Testfallerzeugung mit einer symbolischen virtuellen Maschine und Constraint Solvern

Christoph Lembeck Roger A. Müller

chle4@wi.uni-muenster.de romu@wi.uni-muenster.de

Herbert Kuchen

kuchen@uni-muenster.de

Abstract: Der Softwaretest wird heute als wichtiger Teil der Softwareentwicklung wahrgenommen. Da manuelles Testen teuer und unpräzise ist, werden verstärkt Werkzeuge zum Test-Management eingesetzt. Allerdings bleibt es dem Benutzer in der Regel selbst überlassen, die Testfälle zu erzeugen. Das Werkzeug, das wir in diesem Paper präsentieren, verwendet einen neuartigen Ansatz, um Testfälle zu vorgegebenen strukturellen Kriterien für einzelne oder mehrere Java-Methoden und Objekte zu produzieren. Dafür verwenden wir eine symbolische Java Virtual Machine, die passend zu den Kontrollflüssen in dem Programm Bedingungen generiert. Verzweigungen im Programm werden in Abhängigkeit von den aktuellen Kontrollfluss-Bedingungen, dem Testkriterium und der Verzweigungsstrategie gewählt. Der symbolischen virtuellen Maschine stehen (nicht-)lineare Constraint Solver (CS) zur Verfügung, die in einen CS Manager eingebettet sind. Dieser wählt dynamisch einen angemessenen CS aus und und bereitet die Constraints für die CS auf.

1 Einleitung

Software-Qualitätsmanagement und insbesondere Software-Testen finden mehr und mehr den Weg in das Bewusstsein der Öffentlichkeit. Dafür sprechen „Qualitätsoffensiven“ einiger großer Softwarehäuser, und die Betonung von Testen in Prozessmodellen wie etwa Extreme Programming. Während dem Benutzer beim Sammeln, Bewerten und Ausführen von Tests schon Werkzeuge zur Verfügung stehen, ist eine solche Unterstützung für die Erzeugung von Testfällen noch nicht in Sicht. Diese Lücke versuchen wir mit unserem hier vorgestellten Werkzeug für strukturorientiertes Testen zu schließen. Strukturorientiertes Testen, auch als White-Box-Testen bekannt, orientiert sich an dem Code, um Testfälle zu generieren. Es ist abzugrenzen vom funktionalen oder Black-Box Testen, das sich auf Anforderungen des Benutzers konzentriert und die Struktur des Codes außer Acht lässt. Prominente Vertreter des strukturorientierten Testens sind Def-Use-Ketten, Kantenüberdeckung und Schleifenüberdeckung [Be90].

Symbolische Ausführung [Cl76, Ki76], die das Kernstück der Testfallerzeugung bildet, gehört zu den statischen Methoden der Testfallerzeugung. Diese sind abzugrenzen von den dynamischen Methoden, etwa mittels genetischen Algorithmen, und der zufälligen Erzeugung von Testfällen, die keine Garantie über die Güte der erzeugten Testfälle machen kann [Ed99, TC98].

Das Ziel unserer Arbeit ist es, für einzelne *Einheiten*, wie Methoden, einzelne Klassen oder kleinere Systeme von Klassen Testfälle zu erzeugen, insb. für die Teile eines Programms, die *algorithmisch anspruchsvoll* sind, da hier beim Testen von Hand leicht Fälle übersehen werden. Wie beim White-Box-Testen üblich ist unser Werkzeug nicht darauf ausgelegt, ganze Programmsysteme zu behandeln, da dann die Anzahl der zu betrachteten Pfade zu groß würde. Java-Compiler übersetzen den Quellcode nicht direkt in maschinenlesbaren Code, sondern in so genannten Zwischencode, der auf einer virtuellen Maschine ausgeführt wird. Da es für die virtuelle Maschine von Java eine Fülle verschiedener Sprachen außer Java gibt, wurde die Entscheidung getroffen, die Testfallerzeugung nicht auf Source- sondern auf Bytecode aufzubauen. Die binäre Suche, die in diesem Paper als Beispiel dient, wird daher in Abbildung 1 als Bytecode angegeben.

```
0: iload_1      15: iaload        32: iload_3
1: iload_2      16: iload_3       33: if_icmple 44
2: if_icmpgt 47 17: if_icmpge 28  36: iload 4
5: iload_1      20: iload 4       38: iconst_1
6: iload_2      22: iconst_1      39: isub
7: iadd         23: iadd          40: istore_2
8: iconst_2     24: istore_1      41: goto 0
9: idiv         25: goto 0        44: iload 4
10: istore 4    28: aload_0       46: ireturn
12: aload_0     29: iload 4       47: iconst_m1
13: iload 4     31: iaload        48: ireturn
```

Abbildung 1: Binäre Suche im Bytecode

Dieser Artikel gliedert sich wie folgt: Nach der gerade gegebenen Einleitung folgt ein Überblick über die Details der symbolischen virtuellen Maschine, dann werden der Constraint Solver Manager erläutert und die einzelnen Constraint Solver kurz vorgestellt. Nach einem Überblick über verwandte Arbeiten folgt eine Zusammenfassung und ein Ausblick.

