

Hauptseminar im Wintersemester 2001/2002

Nachweis von Sicherheitseigenschaften für JavaCard durch approximative Programmauswertung

TU–München Lehrstuhl IV
Prof. Tobias Nipkow, Ph.D.

Das ESC/Java Projekt, eine Übersicht

Jens Wilke

24. Januar 2002

Betreuer: Martin Strecker

Hauptseminar Nachweis von Sicherheitseigenschaften

Das ESC/Java Projekt

24. Januar 2002
Jens Wilke (wilke@jwdt.com)

Einführung

Das "Extended Static Checker for Java" Projekt von Compaq ist ein Programmierwerkzeug das mögliche Laufzeitfehler eines Java-Programms über eine statische Analyse der Codes erkennen kann. Benutzer können ihren Programmcode mit zusätzlichen Prüfbedingungen versehen.

Ähnlich wie die Typprüfung im Compiler sollen weitergehende Bedingungen verifiziert werden. Damit soll es möglich sein, typische Programmierfehler automatisiert zu erkennen. Die vollständige Programmverifikation ist aber nicht Ziel des Projektes.

Das vorliegende Papier soll eine Übersicht über das Projekt und die typische Anwendung geben.

Aufbau

Die weiteren Abschnitte sind wie folgt gegliedert: Nach einer groben Übersicht wird auf die einzelnen Komponenten des ESC/Java Tools näher eingegangen. Es folgt eine strukturierte Einführung in die Annotierungssprache JML (Java Modeling Language).

Die eingeführten Konstrukte werden dann anhand eines einfachen Beispiels verdeutlicht. Im anschließenden komplexeren Beispiel werden weitere Prüfmöglichkeiten näher erklärt, und außerdem wie man nach und nach die nötigen Programmannotierungen erarbeitet. Somit wird der Ablauf beim täglichen Einsatz des Tools klar.

Zum Abschluss wird noch kurz auf die Problematik Schleifen und Threads eingegangen und ein Fazit gezogen.

Eigenschaften von ESC/Java

Zur Prüfung der Java-Programme werden innerhalb Kommentarzeilen zusätzliche Annotierungen (genannt Pragmas) vorgenommen (siehe Beispiel Abbildung 4). Diese Pragmas enthalten Regeln, die zum einen Zusatzbedingungen für Methodenaufrufe sein können, also die API Definitionen ergänzen. Zum anderen können diese Regeln aber auch eine Beschreibung des Programmverhaltens angeben.

Die Prüfung ist modular. Das bedeutet in diesem Fall, dass Methode für Methode einzeln geprüft wird. Dadurch ist es möglich, die Prüfung inkremental bzw. in Ausschnitten vorzunehmen. Falls man Fremdsoftware einsetzt, ist es nicht unbedingt erforderlich, deren Source-Code zu besitzen. Über den modularen Ansatz ist es möglich in Spezifikations-Files Verifizierungsbedingungen für Fremd-APIs anzugeben. Wie wir später noch an den Beispielen sehen werden, bringt die Modularität aber auch Nachteile mit sich: Offensichtliche Eigenschaften, die methodenübergreifend gelten, müssen vom Programmierer selbst angegeben werden (Objekt-Invarianten).

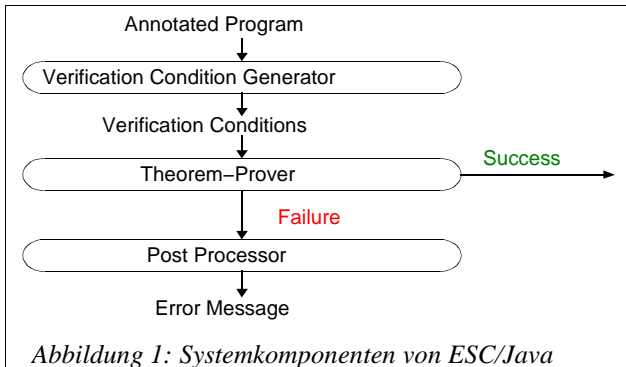
Vom ESC/Java werden nahezu alle Sprachmerkmale von Java 1.2 unterstützt. Außerdem werden für alle gängigen Klassen aus der JDK (z.B. String und Hashtable) fertige Spezifikationsfiles mitgeliefert.

Die Benutzung des ESC/Java ist ähnlich wie beim Java-Compiler. Das Tool ist zeilenorientiert und wird mit dem Namen der zu prüfenden Datei aufgerufen. Als Ergebnis erhält man Fehlermeldungen, die ebenfalls ähnlich wie bei einem Java-Compiler die Zeilennummer,

einen Quelltextausschnitt und eine Warnmeldung enthalten (siehe Abbildung 5).

Systemkomponenten

Das ESC/Java System besteht aus drei Komponenten: "Verification Condition Generator", Theorem-Beweiser und "Post Processor". Wie man aus Abbildung 1 entnehmen kann, ist der Nachrichtenfluss genau in dieser Aufzählungsreihenfolge. Der "Verification Condition Gene-



rator" wandelt ein annotiertes Java-Programm in zu beweisende Regeln um. Im nächsten Verarbeitungsschritt versucht der Theorem-Beweiser diese Regeln zu verifizieren. Wird kein Widerspruch festgestellt, so geschieht keine weitere Ausgabe. Kommt es zu einem Widerspruch, wird die Ausgabe von einem "Post Processor" weiterverarbeitet, um für den Benutzer brauchbare

einem separaten Projekt, genannt JML (Java Modeling Language, siehe [JML]), spezifiziert.

