Automatische Erzeugung von Testfällen

Herbert Kuchen, Christoph Lembeck und Roger A. Müller

Universität Münster, Institut für Wirtschaftsinformatik, kuchen@uni-muenster.de

Extended Abstract

Bei den Ansätzen zum Testen von Software unterscheidet man zwischen dem funktionalen (black-box) Testen und dem strukturellen (glass-box) Testen [Be90, Li03]. Bei Ersterem wird ausgehend von der Spezifikation eine Menge von Testfällen betrachtet, durch die jedes Anwendungsszenario durch einen Testfall abgedeckt wird. Dieser Ansatz eignet sich nicht nur für kleinere Softwareeinheiten wie Module oder Klassen sondern auch für ganze Pakete und Subsysteme bis hin zu Gesamtsystemen. Allerdings werden Sonderfälle und durch die gewählte Implementierung hervorgerufene algorithmische Besonderheiten leicht übersehen, da diese aus der Spezifikation nicht ersichtlich sind. Zu nennen sind hier beispielsweise Reorganisationsoperationen in komplexen Datenstrukturen wie AVL-Bäumen und B-Bäumen. Dieser Nachteil tritt beim strukturorientieren Testen nicht auf. Hier geht man vom Programmcode aus und versucht, ihn durch Kontrollfluss-Pfade sinnvoll zu überdecken. Denkbar ist z.B. alle Kanten und Knoten des mit dem Code korrespondierenden Kontrollflussgraphen zu durchlaufen. Für die Praxis reicht dies allerdings oft nicht aus. Hier strebt man meist an, zumindest alle möglichen Datenflüsse zu überdecken. Hierbei betrachtet man Paare von Anweisungen, bei denen ein Wert berechnet wird, und Anweisungen, bei denen dieser Wert benutzt wird. Ziel ist es, alle solchen Anweisungspaaare, sogenannte defuse-Ketten, zu überdecken. Nachteil des strukturorientierten Testens ist, dass der Aufwand durch die Vielzahl der zu betrachtenden Pfade mit dem Umfang der zu testenden Softwarebausteine stark ansteigt und daher nur für das sogenannte Unit-Testing von einzelnen Modulen oder Klassen geeignet ist.

Im folgenden werden wir uns auf das strukturelle Testen konzentrieren. Hier sind vor allem zwei Probleme zu lösen. Zunächst muss ein System von Pfaden zusammengestellt werden, dass die gewünschte Überdeckung gewährleisten kann. Weiterhin muss für jeden Pfad ein Testfall konstruiert werden, der zu einem Durchlauf des Pfades führt. Ein Testfall besteht aus einer Testeingabe und einer dazu passenden, erwarteten Ausgabe. Bei der Erzeugung der Testfälle bekam der Tester bisher wenig Unterstützung.

Grundsätzlich lässt sich die Erzeugung von Testfällen für das strukturelle Testen in drei Klassen aufteilen: die zufällige, statische und dynamische Testfallerzeugung [TC98, Li03, Be90]. Bei der zufälligen Testfallerzeugung werden die Testfälle unter Verwendung eines Zufallszahlengenerators erstellt. Dies ist zwar einfach zu implementieren, lässt aber trotz hohen Aufwands keine Garantien über die Güte der Ergebnisse zu. Beim dynamischen Ansatz wird die Software tatsächlich ausgeführt, allerdings wird während der Ausführung eine Suche nach Testfällen durchgeführt. Die älteren Ansätze arbeiten dabei mit Minimierungen von Funktionen [MS76] in einem eingeschränkten Suchraum oder mit direkter, lokaler Suche [Ko90, Ko96] nach Testfällen während der Programmausführung. Aktuellere Ansätze verfolgen globale (heuristische) Optimierung zur Generierung struktureller Testdaten [TC98, Wa95, JS95, JS96]. Auf dieser Grundlage haben sich bisher keine praxistauglichen Tools erstellen lassen. In der Praxis verwendete Werkzeuge können i.d.R. lediglich für eine vorgegebene oder zufällig generierte Menge von Testfällen feststellen, ob die gewünschte Überdeckung erreicht wird.

Unser Beitrag besteht darin, dass die gewünschte Testfallmenge systematisch und automatisch generiert wird. Unser Ansatz ist zunächst auf die Programmiersprache Java beschränkt, lässt sich aber auf andere Sprachen übertragen. Die Grundidee besteht darin, dass die Eingabeparameter einer zu testenden Java-Methode als logische Variablen aufgefasst werden und ein Testfall dadurch bestimmt wird, dass das betrachtete Java-Programmstück in Anlehnung an Prolog als eine Art logisches Programm ausgeführt wird. Hierbei werden an Verzweigungsstellen für die Eingabeparameter Constraints erzeugt, die erfüllt werden müssen, damit der betrachtete Programmpfad durchlaufen wird. Ein Testfall ergibt sich dann als Lösung des Constraint-Systems, das sich nach Durchlaufen des betrachteten Pfades ergeben hat. Der Testfall ist dann ein Repräsentant der Äquivalenzklasse derjenigen Testfälle, die alle zum Durchlaufen dieses Pfades führen. Ein Repräsentant reicht für das Testen i.d.R. aus. Weitere äquivalente Testfälle würden nur die Testkosten erhöhen aber wegen des äquivalenten Verhaltens keine weiteren Erkenntnisse liefern.

