JaWA: Java with Assertions

Clemens Fischer and Dieter Meemken

Universität Oldenburg, Fachbereich Informatik
Postfach 2503, 26111 Oldenburg
fischer@informatik.uni-oldenburg.de

Zusammenfassung Methoden zur Entwicklung von korrekter Software und deren Einführung in die Praxis stellen heute immer noch ein schwieriges Problem dar. Bertrand Meyer [Mey97a] hat unter dem Stichwort 'Programmieren mit Vertrag' einen Vorschlag gemacht und durch Eiffel implementiert, wie man formale Spezifikationsanteile mit Programmcode mischen und zur Laufzeit überprüfen kann. In diesem Paper geben wir einen Überblick über die Sprache JaWA, mit der dieses Konzept auf Java übertragen wurde.
JaWA Programme bestehen aus herkömmlichem Java-Code, der Zusicherungen (Invarianten, Vor- und Nachbedingungen) in Form von Kommentaren enthält. Der JaWA-Präcompiler übersetzt JaWA in Java. Er erzeugt zusätzliche Bedingungen, mit denen die spezifizierten Eigenschaften zur Laufzeit überprüft werden können.
Dabei wird die Java Ausnahmebehandlung ausgenutzt und um Eiffel ähnliche 'rescue' und 'retry' Anweisungen ergänzt. Die erzeugten Fehlermeldungen und Warnungen sind frei konfigurierbar.

1 Einleitung

Für die Entwicklung von Software mit sehr hohen Anforderungen an das Qualitätsmerkmal Korrektheit existieren kaum praktisch einsetzbare Lösungen und Werkzeuge. Dies steht in krassem Widerspruch zu den Anforderungen, die an heutige Softwaresysteme gestellt werden. Die Komplexität durch verteilte Architekturen, Anbindung an Fremdsysteme, das Nachladen von Modulen zur Laufzeit und nicht zuletzt die kurzen Lebenszyklen sind Aspekte, die in diesem Zusammenhang beachtet werden müssen.

Zur Überwindung dieser Schwierigkeiten wurden eine Reihe von Modellen vorgeschlagen, mit denen die Softwareentwicklung in verschiedene Phasen eingeteilt wird, und es wird an Methoden zur Lösung der Probleme in den verschiedenen Phasen gearbeitet. Dabei ist nahezu immer die Entwicklung einer Spezifikation – d. h. einer möglichst abstrakten Beschreibung des gewünschten Verhaltens – vorgesehen.

Ein schwieriges Problem ist es aber, den Nachweis zu führen, daß ein erstelltes Programm auch wirklich die gewünschte Spezifikation erfüllt. Den konsequentesten Vorschlag machen dabei *Formale Methoden*, mit denen man basierend auf einer formalen Sprache mathematisch beweisen kann, daß ein Programm eine gegebene Spezifikation implementiert. Allerdings konnten sich Formale Methoden in der Praxis bisher kaum durchsetzen – was sicherlich auch am hohen Aufwand der eingesetzten Verfahren liegt. In der Praxis wird daher häufig nur rudimentär getestet, ob das gewünschte Verhalten implementiert wurde. Selbst systematische Testverfahren werden häufig nicht als wirtschaftlich erachtet.

Bertrand Meyer hat mit dem Konzept des 'Programmierens mit Vertrag' [Mey97a] und dessen Realisierung in der Sprache Eiffel [Mey92] eine Möglichkeit gezeigt, wie eine formale Spezifikation direkt in die Implementierung aufgenommen werden kann. Dazu werden Zusicherungen in die Programme eingefügt, deren Gültigkeit zu fest definierten Zeitpunkten der Programmausführung geprüft werden, womit die frühe Erkennung von Fehlern verbessert wird. Je nach Umfang der eingesetzten Zusicherungen können Eigenschaften der Software formal bewiesen oder in Tests belegt werden. Zusätzlich ist es möglich, im Programm selbst auf die Verletzung von Zusicherungen zu reagieren und so die Robustheit der Software erheblich zu steigern.