2 Symbolic Java Virtual Machine

Wie schon in der Einleitung erwähnt führt die symbolische Java Virtual Machine (SJVM) Befehle symbolisch aus. Das bedeutet, dass die Maschine nicht mit Zahlen rechnet, sondern für jede Variable einen Ausdruck speichert, der von Eingabeparametern (etwa denen einer Javamethode) und Konstanten abhängt. Betrachtet man etwa die Zeilen 5 bis 10 in Abbildung 1, so werden zunächst `low` und `high` auf den Stack gepackt, die beiden durch eine Addition verbunden und der komplette Ausdruck auf den Stack gelegt, dann die `2` auf den Stack gelegt, und mittels der Division mit dem Ausdruck mit der Addition verbunden. Mit dem `istore_4` wird dann der Ausdruck `(low+high)/2` unausgerechnet der Variablen `mid` zugewiesen.
In der bisher beschriebenen Form eignet sich die symbolische Ausführung nur für die Ausführung rein sequentieller Programme ohne Verzweigungen. Verzweigungen sind bei unserem Vorgehen aber an mehreren Stellen nötig:

1. bei bedingten Sprüngen, die es in Java sowohl mit einem Sprungziel als auch als Mehrfachverzweigung gibt.
2. bei im- oder expliziten Exceptions.
3. bei virtuellen Methodenaufrufen.
4. bei Array- oder Objektinitialisierung.

Bei expliziten Exceptions kommt in dem Java Bytecode ein `athrow` vor, das eine Exception in den aktuellen Kontext wirft. Implizite Exceptions werden von einzelnen Bytecode-Befehlen situationsabhängig geworfen, etwa wenn ein `idiv` eine `0` auf dem Stack als Divisor vorfindet. Welche Exceptions von welchem Code verarbeitet werden, wird in so genannten Exception-Handlern festgelegt, die für einen Exception-Typ und einen Code-Bereich das Sprungziel für die Behandlung der Exception beinhalten.
Bei Methodenaufrufen ist eindeutig zu bestimmen, welches Objekt zum Aufruf der Methode in dem aktuellen Kontext verwendet wird. Um dem dynamischen Binden, das Java unterstützt, gerecht zu werden, sollten auch Objekte von (Unter-)Klassen mit in betracht gezogen werden. In diesem Sinne findet also auch hier beim Methodenaufruf eine bedingte Verzweigung statt, da hier ggf. sukzessive Aufrufe unterschiedlicher Methoden zu betrachten sind.
Wie schon in der Einleitung beschrieben baut die SJVM während der Ausführung ein Bedingungssystem auf, das den aktuellen Pfad durch das Programm beschreibt. Betrachten wir als Beispiel die Instruktion `if (a>3) B else C`. Für die Ausführung von `B` müssen wir `a>3` dem globalen Bedingungssystem hinzufügen, für die Ausführung von `C` hingegen die Negation von `a>3`, nämlich `a<=3`. Alle numerischen und boolschen Bedingungen, die als Verzweigungsbedingungen oder Exceptionbedingung den Programmfluss bestimmen, gehen dabei in ein globales Bedingungssystem ein. Dieses wird von dem weiter unten beschriebenen Constraint Solver Manager verwaltet und bei Bedarf gelöst.
Um aber überhaupt eine Entscheidung treffen zu können, ob (zuerst) `B` oder `C` ausgeführt werden soll, werden eine Verzweigungsstrategie und Constraint Solver benötigt, und um schließlich sowohl `B` und `C` nacheinander ausführen zu können, wird ein Backtrackingmechanismus benötigt.
In der Verzweigungsstrategie wird festgelegt, welche und wieviel Analyse durchgeführt wird, um ein möglichst günstiges Verzweigungsziel zu bestimmen. Ein Beispiel hierfür ist die Analyse, ob weitere Testfälle überdeckt werden können. Für den Befehl `if_icmpgt 47` in Zeile 2 in Abbildung 1 kommen als Sprungziel Zeile 5 oder 47 in Frage. Da bereits eine einfache Analyse ergibt, dass in Zeile 47 bis zum Programmende in Zeile 48 keine weitere Variable ausgelesen wird. Sollen für diesen Programmcode nun etwa Def-Use-Ketten überdeckt werden, so kann dementsprechend von einem Sprung zu Zeile 47 abgesehen werden und statt dessen direkt Zeile 5 angesprungen werden.
Bevor die virtuelle Maschine einen Sprung dann tatsächlich ausführt, fügt sie die Sprungbedingung dem globalen Bedingungssystem hinzu und überprüft mit Hilfe des Constraint Solver Managers, ob das globale Bedingungssystem weiterhin lösbar bleibt. Sollte das nicht der Fall sein, so ist das Sprungziel nicht erreichbar, und weitere Sprungziele werden untersucht.
Bei dieser Untersuchung weiterer Sprungziele kann sich aber auch herausstellen, dass es kein weiteres gültiges Sprungziel gibt, und somit eine weitere Programmausführung

