peter Müller, Clyde Ruby. *JML Reference Manual*. Iowa State University, Department of Computer Science, 2006.

[CCL05] Yoonsik Cheon, David Cok, Gary T. Leavens. *Demonstration of JML Tools*. Department of Computer Science, Iowa State University, 2005.

[CELS04] Yoonsik Cheon, Stephen Edwards, Gary T. Leavens, Murali Sitaraman. *Model Variables: Cleanly Supporting Abstraction in Design by Contract*. Department of Computer Science, Iowa State University, 2004.

[CK98] Elaine Cheong, Joseph R. Kiniry. *JPP: A Java Pre-Processor*. Caltech Technical Report CS-TR-98-15, California Institute of Technology, 1998.

[CL05a] Yoonsik Cheon, Gary T. Leavens. *Design by Contract with JML*. Department of Computer Science, Iowa State University, 2005.

[CL05b] Yoonsik Cheon, Gary T. Leavens. *A Contextual Interpretation of Undefinedness for Runtime Assertion Checking*. In: Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG2005), Monterey, Kalifornien, 2005.

[DFH+05] Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Edwin Rodríguez. *Extending JML for Modular Specifcation and Verification of Multi-Threaded Programs*. In: Technical Report SAnToS-TR2004-10, Department of Computing and Information Sciences, Kansas State University, 2005.

[DM02] Werner Dietl, Peter Müller. *Universes: Lightweight Ownership for JML*. Journal of Object Technology, ETH Zürich, 2002.

[DM05] Ádám Darvas, Peter Müller. *Reasoning About Method Calls in JML Specifications*. ETH Zürich, Schweiz, 2005.

[ELM+03] Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, Erik Poll. *Formal Techniques for Java-like Programs 2003*. Darmstadt, 2003.

[FLL+02] Cormac Flanagan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata. *Extended Static Checking For Java*. PLDI '02, Berlin, 2002.

[Hoa69] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM 12(10), S.576 - 583,ACM Press, 1969.

[JP00] Bart Jacobs, Eric Poll. *A Logic for the Java Modeling Language JML*. Department of Computer Science, University of Nijmegen, 2000.

[LL00] John Lewis, William Loftus. *Java Software Solutions. Foundations of Program Design*. Addison Wesley, New York, 2000.

[LMP04] Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter. *Modular Invariants for Layered Object Structures*. Department of Computer Science, Iowa State University, 2004.

[LR00] Gary T. Leavens, Clyde Ruby. *Safely Creating Correct Subclasses without Seeing Superclass Code*. In: OOPSLA 2000, Minneapolis, Minnesota, 2000.

[Mey92] Bertrand Meyer. *Applying „design by contract“*. In: Computer 25(10), S.40 - 51, IEEE Computer, 1992

[Mee97] Dieter Meemken. *Programmieren mit Vertrag in Java*. Diplomarbeit, Universität Oldenburg, 1997.

[Pan05] Sven Eric Panitz. *Compilerbau*. Skript zur Vorlesung, FH Wiesbaden, 2005.

[PS02] Erik Poll, Fausto Spoto. *Static Analysis for JML's* `assignable` *Clause*. University of Nijmegen, Niederlande, und Dipartimento di Informatica, Verona, Italien, 2002.

[RL00] Arun D. Raghavan, Gary T. Leavens. *Desugaring JML Method Specifications*. Technical Report 00-03a, Department of Computer Science, Iowa State University, 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.ps.gz (Stand: Februar 2006).

[Sun04] Sun. *Java Language Guide*. Online-Dokumentation, 2004. Zu beziehen unter: http://java.sun.com/j2se/1.5.0/docs/guide/language (Stand: Feburar 2006).

[Tya04] Rahul Tyagi. *Java: Erweiterte Funktionen von JDK 1.5*. Online-Artikel, ZDnet, 2004. http://www.zdnet.de/builder/ program/0,39023551,39121521,00.htm (Stand: Februar 2004).

[Ver03] Joe Verzulli. *Getting started with JML*. In: Java technology articles, IBM developerWorks, 2003. Zu beziehen unter: http://www-105.ibm.com/developerworks/java (Stand: Februar 2006).

[Vis05] Sreenivasa Viswanadha. *JavaCC - The Java Parser Generator*. Online-Dokumentation, 2005. http://javacc.dev.java.net (Stand: Februar 2006).

Anhang A

JMLjass: Grammatik

Im Folgenden ist die Grammatik von JMLjass aufgeführt. Terminalsymbole sind dabei in spitzen Klammern und Großbuchstaben aufgeführt; sie werden später im Anhang genauer behandelt.

A.1 Compilation Units

```
CompilationUnit ::= ( <FORMAL_COMMENT> )*
                    ( PackageDeclaration )?
                    ( RefinePrefix )?
                    ( ImportDeclaration )*
                    ( TypeDeclaration )*
                    <EOF>
PackageDeclaration ::= <PACKAGE> Name <SEMICOLON>
ImportDeclaration ::= ( <MODEL> )?
                      <IMPORT>
                      ( <STATIC> )?
                      Name
                      ( <DOT> <STAR> )?
                      <SEMICOLON>
```

A.2 Modifier

```
Modifiers ::= ( JavaModifiers | JMLModifier | Annotation )*
JavaModifiers ::= <PUBLIC>
                | <STATIC>
                | <PROTECTED>
                | <PRIVATE>
```

```
                    | <FINAL>
                    | <ABSTRACT>
                    | <SYNCHRONIZED>
                    | <NATIVE>
                    | <TRANSIENT>
                    | <VOLATILE>
                    | <STRICTFP>
JMLModifier ::= <SPEC_PUBLIC>
                  | <SPEC_PROTECTED>
                  | <MODEL>
                  | <GHOST>
                  | <PURE>
                  | <INSTANCE>
                  | <HELPER>
                  | <UNINITIALIZED>
                  | <NON_NULL>
```