Als Zusicherungen sind die folgenden Komponenten vorgesehen:

- Klasseninvarianten spezifizieren Eigenschaften, die zur gesamten Lebenszeit eines Objektes gelten müssen.
- Vorbedingungen sind Bedingungen, die vor der Ausführung einer Methode gelten sollen.
- Nachbedingungen spezifizieren die Zustände, die nach dem Abarbeiten einer Methode erreicht werden dürfen.
- Schleifeninvarianten müssen vor und nach jedem Schleifendurchlauf gelten.
- Varianten schließlich sind Terme, die bei jedem Schleifendurchlauf echt verkleinert werden müssen, aber keine negativen Werte annehmen dürfen. Auf diese Weise wird die Terminierung von Schleifen garantiert.
- Mit der Check-Anweisung kann man eine Überprüfung an einer beliebigen Stelle in den Programmcode einfügen.

Dabei sind Invarianten, Vor- und Nachbedingungen boolesche Ausdrücke über den Attributen und Methoden einer Klasse. Eine Variante ist ein Integerterm.

Der Ausdruck 'Programmieren mit Vertrag' leitet sich daraus her, daß symbolisch ein Vertrag zwischen Benutzer und Entwickler einer Methode geschlossen wird: Wenn der Benutzer die Vorbedingung einer Methode erfüllt, wird ihm die Einhaltung der Nachbedingung 'vertraglich' garantiert.

Das Prinzip, Eigenschaften von Programmen durch logische Formeln über dem Zustandsraum zu spezifizieren, geht auf grundlegende Arbeiten von Floyd [Flo67], Hoare [Hoa69] und Dijkstra [Dij76] zurück. Daraus hat sich die Methode entwickelt, Programme um logische Formeln zu erweitern, die immer gelten sollen, wenn der Kontrollfluß die entsprechende Stelle erreicht [Ros92,LH85]. In der Programmverifikation [AO97] versucht man zu beweisen, daß dies bei jeder möglichen Ausführung eines Programms gilt.

Meyer hat diese Ideen aufgegriffen, vermeidet aber die Komplexität logischer Beweise zugunsten verbesserter Testmöglichkeiten. Außerdem werden weitere pragmatische Konzepte – wie zum Beispiel die automatische Erzeugung von Dokumentationen aus den Zusicherungen und eine ausgefeilte Fehlerbehandlung – kombiniert [Mey97a].

Durch JaWA wird dieses Konzept auf Java übertragen. Ein JaWA-Programm enthält – zusätzlich zum normalen Java-Code – Kommentare, die Schlüsselworte wie `require` oder `ensure` und eine Liste von booleschen Bedingungen enthalten. Der JaWA Präcompiler übersetzt JaWA-Programme in Java. Dabei werden zusätzliche Abfragen eingefügt, wodurch die spezifizierten Eigenschaften zur Laufzeit überprüft werden. Der Präcompiler bietet verschiedene Modi an, so daß die Bedingungen evtl. auch nur teilweise oder gar nicht geprüft werden.

Mit dieser Vorgehensweise sind folgende Vorteile verbunden:

- Das JDK-Tool `javadoc` kann dazu genutzt werden, automatisch eine Dokumentation zu erstellen, die die formalen Spezifikationselemente enthält.
- Ein JaWA-Programm ist auch ein ausführbares Java-Programm, weil die zusätzlichen Anweisungen ausschließlich in Form von Kommentaren in den Programmtext eingefügt werden.
- JaWA kann schrittweise in die Softwareentwicklung eingeführt werden. JaWA und Java können beliebig gemischt werden. Z. B. besonders fehlerkritische Teile oder häufig benutzte Bibliotheken sollten durch den Einsatz von JaWA robuster programmiert werden, wohingegen man andere Komponenten herkömmlich implementieren kann.
- Die Einarbeitungszeit für Entwickler bleibt klein, weil bis auf einige Schlüsselwörter keine neue Syntax für Zusicherungen eingeführt wird.