nicht mehr möglich ist. Eine vergleichbare Situation tritt ebenfalls ein, wenn eine geworfene Exception nicht gefangen wird und zur Programmtermination führt, oder die Methodenausführung regulär mit einem Rücksprung beendet wird. In diesem Fall bestimmt die SJVM mittels des Constraint Solver Managers eine ausgewählte Lösung des globalen Bedingungssystems, und konstruiert hieraus einen Testfall bestehend aus den Parametern der Methode und ggf. dem Rückgabewert. An dieser Stelle wird auch überprüft, ob bereits alle gewünschten Testfälle überdeckt sind oder nicht. Sollte dies der Fall sein, so kann die Ausführung des Werkzeugs enden.
Ist das nicht der Fall, so müssen weitere alternative Ausführungspfade betrachtet werden. Die SJVM geht dabei zu dem jeweils letzten Verzweigungspunkt zurück, um noch nicht ausgeführte Verzweigungen zu auszuprobieren. Dieses Verfahren, dass aus der Implementierung (funktional) logischer Programmiersprachen bekannt ist, nennt sich Backtracking [HK95, MK90].
Der Grundgedanke des Backtracking ist es, für jede Verzweigung so genannte Choice Points zu erzeugen und auf einen Stack zu legen, wobei sich mit denen im Choice Point gespeicherten Informationen der Programmzustand unmittelbar vor der Verzweigung wiederherstellen lässt.
Ein naiver Ansatz zur Implementierung eines solchen Choice Points ist es, den gesamten Zustandsraum der virtuellen Maschine zu kopieren und nach Ausführung wieder herzustellen. Da dieses Verfahren jedoch mit einem hohen Speicherverbrauch belastet ist, soll an dieser Stelle ein Verfahren vorgestellt werden, dass in den Choice Points nur die wesentlichen Daten speichert. Um diese Beschreibung besser verständlich zu gestalten, wird dieses Verfahren anhand ausgewählter Bytecode-Befehle vorgestellt.
Als erstes wird eine bedingte Verzweigung betrachtet und am Beispiel von `if_icmpgt 47` aus Zeile 2 von Abbildung 1 erläutert und in Abbildung 2 dargestellt. Bei dieser Ver-

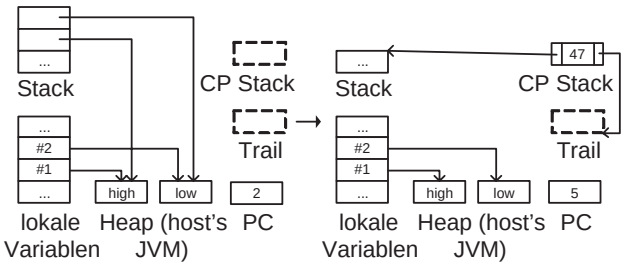

Abbildung 2: Ausführung von `if_icmpgt 47`

zweigung soll ein Sprung erfolgen, wenn `low` größer als `high` ist. Nun betrachten wir zunächst den Fall, dass wir den Sprung zur Zeile 47 nicht durchführen wollen, sondern die Ausführung bei Zeile 5 fortsetzen. Zunächst reicht die SJVM dafür die Bedingung `low<=high` an den Constraint Solver Manager weiter. Davon ausgehend, dass diese den globalen Bedingungsspeicher nicht unlösbar macht, wird nun ein neuer Choice Point instanziiert. In ihm wird die Adresse der noch verbleibenden Alternative und die aktuelle Methode vermerkt, und ein Zeiger auf den Stack und den Trail gesetzt. Der Trail ist ein Stack, auf dem Veränderungen des Operandenstack, der lokalen Variablen und im Heap gespeichert werden. Die Veränderungen werden dabei nur dann auf den Trail geschrieben, wenn Veränderungen an lokalen Variablen oder im Heap vorgenommen werden oder ein