Das ESC/Java Projekt ist eines von mehreren Projekten, die sich mit der Verifikation von JML-annotierten Programmen befassen. Interessant sind zum einen die Ansätze von Leavens, der die Pragmas in Java-Code umwandelt und dann in Kombination mit einer JUnit Testbench dynamische Tests durchführt. D.h. das Programm wird tatsächlich mit vorprogrammierten Testfällen ausgeführt. Die JML-Annotationen dienen als zusätzliche Integritätsbedingungen im zu testenden Code. Genau das andere Extrem ist das LOOP-Projekt von Jacobs. Hier wird versucht, mittels JML eine vollständige Verifikation des Programmes zu bewerkstelligen. Links zu diesen Projekten findet man auf der JML-Homepage.

Verification Condition Generator

Basierend auf Konvertierungsregeln werden über mehrere Zwischensprachebenen die relativ mächtigen Java-Konstrukte in einfache, primitive Kommandos übersetzt. Die Semantik dieser Kommandos wird über die "weakest liberal precondition" (Eingeführt von Dijkstra) definiert. Für jeden Befehl C und die Prädikate N, X und W (die den Status nach der Ausführung wiedergeben), ergibt $wlp.C.(N, X, W)$ den Wert N, X

```

wlp.( v=e ).(N,X,W) = N[ v <- e ]
wlp.skip.(N,X,W)   = N
wlp.raise.(N,X,W)  = X
wlp.(assert e).(N,X,W) = (e && N) || (!e && W)
wlp.(C1; C1).(N,X,W) = wlp.C0.(wlp.C1(N,X,W),X,W)
  
```

Abbildung 2: Umsetzung der Weakest Liberal Precondition von Dijkstra

Fehlermeldungen zu generieren.

Annotierungen des Java-Programms

Java-Programme werden mit sog. Pragmas annotiert. Diese Pragmas werden nach einer bestimmten Konvention innerhalb des Kommentars eines Programms eingefügt; d.h. für den normalen Java-Compiler sind sie nicht sichtbar. Der Sprachumfang dieser Annotierungen wurde in

oder W, je nachdem ob die Ausführung normal terminiert (N), eine Exception produziert (X) oder eine Prüfbedingung nicht einhält (W).

In Abbildung 2 sieht man einen Ausschnitt der verwendeten Ausdrücke und deren Umsetzung in Logik.

Die Methode wird mit dem Ausdruck "BP => wlp.C.(true,true,false)" verifiziert. BP

ist das sog. "Background Predicate". Es enthält z.B. die Vererbungshierarchie der verwendeten Java-Klassen (für Untertypentests) und generelle Regeln der Java-Programmiersprache.

Vereinfachung, die nur für diesen Theorem-Beweiser gilt, denn bei Einbeziehung der Multiplikation werden die Beweise erheblich aufwendiger.

Ein Beispiel dafür wie eine Verifizierungsbedingung gebildet wird, gibt Abbildung 3. Eine Zusicherung und eine Zuweisung werden in einen zu verifizierenden Ausdruck umgewandelt. Das Beispiel ist stark vereinfacht, da in Wahrheit

Fehlerausgabe

In der Fehlerausgabe erhält man, neben der Information welche Prüfbedingung fehlgeschlagen

```
void xy(int num) {
    //@ assert num>5;
    cnt = num;
}

wlp.(assert num>5; cnt = num).(N,X,W) =
    (num > 5) && N[ cnt <- num ] || (!(num > 5) && W)
```

Abbildung 3: Beispielumwandlung einer Java-Methode in eine Verifizierungsbedingung

noch weitere Verifizierungsbedingungen hinzukommen, z.B. für Objekt-Invarianten (wird später erklärt). Außerdem werden noch Labels eingefügt, die es später ermöglichen, im Postprozessor die Quellcode-Zeilenummer einer fehlgeschlagenen Bedingung zu extrahieren.

ist, auch die entsprechende Sourcecodestelle angezeigt.

Der Theorem-Beweiser

Der Theorem-Beweiser (genannt Simplify) wurde speziell auf das Einsatzgebiet der Programmverifikation zugeschnitten. Es gibt automatisierte Entscheidungsprozeduren, die für Funktionen und Prädikate in der Programmierung wichtig sind (z.B. Gleichheit und Arithmetik). Außerdem wurde besonderes Augenmerk darauf gerichtet, dass aus fehlgeschlagenen Beweisen brauchbare Fehlermeldungen extrahiert werden können.

Es ist ebenfalls möglich, noch zusätzliche Hinweise in der Fehlerausgabe zu aktivieren. Z.B. gibt ESC/Java mit der Kommandozeilenoption `-suggest` Vorschläge aus, welche Pragmas zur Beseitigung von Warnmeldungen führen könnten.

Die essenzielle Eigenschaft des Theorem-Beweisers Simplify für das ESC/Java Projekt ist aber, dass Beweise komplett ohne manuelle Eingriffe durchgeführt werden.

Für unsere Beispiele wurden immer die Parameter `-notrace` und `-quiet` benutzt. Dadurch wird die Ausgabe übersichtlicher und enthält keine unnötigen Informationen wie die Ausführungszeit des Programms.

Aber auch Kompromisse wurden eingegangen: Die mathematische Induktion wird nicht unterstützt. Der Theorem-Beweiser hat keine eingebaute Semantik für Multiplikation. Das Weglassen der Multiplikation ist allerdings keine

Annotierungen des Java-Programms

Wie schon erwähnt werden die Java-Programme mit sog. Pragmas annotiert. Diese Pragmas werden nach einer bestimmten Konvention innerhalb des Kommentars eines Programms eingefügt. Die Pragmas müssen dabei am Anfang einer Kommentarzeile stehen und mit `@` beginnen. Es ist möglich, mehrere Pragmas innerhalb eines Kommentars mit Semikolon getrennt anzugeben.