A.3 Typdeklarationen

```
TypeDeclaration ::= <SEMICOLON>
                    | ClassOrInterfaceDeclaration
                    | EnumDeclaration
                    | AnnotationTypeDeclaration
ClassOrInterfaceDeclaration ::= ( <FORMAL_COMMENT> )*
                                Modifiers
                                ( <CLASS> | <INTERFACE> )
                                <IDENTIFIER>
                                ( TypeParameters )?
                                ( ExtendsList )?
                                ( ImplementsList )?
                                ClassOrInterfaceBody
ExtendsList ::= <EXTENDS>
                ClassOrInterfaceType
                ( <COMMA> ClassOrInterfaceType )*
ImplementsList ::= <IMPLEMENTS>
                   ClassOrInterfaceType
                   ( <COMMA> ClassOrInterfaceType )*
EnumDeclaration ::= Modifiers
                    <ENUM>
                    <IDENTIFIER>
                    ( ImplementsList )?
                    EnumBody
EnumBody ::= <LBRACE>
```

```
                EnumConstant
                ( <COMMA> EnumConstant )*
                ( <SEMICOLON> ( ClassOrInterfaceBodyDeclaration )* )?
                <RBRACE>
EnumConstant ::= <IDENTIFIER> ( Arguments )? ( ClassOrInterfaceBody )?
TypeParameters ::= <LT> TypeParameter ( <COMMA> TypeParameter )* <GT>
TypeParameter ::= <IDENTIFIER> ( TypeBound )?
TypeBound ::= <EXTENDS> ClassOrInterfaceType ( <BIT_AND> ClassOrInterfaceType )*
```

A.4 Member Deklarationen

```
ClassOrInterfaceBody ::= <LBRACE> ( ClassOrInterfaceBodyDeclaration )* <RBRACE>
ClassOrInterfaceBodyDeclaration ::= Initializer
                                  | JMLDeclaration
                                  | MemberDeclaration
                                  | <SEMICOLON>
MemberDeclaration ::= ClassOrInterfaceDeclaration
                    | EnumDeclaration
                    | ConstructorDeclaration
                    | FieldDeclaration
                    | MethodDeclaration
FieldDeclaration ::= ( <FORMAL_COMMENT> )* Modifiers VariableDeclarations
VariableDeclarations ::= ( <FIELD> )?
                         TypeSpec
                         VariableDeclarators
                         <SEMICOLON>
                         ( JMLDataGroupClause )*
VariableDeclarators ::= VariableDeclarator ( <COMMA> VariableDeclarator )*
VariableDeclarator ::= VariableDeclaratorId ( <ASSIGN> VariableInitializer )?
VariableDeclaratorId ::= <IDENTIFIER> ( <LBRACKET> <RBRACKET> )*
VariableInitializer ::= ArrayInitializer | Expression
ArrayInitializer ::= <LBRACE>
                     ( VariableInitializer ( <COMMA> VariableInitializer )* )?
                     ( <COMMA> )?
                     <RBRACE>
MethodDeclaration ::= ( <FORMAL_COMMENT> )*
                      Modifiers
                      ( TypeParameters )?
                      ( <METHOD> )?
                      ResultType
                      MethodDeclarator
                      ( <THROWS> NameList )?
```

```
                        ( MethodSpecification )?
                        ( Block | <SEMICOLON> )
                        |
                        ( <FORMAL_COMMENT> )*
                        MethodSpecification
                        Modifiers
                        ( TypeParameters )?
                        ( <METHOD> )?
                        ResultType
                        MethodDeclarator
                        ( <THROWS> NameList )?
                        ( Block | <SEMICOLON> )
MethodDeclarator ::= <IDENTIFIER>
                     FormalParameters
                     ( <LBRACKET> <RBRACKET> )*
FormalParameters ::= <LPAREN>
                     ( FormalParameter ( <COMMA> FormalParameter )* )?
                     <RPAREN>
FormalParameter ::= ( <FINAL> | <NON_NULL> )?
                    TypeSpec
                    ( <ELLIPSIS> )?
                    VariableDeclaratorId
ConstructorDeclaration ::= ( <FORMAL_COMMENT> )*
                           MethodSpecification
                           Modifiers
                           ( TypeParameters )?
                           ( <CONSTRUCTOR> )?
                           <IDENTIFIER>
                           FormalParameters
                           ( <THROWS> NameList )?
                           <LBRACE>
                           ( ExplicitConstructorInvocation )?
                           ( BlockStatement )*
                           <RBRACE>
                           |
                           ( <FORMAL_COMMENT> )*
                           Modifiers
                           ( TypeParameters )?
                           ( <CONSTRUCTOR> )?
                           <IDENTIFIER>
                           FormalParameters
                           ( <THROWS> NameList )?
                           ( MethodSpecification )?
                           <LBRACE>
                           ( ExplicitConstructorInvocation )?
```

```
                                ( BlockStatement )*
                                <RBRACE>
ExplicitConstructorInvocation ::= <THIS>
                                  Arguments
                                  <SEMICOLON>
                                  |
                                  ( PrimaryExpression <DOT> )?
                                  <SUPER>
                                  Arguments
                                  <SEMICOLON>
Initializer ::= MethodSpecification
                ( <STATIC_INITIALIZER> | <INITIALIZER> )
                |
                ( MethodSpecification )?
                ( <STATIC> )?
                Block
TypeSpec ::= Type
             | <PRED_UPPERCASE_TYPE> ( <LBRACKET> <RBRACKET> )*
```