Operandenstack-Element entfernt wird, das unterhalb des Stackpointers auf dem Choice Point liegt.
Ein Beispiel für die Verwendung des Trails findet sich in Abbildung 3 für den Befehl `istore 4` in Zeile 10. Wie aus der Graphik ersichtlich, wird in den Befehlen 5-9, die ja

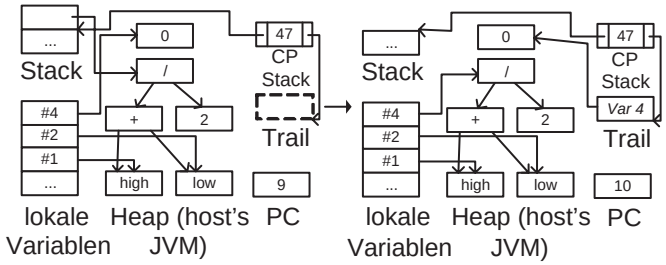

Abbildung 3: Ausführung von `istore 4`

in dem Beispiel direkt der Ausführung des Befehls `if_icmpgt 47` folgen, der Ausdruck `(low+high)/2` aufgebaut. Mit Ausführung von `istore_4` wird `(low+high)/2` jetzt der Variable `mid` zugewiesen, die vorher den Ausdruck `0` bei Initialisierung der Methode erhalten hatte. Dementsprechend wird auf den Trail ein neuer Eintrag gelegt, der den alten Ausdruck für `mid` sichert.
Ein weiterer interessanter Aspekt beim Backtracking ist das Verhalten von Referenztypen wie Klassen, Interfaces und Arrays, das in der Folge am Beispiel von Arrays exemplarisch beschrieben werden soll.
Die Länge eines Arrays kann entweder zum Zeitpunkt der Kompilierung feststehen (z.B. `int[] x = new int[2]`) oder erst zur Laufzeit bestimmt werden (etwa bei Übergabe eines Arrays als Parameter einer Methode). Während sich die Arraybehandlung im ersten Fall einfach handhaben lässt, kommt für den zweiten erneut der Backtrackingmechanismus zum Einsatz. Dabei wird die symbolische Ausführung zunächst mit einem Verweis auf das „kleinstmögliche Array "begonnen, nämlich mit n̂ull. Dies entspräche im Fall der realen Ausführung für die Beispielmethode `binsearch` dem Aufruf `binsearch(null, low, high, x)`. Diese Ausführung endet bereits bei dem ersten Aufruf von `iaload` in Zeile 15, da hier für eine `null`-Referenz eine nicht gefangene `NullPointerException` geworfen wird. Dementsprechend wird ein Backtracking zu der Stelle ausgeführt, an der das Array bereitgestellt wurde, und die Ausführung mit einem Array der Länge 0 fortgesetzt. Diese endet ebenfalls beim ersten Erreichen der Zeile 15, da hier eine nicht gefangene `ArrayIndexOutOfBoundsException` ausgelöst wird. Nach einem erneuten Backtracking zu der Stelle, an der das Array bereitgestellt wurde, wird die Ausführung dann mit einem einelementigen Array fortgesetzt, dieses Verfahren wiederholt sich mit steigender Arraygröße, bis entweder alle Testfälle überdeckt worden sind oder eine festgelegte Arraymaximalgröße erreicht worden ist.
Am Ende der Testfallerzeugung werden die Testfälle noch überprüft, ob die reale Ausführung den gleichen Pfad durchläuft wir die symbolische Ausführung. Ein Unterschied kann in seltenen Fällen durch von der JVM abweichende Rechenungenauigkeiten auftreten, z.B. bei numerisch instabilen Rechnungen, etwa bei dem Vergleich zweier `double`-Zahlen über ein `==`. Sollte sich durch leichte Korrekturen der Eingabewerte der von der

SJVM eingeschlagene Pfad nicht mit dem von der JVM eingeschlagenen Pfad zur Deckung bringen lassen, so wird eine entsprechende Meldung ausgegeben. In jedem Fall werden dem Benutzer aber nur korrekte Testfälle. Auf Grund des Halteproblems kann die Vollständigkeit des Werkzeugs nicht prinzipiell garantiert werden, was aber in der Praxis kein Problem darstellt.