Die Grundidee wird dadurch implementiert, dass wir den Java-Byte-Code mit einer symbolischen Java Virtual Machine bearbeiten, bei der der Wert einer Variablen i.a. keine Zahl sondern ein Ausdruck über den Eingabeparametern ist, die, wie erwähnt, als logische Variablen aufgefasst werden, denen noch kein Wert zugeordnet worden ist. Wenn die symbolische Rechnung an eine Verzweigungsinstruktion wie beispielsweise einen bedingten Sprung stößt, kann die auszuwählende Alternative nicht einfach wie bei Rechnungen auf konkreten Zahlen ermittelt werden, sondern es wird für jede Alternative ein Constraint generiert, das auf Verträglichkeit zum dem bereits generierten System von Constraints überprüft werden muss. Ist das neue Constraint hierzu wi-

derspruchsfrei, so kann die Alternative gewählt und das Constraint dem Constraintsystem hinzugefügt werden. Wenn nicht, braucht die Alternative nicht weiter betrachtet zu werden. Bleiben mehrer Alternativen möglich, so werden diese mit Hilfe eines Backtrackingmechanismus sukzessiv betrachtet. Hierdurch lassen sich dann auch Testfälle zu anderen Kontrollflusspfaden generieren.

Die symbolische Java Virtual Machine umfasst alle Komponenten einer konkreten Java Virtual Machine; insbesondere also Frame Stack und Heap. Zusätzlich enthält sie zur Realisierung des Backtracking-Mechanismus Komponenten, die aus abstrakten Machinen für Logikprogrammiersprachen wie Prolog bekannt sind. Zu nennen sind hier insbesondere ein Choice-Point-Stack, auf dem über Verzweigungspunkte buchgeführt wird, und ein Trail, auf dem festgehalten wird, welche durchgeführten Veränderungen bei einem Backtracking rückgängig gemacht werden müssen. Solche Komponenten kennt man insbesondere aus der Warren Abstract Machine für Prolog.

Zur Lösung der erzeugten Constraintsysteme verwenden wir ein Bündel von Constraint-Solvern. Jedes Constraint wird dem hierfür am besten geeigneten Constraint-Solver zugeordnet. U.a. verwenden wir neben der Gauß'schen Elimination zur Lösung linearer Gleichungssysteme einen Variablen-Emininationssolver zur Lösung von Systemen linearer Ungleichungen sowie einen (numerischen) Bisektionssolver zur Lösung von Polynomgleichungssystemen. Auch Ganzzahligkeitsforderungen werden bei Bedarf berücksichtigt.

Weitere Details findet man in [LC04, LM04, ML03, ML04].

Literatur

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

[JS95] B. Jones, H.H. Sthamer und D.E. Eyres: Generating test-data for Ada procedures using genetic algorithms. Genetic Algorithms in Engineering Systems: Innovations and Applications, 65-70, 1995.

[JS96] B. Jones, H.H. Sthamer und D.E. Eyres: Automatic structural testing using genetic algorithms, Software Engineering Journal 11(5), 299-306, 1996.

[Ko90] B. Korel: Automated software test data generation, IEEE Transactions on Software Engineering 16 (8), 134-146, 1990.

[Ko96] B. Korel: Automated test data generation for programs with procedures, ISSTA, 209-215, 1996.

[LC04] C. Lembeck, R. Caballero, R. Müller, H.Kuchen: Constraint Solving for Generating Glass-Box Test Cases, Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP), 19-32, Aachen, 2004.

[Li03] P. Liggesmeyer. Software Qualität – Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag 2003.

[LM04] C. Lembeck, R. Müller, H. Kuchen: Die Erzeugung von Testfällen mit einer symbolischen virtuellen Maschine und Constraint Solvern (in German), Informatik 2004, Bd. 2, Lecture Notes in Informatics P-51, 418-427, 2004.

[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.

[ML04] R. Müller, C. Lembeck, H. Kuchen: A Symbolic Java Vitual Machine for Test-Case Generation, Proceedings IASTED, 2004.

[MS76] W. Miller und D. Spooner: Automatic generation of floating-point test data, IEEE Transactions on Software Engineering 2 (3), 223-226, 1976.

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

[Wa95] A.L. Watkins: The automatic generation of test data using genetic algorithms, Proceedings of the 4th Software Quality Conference, 300-309, 1995.