A.5 Typspezifikationen

```
JMLDeclaration ::= Modifiers
                   ( Invariant
                   | HistoryConstraint
                   | RepresentsDeclaration
                   | InitiallyClause )
Invariant ::= <INVARIANT> Predicate <SEMICOLON>
HistoryConstraint ::= <CONSTRAINT>
                      Predicate
                      ( <FOR> ConstrainedList )?
                      <SEMICOLON>
ConstrainedList ::= MethodNameList | <PRED_EVERYTHING>
MethodNameList ::= MethodName ( <COMMA> MethodName )*
MethodName ::= MethodReference ( <LPAREN> ParameterDisambiguityList <RPAREN> )?
MethodReference ::= MethodReferenceStart ( <DOT> MethodReferenceRest )*
                    | <NEW> ReferenceType
MethodReferenceStart ::= <SUPER>
                         | <THIS>
                         | <IDENTIFIER>
                         | <PRED_OTHER>
MethodReferenceRest ::= <THIS>
                        | <IDENTIFIER>
ParameterDisambiguityList ::= ParameterDisambiguity ( <COMMA> ParameterDisambiguity )*
```

```
ParameterDisambiguity ::= TypeSpec ( <IDENTIFIER> ( <LBRACKET> <RBRACKET> )* )?
RepresentsDeclaration ::= <REPRESENTS>
                          StoreReferenceExpression
                          AbstrFuncOrAssign
                          SpecificationExpression
                          <SEMICOLON>
                          |
                          <REPRESENTS>
                          StoreReferenceExpression
                          <PRED_SUCH_THAT>
                          Predicate
                          <SEMICOLON>
AbstrFuncOrAssign ::= <ABSTR_FUNC_BACKWARDS> | <ASSIGN>
InitiallyClause ::= <INITIALLY> Predicate <SEMICOLON>
Type ::= ReferenceType | PrimitiveType
ReferenceType ::= PrimitiveType ( <LBRACKET> <RBRACKET> )+
                | ( ClassOrInterfaceType ) ( <LBRACKET> <RBRACKET> )*
ClassOrInterfaceType ::= PossiblyGenericClassOrInterfaceType
                         ( <DOT> PossiblyGenericClassOrInterfaceType )*
PossiblyGenericClassOrInterfaceType ::= <IDENTIFIER> ( TypeArguments )?
TypeArguments ::= <LT> TypeArgument ( <COMMA> TypeArgument )* <GT>
TypeArgument ::= ReferenceType | <HOOK> ( WildcardBounds )?
WildcardBounds ::= <EXTENDS> ReferenceType | <SUPER> ReferenceType
PrimitiveType ::= <BOOLEAN>
                | <CHAR>
                | <BYTE>
                | <SHORT>
                | <INT>
                | <LONG>
                | <FLOAT>
                | <DOUBLE>
ResultType ::= <VOID> | Type
Name ::= <IDENTIFIER> ( <DOT> <IDENTIFIER> )*
NameList ::= Name ( <COMMA> Name )*
```

A.6 Methodenspezifikationen

```
MethodSpecification ::= Specification | ExtendingSpecification
ExtendingSpecification ::= <ALSO> Specification
Specification ::= SpecificationCaseSequence ( RedundantSpecification )?
                | RedundantSpecification
SpecificationCaseSequence ::= SpecificationCase ( <ALSO> SpecificationCase )*
SpecificationCase ::= LightweightSpecificationCase
```

```
                        | HeavyweightSpecificationCase
                        | CodeContractSpecification
LightweightSpecificationCase ::= GenericSpecificationCase
GenericSpecificationCase ::= ( SpecificationVariableDeclarations )?
                             SpecificationHeader
                             ( GenericSpecificationBody )?
                             |
                             ( SpecificationVariableDeclarations )?
                             GenericSpecificationBody
GenericSpecificationBody ::= SimpleSpecificationBody
                             |
                             <NESTED_SPEC_BEGIN>
                             GenericSpecificationCaseSequence
                             <NESTED_SPEC_END>
GenericSpecificationCaseSequence ::= GenericSpecificationCase
                                     ( <ALSO> GenericSpecificationCase )*
SpecificationHeader ::= RequiresClause ( RequiresClause )*
SimpleSpecificationBody ::= SimpleSpecificationBodyClause
                            ( SimpleSpecificationBodyClause )*
SimpleSpecificationBodyClause ::= DivergesClause
                                  | AssignableClause
                                  | CapturesClause
                                  | EnsuresClause
                                  | SignalsOnlyClause
                                  | SignalsClause
HeavyweightSpecificationCase ::= BehaviourSpecificationCase
                                 | ExceptionalBehaviourSpecificationCase
                                 | NormalBehaviourSpecificationCase
BehaviourSpecificationCase ::= ( Privacy )? <BEHAVIOR> GenericSpecificationCase
Privacy ::= <PUBLIC> | <PROTECTED> | <PRIVATE>
NormalBehaviourSpecificationCase ::= ( Privacy )?
                                     <NORMAL_BEHAVIOR>
                                     GenericSpecificationCase
ExceptionalBehaviourSpecificationCase ::= ( Privacy )?
                                          <EXCEPTIONAL_BEHAVIOR>
                                          GenericSpecificationCase
SpecificationVariableDeclarations ::= ForallVariableDeclarations ( OldVariableDeclarations )?
                                      | OldVariableDeclarations
ForallVariableDeclarations ::= ForallVariableDeclaration ( ForallVariableDeclaration )*
ForallVariableDeclaration ::= <FORALL> QuantifiedVariableDeclarations <SEMICOLON>
OldVariableDeclarations ::= OldVariableDeclaration ( OldVariableDeclaration )*
OldVariableDeclaration ::= <OLD>
                           TypeSpec
                           SpecificationVariableDeclarators
                           <SEMICOLON>
```

```
RequiresClause ::= <REQUIRES> PredicateOrNot <SEMICOLON>
PredicateOrNot ::= Predicate | <PRED_NOT_SPECIFIED>
EnsuresClause ::= <ENSURES> PredicateOrNot <SEMICOLON>
SignalsClause ::= <SIGNALS>
                  <LPAREN>
                  ReferenceType
                  ( <IDENTIFIER> )?
                  <RPAREN>
                  ( PredicateOrNot )?
                  <SEMICOLON>
SignalsOnlyClause ::= <SIGNALS_ONLY>
                        ReferenceType
                        ( <COMMA> ReferenceType )*
                        <SEMICOLON>
                        |
                        <SIGNALS_ONLY>
                        <PRED_NOTHING>
                        <SEMICOLON>
DivergesClause ::= <DIVERGES> PredicateOrNot <SEMICOLON>
AssignableClause ::= ( <ASSIGNABLE> | <MODIFIABLE> | <MODIFIES> )
                       StoreReferenceList
                       <SEMICOLON>
CapturesClause ::= <CAPTURES> StoreReferenceList <SEMICOLON>
```