3 Lösen der symbolischen Constraints

Der Constraint-Solver-Manager (CSM) fungiert als ausschließliche Schnittstelle zwischen der symbolischen Ausführungseinheit des Test-Tools und den einzelnen Lösungs-Algorithmen für die Verarbeitung verschiedenster Constraint-Typen. Seine Hauptaufgabe besteht darin, die während der symbolischen Ausführung erzeugten Bedingungen schrittweise entgegenzunehmen und bei Bedarf zu überprüfen, ob die momentan vorhandenen Gleichungs- und Ungleichungssysteme noch innerhalb der Domänen der involvierten Variablen lösbar sind. Gelangt die SJVM an einen Punkt, an dem sie das Ende eines Testpfades erkannt hat, ist es Aufgabe des CSM, eine exemplarische Lösung auszugeben, die dann genau den Vorbedingungen zur Ausführung des Testfalls entspricht. Um dies gewährleisten zu können, greift das Constraint-Solver-System auf eine breite Basis unterschiedlicher Lösungs-Algorithmen zurück, welche eine Handhabung der durch Java-Operatoren generierbarer Bedingungen ermöglichen. Bei dem von uns verfolgten Ansatz kann diese Basis mittels einer flexiblen Schnittstelle, über die weitere Solver auch nachträglich eingebunden werden können, beliebig erweitert werden.

Um während der Laufzeit nicht nach jeder zusätzlich eintreffenden Bedingung auch die Transformationen und Berechnungen auf den zuvor eingetroffenen Constraints erneut durchführen zu müssen, liegt die Forderung nach einer intelligenten Speicherung und Verwaltung der für ihn relevanten Informationen nahe. In dem aktuellen Prototyp wird dies über einen Constraint-Stack realisiert, der neben den inkrementell hinzugefügten und später wieder entfernten Constraints auch Informationen über bereits erkannte Unabhängigkeiten von Constraintgruppen, zuletzt verwendete Lösungsalgorithmen, Lösungen von Teilsystemen und interne Zustände der Solver gespeichert werden. Zudem werden momentan als weitere effizienzsteigernde Maßnahme verschiedene Strategien der frühzeitigen Vereinfachung bzw. Minimierung einzelner Teilbedingungen zu unterschiedlichen Zeitpunkten erprobt.

3.1 Transformation der Ausdrücke in Polynomgleichungssysteme

Die meisten bekannten Algorithmen zur maschinellen Lösung mathematischer Gleichungs- bzw. Ungleichungssysteme sind nicht dafür ausgelegt, Kombinationen beliebig kompliziert aufgebauter Terme zu verarbeiten, sondern gehen in der Regel davon aus, dass die Problemstellungen durch Linearkombinationen einzelner Variablen oder, bei nichtlinearen Problemen, in polynomieller Form vorliegen. Bei der symbolischen Auswertung von Java Programmen hat die eingesetzte Virtuelle Maschine jedoch keinen direkten Einfluss auf die Struktur der später zu verarbeitenden Bedingungen, weshalb vor der Übergabe der durch die SJVM gesammelten Bedingungen eine Transformation in eine polynomielle Darstellung erfolgen muss. Nach einem Blick in die Spezifikation des Befehlssatzes der

Java Virtual Machine fallen dabei besonders die ganzzahlige Division, der Modulo- bzw. Remainder-Operator, die verschiedenen Arten der Typumwandlungen (Typecasts) und die bitmanipulierenden Operationen über den ganzzahligen Datentypen (`>>`, `<<`, `<<<`, `&`, $|$, $\sim$) auf, welche durch die traditionelle Algebra nur bedingt unterstützt werden. Erste prototypische Implementierungen habend dabei ergeben, dass die erstgenannten Operationen (Division, Modulo, Typecast) noch mit vertretbarem Aufwand in den Griff zu bekommen sind, während die Bitoperationen zwar nicht als unmöglich handhabbar erscheinen, jedoch mit einem erheblich größeren Overhead verbunden sind. Eigene Beobachtungen und eine Untersuchung [DJ01] des Laufzeitverhaltens einer Java Virtual Machine anhand verschiedener Benchmark-Suites haben jedoch ergeben, dass Bitoperationen in Anwendungssystemen eine äußerst untergeordnete Rolle spielen, so dass auch wir ihre Berücksichtigung vorerst zurückgestellt haben.

Aus Platzgründen soll die Transformation von Constraints mit enthaltenen Modulo- Divisions oder Typecast-Operationen in eine polynomielle Darstellung hier nur exemplarisch am Beispiel der ganzzahligen Division ausführlich dargestellt werden:

Während bei einer Division, deren Operanden Ausdrücke der Datentypen `float` oder `double` sind, eine einfache Multiplikation mit dem Nenner des Quotienten (unter Berücksichtigung des Vorzeichens bei Ungleichungen) zur Beseitigung der Division ausreicht, muss bei einer Division zweier Ausdrücke der Datentypen `byte`, `char`, `short`, `int` oder `long` die Rundung des Ergebnisses in Richtung 0 berücksichtigt werden, welche von der Java Language Specification vorgeschrieben wird. Diese Vorgabe hat zur Folge, dass keine einheitliche Auf- oder Abrundung angewandt werden kann, sondern im Falle eines negativen Ergebnisses aufgerundet und bei positiven Ergebnissen abgerundet werden muss. Bei Berücksichtigung der möglichen Vorzeichen des Nenners und Zählers ergibt sich die folgende Definition für ganzzahlige Divisionen in Java:

$$\left(\frac{a}{b}\right)_{\texttt{int}} = sgn(a) \cdot sgn(b) \cdot \left\lfloor \frac{\|a\|}{\|b\|} \right\rfloor \tag{1}$$

