Daikon - Auffinden von Programminvarianten zur Laufzeit

Philipp Glaser

21.06.2008

Daikon [Dai] ist ein Programm zum Auffinden von wahrscheinlichen Invarianten zur Laufzeit. Diese können dann in Zusicherungen (assertions), in Programmdokumentationen und formalen Spezifikationen verwendet werden. Dazu beobachtet Daikon die Variablen und gibt die Eigenschaften aus, die zur Laufzeit eintrafen. Es funktioniert mit C/C++, Java und Perl-Programmen.

Daikon wurde bereits benutzt zum Erstellen von Testdaten, Prognostizieren von Inkompatiblilitäten von Komponenten, zum Reparieren von inkonsistenten Datenstrukturen und für andere Aufgaben.

1 Einführung

In dieser Seminarausarbeitung wird nach einem kurzen Vorstellen der Nutzen von Invarianten auf die Funktionsweise des Programms Daikon eingegangen. Dabei wird zuerst darauf eingegangen wie Daikon Invarianten findet und diese filtert. Anschließend wird kurz auf die Effizienz eingegangen und abschließend wird kurz die Anwendung des Programms vorgestellt. Da es unwahrscheinlich ist von dem Author eines Programms zu erwarten, dass er in seinem Code Invarianten explizit dokumentiert, stellt die automatische Ableitung von Invarianten eine Alternative dar. Einen kurzen Überblick über Daikon geben [ECGN01] und [EPG$^+$07].

2 Nutzen von Invarianten

Programminvarianten sind hilfreich beim Verstehen, Ändern und Testen von Programmen. Allerdings sind nur eine kleiner Teil der Invarianten eines Programms explizit im Programmcode aufgeführt. Viele Invarianten werden daher häufig bei Programmmodifikationen verletzt und führen so zu neuen Fehlern in Programmen.[Ern00]

Mit Invarianten lassen sich das Verhalten und die Aufgaben eines Programmteils präzise und formal festhalten. Das explizite Festhalten von Invarianten kann zu einem saubereren Entwurf und zu einer besseren Implementierung führen.

Invarianten dienen ebenfalls der Dokumentation von Programmcode. Als solche helfen sie beim Verstehen von Programmen, was wiederum eine Voraussetzung für Programmmodifikationen darstellt. Zusätzlich zu den in der Dokumentation erwähnten Invarianten helfen dynamisch aufgefundene Invarianten die dokumentierten zu überprüfen und zu präzisieren.

Zum Überprüfen von Annahmen zu einem Programm sind Invarianten ebenfalls hilfreich. Entdeckte Invarianten können explizit als Zusicherungen in den Programmcode geschrieben werden und verhindern so, dass sich möglicherweise bei Veränderungen des Codes Fehler einschleichen. Viele Programmfehler entstehen dadurch, dass viele Invarianten nicht explizit im Code aufgeführt werden, da die Programmierer diese übersehen. Dies war auch die ursprüngliche Motivation Programme zu entwickeln, die beim Auffinden von Invarianten helfen.

Desweiteren können Invarianten die zur Laufzeit gefunden werden, helfen bestimmte Muster beim Programmablauf zu erkennen. So können zum Beispiel typische Pfade beim Programmablauf (bei typischen Eingabedaten) erkannt werden. Ungewöhnliche und seltene Voraussetzungen können auf Fehler oder Spezialfälle deuten.

Zur Laufzeit gefundene Invarianten können jedoch nicht nur Erkenntnisse über ein Programm bringen, sondern ebenfalls über die verwendeten Testdaten. So kann man die Testsuite anpassen, falls die Abdeckung der Programmpfade mit den verwendeten Testdaten zu gering ist.

Diese Invarianten können ebenfalls genutzt werden, um die Laufzeit eines Programm zu optimieren. Übersetzer könnten so optimierten Code für häufige Fälle einfügen (*profiling*).

Bei vielen Methoden zur Validierung von Programmen ist es schwierig oder aufwändig die zu überprüfenden Eigenschaften zu spezifizieren. Automatisch gefundene Invarianten können zur (semi-)automatischen Prüfung herangezogen werden.