Im folgenden soll eine knappe, strukturierte Einführung der unterstützten Pragmas gegeben

werden. Auf ein ausführliches Beispiel für jedes Pragma wurde an dieser Stelle verzichtet. Durch die Anwendungsbeispiele im nächsten Kapitel sollte aber der Verwendungszweck klar werden.

Ausdrücke

Ausdrücke innerhalb der Annotierungs-Pragmas sind analog zu den Java-Ausdrücken. Auch auf Array-Felder und Längen kann zugegriffen werden, z.B.: `content.length < 20`.

Es gibt spezielle Ausdrücke zum Testen von Typen:

\type(Typ t) – bildet aus einem Java-Typ ein entsprechendes Typ-Konstrukt für den Theorem-Beweiser

t1 <: t2 – testet auf Subtyp: Wahr wenn t1 ein Subtyp (oder gleich) t2 ist

\typeof(Objekt o) – gibt den Typ eines Objektes zurück

\elemtype(Typ t) – extrahiert den Element-Typ aus einem Array-Typ

Außerdem werden die Java-Ausdrücke auch noch um weitere Logik-Ausdrücke erweitert:

b1 ==> b2 – die Implikation

(\exists Typ x; e) – Existenzquantor z.B. `(\exists int i; A[i]==4711)`

(\forall Typ x; e) – Allquantor z.B. `(\forall int i; (0 <= i && i < A.length) ==> A[i]!=null)`

Bei den Quantoren ist auch die Benutzung von Referenztypen erlaubt. In diesem Fall sind ein bzw. alle Objekte dieses Typs gemeint.

Pragmas

Im Kommentar innerhalb der Klassendeklaration können folgende Pragmas angegeben werden:

non_null – vor der Variablendeklaration eines Referenztyps. Gibt an dass diese Variable nicht uninitialized bleiben darf und nie den Wert null zugewiesen bekommen darf

invariant e – Objekt-Invariante: Während der gesamten Objektlebenszeit muss diese Bedingung erfüllt sein

Im Kommentar vor einer Methode gibt es diese Pragmas:

requires e – Notwendige Bedingung vor der Ausführung dieser Methode

ensures e – Bedingung der nach erfolgreicher (ohne Exception) Beendigung der Methode gelten muss

exsures e – Bedingung die bei Beendigung der Methode mit Exception gilt

Die Annotierungen in der Klasse wie auch im Methodenkopf werden bei der Verifikation von Programmen, die die so annotierten Klassen verwenden, ebenfalls berücksichtigt.

Im Methodenrumpf können diese Pragmas verwendet werden:

assert e – Bedingung muss an dieser Stelle erfüllt sein

unreachable – bei der Ausführung dürfen wir diese Stelle nie erreichen

loop_invariant e – definiert eine Schleifeninvariante

assume e – gibt an, dass an dieser Stelle das Prädikat e gilt. Dies wird verwendet wenn sich der Programmierer sicher ist, dass eine bestimmte Bedingung erfüllt ist, der Beweiser das aber nicht selbst herausfinden kann (z.B. durch bestimmte Verwendung von anderen Methoden)

nowarn t – unterdrückt einfach die Warnmeldung eines bestimmten Typs

Die beiden letzten Pragmas sind dafür zuständig über Unzulänglichkeiten des ESC/Java Tools hinwegzuhelfen. Natürlich sollte dem Pragma `assume` Vorrang gegeben werden, da das angegebene Prädikat auch für die weitere Verifizierung verwendet wird.

Zwei Pragmas seien noch erwähnt:

ghost n – Deklariert eine Ghost-Variable (in der Klassendeklaration)

set n=e – Setzt den Wert einer Ghost-Variable

Ghost-Variablen sind Variablen, die es nur innerhalb des Beweisers gibt. Dadurch ist es möglich, weiterreichende Verifizierungskriterien einzuführen. So z.B. besitzt jedes Objekt die Ghost-Variable `owner`. Mit dieser Variable kann jedem Objekt ein exklusiver Besitzer zugeordnet werden (Notwendig bei Member-Variablen eines Objektes). Ein ausführliches Beispiel für den Einsatz gibt es später.

Fehlermeldungen und Warnhinweise

Schlägt der Test einer Bedingung im Pragma fehl, so wird eine Warnmeldung ausgegeben. ESC/Java hat aber auch implizite Tests: Beim Methodenaufruf z.B. mit `s.toString()` darf `s` kein `NullPointerException` sein oder beim Arrayzugriff mit `a[i]` muss sichergestellt sein, dass sich `i` innerhalb der Grenzen des Arrays bewegt. Das bedeutet das bestimmte `Runtime-Exceptions` nicht auftreten dürfen (im Fall von oben ist das `NullPointerException` und `ArrayIndexOutOfBoundsException`), bzw. durch entsprechende Pragma-Deklarationen notwendige Bedingungen hinzugefügt werden müssen, die diese `Exceptions` verhindern.

Insgesamt wird auf folgende `Exceptions` geprüft: `ArrayIndexOutOfBoundsException`, `NullPointerException`, `ClassCastException`, `DivisionByZeroException`, `ArrayStoreException`, `NegativeArraySizeException`.

Tritt eine solche `Exception` auf, so gilt dies als Fehlerbedingung. Eine mögliche Fehlerbehandlung einer solchen `Exception` mittels `catch` wird nicht berücksichtigt.

