Beweisen statt Testen – Höhere Zuverlässigkeit durch die Java Modeling Language

Unser tägliches Leben wird immer stärker von Software durchdrungen; daher ist die Korrektheit von Software von fundamentaler Bedeutung. Gleichzeitig wird Software immer komplizierter; daher sind mächtige Methoden zum Nachweis dieser Korrektheit erforderlich. Der wichtigste Ansatz in dieser Sache – der mathematische Beweis der Korrektheit – gewinnt zusehends an Bedeutung für die industrielle Praxis. Im Folgenden werden die grundlegenden Ideen zu diesem Ansatz erläutert sowie eine aktuelle Sprache, die Java Modeling Language, zusammen mit ihren wichtigsten Werkzeugen zur Realisierung dieses Ansatzes vorgestellt.

Edgar Lederer | edgar.lederer@fhnw.ch

Das vielleicht vordringlichste Problem der Informatik ist die Korrektheit von Software – was nützt die benutzerfreundlichste, effizienteste und innovativste Software, wenn sie nicht richtig funktioniert? Die meisten Anwenderinnen und Anwender haben sich schon oft über die Eigentümlichkeiten ihres PCs oder ihrer Büroprogramme die Haare gerauft; das einzige, worauf man sich wirklich verlassen kann, ist, etwas überspitzt formuliert, dass sie im entscheidenden Moment ihren Dienst versagen. Das ist ärgerlich, aber immerhin nicht tödlich. Anders sieht die Situation aus, wenn Software zur Steuerung von Flugzeugen, Eisenbahnen und Autos, oder aber Atomkraftwerken, militärischen Einrichtungen und medizinischen Geräten eingesetzt wird. Hier kann ein Fehlverhalten leicht zur Gefahr für Leib und Leben werden (ein berühmtes Beispiel ist das medizinische Bestrahlungsgerät Therac-25, dessen Fehlverhalten zwischen 1985 und 1987 zu sechs Unfällen führte, von denen drei tödlich endeten); solche Software bezeichnet man daher auch als *sicherheitskritisch* (safety-critical). Ebenfalls äusserst heikel ist es, wenn Software Finanztransaktionen oder unbemannte Raketen steuert. Ein Fehlverhalten solcher Software kann zum Scheitern der Mission führen, für deren Durchführung eben diese Software benötigt wird (ein berühmtes Beispiel ist der fehlgeschlagene Jungfernflug der Trägerrakete Ariane 5 von 1996); solche Software bezeichnet man daher auch als *missionskritisch* (mission-critical).

Korrekte Software

Was bedeutet aber genau, dass Software *korrekt* ist? Ein Programm beispielsweise, das den Sinus einer gegebenen reellen Zahl perfekt ausrechnet, ist dennoch nicht korrekt, wenn die Aufgabe des Programms darin besteht, die Quadratwurzel aus der Zahl zu ziehen. Wir sehen also, dass Korrektheit kein absoluter, sondern nur ein relativer Begriff ist: Eine Software kann nur gegenüber den Anforderungen, die an sie gestellt werden, korrekt sein – nicht aber *per se*.

Um bei der Entwicklung der Software stets ein klares Ziel vor Augen zu haben, erweist es sich als äusserst zweckmässig, diese Anforderungen schriftlich festzuhalten, und zwar *bevor* mit dem Schreiben des eigentlichen Programms begonnen wird. Oft wird das so entstandene Dokument die *Spezifikation* des Programms genannt. Leider ist das Aufschreiben der Spezifikation in der Regel auch kein Kinderspiel: Haben wir wirklich alle Anforderungen aufgelistet (Vollständigkeit)? Widersprechen sich unsere Anforderungen auch nicht (Widerspruchsfreiheit)?

Mit der Spezifikation zerfällt die Frage nach der Korrektheit eines Programms also in zwei Teile:

- Erfüllt die Spezifikation unsere Anforderungen?
- Erfüllt das Programm unsere Spezifikation?

