EvoSuite

Automatic Test Suite Generation for Java

Adrian Uffmann
Matrikelnummer: 12043921
adrian.uffmann@campus.lmu.de

09.02.2020

Masterseminar Fuzz Testing

Zusammenfassung

EvoSuite ist ein Programm, welches mithilfe eines genetischen Algorithmus automatisiert JUnit-Testfälle aus Java-Bytecode generieren kann. Auf der Internetseite des Projektes (`http://www.evosuite.org/publications/`) werden 54 Publikationen zu EvoSuite und dessen Vorgänger $\mu Test$ aufgelistet. Diese Seminararbeit gibt einen Überblick über die Funktionsweise von EvoSuite.

1 Einleitung

Das Testen von Software ist ein wichtiger Teil der Softwareentwicklung, so schreiben Myers et al. [3]:

> it [is] a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost [are] expended in testing the program or system being developed.

Daher ist es kaum verwunderlich, dass es zahlreiche Werkzeuge gibt, die helfen sollen das Testen von Software zu automatisieren. EvoSuite geht noch einen Schritt weiter: hier wird nicht versucht das Testen der Software zu automatisieren, sondern das Erstellen von automatisierten Tests selbst durch Automatisierung zu vereinfachen. Der Fokus liegt dabei auf der Optimierung eines *test last* Ansatzes, d. h. getestet wird erst, nachdem das eigentliche Programm fertig implementiert ist. Bei diesem Ansatz würde sich ein Softwaretester zunächst Szenarien überlegen, mit denen er möglichst viele Pfade des Programms abdeckt und diese als Unittests ausformulieren. Mithilfe von sogenannten Orakeln wird dabei ein bestimmtes Verhalten des Programms

definiert. Verhält sich das Programm anders als von den Orakeln vorgegeben, so schlägt der Test fehl und es wurde ein Bug gefunden.

EvoSuite versucht nun diesen Prozess zu vereinfachen, indem es Unittests generiert, die das Verhalten des Programms möglichst gut beschreiben. Ein Softwaretester muss sich dann nicht mehr selbst Szenarien überlegen, sondern nur noch die generierten Testfälle auf unerwartetes Verhalten prüfen. Dazu nutzt EvoSuite einen genetischen Algorithmus mit dem zufällige Testfälle immer weiter verbessert werden, bis sie eine möglichst hohe Testabdeckung erreichen. Die Testfälle werden dann minimiert um die Arbeit des Softwaretesters zu vereinfachen und es werden mit sogenanntem *Mutation-Testing* möglichst repräsentative Orakel ausgewählt.

EvoSuite wurde seit 2010 immer weiter entwickelt und es sind verschiedene Algorithmen für das generieren von Testfällen implementiert, die über die Kommandozeile ein- und ausgeschaltet werden können. Dazu haben die Autoren von EvoSuite auf der Internetseite (`http://www.evosuite.org/publications/`) 54 Artikel über die Konzepte in und die Evaluation von EvoSuite und dessen Vorgänger $\mu Test$ veröffentlicht. Um den vorgegebenen Umfang einzuhalten, kann diese Seminararbeit nur Überblick über einen kleinen Teil von EvoSuite geben.

2 Genetische Algorithmen

Genetische Algorithmen gehören zu den Evolutionären Algorithmen. Die Idee dabei ist es, den Prozess der Evolution und natürlichen Auslese aus der Biologie zu simulieren, um Optimierungsprobleme zu lösen. Dazu werden Lösungskandidaten für das Optimierungsproblem als Individuen betrachtet. Die Menge der betrachteten Individuen nennt man auch Population oder Generation. Ausgehend von einer Anfangspopulation werden mit einer bestimmten Wahrscheinlichkeit durch Rekombination und/oder Mutation neue Individuen erzeugt. Bei einer Rekombination werden aus mehreren Eltern-Individuen ein oder mehrere neue Kind-Individuen erzeugt, während bei einer Mutation aus nur einem Individuum durch eine kleine Änderung ein neues Individuum entsteht. Die neuen Individuen werden anhand einer Fitnessfunktion bewertet und anhand von dieser Bewertung wird eine Menge der Individuen ausgewählt, um die neue Population zu bilden. Die restlichen Individuen werden verworfen.