3 Auffinden von Invarianten

Daikon findet Invarianten in mehreren Schritten: Zuerst schreibt ein (programmiersprachenspezifisches) Daikon-Frontend die nötigen Dateien zur Ablaufverfolgung (*trace*) und füttert anschließend das Programm mit Testdaten. Zum Schluss liest Daikon die Logdateien, generiert mögliche Invarianten, überprüft sie und gibt sie gegebenenfalls aus.

Daikon gibt Invarianten bei Funktionseintritt und -austritt sowie Schleifeninvarianten an.

3.1 Arten von gefundenen Invarianten

Selbstverständlich findet Daikon nicht alle Invarianten eines Programms, sondern beschränkt sich auf folgende Arten:

Invarianten über einfache Variablen Dazu gehören Variablen mit konstanten Wert, uninitialisierte Variablen und Variablen die nur wenige Werte annehmen.

Invarianten über einzelne numerische Variablen Variablen deren Wertebereich beschränkt ist, Variablen die modulo a immer/nie b ergeben und unter bestimmten Bedingungen auch Variablen, die immer ungleich 0 sind.

Invarianten über zwei numerische Variablen Lineare Beziehungen ($y = ax + c$), Vergleiche ($x < y$), Funktionen ($x = fn(y)$), Invarianten über $a + b$ und $a - b$ (siehe Invarianten über einzelne numerische Variablen)

Invarianten über drei numerische Variablen Lineare Beziehungen und Funktionen

Invarianten über eine Sequenz Wertebereich (Strings, Arrays), Ordnung, Invarianten über alle Elemente

Invarianten über zwei Sequenzen Lineare Beziehung (Elementweise),Vergleich, Untermengen und Umkehrung

Invarianten über eine Sequenz und eine numerische Variable Mitgliedschaft ($a \in b$)

Allerdings ist Daikon darauf ausgelegt, das man relativ einfach neue Arten von aufgespürten Invarianten hinzufügen kann. Außerdem leitet Daikon aus den Sequenzen noch mehrere Variablen ab: Die Länge der Sequenz, die ersten und die letzten zwei Elemente. Bei Zahlen ebenfalls die Summe so wie das Minimum und das Maximum. Aus einer Sequenz und einer numerischen Variable leitet Daikon zusätzlich noch das Element das an dieser Stelle in der Sequenz steht ab sowie das vorhergehende Element und Teilsequenzen bis zu diesen beiden Elementen. Auch die Anzahl der Funktionsaufrufe führt Daikon als Variable ein.

Jede (abgeleitete) Variable und jedes Variablen-Tupel wird auf die vorhergenannten Invarianten geprüft. Dazu erstellt Daikon zu jeder Variable/Tupel eine Liste potentieller Invarianten. Diese werden dann anhand der Ergebnisse der Testläufe entweder als wahr angenommen oder verworfen. Daikon unterstützt dabei nur skalare Variablen (int, boolean, char) und Sequenzen solcher Variablen. Dadurch soll die Komplexität von Daikon reduziert werden und die Implementierung unabhängiger von der Programmiersprache der zu testenden Programme.

Daikon findet Invarianten über globale Variablen, Felder eines Objekts, Prozedur-Parameter und -Rückgabewerte (und andere in der Prozedur veränderte Werte) und Rückgaben von nebenwirkungsfreien Methoden.

3.2 Methodik

Daikon liest die vom Frontend mit Hilfe der Testdaten erzeugten Trace-Dateien ein und generiert daraus wahrscheinliche Invarianten (siehe Abbildung 1). Daikon erzeugt zuerst mögliche Invarianten und testet diese dann mit dem Trace aus der Programmausführung. Alle Invarianten, die ausreichend ohne Falsifizierung getestet wurden, werden dann von Daikon ausgegeben.