A.7 Data Groups

```
JMLDataGroupClause ::= InGroupClause | MapsIntoClause
InGroupClause ::= <IN> GroupList <SEMICOLON>
GroupList ::= GroupName ( <COMMA> GroupList )*
GroupName ::= ( <SUPER> <DOT> | <THIS> <DOT> )? <IDENTIFIER>
MapsIntoClause ::= <MAPS>
                     MemberFieldReference
                     <PRED_INTO>
                     GroupList
                     <SEMICOLON>
MemberFieldReference ::= MapsArrayReferenceExpression
                           ( <DOT> MapsMemberReferenceExpression )?
                           |
                           <IDENTIFIER>
                           <DOT>
                           MapsMemberReferenceExpression
MapsMemberReferenceExpression ::= <IDENTIFIER> | <STAR>
MapsArrayReferenceExpression ::= <IDENTIFIER>
```

```
                                        MapsSpecificationArrayDimension
                                        ( MapsSpecificationArrayDimension )*
MapsSpecificationArrayDimension ::= <LBRACKET>
                                        SpecificationArrayReferenceExpression
                                        <RBRACKET>
```

A.8 Prädikate und Spezifikationsausdrücke

```
Predicate ::= SpecificationExpression
SpecificationExpressionList ::= SpecificationExpression ( SpecificationExpression )*
SpecificationExpression ::= Expression
Expression ::= ConditionalExpression ( AssignmentOperator Expression )?
AssignmentOperator ::= <ASSIGN>
                     | <STARASSIGN>
                     | <SLASHASSIGN>
                     | <REMASSIGN>
                     | <PLUSASSIGN>
                     | <MINUSASSIGN>
                     | <LSHIFTASSIGN>
                     | <RSIGNEDSHIFTASSIGN>
                     | <RUNSIGNEDSHIFTASSIGN>
                     | <ANDASSIGN>
                     | <XORASSIGN>
                     | <ORASSIGN>
ConditionalExpression ::= EquivalenceExpression
                        ( <HOOK> Expression <COLON> Expression )?
EquivalenceExpression ::= ImpliesExpression
                        ( EquivalenceOperator EquivalenceExpression )?
EquivalenceOperator ::= <IF_AND_ONLY_IF> | <NEG_IF_AND_ONLY_IF>
ImpliesExpression ::= ConditionalOrExpression
                    ( <IMPLIES_BACKWARDS>
                    ConditionalOrExpression
                    ( <IMPLIES_BACKWARDS> ConditionalOrExpression )*
                    |
                    ( <IMPLIES> ImpliesNonBackwardExpression )?
                    )
ImpliesNonBackwardExpression ::= ConditionalOrExpression
                               ( <IMPLIES> ImpliesNonBackwardExpression )?
ConditionalOrExpression ::= ConditionalAndExpression
                          ( <SC_OR> ConditionalOrExpression )?
ConditionalAndExpression ::= InclusiveOrExpression
                           ( <SC_AND> ConditionalAndExpression )?
InclusiveOrExpression ::= ExclusiveOrExpression
```

```
                              ( <BIT_OR> InclusiveOrExpression )?
ExclusiveOrExpression ::= AndExpression ( <XOR> ExclusiveOrExpression )?
AndExpression ::= EqualityExpression ( <BIT_AND> AndExpression )?
EqualityExpression ::= InstanceOfExpression
                           ( EqualityOperator EqualityExpression )?
EqualityOperator ::= ( <EQ> | <NE> )
InstanceOfExpression ::= RelationalExpression ( <INSTANCEOF> TypeSpec )?
RelationalExpression ::= ShiftExpression
                             ( RelationalOperator RelationalExpression )?
RelationalOperator ::= ( <LT> | <GT> | <LE> | <GE> | <SUBTYPE> )
ShiftExpression ::= AdditiveExpression
                    ( ( LSIGNEDSHIFT
                    | RSIGNEDSHIFT
                    | RUNSIGNEDSHIFT )
                    ShiftExpression )?
AdditiveExpression ::= MultiplicativeExpression
                           ( AdditiveOperator AdditiveExpression )?
AdditiveOperator ::= ( <PLUS> | <MINUS> )
MultiplicativeExpression ::= UnaryExpression
                                 ( MultiplicativeOperator MultiplicativeExpression )?
MultiplicativeOperator ::= ( <STAR> | <SLASH> | <REM> )
UnaryExpression ::= ( <PLUS> | <MINUS> ) UnaryExpression
                    | PreIncrementExpression
                    | PreDecrementExpression
                    | UnaryExpressionNotPlusMinus
PreIncrementExpression ::= <INCR> PrimaryExpression
PreDecrementExpression ::= <DECR> PrimaryExpression
UnaryExpressionNotPlusMinus ::= ( <TILDE> | <BANG> ) UnaryExpression
                                    | CastExpression
                                    | PostfixExpression
PostfixExpression ::= PrimaryExpression ( <INCR> | <DECR> )?
CastExpression ::= <LPAREN> Type <RPAREN> UnaryExpression
                   | <LPAREN> Type <RPAREN> UnaryExpressionNotPlusMinus
PrimaryExpression ::= PrimaryPrefix ( PrimarySuffix )*
MemberSelector ::= <DOT> TypeArguments <IDENTIFIER>
PrimaryPrefix ::= Literal
                  | <THIS>
                  | <SUPER> <DOT> <IDENTIFIER>
                  | JMLPrimary
                  | <LPAREN> Expression <RPAREN>
                  | AllocationExpression
                  | ResultType
                  | <DOT> <CLASS>
                  | Name
PrimraySuffix ::= <DOT> <THIS>
```

```
                    | <DOT> AllocationExpression
                    | MemberSelector
                    | <LBRACKET> Expression <RBRACKET>
                    | <DOT> <IDENTIFIER>
                    | Arguments
JMLPrimary ::= ResultExpression
                  | OldExpression
                  | NotAssignedExpression