Der Einsatz von JaWA bietet sich natürlicherweise immer dann an, wenn Korrektheit eine besonders große Rolle spielt. Weil aber die Hürde für die Benutzung von JaWA relativ gering ist, halten wir den Einsatz von JaWA generell für sinnvoll. Es bietet sich z. B. auch

an, bei der Zusammenarbeit mit anderen Programmiersprachen bessere Schnittstellendokumentationen zu erzeugen. So ist ein typisches Anwendungsgebiet von Java die Programmierung von intelligenten Eingabemasken in Client/Server-Anwendungen. Dabei werden bereits umfangreiche Konsistenzprüfungen vorgenommen, bevor eine Kommunikation mit dem Server, der häufig in einer anderen Programmiersprache implementiert ist, erfolgt. Mit JaWA können die erlaubten Datenwerte genau spezifiziert werden.

JaWA bietet darüberhinaus einen umfangreichen Mechanismus zum Abfangen von Ausnahmen, der den Möglichkeiten von Eiffel um nichts nachsteht. Daher kann die Überprüfung nicht nur beim Debuggen, sondern auch in einem Endprodukt benutzt werden, wenn es die Performance zuläßt. Dadurch steigt die Robustheit von Programmen.

Von Eiffel ist bekannt, daß der Overhead bei einer teilweisen Überprüfung der Zusicherungen bei ca. 20 % liegt [Mey97a]. Bei einer vollständigen Überprüfung liegt dieser Overhead deutlich höher. Ähnliche Erfahrungen haben wir mit JaWA auch gemacht.

Der Rest dieses Papers ist wie folgt aufgebaut: In den nächsten beiden Abschnitten wird JaWA beispielhaft erläutert und der Übersetzungsvorgang des Präcompilers erklärt. In Abschnitt 4 sind die Optionen des Präcompilers zusammengefaßt. Den Schluß bilden eine Diskussion verwandter Ansätze und Perspektiven für weitere Arbeiten.

In dieser Arbeit geben wir nur einen Überblick über JaWA. Eine vollständige Beschreibung findet man in [Mee97]. Die Sourcen des JaWA-Präcompilers und ein Benutzerhandbuch stehen im Internet unter

```
http://theoretica.informatik.uni-oldenburg.de/~jawa
```

zur Verfügung.

Wir begründen hier nicht, warum ‘Programmieren mit Vertrag’ ein sinnvolles Konzept ist. Argumente dafür findet man z. B. in [Mey97a,Mey97b].

2 Die Sprachelemente von JaWA

Als durchgehendes Beispiel für die Einführung von JaWA benutzen wir die Klasse `Stack` für einen Keller mit endlicher Kapazität von beliebigen Datenelementen. Sie ist intern als verkettete Liste organisiert.

2.1 Klasseninvarianten

Klasseninvarianten sind Bedingungen, die für die gesamte Klasse und nicht nur für einzelne Methoden gelten sollen. Sie werden bei jedem Methodenaufruf zu Beginn und an seinem Ende geprüft. Somit beschreiben sie den Zustandsraum, in dem sich jede Instanz dieser Klasse nach dem Abschluß einer Berechnung befinden muß. Klasseninvarianten werden durch das Schlüsselwort `invariant` eingeleitet und wie eine Methode am Ende der Klasse angegeben. In der Klasse `Stack` definiert die Invariante eine obere und eine untere Schranke für die Anzahl der Elemente:

```
public class Stack {
  private Linkable head;        // Kopf der Liste
  private int      items;       // Anzahl der Elemente
  ...
  /** invariant items>=0; items<=max **/
}
```

Die Variable `max` muß in der Umgebung von `Stack` definiert sein.