Dieser iterative Prozess wird so lange ausgeführt, bis ein Individuum gefunden wurde, welches das Optimierungsproblem optimal löst, bis eine vorgegebene Anzahl an Generationen entwickelt wurde, oder bis die verfügbare Zeit aufgebraucht ist. In jedem Fall ist das Ergebnis des genetischen Algorithmus das beste Individuum.

Um ein konkretes Optimierungsproblem mit einem genetischen Algorithmus zu lösen, muss genau definiert werden: was ein Individuum ist, aus

welchen Individuen die Anfangspopulation besteht, wie die Rekombinations- und Mutationsoperatoren neue Individuen erzeugen und wie die Fitnessfunktion aussieht. Dies ist von Problem zu Problem unterschiedlich.

3 Erzeugen von Testfällen

3.1 Betrachten von ganzen Testsuites auf einmal

EvoSuite beginnt mit dem Generieren von Testfällen ohne Orakel. Dafür werden verschiedene Algorithmen angeboten, in dieser Seminararbeit wird jedoch nur das Verbessern von ganzen Testsuites hinsichtlich der Testabdeckung mit einem genetischen Algorithmus nach [1] beschrieben.

Die Verbesserung von ganzen Testsuites auf einmal ist dabei eine wichtige Neuerung, die vor EvoSuite, nach bestem Wissen der Autoren, nicht verwendet wurde. Nach [**?**] betrachteten ältere Werkzeuge nur einzelne Programmpfade auf einmal und versuchten einzelne Testfälle zu generieren, die die ausgewählten Pfade abdeckten. Ein großes Problem bei diesem Ansatz ist jedoch, dass nicht jeder Programmpfad auch erreichbar ist. Wenn versucht wird einen unerreichbaren Pfad zu erreichen, dann ist der dafür verwendete Aufwand verschwendet. Ob ein bestimmter Pfad überhaupt erreichbar ist, ist allerdings ein unentscheidbares Problem, da man sonst das Halteproblem lösen könnte. Bei diesem Ansatz ist es demnach sehr wichtig, mit welchen Programmpfaden begonnen wird.

Ein weiteres Problem ist Kollateralabdeckung. Wenn zunächst Testfälle generiert werden, die einfache Pfade nahe der Oberfläche des Programms abdecken, kann es passieren, dass dieselben Pfade auch erreicht werden, wenn später Testfälle für tiefere Pfade des Programms generiert werden. Dadurch werden die zuerst generierten Testfälle nutzlos, da sie keinen Beitrag mehr zur Testabdeckung beitragen und der Aufwand sie zu generieren war verschwendet.

EvoSuite vermeidet diese Probleme, indem die Testabdeckung der gesamten Testsuite auf einmal verbessert wird. Statt einzelne Programmpfade auszuwählen, die erreicht werden sollen, wird bei EvoSuite gemessen, wie weit die gesamte Testsuite davon entfernt ist, alle Programmpfade abzudecken.

3.2 Genetischer Algorithmus für das Generieren von Testsuites

Wie in Abschnitt 2 erwähnt, muss für die Nutzung eines genetischen Algorithmus zunächst definiert werden, wie ein Individuum aussieht. Da EvoSuite ganze Testsuites auf einmal betrachtet, ist ein Individuum eine Testsuite. Eine Testsuite ist dabei eine Menge von Testfällen ($T = \{t_1, t_2, ..., t_n\}$), wobei ein Testfall eine Sequenz von Anweisungen ist ($t = \langle s_1, s_2, ..., s_l \rangle$). In

EvoSuite wird zwischen 5 Arten von Anweisungen unterschieden:

- Primitive Anweisungen, wie `int var0 = 54`, oder `Object[] var1 = new Object[10]`.
- Zuweisungen, wie `var1[0] = new Object()`, oder `var2.maxSize = 10`.
- Feld-Anweisungen, wie `int var3 = var2.size`.
- Konstruktor-Anweisungen, wie `Stack var2 = new Stack()`.
- Methoden-Anweisungen, wie `int var4 = var2.pop()`.