Die Überprüfung der ersten Frage wird als *Validierung* bezeichnet und oft durch die Frage «Bauen wir das richtige Produkt?» apostrophiert, während die Überprüfung der zweiten Frage als *Verifikation* bezeichnet wird und durch die Frage «Bauen wir das Produkt richtig?» charakterisiert wird.

Im Idealfall erfolgt die Validierung, bevor mit dem Schreiben des eigentlichen Programms begonnen wird. Wie kann dies geschehen? Möglich und gängig sind das sorgfältige Review der Spezifikation im Team oder die Entwicklung und Anwendung eines Prototypen. Besonders attraktiv ist es auch, die Spezifikation in einer sehr hohen, meist deklarativen (funktionalen oder logischen) Programmiersprache zu schreiben. Dann ist die Spezifikation nämlich direkt ausführbar. Dies ist für die Endanwendung meist nicht effizient

genug, dafür ist aber das Schreiben der ausführbaren Spezifikation vergleichsweise schnell erledigt. Nun können praktische Erfahrungen beim Ausführen der Spezifikation gesammelt werden, durch die die Spezifikation schrittweise verbessert werden kann, bevor noch eine einzige Zeile in der endgültigen Programmiersprache geschrieben ist. (Es ist aber bei diesem Ansatz unbedingt zu beachten, dass die Ausführbarkeit der Spezifikation einen Verlust an Abstraktion nach sich ziehen kann, was der Idee einer Spezifikation genau zuwiderläuft.)

Software-Verifikation

Schwerpunkt dieses Artikels ist aber nicht die Validierung, sondern die Verifikation. In der industriellen Praxis erfolgt die Verifikation in den meisten Fällen mittels Testens: Das zu verifizierende Programm wird für geeignet ausgewählte Testfälle ausgeführt, und die dabei erhaltenen Ergebnisse werden mit den erwarteten Ergebnissen verglichen; welche Ergebnisse erwartet werden, sagt die Spezifikation. (Die Instanz, welche die richtigen Ergebnisse voraussagt, nennt man allgemein auch *Test-Orakel*; hier ist also die Spezifikation das Test-Orakel.) Welche Schlüsse kann man nun aus den Tests ziehen? Wenn man bedenkt, dass ein Programm dann und nur dann korrekt ist, wenn es *für alle denkbaren* Eingaben richtig funktioniert, die folgenden: Weichen erhaltenes und erwartetes Ergebnis für irgendeinen der ausgewählten Testfälle voneinander ab, so weiss man definitiv, dass das Programm fehlerhaft ist; stimmen sie hingegen für alle ausgewählten Testfälle überein, so weiss man zwar, dass das Programm für eben diese Testfälle korrekt funktioniert, aber über alle übrigen, nicht getesteten Fälle (und dies sind «astronomisch» viel mehr als die getesteten), weiss man leider absolut nichts. Es gibt eben keinen logischen Schluss von dem Speziellen auf das Allgemeine («Weil ich fünf weisse Schwäne gesehen habe, sind alle Schwäne weiss.»), und das Stetigkeitsargument aus den empirischen Wissenschaften wie der Physik («Weil die Brücke sowohl unbelastet als auch stark belastet hält, hält sie auch bei mittlerer Belastung.») ist bei Programmen nicht anwendbar, da Programme eben nicht auf stetigen, sondern auf diskreten Strukturen basieren. Diese Überlegungen führen zu der etwas ernüchternden und bereits 1969 von Edsger W. Dijkstra formulierten Erkenntnis, dass man durch Testen immer nur die Anwesenheit, niemals aber die Abwesenheit von Fehlern in Programmen zeigen kann. (Es ist in diesem Zusammenhang interessant festzuhalten, dass diese Erkenntnis konsistent ist mit der *Falsifikationstheorie* des Philosophen Sir Karl Popper, derzufolge sich wissenschaftliche Theorien immer nur falsifizieren, niemals aber verifizieren lassen; solange eine Theorie noch nicht falsifiziert ist, kann man lediglich sagen, dass sie sich bisher *bewährt* hat.) Die in der Industrie häufig durchgeführten *Akzeptanztests* zeigen auch nicht etwa, dass ein Programm korrekt ist, sondern lediglich – wie der Name auch treffend ausdrückt – dass es vom Auftraggeber *akzeptiert* wird, er also bereit ist, die Rechnung des Auftragnehmers zu begleichen. Auch das Unwort «ausgetestet» ist mehr als gefährlich, da es durch seine Vorsilbe «aus» suggeriert, dass alle denkbaren Fälle getestet wurden.

