JINSI: Isolation fehlerrelevanter Interaktion in Produktivsystemen

Martin Burger, mburger@cs.uni-sb.de
Universität des Saarlandes, Fachrichtung Informatik
Saarbrücken, Deutschland

1 Einleitung

Die Fehlerbeseitigung in einem Softwaresystem beginnt zumeist mit der Reproduktion des Fehlers. Bei einem Produktivsystem ist dies besonders schwierig, da es von zahlreichen Diensten wie z.B. Datenbanken abhängt. Daher kann das System nicht ohne weiteres unter Laborbedingungen gestartet und beobachtet werden. Wir schlagen als Hilfsmittel für JAVA-Systeme JINSI vor: JINSI führt Teilkomponenten eines Produktivsystemes unter Laborbedingungen aus. Dazu zeichnet es zunächst die Interaktion dieser Komponente mit ihrer Umgebung im Produktivbetrieb auf und versorgt sie später im Laborbetrieb mit dieser Interaktion. Dort kann die Komponente nicht nur nach Belieben ausgeführt werden, sondern JINSI ermittelt auch den Bruchteil der Interaktion, der tatsächlich fehlerrelevant ist. Damit automatisiert JINSI zwei wesentliche Aufgaben der Fehlersuche: die Reproduktion und die Isolation eines Fehlers.

Bisherige Arbeiten zeichnen ganze Programmzustände auf, um einen Lauf zu wiederholen und zu untersuchen. Hierbei fallen allerdings sehr große Datenmengen an. Eine weitere Möglichkeit, um relevante Daten wie Parameter und Rückgabewerte zu speichern, besteht in der Serialisierung von Objekten. Diese ist allerdings bezüglich Speicher und Zeit sehr teuer. Die von JINSI aufgezeichnete Datenmenge ist im Vergleich winzig, da es die zu speichernden Daten auf ein Minimum beschränkt: (1) es werden nur die Objekte gespeichert, die Einfluss auf die Berechnung haben und (2) JINSI benötigt nur Objekt-Ids, Typen und skalare Werte für die spätere Wiedergabe. Dadurch werden benötigter Speicher und Zeit drastisch reduziert, was das Aufzeichnen in Produktivsystemen erlaubt.

JINSI arbeitet erfolgreich auf großen Systemen wie dem ASPECTJ-Compiler. Mit unseren Ansatz beträgt der Faktor für den zeitlichen Mehraufwand durch das Aufzeichnen nur 1,1 bis etwa 2. JINSI konnte in ersten Versuchen die Interaktion auf wenige fehlerrelvante Methodenaufrufe minimieren.

2 JINSI

Mittels Instrumentierung zieht JINSI[1] zunächst eine Grenze zwischen einer Komponente und ihrer Umgebung (Abb. 1); die Definition einer Komponente erfolgt durch Auswahl von Klassen oder Paketen. JINSI identifiziert die dortige Interaktion und instrumentiert den Bytecode so, dass ein- und ausgehende *Ereignisse*, wie Methodenaufrufe, Rückgabewerte und Feldzugriffe, abgefangen werden. Die Instrumentierung stellt auch sicher, dass nur Ereignisse aufgezeichnet werden, die die Grenze passieren. Die notwendige Instrumentierung kann vor Auslieferung oder auch zur Laufzeit im Produktivbetrieb geschehen und dauert auch bei großen Systemen nur wenige Sekunden. So kann die erneute Definition einer alternativen Grenze schnell erfolgen.

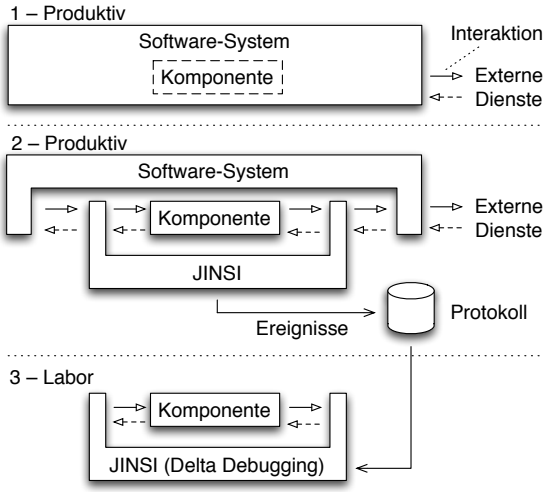

Abbildung 1: Eine Komponente in einem Softwaresystem (1) wird mittels Instrumentierung abgegrenzt. Danach kann JINSI die Interaktion im Produktiveinsatz abfangen und aufzeichnen (2). Im Labor kann der Fehler reproduziert und isoliert werden (3).

Daten, die während der Ausführung die Grenze passieren, können von skalaren Werten bis zu komplexen Objektgraphen reichen. Das Speichern von kompletten Objekten ist sehr aufwendig. So ist Serialisierung hinsichtlich Zeit und Speicher kostspielig. JINSI speichert stattdessen für Objekte, die für die Berechnung relevant sind, lediglich eine eindeutige Id und den Typ. Da auch alle ausgehenden Ereignisse aufgezeichnet werden, sind genügend Informationen für das spätere Abspielen vorhanden: Wird beispielsweise ein ausgehender Methodenaufruf abgefangen, so wird der Rückgabewert aufgezeichnet. Während der Wiedergabe ist der reale Wert nicht vonnöten; JINSI kann dann

[1] Für "JINSI Isolates Noteworthy Software Interactions"; "Jinsi" ist auch Suaheli und bedeutet "Methode".

den passenden Rückgabewert liefern: entweder einen skalaren Wert oder ein passendes Objekt[2]. Durch dieses *teilweise* Aufzeichnen fällt der Faktor des zeitlichen Overheads mit etwa 1,1 bis 2 klein aus, und die in XML gespeicherten Ereginisse sind auch bei langen Läufen komprimiert maximal einige Megabyte groß.