2.2 Vor- und Nachbedingungen

Für die einzelnen Methoden der Klasse `Stack` sollen die folgenden Bedingungen gelten :

- Jede Instanz der Klasse `Stack` ist anfangs leer.
- Die Methode `IsEmpty` liefert genau dann 'True', wenn der Keller leer ist, sonst 'False'. Der Keller wird nicht verändert.
- Die Methode `Push` zum Hinzufügen eines Elementes darf nur dann aufgerufen werden, wenn der Keller nicht voll ist. Das Element wird oben auf den Keller gelegt. Die Anzahl der Elemente erhöht sich um eins.
- Die Methode `Pop` zum Herunternehmen eines Elementes darf nur dann aufgerufen werden, wenn der Keller nicht leer ist. Die Anzahl der Elemente verringert sich um eins.
- Die Methode `Top` liefert das oberste Element des Kellers. Hierbei wird der Keller nicht verändert.

Vor- bzw. Nachbedingungen werden in JaWA durch die Schlüsselwörter `require` bzw. `ensure` eingeleitet. In Zusicherungen dürfen alle Attribute und Methoden der Klasse benutzt werden. Handelt es sich beim Typ eines Feldes selbst wieder um eine Klasse, so kann mit der Punktnotation auch auf dessen Attribute und Methoden zugegriffen werden. In der Nachbedingung kann auf den Wert eines Attributs *x* zu Beginn des Methodenaufrufs mit `old_`*x* zugegriffen werden. Damit können Veränderungen überprüft werden. Die Bedingung `nochange` besagt, daß kein Wert eines Attributs verändert worden ist. Auf den Rückgabewert einer Methode kann man durch `result` zugreifen.

Im folgenden JaWA-Programm sind die oben angeführten Bedingungen spezifiziert. Implementierungsdetails und die Methode `IsFull` haben wir aus Platzgründen weggelassen. Das vollständige Beispiel kann man in [Mee97] nachlesen.

```
public Stack(){
   head = null;
   items= 0;
   /** ensure items==0 **/
}
public boolean IsEmpty() {
   if( items==0 ) return true;
   else return false;
   /** ensure result==(items==0); nochange **/
}
public void Push(Object v) {
   /** require !IsFull(); v!=null **/
   ...
   /** ensure !IsEmpty(); items==old_items+1; v==Top() **/
}

public Object Pop() {
   /** require !IsEmpty() **/
   ...
   /** ensure !IsFull(); items==old_items-1; **/
}
public Object Top() {
   /** require !IsEmpty() **/
   ...
   /** ensure nochange **/
}
```

2.3 Schleifeninvarianten und -varianten

Eine Schleifeninvariante ist eine Bedingung, die durch die Ausführung der Schleife nicht verändert wird. Solche Invarianten spielen eine Schlüsselrolle, um sich von der Korrektheit von Schleifen zu überzeugen.

Eine Schleifenvariante ist ein ganzzahliger Ausdruck mit einem Wert größer als Null, der durch jede Schleifeniteration echt verringert wird. Auf diese Weise kann man die Terminierung von Schleifen garantieren.

Die folgende Schleife aus der Methode `Push` berechnet das letzte Element des Kellers.

```
Linkable actual = head;
while (i<items) {
  /* invariant items==old_items */
  /* variant items-i */
  actual = actual.GetNext(); i++;
}
```

Neben der While-Schleife existieren in Java noch die Do-Schleife und die For-Schleife. Beide können in analoger Weise mit Zusicherungen versehen werden.

2.4 Die Check-Anweisung

Die Check-Anweisung bietet die Möglichkeit, an einer beliebigen Stelle im Programmcode eine Zusicherung einzubauen. Sie entspricht der C-Anweisung `assert` und erweitert die Funktionalität eines einfachen, beschreibenden Kommentars zu einer prüfbaren und an den Ausnahmemechanismus angebundenen Anweisung.

Mit der folgenden Bedingung findet man heraus, ob im Keller noch Platz für zwei Elemente ist.

```
/* check items <= max - 2 */
```

3 Weitere Aspekte von JaWA

3.1 Vererbung

Wird von einer JaWA-Klasse ein Nachkomme abgeleitet, so erbt dieser auch dessen Klasseninvarianten. Es werden daher die Invarianten aller Väter und die eventuell neu hinzugefügten Invarianten überprüft.