Glücklicherweise gibt es eine weit aussagekräftigere Methode als die des Testens, Programme gegenüber ihrer Spezifikation zu verifizieren, nämlich die *mathematische* oder *deduktive* Methode. Möchten wir in der Mathematik beispielsweise wissen, dass die binomische Formel $(a + b)^2 = a^2 + 2ab + b^2$ für alle Zahlen a und b gilt, so verschaffen wir uns dieses Wissen ja auch nicht dadurch, dass wir einige konkrete Zahlen für a und b einsetzen (z. B. $a = 2$, $b = 3$ und $a = 17$, $b = -5$), sondern indem wir mathematische Gesetze anwenden:

$$
\begin{aligned}
&(a + b)^2 \\
&= \text{// Definition von Quadrat} \\
&(a + b) \cdot (a + b) \\
&= \text{// Distributivgesetz} \\
&a \cdot (a + b) + b \cdot (a + b) \\
&= \text{// Distributivgesetz zweimal} \\
&a \cdot a + a \cdot b + b \cdot a + b \cdot b \\
&= \text{// Kommutativgesetz der Multiplikation} \\
&a \cdot a + a \cdot b + a \cdot b + b \cdot b \\
&= \text{// neutrales Element 1 der Multiplikation} \\
&a \cdot a + 1 \cdot (a \cdot b) + 1 \cdot (a \cdot b) + b \cdot b \\
&= \text{// Distributivgesetz} \\
&a \cdot a + (1 + 1) \cdot (a \cdot b) + b \cdot b \\
&= \text{// Definition von 2 und Definition von Quadrat} \\
&a^2 + 2ab + b^2
\end{aligned}
$$

Nun wissen wir, zweifelsfrei und nach nur endlich vielen Schritten, dass unsere binomische Formel für alle, also unendlich viele, Zahlen gilt. Es ist eine der faszinierendsten Eigenschaften der Mathematik, dass man Aussagen über unendlich viele Gegenstände nach nur endlich vielen Überlegungen erhält, und das mit (bis auf Rechenfehler) absoluter Sicherheit. Warum sollten wir dieses mächtige Werkzeug nicht auch für Programme einsetzen?

Dies setzt natürlich voraus, dass Programme und Spezifikationen mit mathematischer Präzision behandelt werden können. Insbesondere reicht es dann nicht mehr aus, Spezifikationen nur umgangssprachlich zu beschreiben; eine formale Beschreibung ist stattdessen erforderlich, und dafür wiederum werden formale Spezifikationssprachen benötigt. Des Weiteren ist es erforderlich, den Programmiersprachen und den Spezifikationssprachen mathematisch exakt definierte Bedeutungen zu geben. Schliesslich – und das erahnt man schon bei unserem trivialen Beispiel mit der binomischen Formel – sind mathematische Korrektheitsbeweise für kom-

plexe Programme extrem aufwändig und damit selbst fehleranfällig. Daher muss die Entwicklung solcher Beweise wiederum durch geeignete (fehlerfreie!) Programme unterstützt werden. All diese Anforderungen sind aber inzwischen durch jahrzehntelange weltweite Forschungsanstrengungen prinzipiell und für relevante Fallstudien erfüllt worden – das Hauptaugenmerk der Forschung liegt aktuell auf der Umsetzung dieser Methoden in die industrielle Praxis. Nicht zuletzt Microsoft Research arbeitet intensiv an der Umsetzung dieser Ideen; sowohl Microsofts *Spec#*, eine Sprache sehr ähnlich wie das hier vorgestellte JML, als auch Microsofts *Static Driver Verifier*, ein Programm zur statischen Verifikation bestimmter Eigenschaften von insbesondere fremd entwickelten Gerätetreibern, geben ein beredtes Zeugnis davon ab.