Vor dem Abspielen wird die Komponente erneut instrumentiert, so dass ausgehende Konstruktoraufrufe und Feldzugriffe behandelt werden können. Mittels der aufgezeichneten Interaktion kann dann der fehlerhafte Lauf auf der Komponente nach Belieben wiederholt werden. Diese Interaktion kann aber sehr viele irrelevant Ereignisse umfassen, z.B. wenn es sich um einen lange laufenden Serverdienst handelt. JINSI nutzt *Delta Debugging* (Zeller, 2005), um die Menge der Ereignisse durch systematisches Wiederholen des Laufes zu minimieren. Dabei werden alle nicht relevanten eingehenden Methodenaufrufe beseitigt. Da die Isolation ohne Umgebung, also z.B. ohne Anfragen an eine Datenbank, geschieht, kann der Fehler innerhalb weniger Sekunden bis Minuten vollautomatisch eingegrenzt werden.

Nach der Reduzierung auf die relevanten Ereignisse kann JINSI einen minimalen JUNIT Testfall erzeugen, der den Fehler in einer IDE wie ECLIPSE reproduziert. Dort können alle verfügbaren Werkzeuge genutzt werden, um den Defekt zu beseitigen.

3 Andere Arbeiten

JINSI ist der erste Ansatz, der das Aufzeichnen und Abspielen von Komponenteninteraktion mit der Isolierung relevanter Methodenaufrufe kombiniert. Frühere Arbeiten behandeln diese Ansätze getrennt. Lei and Andrews (2005) nutzen Delta Debugging um eine *zufällige* Sequenz von Methodenaufrufen zu minimieren, im Gegensatz zu zuvor unter realen Bedingungen aufgezeichneten Interaktionen. Delta Debugging beschreibt allgemein die Isolierung fehlerrelevanter Umstände (Zeller, 2005). Omniscient Debugging zeichnet *alle Zustände* eines Laufes auf (Lewis, 2003). JINSI reduziert den Lauf auf relevante Aufrufe, diese können mit ähnlichen Techniken untersucht werden. SCARPE (Orso and Kennedy, 2005) ist ein Werkzeug für das *selektive* Aufzeichnen und Abspielen von Interaktion in JAVA-Programmen; JINSI nutzt ähnliche Techniken (Orso et al., 2006).

4 Ergebnisse und Ausblick

JINSI implementiert eine kombinierte Lösung für zwei nicht triviale Aufgaben: Reproduzieren eines Fehlers und Beschränken auf die fehlerrelevante Komponenteninteraktion. Aus dieser Interaktion wird ein minimaler JUNIT-Testfall erzeugt, der den Fehler reproduziert. Unser Ansatz kann für komplexe Systeme im Produktiveinsatz angewandt werden und erlaubt die Analyse eines realen Fehlers im Labor. Neben ASPECTJ haben wir JINSI bereits erfolgreich auf dem E-Mail-Programm COLUMBA angewandt.

Durch die begrenzte Beobachtung einer Komponente und die Aufzeichnung weniger Objekt-Informationen wird die Menge der anfallenden Daten gering gehalten: Werden alle Interaktionen der zentralen ASPECTJ-Klasse `BcelWorld` beim Kompilieren des Timing Aspektes des mitgelieferten Beispieles "Telecom Simulation"[3] aufgezeichnet, so werden 5496 Ereignisse abgefangen. Diese sind als gzip-komprimierte XML-Datei lediglich 68 Kilobyte groß.

Die Instrumentierung der 3494 Klassen umfassenden Datei aspectjtools.jar dauert auf einem iMac mit Intel 2,0GHz Core Duo CPU nur 20,8s. Das Kompilieren des Aspektes benötigt mit der originalen Bibliothek 2,1s, mit der instrumentierten 4,5s. Hier beträgt der Faktor des zeitlichen Overheads 2,1. In länger laufenden Programmen ist mit einem Faktor von 1,1 bis 1,2 zu rechnen.

Mittels der aufgezeichneten Ereignisse kann ein Fehler reproduziert werden. JINSI hat in ersten Versuchen diese Interaktionen minimieren können. Der daraus resultierende JUNIT-Testfall umfasst nur einige wenige Methodenaufrufe auf der fehlerhaften Komponente.

JINSI ist noch ein Prototyp und wird weiterentwickelt. Eine Fallstudie mit dem Benchmark iBugs[4], welcher 369 echte Fehler in ASPECTJ enthält, soll den Nutzen belegen. Neben der Minimierung ausgehender Interaktion erscheint die von eingehender Interaktion interessant, um die Menge der relevanten Ereignisse weiter einzugrenzen. Eine Integration in ECLIPSE als Plug-In soll den Ansatz frei verfügbar machen.

Literatur

Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proc. 16th IEEE Intl. Symp. on Software Reliability Engineering*, 2005.

B. Lewis. Debugging backwards in time. In *Proc. 5th Int. Workshop on Automated and Algorithmic Debugging*, 2003.

A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. of the 3rd Intl. ICSE Workshop on Dynamic Analysis*, 2005.

A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with jinsi. In *Proc. of the 4th Intl. ICSE Workshop on Dynamic Analysis*, 2006.

A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 1st edition, 2005.

[2] JINSI verwendet hier sog. Mockobjekte. Diese implementieren die Schnittstelle des echten Objektes, ahmen das eigentliche Verhalten aber nur nach; das Verhalten kann von außen vorgegeben werden.

[3] `http://www.eclipse.org/aspectj/doc/released/progguide/examples-production.html`. Version 1.5.3.

[4] `http://www.st.cs.uni-sb.de/ibugs/`