Der Sprachumfang von Java erlaubt zwar deutlich mehr Arten von Anweisungen, wie z. B. `if`-Anweisungen, `while`- und `for`-Schleifen, diese sind jedoch nicht in Testfällen erwünscht, da Testfälle deterministisch sein sollen und keine Parameter erhalten, die sich auf den Kontrollfluss auswirken könnten.

Für den noch zu definierenden Mutationsoperator ist es wichtig festzuhalten, dass jede Anweisung einen Wert von einem bestimmten Typ bereitstellt, der in späteren Anweisungen verwendet werden kann. Für eine Anweisung s wird dieser Wert im folgenden als $v(s)$ bezeichnet.

Als nächstes müssen die Rekombination und Mutation definiert werden. Die Rekombination ist in EvoSuite sehr einfach gehalten. Hierbei werden mit einer bestimmten Wahrscheinlichkeit aus den beiden Eltern-Testsuites P_1 und P_2 die beiden Kind-Testsuites O_1 und O_2 erstellt. Zunächst wird ein Anteil $\alpha \in [0; 1]$ gewählt. Das Kind O_1 ergibt sich, indem der Anteil α der Testfälle von P_1 mit einem Anteil $1 - \alpha$ der Testfälle von P_2 kombiniert wird. Das Kind O_2 besteht dann aus den übrigen $(1 - \alpha) \times |P_1|$ Testfällen von P_1 und den übrigen $\alpha \times |P_2|$ Testfällen von P_2. Individuen können durch diese Rekombination nicht länger oder kürzer werden, als der längste, bzw. kürzeste Elternteil. Im Durchschnitt nähern sich dadurch alle Individuen der gleichen Anzahl an Testfällen an.

Die Mutation einer Testsuite T ist deutlich komplexer, als die Rekombination. Hierbei wird jeder Testfall t mit einer Wahrscheinlichkeit von $\frac{1}{|T|}$ mutiert. Im Durchschnitt wird bei der Mutation einer Testsuite also genau ein Testfall mutiert. Es ist aber auch möglich, dass keiner oder mehrere Testfälle auf einmal mutiert werden. Bei der Mutation eines Testfalls werden der Reihe nach folgende Operationen jeweils mit einer Wahrscheinlichkeit von $\frac{1}{3}$ durchgefüht:

1. Entfernen von Anweisungen.
2. Ändern von Anweisungen.
3. Einfügen von Anweisungen.

Beim Entfernen von Anweisungen wird jede Anweisung s_i mit einer Wahrscheinlichkeit von $\frac{1}{|t|}$ entfernt. Da der Wert $v(s_i)$ in einer späteren Anweisung benutzt werden könnte, müssen alle Anweisungen $\{s_j \in t | j > i\}$ überprüft werden, um sicherzustellen, dass ein valides Individuum entsteht. Wenn ein s_j den Wert $v(s_i)$ verwendet, dann wird dieser durch einen anderen verfügbaren Wert desselben Typs ersetzt. Wenn in dem Testfall keine andere Anweisung einen Wert mit demselben Typ bereitstellt, dann wird s_j rekursiv entfernt. Bleibt nach dem Entfernen von Anweisungen ein leerer Testfall übrig, dann wird dieser komplett gelöscht.

Beim Ändern von Anweisungen, wird wieder jede Anweisung mit einer Wahrscheinlichkeit von $\frac{1}{|t|}$ geändert. Hier werden verschiedene Arten von Anweisungen verschieden behandelt.