Zusicherungen, Hoare-Tripel und Hoare-Logik

Aber was sind nun eigentlich die grundlegenden Ideen, um mathematische Beweise auch auf Programme anwenden zu können? Diese Ideen wurden erstmals 1969 von C.A.R. (nun Sir Tony) Hoare in einem Artikel [H69] vorgestellt (seinerseits inspiriert durch einen Artikel [F67] aus dem Jahre 1967 von Robert W. Floyd) und basieren auf Zusicherungen, Hoare-Tripeln und Hoare-Logik, wie im Folgenden kurz skizziert wird.

Eine *Zusicherung* ist ein boolescher Ausdruck, der in der Regel einige der deklarierten Variablen des Programms enthält. In einem gegebenen Zustand der das Programm ausführenden Maschine hat eine Zusicherung also immer einen der beiden Werte *wahr* oder *falsch*. Damit lässt sich eine Zusicherung aber auch auffassen als Beschreibung einer Menge von Zuständen, nämlich der Menge aller derjenigen Zustände, in denen die Zusicherung den Wert *wahr* hat.

Ein *Hoare-Tripel* ist nun ein Ausdruck der Form «{P} A {Q}» wobei *P* und *Q* Zusicherungen sind, und *A* eine Anweisung ist. *P* und *Q* nennt man dabei auch *Vorbedingung* bzw. *Nachbedingung* des Hoare-Tripels. Ein Hoare-Tripel heisst *gültig*, wenn die folgende zentrale Bedingung erfüllt ist:

> *Wird die Anweisung* A *in einem Zustand ausgeführt, in dem die Vorbedingung* P *wahr ist, und wird die Ausführung in endlicher Zeit beendet (geht also nicht in eine Endlosschleife), so ist die Nachbedingung* Q *im resultierenden Zustand erfüllt.*

Ein Beispiel für ein gültiges Hoare-Tripel ist also «{ $x = 5$ } $x := x + 1$ { $x > 5$ }»; ein Beispiel für ein ungültiges Hoare-Tripel hingegen «{ $x = 5$ } $x := x + 1$ { $x > 6$ }». Hier ein weiteres Beispiel für ein Hoare-Tripel:

> { *p* ist ein syntaktisch korrektes Java-Programm }
> *c* := Java-Compiler(*p*)
> { *c* ist der korrekte Bytecode für *p* }

Es ist offensichtlich, dass die Entwickler des Java-Compilers daran interessiert sind, dass dieses Hoare-Tripel gültig ist. Aber wie lässt sich entscheiden, ob ein Hoare-Tripel gültig ist oder nicht? Dies ist möglich mittels der *Hoare-Logik*, einem Satz von Regeln, die es gestatten, aus der Gültigkeit von Hoare-Tripeln mit gegebenen Anweisungen auf die Gültigkeit von Hoare-Tripeln mit daraus zusammengesetzten Anweisungen zu schliessen. Betrachten wir als Beispiel die wichtigste dieser Regeln, nämlich die für Schleifen:

> *Ist das Hoare-Tripel* «{ I and B } A { I }» *gültig, so ist auch das Hoare-Tripel* «{ I } while B do A end { I and not B }» *gültig.*

Diese Regel ist unmittelbar einleuchtend: Wenn die Ausführung des Schleifenrumpfes *A* in einem Zustand, der die Zusicherung *I* und die Schleifenbedingung *B* erfüllt, zu einem resultierenden Zustand führt, der *I* immer noch erfüllt, so bleibt *I* auch noch nach beliebig oft wiederholter Ausführung des Schleifenrumpfes erfüllt – insbesondere auch noch nach der allerletzten. Nach dieser allerletzten Ausführung ist aber gleichzeitig die Schleifenbedingung nicht mehr erfüllt – ansonsten wäre der Schleifenrumpf ja nochmals ausgeführt worden.