Die Vererbung von Vor- und Nachbedingungen kann leider nicht auf ebenso einfache Weise durchgeführt werden. Gemäß der Regeln für Datenverfeinerung [Mor90] darf die Vorbedingung einer abgeleiteten Klasse abgeschwächt und die Nachbedingung verstärkt werden.

Die Einhaltung dieser Regel ist schwierig zu überprüfen, weil lokale Variablen durch Vererbung geändert werden können und daher geerbte Bedingungen unter Umständen nicht mehr auswertbar sind. In JaWA wird daher nur kontrolliert, ob in Nachkommen die entsprechenden Bedingungen syntaktisch vorhanden sind. Dieser Punkt soll in zukünftigen JaWA Versionen verbessert werden. In Eifel wird die Datenverfeinerung überhaupt nicht überprüft.

Alle weiteren Bedingungen sind Implementierungsinformationen und werden notwendigerweise nicht mehr berücksichtigt, wenn eine Methode bei Vererbung überschrieben wird.

3.2 Nebenläufigkeit

Zusicherungen bestehen aus mehreren booleschen Bedingungen, die als atomare Anweisung ohne Einflußmöglichkeiten von anderen nebenläufigen Prozessen geprüft werden müssen. Wenn z. B. ein anderer Prozeß die Zuweisung `y=z` während der Prüfung der Nachbedingung `/** ensure x==y; y==z **/` durchführen kann, wird keine Fehlermeldung ausgelöst, obwohl evtl. die Bedingung `y==z` verletzt war.

Deshalb werden alle Prüfungen durch den JaWA-Präcompiler automatisch in synchronised-Blöcke eingeschlossen. Allerdings müssen alle Methoden anderer nebenläufiger Klassen, die auf relevante Attribute zugreifen können, von Hand ebenfalls in synchronised Blöcke eingeschlossen werden, weil nur dann die Überprüfungen als atomarer Schritt durchgeführt wird. Für die Einhaltung dieser Bedingung ist der Benutzer selbst verantwortlich. Weitere Details zu diesem Problem sind in [Mee97, S. 83] nachzulesen.

3.3 Vertragsverletzung

Wie bereits erwähnt, werden bei verletzten Zusicherungen Ausnahmen ausgelöst. Diese können im *Rescue-Block* am Ende der Methode aufgefangen und behandelt werden. Bevor die Ausnahme an den Aufrufer weitergeleitet wird, können alle Datenattribute auf gültige Werte gesetzt werden. Somit bleibt die Instanz und damit letztlich das gesamte Programm stabil. Das folgende Beispiel zeigt eine Methode `Div`, in der eine ganzzahlige Division durchgeführt wird. Ist der Divisor gleich 0, so wird durch die Vorbedingung eine Ausnahme ausgelöst, die im Rescue-Block wieder aufgefangen wird und das Ergebnis auf einen definierten Wert setzt:

```
public void Div() {
  /** require z!=0 **/
  x =y/z;
  /* rescue catch (RuntimeCheck.AssertionException e)
     { x=1; }
  */
}
```

Ein Rescue-Block wird in eine try-catch-finally-Anweisung übersetzt, welche die gesamte Methode umschließt und generierte Ausnahmen auffängt.

An der Stelle, an der die ausgelöste Ausnahme wieder aufgefangen wird, kann in einer vom Benutzer festlegbaren Weise eine Fehlermeldung erzeugt werden. Dazu kann die Methode der Klasse für die Ausnahmebehandlung einen Fehlercode und/oder eine Fehlermeldung in Textform erzeugen. Diese können dann entsprechend bearbeitet und in beliebiger Form ausgegeben werden, z. B. direkt auf Standard-Out oder in Form einer Dialog-Box. Die Fehlerbehandlung kann sowohl global in der Hauptklasse der Applikation, als auch verteilt in den übrigen Klassen erfolgen.

Es existieren zwei Klassen für JaWA-Ausnahmen, je eine für geprüfte und ungeprüfte Ausnahmen. Von ihnen können für benutzereigene Erweiterungen weitere Klassen abgeleitet und an den bestehenden JaWA-Ausnahmemechanismus angebunden werden.