                  | NotModifiedExpression
                  | NonNullElementsExpression
                  | <INFORMAL_DESCRIPTION> | TypeOfExpression
                  | ElemtypeExpression
                  | TypeExpression
                  | MaxExpression
                  | IsInitializedExpression
                  | InvariantForExpression
                  | SpecificationQuantifiedExpression
ResultExpression ::= <PRED_RESULT>
OldExpression ::= <PRED_OLD>
                    <LPAREN>
                    SpecificationExpression
                    ( <COMMA> <IDENTIFIER> )?
                    <RPAREN>
                    | <PRED_PRE>
                    <LPAREN>
                    SpecificationExpression
                    <RPAREN>
NotAssignedExpression ::= <PRED_NOT_ASSIGNED>
                            <LPAREN>
                            StoreReferenceList
                            <RPAREN>
NotModifiedExpression ::= <PRED_NOT_MODIFIED>
                            <LPAREN>
                            StoreReferenceList
                            <RPAREN>
NonNullElementsExpression ::= <PRED_NON_NULL_ELEMENTS>
                                <LPAREN>
                                SpecificationExpression
                                <RPAREN>
TypeOfExpression ::= <PRED_TYPE_OF> <LPAREN> SpecificationExpression <RPAREN>
ElemtypeExpression ::= <PRED_ELEMTYPE>
                         <LPAREN>
                         SpecificationExpression
                         <RPAREN>
TypeExpression ::= <PRED_TYPE> <LPAREN> Type <RPAREN>
```

```
MaxExpression ::= <PRED_MAX>
                  <LPAREN>
                  SpecificationExpression
                  <RPAREN>
IsInitializedExpression ::= <PRED_IS_INITIALIZED>
                            <LPAREN>
                            ReferenceType
                            <RPAREN>
InvariantForExpression ::= <PRED_INVARIANT_FOR>
                           <LPAREN>
                           SpecificationExpression
                           <RPAREN>
SpecificationQuantifiedExpression ::= <LPAREN>
                                      Quantifier
                                      QuantifiedVariableDeclarations
                                      <SEMICOLON>
                                      ( Predicate <SEMICOLON> )?
                                      SpecificationExpression
                                      <RPAREN>
Quantifier ::= <PRED_FORALL>
             | <PRED_EXISTS>
             | <PRED_MAX>
             | <PRED_MIN>
             | <PRED_NUM_OF>
             | <PRED_PRODUCT>
             | <PRED_SUM>
QuantifiedVariableDeclarations ::= TypeSpec
                                   QuantifiedVariableDeclarator
                                   ( <COMMA> QuantifiedVariableDeclarator )*
QuantifiedVariableDeclarator ::= <IDENTIFIER> ( <LBRACKET> <RBRACKET> )*
SpecificationVariableDeclarators ::= SpecificationVariableDeclarator
                                     ( <COMMA> SpecificationVariableDeclarator )*
SpecificationVariableDeclarator ::= <IDENTIFIER>
                                    ( <LBRACKET> <RBRACKET> )*
                                    ( <ASSIGN> SpecificationInitializer )?
SpecificationArrayInitializer ::= <LBRACE>
                                  ( SpecificationInitializer
                                  ( <COMMA> SpecificationInitializer )*
                                  ( <COMMA> )? )?
                                  <RBRACE>
SpecificationInitializer ::= SpecificationExpression
                           | SpecificationArrayInitializer
StoreReferenceList ::= StoreReference ( <COMMA> StoreReference )*
StoreReference ::= StoreReferenceExpression
                 | <INFORMAL_DESCRIPTION>
```

```
                    | StoreReferenceKeyword
StoreReferenceExpression ::= StoreReferenceName
                             ( StoreReferenceNameSuffix )*
StoreReferenceName ::= <IDENTIFIER>
                       | <SUPER>
                       | <THIS>
StoreReferenceNameSuffix ::= <DOT>
                             ( <IDENTIFIER> | <THIS> | <STAR> )
                             | <LBRACKET>
                             SpecificationArrayReferenceExpression
                             <RBRACKET>
SpecificationArrayReferenceExpression ::= SpecificationExpression
                                          <DOT_DOT>
                                          SpecificationExpression
                                          | SpecificationExpression
                                          | <STAR>
StoreReferenceKeyword ::= <PRED_NOTHING>
                          | <PRED_EVERYTHING>
                          | <PRED_NOT_SPECIFIED>
Literal ::= <INTEGER_LITERAL>
          | <FLOATING_POINT_LITERAL>
          | <CHARACTER_LITERAL>
          | <STRING_LITERAL>
          | BooleanLiteral
          | NullLiteral
BooleanLiteral ::= <TRUE> | <FALSE>
NullLiteral ::= <NULL>
Arguments ::= <LPAREN> ( ArgumentList )? <RPAREN>
ArgumentList ::= Expression ( <COMMA> Expression )*
AllocationExpression ::= <NEW> PrimitiveType ArrayDimsAndInits
                         | <NEW>
                         ClassOrInterfaceType
                         ( ArrayDimsAndInits | Arguments ( ClassOrInterfaceBody )? )
ArrayDimsAndInits ::= ( <LBRACKET> Expression <RBRACKET> )+
                      ( <LBRACKET> <RBRACKET> )*
                      | ( <LBRACKET> <RBRACKET> )+ ArrayInitializer
```