Die Zusicherung *I* wird also durch Ausführung des Schleifenrumpfes nicht zerstört; man bezeichnet sie daher auch als *Invariante* der Schleife. Invarianten beschreiben die eigentliche Idee, die hinter einer Schleife verborgen liegt, und stellen den kreativen Teil bei der Entwicklung einer Schleife dar. Entsprechend ist es auch nicht verwunderlich, dass Invarianten im Allgemeinen nicht automatisch hergeleitet werden können (für einfachere Fälle lassen sich inzwischen jedoch wahrscheinliche Invarianten automatisch bestimmen; siehe dazu weiter unten). Von Seiten des Unterrichts ist noch folgendes anzumerken: Das Konzept der Invarianten gehört zu den für das grundlegende Verständnis der Informatik mächtigsten und daher auch wichtigsten Konzepten; umso trauriger ist es, dass dieses Konzept auch heute noch an vielen Universitäten im Unterricht nur stiefmütterlich behandelt wird [G06]; an unserer Schule findet es jedoch im Modul «Algorithmen und Datenstrukturen» einen breiteren Raum.

Design by Contract

Diese grundlegenden Ideen wurden bald auch auf die objektorientierte Programmierung angewendet; besonders einflussreich ist dabei die juristische Metapher des *Design by Contract* von Bertrand Meyer [M97]. Im Design by Contract gehört zu jeder Methode einer Klasse eine Vorbedingung und eine Nachbedingung; diese bilden einen Kontrakt zwischen dem Aufrufer der Methode und der Methode selbst:

- Der Aufrufer *verpflichtet* sich, die Methode nur in solchen Zuständen aufzurufen, die die

Vorbedingung erfüllen; entsprechend ist die Methode *berechtigt* davon auszugehen, dass die Vorbedingung erfüllt wird.
- Die Methode *verpflichtet* sich (in den Fällen, in denen die Vorbedingung erfüllt ist), einen Zustand zu erzeugen, der die Nachbedingung erfüllt; entsprechend ist der Aufrufer *berechtigt* (in den Fällen, in denen die Vorbedingung erfüllt ist) davon auszugehen, dass die Nachbedingung erfüllt wird.

Zusätzlich zu diesen Vor- und Nachbedingungen gibt es im Design by Contract Konsistenzbedingungen, die während der gesamten Lebensdauer eines Objektes einer Klasse erfüllt sein müssen; diese werden entsprechend als *Klasseninvarianten* bezeichnet.

Leider reichen diese grundlegenden Ideen noch nicht aus, um voll ausgebildete objektorientierte Programme zu spezifizieren und zu verifizieren. Weitere wesentliche Punkte sind die folgenden: (1) *Exceptions*: Das Verhalten bei Ausnahmen muss spezifiziert werden können. (2) *Frame problem*: Es muss spezifiziert werden können, welche Speicherzellen durch Ausführung des Programms *nicht* geändert werden dürfen, und das unter Berücksichtigung von Sichtbarkeit und Subtyping. (3) *Call-back problem*: Klasseninvarianten gelten stets zu Beginn und am Ende einer Methode, aber zunächst keineswegs notwendigerweise auch dazwischen. Ruft nun eine Methode eine Methode derselben Klasse auf (*call back*), so wird diese aufgerufene Methode im Allgemeinen in einem Zustand aufgerufen, in dem die Invariante verletzt ist, womit sich natürlich keinerlei Voraussagen mehr über das Verhalten der aufgerufenen Methode machen lassen. Stellt man aber sicher, dass die Invariante auch vor und nach jedem Methodenaufruf gilt, so ist das Problem gelöst; leider ist diese Lösung sehr restriktiv – andere Ansätze sind Gegenstand aktueller Forschung. (4) *Behavioral subtyping*: Das Verhalten eines Subtyps muss stets mit dem Verhalten seines Supertyps übereinstimmen (*behavioral subtyping*), so dass anstelle eines Objektes des Supertyps gefahrlos auch immer ein Objekt des Subtyps eingesetzt werden kann (*Liskov's substitution principle*). Dies lässt sich dadurch erreichen, dass ein Subtyp alle Spezifikationen seines Supertyps erbt (*specification inheritance*), also alle Klasseninvarianten des Supertyps und die Kontrakte aller überschriebenen Methoden. (5) *Data abstraction*: Besonders wesentlich ist die Spezifikation des Verhaltens aller Klassen, die ein gegebenes Interface implementieren; diese Spezifikation muss also im Interface selbst stehen. Dort aber sind keine Felder deklariert, so dass sich eigentlich gar keine Kontrakte und Klasseninvarianten formulieren lassen. Dieses Problem lässt sich aber lösen, indem man durch Deklaration von *Modell-Feldern* im Interface einen abstrakten Zustand schafft; mit diesen lassen sich dann Kontrakte und Invarianten formulieren. Dass eine Klasse ein Interface dann korrekt implementiert, lässt sich durch Relationen in der Klasse ausdrücken, die den konkreten Zustand der Klasse mit dem abstrakten Zustand des Interfaces in Beziehung setzen.