Um Ausnahmen ganz zu vermeiden, kann die *Retry-Anweisung* eingesetzt werden. Dadurch wird ein erneuter Versuch mit geänderten Werten unternommen, die Methode doch erfolgreich abzuarbeiten. Das folgende Beispiel erweitert die Methode `Div` um die Retry-Anweisung. Der Wert von `z` wird auf 1 gesetzt, und die Methode kann im zweiten Versuch nicht mehr scheitern.

```
public void Div() {
   /* check z!=0 */
   x =y/z;
   /* rescue catch (RuntimeCheck.AssertionException e)
      { z=1; retry; }
   */
}
```

Retry-Anweisungen werden in eine try-catch-finally-Anweisung übersetzt, die von einer While-Schleife umschlossen wird.

3.4 Dokumentation

Alle Zusicherungen können sowohl als normale Kommentare in /* */, als auch als Dokumentationskommentare in /** **/ geschrieben werden. Für die Übersetzung des JaWA-Präcompilers ist dies unerheblich. Der Unterschied besteht darin, daß letztere mit Hilfe von `javadoc` automatisch in die Dokumentation der Software aufgenommen werden. Zusicherungen wie Vorbedingungen, Nachbedingungen und Klasseninvarianten sollten als Dokumentationskommentare geschrieben werden, da sie die Funktionalität der Klasse beschreiben, während Check-Anweisungen, Schleifeninvarianten, Schleifenvarianten, Rescue-Blocks und Retry-Anweisung die Implementation beschreiben oder lenken. Sie sind als Hilfe für den Entwickler der Klasse selbst zu sehen und sollten lediglich als normale Kommentare erscheinen.

3.5 Einschränkungen

Gegenüber Java unterliegt JaWA nur der naheliegenden syntaktischen Einschränkung, daß Variablennamen, die vom Präcompiler erzeugt werden, nicht auch im Programm benutzt werden dürfen. Das betrifft die Namen `result, rescue, retry` und `depthOfCall` sowie alle Namen mit dem Präfix `old_` oder `variant`.

Semantisch ist zu beachten, daß in JaWA Invarianten nur bei Methodenaufrufen und nicht auch bei Zuweisungen an öffentliche Attribute überprüft werden. Dies wurde aus Effizienzgründen vermieden, weil sonst jede Zuweisung an globale Attribute durch einen synchronisierten Methodenaufruf realisiert werden müßte.

Das bedeutet aber, daß Invarianten durch Zuweisungen an öffentliche Attribute verletzt werden können, ohne daß eine entsprechende Fehlermeldung ausgelöst wird. Um diese Situation zu vermeiden, sollten daher Attribute, die in Invarianten benötigt werden – entsprechend dem Prinzip der Datenkapselung – nur mittels Methodenaufrufen zugreifbar sein.

Problematisch ist es auch, Eigenschaften zu überprüfen, die mehrere Klassen umfassen. Man muß dafür sorgen, daß die entsprechenden Bedingungen in jeder Klasse vorkommen, die die Eigenschaft verletzen könnte.

Die letzte Einschränkung betrifft die Mächtigkeit der Spezifikationssprache. Zum einen sind nur boolesche Ausdrücke und keine volle Prädikatenlogik mit Quantoren erlaubt. Daher können im Prinzip nicht alle interessanten Eigenschaften formuliert werden. Zum anderen sind aber in den Zusicherungen beliebige Methodenaufrufe möglich, wodurch unerwünschte Seiteneffekte erzeugt werden können.[1] Dies kann z. B. dadurch vermieden werden, daß alle in Bedingungen benutzten Methoden die Nachbedingung `nochange` erfüllen.

[1] Dadurch wäre es auch möglich, Quantoren zu implementieren und so die Ausdrucksstärke der Spezifikationssprache zu erweitern. Dies ist aber aus Performancegründen kaum empfehlenswert.