A.9 Statements und Annotation Statements

```
Statement ::= LabeledStatement
            | AssertStatement
            | Block
            | EmptyStatement
```

```
                | PossiblyAnnotatedLoop
                | PossiblyAnnotatedLoop
                | PossiblyAnnotatedLoop
                | StatementExpression ";"
                | SwitchStatement
                | IfStatement
                | BreakStatement
                | ContinueStatement
                | ReturnStatement
                | ThrowStatement
                | SynchronizedStatement
                | TryStatement
PossiblyAnnotatedLoop ::= ( LoopInvariant )*
                          ( VariantFunction )*
                          LoopStatement
LoopStatement ::= WhileStatement
                | DoStatement
                | ForStatement
LoopInvariant ::= ( <MAINTAINING> | <LOOP_INVARIANT> )
                  Predicate
                  <SEMICOLON>
VariantFunction ::= ( <DECREASING> | <DECREASES> )
                    SpecificationExpression
                    <SEMICOLON>
AssertStatement ::= <ASSERT> Predicate ( <COLON> Expression )? <SEMICOLON>
LabeledStatement ::= <IDENTIFIER> <COLON> Statement
Block ::= <LBRACE> ( BlockStatement )* <RBRACE>
BlockStatement ::= LocalVariableDeclaration <SEMICOLON>
                 | Statement
                 | ClassOrInterfaceDeclaration
LocalVariableDeclaration ::= ( <FINAL> | <NON_NULL> )?
                             Type
                             VariableDeclarator
                             ( <COMMA> VariableDeclarator )*
EmptyStatement ::= <SEMICOLON>
StatementExpression ::= PreIncrementExpression
                      | PreDecrementExpression
                      | PrimaryExpression
                      ( <INCR> | <DECR> | AssignmentOperator Expression )?
SwitchStatement ::= <SWITCH>
                    <LPAREN>
                    Expression
                    <RPAREN>
                    <LBRACE>
                    ( SwitchLabelStatement )*
```

```
                    <RBRACE>
SwitchLabelStatement ::= SwitchLabel ( BlockStatement )*
SwitchLabel ::= <CASE> Expression <COLON>
              | <DEFAULT> <COLON>
IfStatement ::= <IF>
                <LPAREN>
                Expression
                <RPAREN>
                Statement
                ( <ELSE> Statement )?
WhileStatement ::= <WHILE> <LPAREN> Expression <RPAREN> Statement
DoStatement ::= <DO>
                Statement
                <WHILE>
                <LPAREN>
                Expression
                <RPAREN>
                <SEMICOLON>
ForStatement ::= <FOR>
                 <LPAREN>
                 ( EnhancedForStatement | StandardForStatement )
                 <RPAREN>
                 Statement
EnhancedForStatement ::= Type <IDENTIFIER> <COLON> Expression
StandardForStatement ::= ( ForInit )?
                         <SEMICOLON>
                         ( Expression )?
                         <SEMICOLON>
                         ( ForUpdate )?
ForInit ::= LocalVariableDeclaration
          | StatementExpressionList
StatementExpressionList ::= StatementExpression
                            ( <COMMA> StatementExpression )*
ForUpdate ::= StatementExpressionList
BreakStatement ::= <BREAK> ( <IDENTIFIER> )? <SEMICOLON>
ContinueStatement ::= <CONTINUE> ( <IDENTIFIER> )? <SEMICOLON>
ReturnStatement ::= <RETURN> ( Expression )? <SEMICOLON>
ThrowStatement ::= <THROW> Expression <SEMICOLON>
SynchronizedStatement ::= <SYNCHRONIZED>
                          <LPAREN>
                          Expression
                          <RPAREN>
                          Block
TryStatement ::= <TRY>
                 Block
```

```
                ( CatchStatement )*
                ( <FINALLY> Block )?
CatchStatement ::= <CATCH> <LPAREN> FormalParameter <RPAREN> Block
RUNSIGNEDSHIFT ::= ( <GT> <GT> <GT> )
RSIGNEDSHIFT ::= ( <GT> <GT> )
LSIGNEDSHIFT ::= <LSHIFT>
```

A.10 Annotationen

```
Annotation ::= NormalAnnotation
             | SingleMemberAnnotation
             | MarkerAnnotation
NormalAnnotation ::= <AT> Name <LPAREN> ( MemberValuePairs )? <RPAREN>
MarkerAnnotation ::= <AT> Name
SingleMemberAnnotation ::= <AT> Name <LPAREN> MemberValue <RPAREN
MemberValuePairs ::= MemberValuePair ( <COMMA> MemberValuePair )*
MemberValuePair ::= <IDENTIFIER> <ASSIGN> MemberValue
MemberValue ::= Annotation
              | MemberValueArrayInitializer
              | ConditionalExpression
MemberValueArrayInitializer ::= <LBRACE>
                                MemberValue
                                ( <COMMA> MemberValue )*
                                ( <COMMA> )?
                                <RBRACE>
```

A.11 Annotation Types

```
AnnotationTypeDeclaration ::= Modifiers
                              <AT>
                              <INTERFACE>
                              <IDENTIFIER>
                              AnnotationTypeBody
AnnotationTypeBody ::= <LBRACE>
                       ( AnnotationTypeMemberDeclaration )*
                       <RBRACE>
AnnotationTypeMemberDeclaration ::= ( Modifiers Type <IDENTIFIER>
                                    <LPAREN> <RPAREN>
                                    ( DefaultValue )? <SEMICOLON>
                                    | ClassOrInterfaceDeclaration
                                    | EnumDeclaration
```

| AnnotationTypeDeclaration
| FieldDeclaration)
| (<COMMA>)
DefaultValue ::= <DEFAULT> MemberValue