Java Modeling Language

Eine konkrete Spezifikationssprache, die alle diese Probleme löst (und zwar in der eben angedeuteten Art und Weise), ist die Java Modeling Language (JML[1]) [BC05, CKLP06, LBR06]. JML dient der Spezifikation des Verhaltens von voll ausgebildeten objektorientierten Programmen in Java. Wesentlich dabei ist, dass JML für den industriellen Einsatz gedacht ist, dass JML-Programmierer keine Experten für mathematische Methoden sein müssen, und dass für JML eine Reihe verschiedenartiger Werkzeuge zur Verfügung steht. Von diesen werden wir später noch einige etwas näher ansehen; vorerst aber betrachten wir ein kleines Beispiel einer JML-annotierten Java-Methode: Abbildung 1 zeigt die bekannte binäre Suche. Dieses Beispiel ist besonders reizvoll, da einerseits die operationelle Idee hinter der binären Suche sofort einleuchtet («Ein von links nach rechts aufsteigend sortiertes Feld wird in der Mitte geteilt: ist das gesuchte Element grösser als das Element in der Mitte, kommen sämtliche Elemente links der Mitte und die Mitte selbst nicht mehr in Frage; ansonsten fallen die Elemente rechts der Mitte flach. Die verbleibenden Elemente werden nach demselben Verfahren durchsucht, bis kein Element mehr verbleibt.»), es andererseits aber ziemlich schwierig ist, das Verfahren wirklich korrekt zu implementieren (die Leserin oder der Leser mögen es einmal selbst versuchen!). Unsere Implementierung basiert auf den Überlegungen in [B03].

Formuliert in JML sind die Vorbedingung, die Nachbedingung und die Schleifeninvariante. (Zusätzlich ist die Methode als *pure* deklariert, was bedeutet, dass sie keine Seiteneffekte hat; ausserdem enthält sie noch eine Zusicherung (Schlüsselwort *assert*) am Ende der Schleife.) Die *Vorbedingung* (Schlüsselwort *requires*) verlangt, dass das Feld nicht null ist, dass seine Länge mindestens 0 ist (dient lediglich der Verdeutlichung, dass auch die Länge 0 erlaubt ist), und dass das Feld aufsteigend sortiert ist, wobei auch gleiche Werte vorkommen dürfen. Die *Nachbedingung* (Schlüsselwort *ensures*) stellt sicher (jedenfalls solange die Vorbedingung erfüllt ist), dass der gesuchte Index zwischen 0 und der Länge des Feldes (jeweils einschliesslich) liegt, und dass alle Elemente des Feldes links vom gesuchten Index (ausschliesslich des Index) kleiner sind als der gesuchte Wert, und alle Elemente rechts davon (einschliesslich des In-