Beispiel Bag.java

In Abbildung 4 sieht man ein typisches Java-Programm. Die Abbildung 5 enthält die entsprechenden Warnmeldungen von ESC/Java.

Die ersten zwei und die vierte Warnmeldung deuten eine mögliche `NullPointerException` an. Um die erste Warnmeldung zu beseitigen, müssen wir für die aufrufenden Programme fordern, dass diese niemals eine `Null-Referenz` übergeben. Dies geschieht mit dem Pragma `requires input!=null` im Kopf der Methode. Die Alternative wäre, dass das Programm um eine sinnvolle Fehlerbehandlungsroutine für diesen Fall ergänzt wird.

Die weiteren Warnungen bzgl. einer `Null-Referenz` betreffen die Member-Variable `a`. Über die Angabe des Pragmas `non_null` geben wir an, dass diese Variable stets mit einem gültigen `Array-Objekt` initialisiert ist. Diese Bedingung wird ebenfalls in jeder Methode überprüft; genauer gesagt: ESC/Java stellt nun sicher, dass im Konstruktor ein brauchbarer (`non-null`) Wert zugewiesen wird und dass bei keiner Zuweisung in allen Methoden dieser Wert mit `null` überschrieben wird.

Bleiben zwei Warnmeldungen, die auf einen möglichen Fehler bei der `Array-Indizierung` hinweisen. Bei der ersten Warnung können wir tatsächlich einen typischen Fehler feststellen: Die `for-Schleife` läuft nicht beim `Index 0` los, sondern beim `Index 1`. Bei jedem Aufruf von `extractMin` wird ein Wert zurückgegeben und die Anzahl der verbleibenden Elemente (die Variable `n`) um eins verringert. Also kann es sein, dass wir in Zeile 21 ins `Negative` laufen. Dies ist dann der Fall, wenn alle Elemente des `Initialisierungs-Array input` von `extractMin` zurückgeliefert wurden. Um diesen Fall zu verhindern, wurde entschieden eine `Exception` zu generieren.

Version 0 von Bag.java:

```
1: class Bag {
2:     int[] a;
3:     int n;
4:
5:     Bag(int[] input) {
6:         n = input.length;
7:         a = input;
8:     }
9:
10:
11:     int extractMin() {
12:         int m = Integer.MAX_VALUE;
13:         int mindex = 0;
14:         for (int i = 1; i <= n; i++) {
15:             if (a[i] < m) {
16:                 mindex = i;
17:                 m = a[i];
18:             }
19:         }
20:         n--;
21:         a[mindex] = a[n];
22:         return m;
23:     }
24:
25: }
```

Version 1 von Bag.java:

```
1: class Bag {
2:     /*@non_null*/ int[] a;
3:     int n;
4:
5:     //@requires input!=null;
6:     Bag(int[] input) {
7:         n = input.length;
8:         a = input;
9:     }
10:
11:     //@invariant 0 <= n & n <= a.length;
12:
13:     int extractMin() throws Exception {
14:         if (n<=0) {
15:             throw new Exception("Nix mehr drin!");
16:         }
17:         int m = Integer.MAX_VALUE;
18:         int mindex = 0;
19:         for (int i = 0; i < n; i++) {
20:             if (a[i] < m) {
21:                 mindex = i;
22:                 m = a[i];
23:             }
24:         }
25:         n--;
26:         a[mindex] = a[n];
27:         return m;
28:     }
29: }
```

Abbildung 4: Eine einfaches Beispielprogramm. Bag implementiert eine Multimenge von Integerzahlen. Mit der Methode extractMin() wird nacheinander jeweils das kleinste Element der Menge zurückgegeben.

Mit diesen Änderungen sind wir aber immer noch nicht an unserem Ziel – keine Warnmeldungen mehr zu erhalten – angekommen. Die Warnmeldung aus Zeile 15 `IndexTooBig` bleibt noch bestehen. Warum ist die `IndexNegative` Meldung verschwunden, aber diese Warnmel-

Warning: Type of right-hand side possibly not a subtype of array element type (ArrayStore). Das heißt ESC/Java sieht nicht sichergestellt, dass in der `content-Variable` tatsächlich auch immer ein `Objekt-Array` ist.

```

$ escjava -quiet -notrace Bag.java

Bag.java:6: Warning: Possible null dereference (Null)
  n = input.length;
      ^
Bag.java:15: Warning: Possible null dereference (Null)
  if (a[i] < m) {
      ^
Bag.java:15: Warning: Array index possibly too large (IndexTooBig)
  if (a[i] < m) {
      ^
Bag.java:21: Warning: Possible null dereference (Null)
  a[mindex] = a[n];
      ^
Bag.java:21: Warning: Possible negative array index (IndexNegative)
  a[mindex] = a[n];
      ^

5 warnings

```

Abbildung 5: Warnmeldungen bei der Überprüfung der Version 0 von `Bag.java`

dung nicht? Der Grund liegt darin begründet, dass ESC/Java innerhalb der Methode nachweisen kann, dass `n` nicht negativ wird. Allein vom Code dieser Methode ist aber nicht ersichtlich, dass `n` nie größer als die `Array-Länge` wird. Um diese `methodenübergreifende` Beziehung ESC/Java klar zu machen (und diese auch sicherzustellen), ist es notwendig eine `Objekt-Invariante` zu definieren. Siehe Zeile 11 in `Version 1` von `Bag.java`.

Ein komplexes Beispiel

Anhand eines ausführlicheren Beispiels soll nun die Anwendung von weiteren `JML-Pragmas` verdeutlicht werden.