4 Konfiguration des JaWA-Präcompilers

Für den Präcompiler kann eine Konfigurationsdatei angelegt werden, die folgende Aspekte steuert:

- Der *Überprüfungsmodus* bestimmt, welche Teile von JaWA bei der Übersetzung berücksichtigt werden. Im Modus *All* werden alle Zusicherungen, wie oben beschrieben überprüft. Dies ist insbesondere während des Debuggings sinnvoll. Im Modus *Preconditions* werden nur Vorbedingungen bei Methodenaufrufen überprüft. Alle anderen Bedingungen, insbesondere auch die Invarianten, bleiben ungeprüft. Dies bietet sich an, wenn man davon ausgeht, daß das Programm korrekt arbeitet, und man nur überprüfen will, ob andere Systeme die erwarteten Vorbedingungen erfüllen. Der Modus Preconditions bietet sich daher besonders bei der Integration von fremder Software an. Der Overhead ist natürlich geringer als beim Modus All.
  Im Modus *Nothing* werden keine Überprüfungen vorgenommen. Der Präcompiler kopiert lediglich eine Datei `name.JaWA` in `name.java`. Dieser Modus wurde eingeführt, damit bei der Erzeugung von Endversionen, in denen aus Performancegründen keine Überprüfungen mehr stattfinden sollen, vorhandene Makefiles weiter genutzt werden können.
- Der *Übersetzungsmodus* bestimmt, wie auf die Verletzung einer Bedingung reagiert werden soll. Im Modus *Contract* wird eine geprüfte Ausnahme ausgelöst. Dafür ist es notwendig, daß der Entwickler die entsprechenden Methoden in einen `try`-Block einschließt. Im Modus *UncheckedContract* wird hingegen nur eine ungeprüfte Ausnahme erzeugt. Dabei ist es möglich, daß eine nicht abgefangene Ausnahme zur Terminierung des Programmes führt. Im Modus *Warning* schließlich werden nur Warnungen ausgegeben, es wird aber nicht in den Kontrollfluß eingegriffen.

Außerdem kann man die Klassen *ExceptionClass* und *WarningClass* frei definieren und somit ein entsprechendes Verhalten bei geprüften Ausnahmen bzw. den Text von Warnungen frei programmieren.

Der Präcompiler selbst wurde in C++ mit Hilfe von Bison und Lex programmiert. Eine Reimplementierung von JaWA in Java wird zur Zeit durchgeführt.

5 Diskussion

Ein kommerzielles Tool, daß in Konkurenz zu JaWA steht, ist AssertMate [Ass98] der Firma Reliable Software Technologies. Dieser Windows-basierte Java Präprozessor erlaubt die Spezifikation von Vor- und Nachbedingungen und Zusicherungen ähnlich wie es JaWA ermöglicht. Dabei wird neben einer booleschen Bedingung der Text der Fehlermeldung spezifiziert, der bei einer Verletzung erscheinen soll. Der Sprachumfang von AssertMate ist deutlich kleiner als der von JaWA: Invarianten von Klassen und Schleifen sowie Varianten werden nicht unterstützt, und es ist nicht möglich, in Nachbedingungen auf den alten Wert eines Attributs zuzugreifen. Ebenso werden nicht die ‘Rescue/Retry’ Möglichkeiten zur Ausnahmebehandlung von Eiffel implementiert. Daher ist JaWA wesentlich mächtiger als AssertMate.

Für JaWA bieten sich noch eine Reihe von Erweiterungsmöglichkeiten an. Durch die Portierung des Präcompilers nach Java kann eine größere Plattformunabhängigkeit erreicht werden.

Ebenso könnten die Warnungen bei einer möglichen Verletzung der in Abschnitt 3.5 gemachten Einschränkungen verbessert werden. Insbesondere beim Einsatz von Nebenläufigkeit bieten sich Erweiterungen an. Z. B. könnte ein Benutzer durch eine Datenflußanalyse auf mögliche Problemfälle aufmerksam gemacht werden. Die Datenflußanalyse könnte auch dazu genutzt werden, die eingefügten Bedingungen zu minimieren, um den Performanceoverhead klein zu halten.