Hierbei sei die untere Gaußklammer wie folgt definiert: $\lfloor y \rfloor = z$ mit $y - 1 < z \leq y$. Da sich die Gaußklammer, die Betragsfunktion und die sgn-Funktion jedoch nur sehr umständlich weiter verarbeiten lassen, werden diese im folgenden durch einfache Umformungen entfernt. Multipliziert man Gleichung (1) mit den durch $sgn(a)$ und $sgn(b)$ repräsentierten Vorzeichen der beiden Operanden, und setzt das Ergebnis in die Definition der Gaußklammer ein, ergibt sich:

$$\frac{\|a\|}{\|b\|} - 1 < sgn(a) \cdot sgn(b) \cdot \left(\frac{a}{b}\right)_{\texttt{int}} \leq \frac{\|a\|}{\|b\|} \tag{2}$$

Nach Multiplikation mit $\|b\|$ führt eine Fallunterscheidung der möglichen Vorzeichen von a und b zu:

$$\begin{cases} a \geq 0, b > 0 & : \quad a - b < b \cdot \left(\frac{a}{b}\right)_{\texttt{int}} \leq a \\ a \geq 0, b < 0 & : \quad a + b < b \cdot \left(\frac{a}{b}\right)_{\texttt{int}} \leq a \\ a < 0, b > 0 & : \quad a + b > b \cdot \left(\frac{a}{b}\right)_{\texttt{int}} \geq a \\ a < 0, b < 0 & : \quad a - b > b \cdot \left(\frac{a}{b}\right)_{\texttt{int}} \geq a \end{cases} \tag{3}$$

Durch die oben aufgeführten Überlegungen lässt sich nun eine ganzzahlige Division mit Hilfe einer einfachen Substitution des Quotienten durch eine Hilfsvariable, an welche

zusätzliche Bedingungen geknüpft werden, ersetzen. Für jede Substitution eines Quotienten $\left(\frac{a}{b}\right)$ durch die Hilfsvariable c gelten dann folgende Bedingungen (Die Bedingungen $b > 0$ bzw. $b < 0$ können dabei vernachlässigt werden, da diese sich implizit aus den restlichen Bedingungen herleiten lassen):

$$\begin{array}{l}((a \geq 0) \wedge (b \cdot c \leq a) \wedge ((a - b < b \cdot c) \vee (a + b < b \cdot c)))\vee \\ ((a < 0) \wedge (b \cdot c \geq a) \wedge ((a + b > b \cdot c) \vee (a - b > b \cdot c)))\end{array} \tag{4}$$

Auf den ersten Blick erscheint dieses neue System von Nebenbedingungen zwar relativ unhandlich. Sind allerdings die Vorzeichen des Nenners und/oder des Zählers bekannt, erweisen sich große Teile der Nebenbedingungen schnell als widersprüchlich, so dass sich die Komplexität der Nebenbedingungen nach entsprechenden Vereinfachungen stark reduziert. Für das einfache Beispiel $\frac{a}{3} = 5$ ergäbe sich so:

$$\begin{array}{l}(c = 5)\wedge \\ (((a \geq 0) \wedge (3 \cdot c \leq a) \wedge ((a - 3 < 3 \cdot c) \vee (a + 3 < 3 \cdot c)))\vee \\ ((a < 0) \wedge (3 \cdot c \geq a) \wedge ((a + 3 > 3 \cdot c) \vee (a - 3 > 3 \cdot c))))\end{array} \tag{5}$$

Hier lässt sich für die Variable c direkt der Wert 5 ablesen, welcher in die restlichen Bedingungen eingesetzt werden kann. Nach Ausmultiplizierung und einigen trivialen Umformungen erhält man:

$$\begin{array}{l}((a \geq 0) \wedge (a \geq 15) \wedge ((a < 18) \vee (a < 12)))\vee \\ ((a < 0) \wedge (a \leq 15) \wedge ((a > 12) \vee (a > 18)))\end{array} \tag{6}$$

Das einzige nicht widersprüchliche, konjugierte System von Bedingungen, welches sich hieraus erzeugen lässt, ist $(a \geq 15) \wedge (a < 18)$. Die Umformung hat den Lösungsraum also nicht verändert.