In `Abbildung 6` sieht man den Code einer `Stack-Datenstruktur`. Die typischen ESC/Java Warnmeldungen, wie wir sie aus dem Beispiel `Bag.java` kennen, sind in der `Version 1` bereits durch die Annotierungen abgestellt. Neu ist allerdings die `Invariante` in Zeile 7. Warum wird sie benötigt? Lassen wir dieses `Pragma` weg, so bekommen wir die Meldung: `Stack.java:15:`

Da dies eine `Feinheit` von Java ist, die scheinbar nicht allgemein geläufig ist, sei hier nochmal kurz auf die `ArrayStoreException` eingegangen. Jedes `Array-Objekt` hat zur Laufzeit einen `Elementtyp`. Durch das `Typenkalkül` von Java wird von einem `Array` mit `Elementtyp` von z.B. `Number` (der Einfachheit halber `Number-Array` genannt) erwartet, dass es nur `Objekte` vom Typ `Number` oder `Subtypen` beim `Elementzugriff` zurückgibt. Die `Sicherstellung` dafür geschieht beim `Schreiben` in das `Array` (`Typprüfung` des zugewiesenen Wertes) und muss zur `Laufzeit` durchgeführt werden. In `Abbildung 7` sieht man einen `fehlerhaften Code`, bei dem diese `Typprüfung` fehlschlägt. Wichtig ist, dass man sich den `Unterschied` des `statischen Elementtyps` der `Variable a` (dies ist `Objekt`) und dem `dynamischen Elementtyp` des `Arrays` (dies ist `String`) veranschaulicht.

Version 0 von Stack.java:

```
1: public class Stack {
2:
3:     int size;
4:
5:     Object content[] = new Object[5];
6:
7:     public void push(Object o) {
8:         if (size>=content.length) {
9:             Object[] oa = new Object[size+5];
10:            System.arraycopy(content,0,oa,0,size);
11:            content = oa;
12:        }
13:        content[size++] = o;
14:    }
15:
16:    public Object pop() {
17:        return content[--size];
18:    }
19:
20:    public int size() {
21:        return size;
22:    }
23:
24: }
```

Version 1 von Stack.java:

```
1: public class Stack {
2:
3:     int size;
4:
5:     /*@non_null*/ Object content[] = new Object[5];
6:     /*@ invariant size >= 0 && size <= content.length;
7:     /*@ invariant \elemtype(\typeof(content)) == \type(Object);
8:
9:     public void push(Object o) {
10:        if (size>=content.length) {
11:            Object[] oa = new Object[size+5];
12:            System.arraycopy(content,0,oa,0,size);
13:            content = oa;
14:        }
15:        content[size++] = o;
16:    }
17:
18:    public Object pop() throws StackEmptyException {
19:        if (size<=0) {
20:            throw new StackEmptyException();
21:        }
22:        return content[--size];
23:    }
24:
25:    public int size() {
26:        return size;
27:    }
28:
29:    public static class StackEmptyException extends RuntimeException {}
30:
31: }
```

Abbildung 6: Version 0 und 1 einer einfachen Stack Implementierung. In der Version 1 wurden schon die typischen Annotierungen, wie wir sie aus dem Beispiel mit Bag.java kennen, vorgenommen.

```

1: public class ArrayStore {
2:
3:     public static void main(String[] arg) {
4:         Object[] a = new String[20];
5:         a[5] = new Object();
6:     }
7:
8: }

```

\$ java ArrayStore

```

java.lang.ArrayStoreException:
    at ArrayStore.main(ArrayStore.java:5)

```

Abbildung 7: Beispiel für das Zustandekommen einer ArrayStoreException.

```

1: public class Stack {
2:
3:     int size;
4:
5:     /*@non_null*/ Object content[] = new Object[5];
6:     //@ invariant size >= 0 && size <= content.length;
7:     //@ invariant \elemtype(\typeof(content)) == \type(Object);
8:
9:     //@ ensures size == 0;
10:    public Stack() {
11:    }
12:
13:    //@ modifies size
14:    //@ ensures size == \old(size) + 1;
15:    public void push(Object o) {
16:        if (size >= content.length) {
17:            Object[] oa = new Object[size+5];
18:            System.arraycopy(content, 0, oa, 0, size);
19:            content = oa;
20:        }
21:        content[size++] = o;
22:    }
23:
24:    //@ modifies size
25:    //@ ensures size > 0 ==> size == \old(size) - 1;
26:    public Object pop() throws StackEmptyException {
27:        if (size <= 0) {
28:            throw new StackEmptyException();
29:        }
30:        return content[--size];
31:    }
32:
33:    //@ ensures \result == size;
34:    public int size() {
35:        return size;
36:    }
37:
38:    public static class StackEmptyException extends RuntimeException {}
39:
40: }

```

Abbildung 8: Version 2 von Stack.java enthält weitere Pragmas, die die Größenänderung modelliert.

Verhaltensmodellierung

Um weitere Programmierfehler entdecken zu können, müssen wir ESC/Java mehr über das gewünschte Programmverhalten mitteilen. Dazu führen wir weitere Pragmas ein, die die Größenveränderung unseres Stacks beschreiben.

Die nächste Version vom Stack Beispiel ist in Abbildung 8. Über das `ensures`-Pragma spezifizieren wir die Zustände nach Beendigung der Methode. Neu ist die Funktion `\old(x)`, die

dem Wert einer Variable vor dem Methodenaufruf entspricht. Damit ist es möglich, Zustandsänderungen zu modellieren. Neu ist auch `\result`, dies ist der Return-Wert eines Methodenaufrufes (Siehe Zeile 33).