Sehr interessant sind in diesem Zusammenhang auch Überlegungen, wo Zusicherungen sinnvoll plaziert werden können und wie die hier implementierten Möglichkeiten mit traditionellen Testanforderungen, (wie vollständige Pfadüberdeckung) kombiniert werden können [VK98].

Eine wichtige langfristige Perspektive ist schließlich die Kombination von JaWA mit Methoden der Programmverifikation. Z. B. könnten automatische Beweiswerkzeuge dazu eingesetzt werden, die Einhaltung zumindest einiger Bedingungen zu beweisen. Dadurch könnten Überprüfungen zur Laufzeit eingespart und gleichzeitig die Korrektheit von Programmen erhöht werden. Auch für das in Abschnitt 3.1 genannte Problem der Vererbung von Vor- und Nachbedingungen bietet sich der Einsatz von Beweiswerkzeugen an. Der Vorteil dabei wäre wiederum, daß man stärker mathematisch orientierte Methoden schrittweise in den Softwareentwurfsprozeß einführen kann, ohne revolutionäre Veränderungen der Arbeitsweise zu erzwingen.

Zum Verhältnis von JaWA und Eiffel ist zu sagen, daß JaWA im Prinzip – was das Konzept des 'Programmierens mit Vertrag' angeht – dieselben Möglichkeiten wie Eiffel bietet. Im Fall der Vererbung von Vor- und Nachbedingungen sind die Überprüfungen sogar erweitert. Damit wird ein wichtiges Argument entkräftet, das von der Eiffel-Community als Argument gegen die Benutzung von Java ins Feld geführt wird: Das Fehlen einer integrierten Methode zur Verbesserung der Softwarequalität [OO98].

Danksagung: Die Autoren bedanken sich bei D. Boles und E.-R. Olderog für wertvolle Kommentare bei der Entwicklung von JaWA. Für hilfreiche Vorschläge bei der Verbesserung dieses Papers danken wir D. Boles, J. Tapken und A. Berendes.

Literatur

[AO97] APT, K. R. und OLDEROG, E.-R.: *Verification of Sequential and Concurrent Programs*. Springer, 2te Auflage, 1997.

[Ass98] RELIABLE SOFTWARE TECHNOLOGIES: *AssertMate User Manual*, 1998. http://rstcorp.com/AMJava.html.

[Dij76] DIJKSTRA, E. W.: *A Discipline of Programming*. Prentice-Hall, 1976.

[Flo67] FLOYD, R.: *Assigning meaning to programs*. In: SCHWARTZ, J. T. (Herausgeber): *Mathematical Aspects of Computer Science*, Seiten 19–32, New York, 1967. AMS.

[Hoa69] HOARE, C. A. R.: *An axiomatic basis for computer programming*. Comm. ACM, 12:576–580, 1969.

[LH85] LUCKHAM, D. und HENKE, F. v.: *An overview of ANNA, a specification language for Ada*. IEEE Software, Seiten 9–22, 1985.

[Mee97] MEEMKEN, D.: *Programmieren mit Vertrag in Java*. Diplomarbeit, Universität Oldenburg, September 1997.

[Mey92] MEYER, B.: *Eiffel: The Language*. Prentice Hall, 1992.

[Mey97a] MEYER, B.: *Object-Oriented Software Construction*. ISE, 2. Auflage, 1997.

[Mey97b] MEYER, B.: *Put it in the contract: The lessons of Ariane*. Computer, 30(2):129–130, Jan 1997.

[Mor90] MORGAN, C.: *Programming from Specifications*. Prentice Hall, 1990.

[OO98] *Object-Oriented Languages: A Comparison*. http://www.eiffel.com/, 1998.

[Ros92] ROSENBLUM, D.: *Towards a method of programming with assertions*. In: *Proceedings of the 14th International Conference on Software Engineering*, Seiten 92–104, 1992.

[VK98] VOAS, J. und KASSAB, L.: *Using Assertions to Make Untestable Software More Testable*. Software Quality Professional, 1998. (noch nicht erschienen).