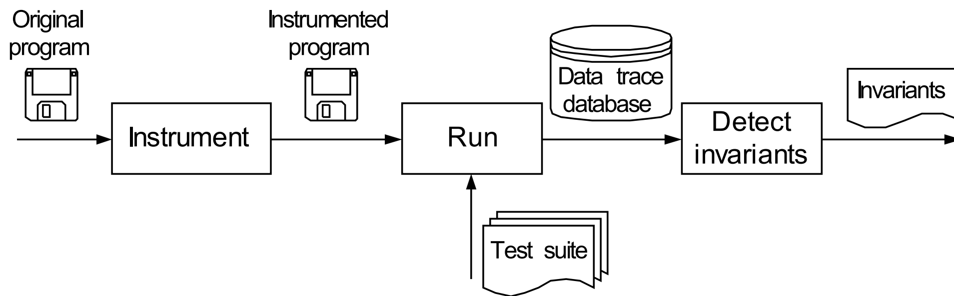

Abbildung 1: Grundlegende Funktionsweise von Daikon

3.3 Relevanz der Invarianten

Daikon gibt eine große Menge an Invarianten aus. Wie bei anderen dynamischen Analyseverfahren hängt die Qualität der Ergebnisse stark von den verwendeten Testdaten ab. Allerdings sind die Anforderungen an die Testdaten andere als zum Beispiel zur Auffindung von Bugs. Schließlich genügt es nicht, jeden möglichen Pfad einmal zu durchlaufen, sondern alle Pfade und Funktionen müssen möglichst häufig ausgeführt werden, damit die Trefferquote der gefundenen Invarianten möglichst hoch ist. Das heißt man benötigt zur Suche nach Invarianten immer sehr viele Testdaten.

Natürlich gibt Daikon auch mit sehr vielen Testdaten nur die Invarianten aus, die aus diesen gefolgert werden können, das heißt, dass ein weiterer Testdurchlauf würde möglicherweise dazu führt, dass einige ausgegebene Invarianten verworfen oder verändert werden würden. Allerdings kann man dies umgekehrt auch nutzen, um die Testsuite anzupassen.

Da jedoch die Korrektheit nicht garantiert werden kann, ist es möglicherweise sinnvoll zusätzlich statische Methoden zur Auffindung von Invarianten zu verwenden. Außerdem unterscheidet Daikon nicht zwischen „echten“ Funktionsinvarianten und solchen die nur bei den speziellen Testdaten auftreten.

Um eine Mindestqualität der ausgegebenen Invarianten sicherzustellen, testet Daikon die gefundenen Invarianten auf ihre Wahrscheinlichkeit bei rein zufälligen Eingaben. So währe zum Beispiel die Invariante $x \neq 0$ bei 100 Werten zwischen -100 und 100 doch sehr wahrscheinlich. Da solche Invarianten sehr häufig auftreten würden, lassen sich alle Invarianten, die mit einer gewissen Wahrscheinlichkeit auch bei zufälliger Eingabe gefunden werden würden filtern. So kann man zum Beispiel nur Invarianten ausgeben, die nur mit einer Wahrscheinlichkeit $< 0{,}1$ % auch bei zufälligen Eingaben aufgetreten wäre.

Schwächen zeigt Daikon auch im Umgang mit komplexeren Datentypen wie z.B. binäre Suchbäumen. Da Daikon diese nicht kennt, liefert Daikon meist nur schlechte Ergebnisse.

3.4 Ausblick

Um Daikon auch in Zukunft als nützliches Tool zu erhalten und zu verbessern, wurde bei der Entwicklung auf Erweiterbarkeit großen Wert gelegt. So gibt es beispielsweise auch

Frontends für den Java PathFinder Model-Checker oder online Datenquellen [EPG+07]. Durch neue Frontends lässt sich Daikon auch auf Programme, die in anderen Programmiersprachen geschrieben sind, anwenden. Durch Erweitern der Klasse „Invariants“ lassen sich andere Arten von Invarianten finden. Auch neue abgeleiteten Variablen lassen sich durch Erweitern einer Klasse hinzufügen.