Eine weitere Verfeinerung unserer Stack-Datenstruktur in Abbildung 9 enthält ein Beispiel, wie Ghost-Variablen zu verwenden sind. Bei abstrakten Datenstrukturen wie z.B. Stack, Streuspeicher oder Liste hat man in Java das Problem, dass diese nicht über den Elementtyp

```
1: public class Stack {
2:
3:     /*@ ghost public \TYPE elementType;
4:
5:     int size;
6:
7:     /*@non_null*/ Object content[];
8:     /*@ invariant size >= 0 && size <= content.length;
9:     /*@ invariant \elemtype(\typeof(content)) == \type(Object);
10:    /*@ invariant (\forall int i; (0 <= i && i < size)
11:        ==> \typeof(content[i]) <: elementType); */
12:    /*@ invariant content.owner == this;
13:
14:    /*@ ensures size == 0;
15:    public Stack() {
16:        content = new Object[5];
17:        /*@ set content.owner = this;
18:    }
19:
20:    /*@ modifies size
21:    /*@ ensures size == \old(size) + 1;
22:    /*@ requires \typeof(o) <: elementType
23:    public void push(Object o) {
24:        if (size >= content.length) {
25:            Object[] oa = new Object[size+5];
26:            /*@ set oa.owner = this;
27:            System.arraycopy(content, 0, oa, 0, size);
28:            content = oa;
29:        }
30:        content[size++] = o;
31:    }
32:
33:    /*@ modifies size
34:    /*@ ensures size > 0 ==> size == \old(size) - 1;
35:    /*@ ensures \typeof(\result) <: elementType
36:    public Object pop() throws StackEmptyException {
37:        if (size <= 0) {
38:            throw new StackEmptyException();
39:        }
40:        return content[--size];
41:    }
42:
43:    /*@ ensures \result == size;
44:    public int size() {
45:        return size;
46:    }
47:
48:    public static class StackEmptyException extends RuntimeException {}
49:
50: }
```

Abbildung 9: Version 3 von Stack.java legt den Elementtyp der Datenstruktur fest

parametrisiert werden können, so wie das in C++ mit Templates oder mit der Generic Types Java-Erweiterung möglich ist. D.h. allgemeine Datenstrukturen können nur mit dem Elementtyp Objekt arbeiten und wir sind gezwungen, bei jedem Zugriff einen Cast auf den wirklich ver-

wendeten Typ zu machen. Siehe die Version 0 von `Rechner.java` Zeile 14 in Abbildung 10. Da wir in der Programmverifikation auch nachweisen wollen, dass dieser Cast tatsächlich erfolgreich ist, müssen wir unserer Datenstruktur

Version 0 von Rechner.java:

```

1: public class Rechner {
2:
3:     Stack stack;
4:
5:     public Rechner() {
6:         stack = new Stack();
7:     }
8:
9:     public void ablegen(float f) {
10:         stack.push(new Float(f));
11:     }
12:
13:     public void addieren() throws RuntimeException {
14:         Float f1 = (Float) stack.pop();
15:         Float f2 = (Float) stack.pop();
16:         stack.push(new Float(f1.floatValue()+f2.floatValue()));
17:     }
18:
19:     public float ergebnis() throws RuntimeException {
20:         Float f = (Float) stack.pop();
21:         return f.floatValue();
22:     }
23:
24: }

```

Version 1 von Rechner.java:

```

1: public class Rechner {
2:
3:     /*@non_null*/ Stack stack;
4:     //@ invariant stack.elementType == \type(Float);
5:
6:     public Rechner() {
7:         stack = new Stack();
8:         //@ set stack.elementType = \type(Float);
9:     }
10:
11:     public void ablegen(float f) {
12:         stack.push(new Float(f));
13:     }
14:
15:     public void addieren() throws RuntimeException {
16:         Float f1 = (Float) stack.pop();
17:         Float f2 = (Float) stack.pop();
18:         stack.push(new Float(f1.floatValue()+f2.floatValue()));
19:     }
20:
21:     public float ergebnis() throws RuntimeException {
22:         Float f = (Float) stack.pop();
23:         return f.floatValue();
24:     }
25:
26: }

```

Abbildung 10: Ein einfacher UPN-Rechner soll die Verwendung der Stack-Struktur demonstrieren

beibringen welchen Elementtyp sie enthält. Dies ist genau ein Einsatzgebiet der Ghost-Variable. In Zeile 3 der Abbildung 9 deklarieren wir uns eine zusätzliche Variable für den Beweisvorgang, die den Elementtyp des Stacks enthält wird – insofern dieser bekannt ist. Nun können

nur über die `push()`-Methode in unseren Stack wandern.