1 http://www.cs.ucf.edu/~leavens/JML/

```
/*@ requires a != null && a.length >= 0
  @   && (\forall int i; 0 <= i && i < a.length-1; a[i] <= a[i+1]);
  @ ensures 0 <= \result && \result <= a.length
  @   && (\forall int i; 0 <= i && i < \result; a[i] < x)
  @   && (\forall int i; \result <= i && i < a.length; a[i] >= x);
  @*/
static /*@ pure @*/ int binarySearch(int[] a, int x) {
    int l = 0;
    int r = a.length - 1;
    /*@ loop_invariant 0 <= l && l <= r+1 && r <= a.length - 1
      @   && (\forall int i; 0 <= i && i < l; a[i] < x)
      @   && (\forall int i; r < i && i < a.length; a[i] >= x);
      @*/
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (a[m] < x) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }
    //@ assert l == r+1;
    return l;
}
```

Abbildung 1: JML-annotierte Java-Methode zur bekannten binären Suche

dex) grösser oder gleich dem Wert. Die *Schleifeninvariante* (Schlüsselwort *loop_invariant*) sagt, dass vor und nach jedem Schleifendurchlauf alle Elemente des Feldes links von einem Index *l* kleiner sind als der gesuchte Wert, und alle Elemente rechts von einem Index *r* grösser oder gleich dem Wert sind, und (hauptsächlich) dass der linke Index den rechten Index „überholen" kann, aber nur um maximal eine Position. An der Schleifeninvariante ist besonders gut zu erkennen, dass das Feld insgesamt in drei Teile unterteilt wird (Teil 1: Elemente, von denen man schon weiss, dass sie kleiner sind als der gesuchte Wert; Teil 2: Elemente, von denen man noch nichts weiss; und Teil 3: Elemente, von denen man schon weiss, dass sie grösser oder gleich dem gesuchten Wert sind). Relevant sind also in Wirklichkeit *drei* Teile, obwohl das Verfahren *binäre* Suche heisst! Am Anfang des Verfahrens ist noch kein Wissen vorhanden: Teile 1 und 3 sind noch «leer», während Teil 2 das gesamte Feld abdeckt. Am Ende des Verfahrens hingegen, also wenn der linke den rechten Index überholt hat, ist das gesamte gewünschte Wissen vorhanden: Teil 2 ist nun leer, während die Teile 1 und 3 zusammen das gesamte Feld abdecken – womit die Suche abgeschlossen ist.

JML-Werkzeuge

Nun aber zu den Werkzeugen für JML: Das Werkzeug *jmlc* ist ein Compiler, der JML-annotierte Java-Programme in Bytecode übersetzt; die JML-Zusicherungen werden dabei in Code zu ihrer Überprüfung zur Laufzeit übersetzt. Wird eine Zusicherung zur Laufzeit des Programms verletzt, so wird eine aussagekräftige Fehlermeldung erzeugt; andernfalls bleibt die Zusicherung ohne Wirkung.

Das Werkzeug *jmlunit* verwendet die JML-Spezifikationen als Test-Orakel; die Testfälle selbst müssen dabei weiterhin von Hand festgelegt werden.

Das Werkzeug *ESC/Java2* ist ein *Extended Static Checker*, also ein Werkzeug zur statischen Analyse des Programmtextes; das Programm braucht dazu also nicht ausgeführt zu werden. ESC/Java2 generiert aus einem JML-annotierten Java-Programm automatisch eine Liste möglicher Schwachstellen des Programms. Dabei ist es weder das Ziel, dass die Liste alle Fehler des Programms enthält, noch dass alle Einträge in der Liste tatsächlich Fehler sind, sondern lediglich, dass die Liste möglichst viele der Fehler des Programms enthält, und dass möglichst viele der Einträge in der Liste tatsächlich Fehler sind, so dass sich der Aufwand für das sorgfältige Studium der Liste unter dem Strich auszahlt.