Auch die wertebereichseinschränkenden Typecasts (z.B. `double`→`int`), welche im übrigen ähnlich der Division einer Rundung in Richtung 0 entsprechen, und die Modulo-Operation lassen sich durch ähnliche Umformungen in einfacher handhabbare Terme transformieren.

3.2 Wahl geeigneter Solver

Wie bereits erwähnt, greift der CSM zur endgültigen Lösung der einzelnen Constraints auf unterschiedliche, für verschiedene Constraint-Klassen optimierte Lösungsalgorithmen zurück. Da es bei der Implementierung des CSM ein wichtiges Ziel war, möglichst exakte Ergebnisse zu liefern, wurde bei der Auswahl der Algorithmen darauf geachtet möglichst viele der zu bearbeitenden Constraints symbolisch zu lösen. Erst wenn die symbolischen Berechnungen keine verwertbaren Ergebnisse mehr liefern können, greift der CSM auf eine numerische Behandlung der Gleichungssysteme zurück.

Die auftretenden Constraints innerhalb unseres Systems lassen sich wie folgt klassifizieren: Als erstes gibt es rein boolsche Constraints, welche, da keine Typumwandlungen von anderen Typen zum Typ `boolean` oder umgekehrt vorgesehen sind, immer als unabhängige Systeme gelöst werden können. Da es zudem keine Bytecode-Instruktionen zur Negation boolscher Werte oder zur Verknüpfung über die Operationen *und* und *oder* gibt, und diese stattdessen vom Compiler durch bedingte Sprünge realisiert werden, können die Werte für die Belegung boolscher Eingabevariablen für gegebene Pfade leicht abgelesen werden. Ein aufwändiges Solver-System ist hierfür nicht notwendig.

Die restlichen, nicht boolschen Probleme lassen sich zunächst in lineare und nichtlineare Probleme aufteilen, wobei gerade bei den linearen Problemen eine Unterscheidung von Gleichungssystemen und Ungleichungssystemen mit starken oder schwachen Ungleichungen sinnvoll erscheint. Während reine Gleichungssysteme über Fließkommazahlen einfach und effizient mit der Gauss-Jordan-Elimination [Ap03] gelöst werden können, bietet sich für Probleme mit Ganzzahligkeitsbedingungen und schwachen Ungleichungen der aus der Optimierung bekannte Simplex-Algorithmus mit aufgesetztem Branch&Bound an. Starke Ungleichungen können hingegen symbolisch korrekt durch die Fourier-Motzkin-Elimination [DE73, Ap03] bearbeitet werden, welche auch dazu verwendet werden kann, gemischt ganzzahlige Probleme mit starken Ungleichungen für eine Bearbeitung mit Hilfe des Simplex-Algorithmus vorzubereiten.

Für nichtlineare Constraints gibt es hingegen nur begrenzte Möglichkeiten der symbolischen Verarbeitung, von denen für unsere Zwecke der Buchberger-Algorithmus [Bu85] am interessantesten erscheint, welcher gegebene Polynomgleichungssysteme in sog. Gröbner-Basen überführen kann, welche in Ihrer Struktur im Optimalfall der dreiecksähnlichen Gestalt eines Gleichungssystems nach der Gauss-Elimination ähnelt. Nicht lösbare Polynomgleichungssysteme können so einfach erkannt werden und auch bei endlichen Lösungsmengen ist eine Bestimmung einzelner Lösungen mit einfachen Mitteln möglich. In vielen Fällen sind die gesammelten Constraint-Systeme jedoch stark unterbestimmt und besitzen unendliche Lösungsräume, was dazu führt, dass die Dreiecksform der Gröbner-Basen weniger stark ausgeprägt ausfällt und eine Bestimmung reeller Lösungsinstanzen ähnlich kompliziert ist, wie beim Ausgangssystem. An dieser Stelle muss der CSM die symbolische Lösung der Constraints abbrechen und auf numerische Verfahren, wie z.B. das Newton-Verfahren, Regula-Falsi oder das Bisektionsverfahren zurückgreifen [BS93].