Bei Primitiven Anweisungen wird einfach der primitive Wert um einen zufälligen Δ-Wert erhöht oder erniedrigt, Arrays werden verlängert oder verkürzt und Zeichenketten werden mutiert. Bei der Verkürzung von Arrays muss genau wie beim Entfernen von Anweisungen darauf geachtet werden, dass spätere Anweisungen keine entfallenen Arrayindizies verwenden. Bei der Mutation von Zeichenketten werden, ähnlich wie bei der Mutation von Testfällen, neue Zeichen eingefügt, bestehende Zeichen geändert, und/oder alte Zeichen entfernt.

Bei Zuweisungen wird zufällig eine der beiden Seiten durch ein anderes Feld, bzw. einen anderen Wert ersetzt.

Für Feld-, Konstruktor- und Methoden-Anweisungen wird zufällig ein anderes Feld, ein anderer Konstruktor, oder eine andere Methode ausgesucht, deren Parameter mit den verfügbaren Werten erfüllt werden können. Dabei ist es z. B. auch möglich, dass aus einer Feld-Anweisung eine Methoden-Anweisung wird.

Beim Einfügen von Anweisungen wird dem Testfall mit einer Wahrscheinlichkeit von σ eine zufällige Anweisung hinzugefügt. Als Parameter für Methoden- oder Konstruktoraufrufe werden dabei die bereitgestellten Werte wiederverwendet oder neue Werte generiert, die direkt eingesetzt werden. Dies wird wiederholt, bis in einem Schritt mit einer Wahrscheinlichkeit von $1 - \sigma$ keine neue Anweisung eingefügt wird. Durch dieses Vorgehen sinkt die Wahrscheinlichkeit weitere Anweisungen einzufügen exponentiell. i oder mehr Anweisungen werden also nur mit einer Wahrscheinlichkeit von σ^i eingefügt.

Mit dem bis hier beschriebenen Mutationsoperator können Testfälle länger und kürzer werden und sogar entfallen. Es ist aber bisher nicht möglich, dass neue Testfälle entstehen und auch durch die Rekombination kann die Anzahl der Testfälle in einer Testsuite nicht größer werden, als die Anzahl der Testfälle im größeren Elternteil. Aus diesem Grund wird bei der Mutation einer Testsuite nach dem Mutieren der Testfälle mit einer Wahrscheinlichkeit von σ' ein zufälliger neuer Testfall erzeugt. Genau wie bei dem Einfügen

von Anweisungen werden hier mit exponentiell sinkender Wahrscheinlichkeit weitere Testfälle hinzugefügt.

Um einen Testfall zu Erzeugen, wird eine zufällig Länge l $[1, L]$ gewählt (wobei L die maximale Länge von Testfällen ist) und es werden so lange neue Anweisungen, wie bei der Mutation von Testfällen, eingefügt, bis der neue Testfall die gewünschte Länge erreicht hat.

Nun da die Mutation von Testsuites definiert ist, ist es leicht die Anfangspopulation zu definieren. Als Anfangspopulation wird eine vorgegebene Anzahl an Testsuites erzeugt, indem jede mit einer vorgegeben Anzahl an zufälligen Testfällen befüllt wird. Die zufälligen Testfälle werden dabei genau wie bei der Mutation von Testsuites erstellt.

4 Erzeugen von Orakeln mit Mutation-Testing

Dieser Abschnitt stützt sich vor allem auf [2].

Mutation bedeutet hier etwas anderes, als bei genetischen Algorithmen.

5 Search-Based Software Testing Competition

6 Praxistauglichkeit

7 Fazit

Literatur

[1] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, feb. 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.14. URL `http://www.evosuite.org/wp-content/papercite-data/pdf/tse12_evosuite.pdf`.

[2] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, march-april 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.93. URL `http://www.evosuite.org/wp-content/papercite-data/pdf/tse12_mutation.pdf`.

[3] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The Art of Software Testing*. New Jersey: John Wiley & Sons, Inc., 2004.