A.12 Redundanz

RedundantSpecification ::= Implications
Implications ::= <IMPLIES_THAT> SpecificationCaseSequence

A.13 Spezifikationen für Subtypen

CodeContractSpecification ::= <CODE_CONTRACT>
CodeContractClause
(CodeContractClause)*
CodeContractClause ::= AccessibleClause
| CallableClause
AccessibleClause ::= <ACCESSIBLE> ObjectStoreReferenceList <SEMICOLON>
ObjectStoreReferenceList ::= StoreReference | OtherReference
OtherReference ::= <PRED_OTHER> (StoreReferenceNameSuffix)*
CallableClause ::= <CALLABLE> CallableMethodsList <SEMICOLON>
CallableMethodsList ::= MethodNameList
| StoreReferenceKeyword

A.14 Verfeinerungen

RefinePrefix ::= (<REFINE> | <REFINES>) <STRING_LITERAL> <SEMICOLON>

Anhang B

JMLjass: Terminalsymbole

Im Folgenden werden die in der Grammatik verwendeten Terminalsymbole vorgestellt.

B.1 Reservierte Worte in JMLjass

ACCESSIBLE ::= “accessible“
ALSO ::= “also“
ASSIGNABLE ::= “assignable“
BEHAVIOR ::= “behaviour“ | “behavior“
CALLABLE ::= “callable“
CAPTURES ::= “captures“
CODE_CONTRACT ::= “code_contract“
CONSTRAINT ::= “constraint“
CONSTRUCTOR ::= “constructor“
DEBUG ::= “debug“
DECREASES ::= “decreases“
DECREASING ::= “decreasing“
DIVERGES ::= “diverges“
ENSURES ::= “ensures“ | “post“
EXCEPTIONAL_BEHAVIOR ::= “exceptional_behaviour“ | “exceptional_behavior“
FIELD ::= “field“
FORALL ::== ::= “forall“
GHOST ::= “ghost“
HELPER ::= “helper“
IN ::= “in“
INITIALIZER ::= “initializer“
INITIALLY ::= “initially“
INSTANCE ::= “instance“
INVARIANT ::= “invariant“

IMPLIES_THAT ::= “implies_that“
LOOP_INVARIANT ::= “loop_invariant“
MAPS ::= “maps“
METHOD ::= “method“
MODEL ::= “model“
MODIFIABLE ::= “modifiable“
MODIFIES ::= “modifies“
NON_NULL ::= “non_null“
NORMAL_BEHAVIOR ::= “normal_behaviour“ | “normal_behavior“
OLD ::= “old“
OR ::= “or“
PURE ::= “pure“
REFINE ::= “refine“
REFINES ::= “refines“
REPRESENTS ::= “represents“
REQUIRES ::= “requires“ | “pre“
RETURNS ::= “returns“
SET ::= “set“
SIGNALS ::= “signals“ | “exsures“
SIGNALS_ONLY ::= “signals_only“
SPEC_PROTECTED ::= “spec_protected“
SPEC_PUBLIC ::= “spec_public“
STATIC_INITIALIZER ::= “static_initializer“
UNINITIALIZED ::= “uninitialized“

B.2 Reservierte Worte in Java

ABSTRACT ::= “abstract“
ASSERT ::= “assert“
BOOLEAN ::= “boolean“
BREAK ::= “break“
BYTE ::= “byte“
CASE ::= “case“
CATCH ::= “catch“
CHAR ::= “char“
CLASS ::= “class“
CONST ::= “const“
CONTINUE ::= “continue“
DEFAULT ::= “default“
DO ::= “do“
DOUBLE ::= “double“
ELSE ::= “else“

ENUM ::= “enum“
EXTENDS ::= “extends“
FALSE ::= “false“
FINAL ::= “final“
FINALLY ::= “finally“
FLOAT ::= “float“
FOR ::= “for“
GOTO ::= “goto“
IF ::= “if“
IMPLEMENTS ::= “implements“
IMPORT ::= “import“
INSTANCEOF ::= “instanceof“
INT ::= “int“
INTERFACE ::= “interface“
LONG ::= “long“
MAINTAINING ::= “maintaining“
NATIVE ::= “native“
NEW ::= “new“
NULL ::= “null“
PACKAGE ::= “package“
PRIVATE ::= “private“
PROTECTED ::= “protected“
PUBLIC ::= “public“
RETURN ::= “return“
SHORT ::= “short“
STATIC ::= “static“
STRICTFP ::= “strictfp“
SUPER ::= “super“
SWITCH ::= “switch“
SYNCHRONIZED ::= “synchronized“
THIS ::= “this“
THROW ::= “throw“
THROWS ::= “throws“
TRANSIENT ::= “transient“
TRUE ::= “true“
TRY ::= “try“
VOID ::= “void“
VOLATILE ::= “volatile“
WHILE ::= “while“