4 Kosten

Die Kosten des Ableitens von Invarianten hängen von der Anzahl der überprüften Programmpunkte, der Anzahl der Variablen, den Arten der überprüften Invarianten und der Menge der Testdaten ab. Da allerdings die abgeleiteten Variablen von den Testdaten abhängen, ist es schwierig die Kosten abzuschätzen. Um Daikon für große Anwendungen zu nutzen, lässt sich die Anzahl der abgeleiteten Variablen oder die Arten von gesuchten Invarianten reduzieren. Möglicherweise muss man sich jedoch auch auf bestimmte Programmteile beschränken [Xia07].

Um die Laufzeit von Daikon zu reduzieren, wurden vor allem vier Optimierungen verwendet:

- Bei Variablen, die immer gleich sind, gilt jede Invariante, die für a gilt selbstverständlich auch für b und wird für b daher nicht weiter getestet.
- Variablen, die an einem bestimmten Programmpunkt immer den gleichen Wert haben, machen die Suche nach weiteren Invarianten, die nur diese Variable betreffen überflüssig. So folgt selbstverständlich aus $x = 9$, dass x ungerade ist und falls auch $y = 12$ feststeht ebenfalls, dass $x < y$ gilt.
- Wenn Variablen an mehreren Stellen im Programm gleich sind, werden Invarianten nur am ersten Punkt gesucht und ausgegeben.
- Wenn aus Invariante a Invariante b folgt, wird nur Invariante a ausgegeben. Zum Beispiel lässt sich aus $x > y$ leicht $x \geq y$ folgern. (Die drei vorhergenannten Optimierungen sind lediglich Spezialfälle dieser Optimierung.)

5 Programmanwendung

Daikon besteht aus zwei Komponenten: sogenannten Frontends (Programmiersprachen-spezifisch) und der eigentlichen Programm zum Auffinden der Invarianten. Um Daikon zu nutzen muss das zu untersuchende Programm selbstverständlich mit Debuginformationen kompiliert worden sein.

Daikon selbst besitzt viele Konfigurationsmöglichkeiten, die das Laufzeit verhalten ändern. So lassen sich beispielsweise bestimmte Invarianten von der Suche ausschließen oder Filter deaktivieren. Die Ausgabe erfolgt in verschiedenen Formen: Neben einem eigenen Ausgabeformat kann es zum Beispiel auch JML(Java Modeling Language). Es ist ebenfalls möglich die Ausgabe in den Quellcode schreiben zu lassen (siehe Abbildung 2).

```
/**
 * Array-based implementation of the queue.
 * @author Mark Allen Weiss
 */
public class QueueAr
{
  /*@ invariant this.theArray != null; */
  /*@ invariant \typeof(this.theArray) == \type(java.lang.Object[]); */
  /*@ invariant this.currentSize >= 0; */
  /*@ invariant this.front >= 0; */
  /*@ invariant this.back >= -1; */
  /*@ invariant (this.theArray.length == 0) ==> (this.currentSize == 0); */
  /*@ invariant this.currentSize <= this.theArray.length; */
  /*@ invariant (this.theArray.length == 0) ==> (this.front == 0); */
  /*@ invariant this.front <= this.theArray.length; */
  /*@ invariant (this.theArray.length-1 == 0) ==> (this.front == 0); */
  /*@ invariant this.back <= this.theArray.length; */
  /*@ invariant DataStructures.QueueAr.DEFAULT_CAPACITY != this.theArray.length-1; */
  /*@ invariant theArray.owner == this; */
...
```

Abbildung 2: Ausgabe von Invarianten direkt in den Quellcode

6 Probleme

Leider habe ich es nicht geschafft eine funktionierende Version des C/C++-Frontends zu kompilieren, da meine glibc-Version zu neu war. Fjalar (das verwendete Analyse Tool für C/C++) lässt sich nur mit glibc 2.2 - 2.5 kompilieren.

Literatur

[Dai] http://groups.csail.mit.edu/pag/daikon/.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng*, 27(2):99–123, 2001.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program*, 69(1-3):35–45, 2007.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, 2000.

[Xia07] Chen Xiao. Performance enhancements for a dynamic invariant detector. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 2007.