4 Verwandte Arbeiten und Zusammenfassung

Gupta, Mathur and Soffa [GM00] haben einen Kantenauswahl-Algorithmus vorgestellt, der Eingabedaten generiert, die zur Ausführung einer gewählten Kante führen. Ihr Ansatz ist streng numerisch und beinhaltet keine virtuelle Maschine. Gotlieb, Botella and Rueher [GB98] einen Constraint-basierten Ansatz für eine C-Untermenge vorgeschlagen, der Pfade für die Überdeckung jeder Anweisung im Programm generiert. Dabei waren weder das Testkriterium austauschbar, noch gab es mehrere, austauschbare Constraint Solver. Korels [Ko96] Ansatz besteht darin, Testdaten durch eine *data dependency* Analyse an Turbo Pascal Programmen durchzuführen und mittels Minimierungstechniken passende Eingabedaten zu produzieren. Ein älterer Ansatz zur Testdatenerzeugung kam von Ramamoorthy, Ho, and Chen [RH76], die Fortran-Code in eine Grundform transformiert haben, und die die Constraints, die den gesuchten Pfad durch das Programm beschreiben durch Vorwärtseinsetzen zu lösen versuchten. Lapierre et al. [LM99] schließlich implementierte ein Werkzeug, dass Pfad-beschreibende Constraints durch gemischt-ganzzahlige lineare Programmierung für C-Programme zu lösen versuchte.

Wir haben in diesem Artikel eine tragfähige Strategie vorgestellt, um Testfälle für die Java-Methoden mit einer symbolischen virtuellen Maschine und einem Constraint Solver Manger zu erzeugen. Der Java Bytecode wird dabei symbolisch ausgeführt, wobei

an Verzweigungen ein geeigneter Pfad in Abhängigkeit vom globalen Bedingungssystem und den bereits überdeckten Testfällen gewählt wird. Wenn mehrere Pfade besucht werden müssen, so verwenden wir Backtracking, um die Verzweigung erneut zu besuchen. Zu der vorgestellten Methode wurde bereits ein Prototyp entwickelt, der genauer in [ML03] beschrieben wurde. Basierend auf den Erfahrungen mit diesem Prototypen werden wir einige Punkte an unserer Arbeit erweitern und verbessern. Unser Constraint Solver System soll noch um weitere, nicht lineare, gemischt-ganzzahlige Constraint Solver erweitert werden und wir wollen die Benutzerkomponente um eine interaktive Komponente erweitern, mit der der Benutzer dynamisch Überdeckungen auf Kanten- und Anwendungsebene ergänzen kann, für die die Maschine dann Testfälle produziert. Schließlich planen wir eine symbolische Ausführungseinheit für die Microsoft Intermediate Language zu ergänzen.

Literatur

[Ap03] K.R. Apt. Principle of Constraint Programming. Cambridge UP, Cambridge, UK, 2003.

[Be90] B. Beizer. Software Testing Techniques. Van Nostr. Reinhold, 1990.

[BS93] M.S. Bazaraa, H.D. Sherali, C.M. Shetty. Nonlinear Programming, Theory and Algorithms. John Wiley & Sons, New York, NY, 1993.

[Bu85] B. Buchberger. Grobner Bases: An Algorithmic Method in Polynomial Ideal Theory. *In Multidimensional Systems Theory*, D. Reidel Publishing, Dordrecht, Holland, 1985.

[Cl76] L.A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering 2(3). 215-222, 1976.

[DE73] G.D. Danzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288-297, 1973.

[DJ01] D. Gregg, J. Power and J. Waldron. Platform independent dynamic Java Virtual Machine analysis: the Java Grande forum benchmark suite, *Concurrency and Computation Practice and Experience*, vol. 15(3-5), pp. 459-484, Wiley Interscience, 2003.

[Ed99] J. Edvardsson. A survey on Automatic Test Data Generation. *In Proc. of the ECSEL*, pages 21-28. 1999.

[GB98] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. *In ISSTA '98, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, March 1998.

[GM00] N. Gupta, A.P. Mathur, and M.L.Soffa. Generating Test Data for Branch Coverage. 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.

[GN72] R.S. Garfinkel, G.L. Nehmhauser. Integer Programming. John Wiley & Sons, NY 1972.

[HK95] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. *In Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

[MK90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodriguez-Artalejo. Graph-based Implementation of a Functional Logic Language, Procs. ESOP, LNCS 432, 1990, 271-290.

[Ko96] B. Korel. Automated Test Data Generation for Programs with Procedures. *In Proceedings of the ISSTA'96, San Diego, USA. Software Engineering Notes 21(3)*, May 1996.

[Ki76] J.C. King. Symbolic execution and Program Testing. Comm. of the ACM, 19(7), 1976.

[LM99] S. Lapierre et al. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. Int. Conf. on Software Maintenance 1999.

[ML03] R.A. Müller, C. Lembeck, and H. Kuchen. GlassTT - a Symbolic Java Virtual Machine Using Constraint Solving Techniques for Glass-Box Test Case Generation, *Technical Report* 102, University of Münster, 2003.

[RH76] C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the Automated Generation of Program Test Data. *In IEEE TSE, Vol. SE-2, No.4*, December 1976.

[TC98] N. Tracey, J. Clark, K. Mander und J. McDermid. An automated framework for structural test-data generation. ASE, 285-298 1998.