B.3 Literale

```
INTEGER_LITERAL ::= DECIMAL_LITERAL (["l","L"])?
                  | HEX_LITERAL (["l","L"])?
                  | OCTAL_LITERAL (["l","L"])?
DECIMAL_LITERAL ::= ["1"-"9"] (["0"-"9"])*
HEX_LITERAL ::= "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+
OCTAL_LITERAL ::= "0" (["0"-"7"])*
FLOATING_POINT_LITERAL ::= (["0"-"9"])+ "." (["0"-"9"])*
                           (<EXPONENT>)? (["f","F","d","D"])?
                           | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
                           | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
                           | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
EXPONENT ::= ["e","E"] (["+","-"])? (["0"-"9"])+
CHARACTER_LITERAL ::= "'"
                      ( (~["'","\","\n","\r"])
                      | ("\" ( ["n","t","b","r","f","\","'","""]
                      | ["0"-'"7"] ( ["0"-"7"] )?
                      | ["0"-"3"] ["0"-"7"] ["0"-"7"] ) ) )
                      "'"
STRING_LITERAL ::= """
                   ( (~["""","\","\n","\r"])
                   | ("\"( ["n","t","b","r","f","\","'","""]
                   | ["0"-"7"] ( ["0"-"7"] )?
                   | ["0"-"3"] ["0"-"7"] ["0"-"7"] ) ) )*
                   """
```

B.4 Identifier

```
IDENTIFIER ::= LETTER ( LETTER | DIGIT )*
LETTER ::= [
           "\u0024",
           "\u0041"-"\u005a",
           "\u005f",
           "\u0061"-"\u007a",
           "\u00c0"-"\u00d6",
           "\u00d8"-"\u00f6",
           "\u00f8"-"\u00ff",
           "\u0100"-"\u1fff",
           "\u3040"-"\u318f",
           "\u3300"-"\u337f",
           "\u3400"-"\u3d2d",
```

```
                “\u4e00“-“\u9fff“,
                “\uf900“-“\ufaff“
                ]
DIGIT ::= [
            “\u0030“-“\u0039“,
            “\u0660“-“\u0669“,
            “\u06f0“-“\u06f9“,
            “\u0966“-“\u096f“,
            “\u09e6“-“\u09ef“,
            “\u0a66“-“\u0a6f“,
            “\u0ae6“-“\u0aef“,
            “\u0b66“-“\u0b6f“,
            “\u0be7“-“\u0bef“,
            “\u0c66“-“\u0c6f“,
            “\u0ce6“-“\u0cef“,
            “\u0d66“-“\u0d6f“,
            “\u0e50“-“\u0e59“,
            “\u0ed0“-“\u0ed9“,
            “\u1040“-“\u1049“
            ]
```

B.5 Trennzeichen

```
LPAREN ::= “(“
RPAREN ::= “)“
LBRACE ::= “{“
RBRACE ::= “}“
LBRACKET ::= “[“
RBRACKET ::= “]“
SEMICOLON ::= “;“
COMMA ::= “,“
DOT ::= “.“
AT := “@“
```

B.6 Spezielle Operatoren in JMLjass

```
IMPLIES ::= "==>"
IMPLIES_BACKWARDS ::= "<=="
IF_AND_ONLY_IF ::= "<==>"
NEG_IF_AND_ONLY_IF ::= "<=! =>"
```

ABSTR_FUNC ::= "− >"
ABSTR_FUNC_BACKWARDS ::= "< −"
DOT_DOT ::= ".."
NESTED_SPEC_BEGIN ::= "{|"
NESTED_SPEC_END ::= "|}"
SUBTYPE ::= "<:"

B.7 Operatoren in Java

ASSIGN ::= “=“
LT ::= “<“
GT ::= “>“
BANG ::= “!“
TILDE ::= “~“
HOOK ::= “?“
COLON ::= “:“
EQ ::= “==“
LE ::= “<=“
GE ::= “>=“
NE ::= “! =“
SC_OR ::= “||“
SC_AND ::= “&&“
INCR ::= “++“
DECR ::= “−−“
PLUS ::= “+“
MINUS ::= “−“
STAR ::= “∗“
SLASH ::= “/“
BIT_AND ::= “&“
BIT_OR ::= “|“
XOR ::= “^“
REM ::= “%“
LSHIFT ::= “<<“
PLUSASSIGN ::= “+ =“
MINUSASSIGN ::= “− =“
STARASSIGN ::= “∗ =“
SLASHASSIGN ::= “/ =“
ANDASSIGN ::= “& =“
ORASSIGN ::= “| =“
XORASSIGN ::= “^=“
REMASSIGN ::= “% =“
LSHIFTASSIGN ::= “<<=“
RSIGNEDSHIFTASSIGN ::= “>>=“

RUNSIGNEDSHIFTASSIGN ::= “>>>=“
ELLIPSIS ::= “...“

B.8 Spezielle Prädikate in JMLjass

PRED_UPPERCASE_TYPE ::= “\TYPE“
PRED_ELEMTYPE ::= “\elemtype“
PRED_EVERYTHING ::= “\“ “everything“
PRED_EXISTS ::= “\exists“
PRED_FRESH ::= “\fresh“
PRED_FORALL ::= “\forall“
PRED_INTO ::= “\into“
PRED_INVARIANT_FOR ::= “\invariant_for“
PRED_IS_INITIALIZED ::= “\is_initialized“
PRED_LOCK_SET ::= “\lockset“
PRED_MAX ::= “\max“
PRED_MIN ::= “\min“
PRED_NON_NULL_ELEMENTS ::= “\nonnullelements“
PRED_NOT_ASSIGNED ::= “\not_assigned“
PRED_NOT_MODIFIED ::= “\not_modified“
PRED_NOT_SPECIFIED ::= “\not_specified“
PRED_NOTHING ::= “\nothing“
PRED_NUM_OF ::= “\num_of“
PRED_OLD ::= “\old“
PRED_OTHER ::= “\other“
PRED_PRE ::= “\pre“
PRED_PRODUCT ::= “\product“
PRED_REACH ::= “\reach“
PRED_RESULT ::= “\result“
PRED_SUCH_THAT ::= “\such_that“
PRED_SUM ::= “\sum“
PRED_TYPE_OF ::= “\typeof“
PRED_TYPE ::= “\type“
PRED_WARN ::= “\warn“
PRED_WARN_OP ::= “\warn_op“

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Martin Schnaidt