Noch nicht erwähnt sind die zusätzlichen Pragmas in Zeile 12, 17 und 26. Diese sind notwendig da wir in Zeile 10 eine Invariante

```

$ escjava -quiet -notrace Rechner.java

Rechner.java:10: Warning: Possible null dereference (Null)
    stack.push(new Float(f));
           ^
Rechner.java:14: Warning: Possible null dereference (Null)
    Float f1 = (Float) stack.pop();
                        ^
Rechner.java:14: Warning: Possible type cast error (Cast)
    Float f1 = (Float) stack.pop();
                ^
Rechner.java:15: Warning: Possible type cast error (Cast)
    Float f2 = (Float) stack.pop();
                ^
Rechner.java:16: Warning: Possible null dereference (Null)
    stack.push(new Float(f1.floatValue()+f2.floatValue()));
                    ^
Rechner.java:16: Warning: Possible null dereference (Null)
    stack.push(new Float(f1.floatValue()+f2.floatValue()));
                    ^
Rechner.java:20: Warning: Possible null dereference (Null)
    Float f = (Float) stack.pop();
                        ^
Rechner.java:20: Warning: Possible type cast error (Cast)
    Float f = (Float) stack.pop();
                ^
Rechner.java:21: Warning: Possible null dereference (Null)
    return f.floatValue();
           ^
9 warnings

```

Abbildung 11: Warnmeldungen bei der Überprüfung der Version 0 von `Rechner.java` mit der Version 3 von `Stack.java`

wir unsere Bedingungen in der Programm-Schnittstelle erweitern. Mit der Bedingung in Zeile 22 wird sichergestellt, dass das Anwendungsprogramm auch die richtigen (Typen) Objekte im Stack ablegt. Andersherum wird durch Zeile 35 bekannt gemacht, dass auch nur Objekte des Elementtyps als Rückgabewert zu erwarten sind. Die Objekt-Invariante in Zeile 10 ist notwendig, da in der `pop()`-Methode klar sein muss, dass nur Objekte des Elementtyps im Array sind (ansonsten würde die `ensure`-Bedingung in Zeile 35 fehlschlagen). An dieser Stelle sei nochmals darauf hingewiesen, dass die Überprüfung modular, also pro Methode erfolgt und der Beweiser beim bearbeiten der Methode `pop()` nicht wissen kann, dass Array-Elemente über einen Objekt-Inhalt haben. Nun kann es aber sein, dass dieses Objekt von einem anderen Programmteil geändert wird. Um den Sachverhalt exklusiver Objekte in den Member-Variablen zu modellieren, besitzt jedes Objekt innerhalb der Beweisers die Ghost-Variable `owner`. Belegt man diese mit dem aktuellen Objekt (so wie im Beispiel), kann der Beweiser davon ausgehen, dass das Objekt nicht mehr außerhalb dieser Klasse verändert wird.

Eine Anwendung

Nun haben wir an unserem Stack-Beispiel schon einigen zusätzlichen Code eingefügt, der zur Überprüfung des eigenen Stack-Codes, wie auch

zur Überprüfung von Anwendungsprogrammen, die diese Datenstruktur benutzen, dient. Konsequenterweise soll nun zum Abschluss ein einfaches Programm vorgestellt werden, das den Stack verwendet. In Abbildung 10 sieht man die Klasse `Rechner.java`. Wie man schnell sieht stellt der Code einen UPN-Rechner dar: Zahlen können mit der Methode `ablegen()` auf den Stack geschrieben werden. Die Methode `addieren()` addiert jeweils die zwei obersten

Stackelemente und legt das Ergebnis wieder auf dem Stack ab. Und zu guter Letzt die Methode `ergebnis()` gibt den Wert des obersten Stackelementes zurück.

Lassen wir die Version 0 von `Rechner.java` mit ESC/Java durchlaufen, ergibt sich die Abbildung 11. Die Warnungen der möglichen Null-Dereferenz der `stack Member-Variable` in Zeile 10 und 20 sind kalter Kaffee: Wie in den

```

1: public class Stack {
2:
3:     /*@ ghost public \TYPE elementType;
4:
5:     int size;
6:
7:     /*@non_null*/ Object content[];
8:     /*@ invariant size >= 0 && size <= content.length;
9:     /*@ invariant \elementype(\typeof(content)) == \type(Object);
10:    /*@ invariant (\forall int i; (0 <= i && i < size)
11:                ==> content[i] != null &&
12:                    \typeof(content[i]) <: elementType); */
13:    /*@ invariant content.owner == this;
14:
15:    /*@ ensures size == 0;
16:    public Stack() {
17:        content = new Object[5];
18:        /*@ set content.owner = this;
19:    }
20:
21:    /*@ modifies size
22:    /*@ ensures size == \old(size) + 1;
23:    /*@ requires \typeof(o) <: elementType
24:    public void push(/*@non_null*/ Object o) {
25:        if (size>=content.length) {
26:            Object[] oa = new Object[size+5];
27:            /*@ set oa.owner = this;
28:            System.arraycopy(content,0,oa,0,size);
29:            content = oa;
30:        }
31:        content[size++] = o;
32:    }
33:
34:    /*@ modifies size
35:    /*@ ensures size > 0 ==> size == \old(size) - 1;
36:    /*@ ensures \typeof(\result) <: elementType
37:    /*@ ensures \result != null
38:    public Object pop() throws StackEmptyException {
39:        if (size<=0) {
40:            throw new StackEmptyException();
41:        }
42:        return content[--size];
43:    }
44:
45:    /*@ ensures \result == size;
46:    public int size() {
47:        return size;
48:    }
49:
50:    public static class StackEmptyException extends RuntimeException {}
51:
52: }

```

Abbildung 12: Version 4 von `Stack.java`: Es dürfen keine NULL-Objekte gespeichert werden

vorangegangenen Beispielen sorgt ein `non_null` Pragma für Abhilfe. Eine neue Problemstellung sind die `Typecast`-Warnungen in den Zeilen 14, 15 und 20. Wie vorher schon angedeutet, kommt nun die `elementType` Ghost-Variable vom Stack ins Spiel. Über die Invariante und das setzen von `elementType` in Zeile 4 und 8 der Version 1 von `Rechner.java` machen wir bekannt, dass nur Elemente vom Typ `Float` im Stack gespeichert werden. Diese Information steht dem Beweiser nun auch beim `Typecast` auf `Float` zur Verfügung: Die Warnmeldungen über den `type cast error` verschwinden.