Das Werkzeug *JACK* ist ein Verifizierer und kommt damit ebenfalls mit dem Programmtext alleine aus. Dieser erzeugt aus einem JML-annotierten Java-Programm einen Satz von Verifikationsbedingungen; sind diese allesamt gültig, ist das Java-Programm korrekt. Diese Verifikationsbedingungen werden dann weitergeleitet an einen automatischen Theorembeweiser, der versucht, diese Bedingungen zu beweisen. Wie aus der theoretischen Informatik bzw. der Logik bekannt, ist es im Allgemeinen aber gar nicht möglich, beliebige Aussagen automatisch zu beweisen. In der Praxis stellt es sich aber heraus, dass doch bis zu 90% der Verifikationsbedingungen automatisch bewiesen werden können. Die übrigen, nicht bewiesenen Bedingungen werden dann dem Anwender zur weiteren Untersuchung vorgelegt; dieser kann sich dann beispielsweise um Beweise mit Hilfe eines interaktiven Theorembeweisers oder

um klassische mathematische Beweise mit Papier und Bleistift bemühen. JACK ist als Plug-in für Eclipse realisiert.

Das Werkzeug *Daikon* führt Java-Programme auf ausgewählten Eingaben aus und erzeugt aus den dabei entstandenen Ausführungsspuren wahrscheinliche Invarianten in Form von JML-Spezifikationen. Damit lässt sich in vielen der einfacheren Fälle der Aufwand zur Entwicklung der JML-Spezifikationen verringern; auch ist das Werkzeug nützlich für die nachträgliche Spezifikation von Legacy-Software, und insbesondere für das Refactoring von Legacy-Software, da beim Refactoring die Spezifikationen weiterhin eingehalten werden müssen – aber dazu ist es praktisch, die Spezifikationen erst einmal zu kennen.

Speziell in unserem Forschungsprojekt *ProMedServices*, bei dem es unter anderem gerade um das Refactoring von Teilen eines existierenden medizinischen Informationssystems geht, wollen wir den Einsatz von JML an einem praktischen Beispiel untersuchen. Des weiteren ist zu bemerken, dass ein grosser Anteil des Interesses an JML und verwandten Technologien aus dem Bereich Java-Smart-Card kommt, da darauf häufig Finanzanwendungen für den Masseneinsatz, also missionskritische Anwendungen, realisiert werden. Könnte man mit JML und verwandten Technologien hohe Zuverlässigkeit auch für andere mobile Geräte sicherstellen, so könnte man auch diese für einen solchen Einsatz in Betracht ziehen.

Seit Anbeginn der Informatik wird der mathematische Nachweis der Korrektheit von Software systematisch erforscht. Inzwischen sind viele der grundlegenden Fragen geklärt und viele leistungsfähige Werkzeuge entwickelt, so dass wir nun am Beginn des grossflächigen industriellen Einsatzes dieser Methoden stehen. Bis dahin sind noch etliche Jahre intensivster Arbeit in Theorie, Praxis und Ausbildung erforderlich – das Ziel, dass man sich auf Software auch im entscheidenden Moment verlassen kann, ist diese Arbeit aber allemal wert.

Referenzen

[B03] Roland Backhouse. Program Construction: Calculating Implementations from Specifications. John Wiley & Sons, 2003.

[F67] Robert W. Floyd. Assigning Meanings to Programs. Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics, Vol. 19), American Mathematical Society, 1967.

[G06] David Gries. What Have We Not Learned about Teaching Programming? Computer, Volume 39, Number 10, 2006.

[H69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming, Communications of the ACM, Volume 12, Number 10, 1969.

[M97] Bertrand Meyer. Object-Oriented Software Construction. Second Edition, Prentice Hall PTR, 1997.

[BC05] Lilian Burdy, Yoonsik Cheon et al. An overview of JML tools and applications. Int. Journal on Software Tools for Technology Transfer, Volume 7, Number 3, 2005.

[CKLP06] Patrice Chalin, Joseph R. Kiniry et al. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, 2006.

[LBR06] Gary T. Leavens, Albert L. Baker, Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes, Volume 31, Number 3, 2006.