Es bleiben aber immer noch die drei Null-Referenz Warnmeldungen von Zeile 16 und 21. Hierbei handelt es sich um Werte die mit `pop()` vom Stack abgerufen wurden. Man könnte nun diese Warnmeldung mit einem `assume f!=null` unterdrücken. Dies wäre sicherlich auch korrekt (bewiesen haben wir es allerdings aber nicht), denn wir wissen ja, dass wir nie einen Null-Wert in den Stack hineinschreiben. Aber ist das eine elegante Lösung? Intuitiv macht es eigentlich keinen Sinn eine Null-Referenz in einen Stack zu schreiben. Also wäre die elegante Variante, die letzte Version von `Stack.java` nochmals zu überdenken und eine Ergänzung vorzunehmen. In Abbildung 12 sieht man das Ergebnis: Bei der `push()`-Methode werden keine Null-Werte mehr angenommen. Nun können wir per Objekt-Invariante nachweisen, dass innerhalb Array-Index 0 bis `size` immer eine gültige Objekt-Referenz steht (Zeile 11). Zu guter letzt die Änderung in Zeile 37: Nun weiß ESC/Java, dass beim Aufruf von `pop()` niemals ein Null-Wert zurückkommen kann. Damit sind nun ndie restlichen Warnmeldungen von `Rechner.java` eliminiert.

Prüfung von Schleifen

Programmcode der in Schleifen ausgeführt wird, wird nur vage überprüft. Gibt man Assertions im Schleifenrumpf oder Schleifeninvarianten an, so wird die Korrektheit standardmäßig nur für den ersten Schleifendurchlauf bewiesen. Die Anzahl der Schleifendurchläufe, die betrachtet werden

sollen, können über die Kommandozeile konfiguriert werden.

Im Vortrag kam die Frage auf, ob diese Strategie nicht abgeändert wird, wenn eine Schleifeninvariante spezifiziert ist. In diesem Fall könnte theoretisch versucht werden, dass für alle Schleifendurchläufe der Nachweis erfolgt. Es geht explizit aus der Dokumentation hervor, dass dies nicht zutrifft; d.h. auch spezifizierte Schleifeninvarianten werden nur über den einfachen Mechanismus (wie oben beschrieben) bearbeitet.

Thread-Synchronisation

Zur Vollständigkeit sei erwähnt, dass ESC/Java einige mächtige Konstrukte zur Verfügung stellt, um die korrekte Thread-Synchronisation von Methoden zu überprüfen. Auf eine ausführliche Behandlung dieser Thematik wird verzichtet, da davon auszugehen ist, dass nicht jeder mit dieser Problemstellung vertraut ist und für eine Einführung der Platz und die Zeit nicht ausreicht.

Entwicklungszeile und Optimierungen

Innerhalb des ESC/Java Projekts sollte ein Software-Framework entstehen in dem verschiedene Kompromisse bei der Programmverifikation erforscht werden können. Folgende Kriterien gingen ein: Anzahl nicht gefundene Fehler, Anzahl von falschen Fehlermeldungen, Einarbeitungsaufwand, Aufwand für Programm Annotation, Laufzeit des Prüfprogramms, Entwicklungsaufwand für das Prüfprogramm.

Die Laufzeit von ESC/Java liegt, bei unseren Beispielen, im Sekundenbereich. Um diese Schnelligkeit zu erreichen, wurden bei der Mächtigkeit des Beweisers Einschnitte gemacht. Beispiele für nicht gefundene Fehler sind die Behandlung von Schleifen. Ein Beispiel für falsche Fehler sind offensichtliche Objekt-Invarianten, die ESC/Java nicht erkennt, da es nicht methodenübergreifend arbeitet.

Schlussfolgerung

Aufgrund der Schnelligkeit des Beweisvorgangs wird der Einsatz des Tools im normalen Entwicklungsprozess möglich. Programmierer werden dadurch angehalten, weitere sinnvolle (da formal definiert) Kommentare zum Code zu machen, die für die Verifikation notwendig sind. Diese zusätzlichen Annotierungen halten sich in Grenzen und dürften den normalen Entwicklungsprozess nicht zu stark behindern.

Typische Flüchtigkeitsfehler können so besser erkannt werden. Nicht zu unterschätzen ist aber der Wert, der zusätzlichen Beweisbedingungen beim nachträglichen Abändern von bestehenden, großen Softwarepaketen. Hier herrscht oftmals die Situation vor, dass dies ein Mitarbeiter durchführen muss, der keinen Überblick über das Gesamtprojekt und die verwendeten APIs hat.

Der Einsatz von ESC/Java kann aber niemals eine vollständige Verifikation (falls diese notwendig ist) oder ausführliche Funktionstests ersetzen.

Literatur-Verweise

[ESC/Java] Die Webseite des ESC/Java Projektes ist:
<http://research.compaq.com/SRC/esc>

[JML] Java Modeling Language, Webseite:
<http://www.cs.iastate.edu/~leavens/JML.html>

Glossary

ESC

Extended Static Checking

Pragma

Prüfungsanweisung/-bedingung im Kommentar eines Quelltextes

Objekt-Invariante

Bedingung die für die ganze Lebenszeit eines Objektes gelten muss

Schleifeninvariante

Bedingung die jeweils am Anfang jedes Schleifendurchlaufs gelten muss

Assertion

Zusicherung: Bedingung die an einer Stelle im Code angegeben ist und zur Laufzeit an genau dieser Stelle zwingend erfüllt sein muss