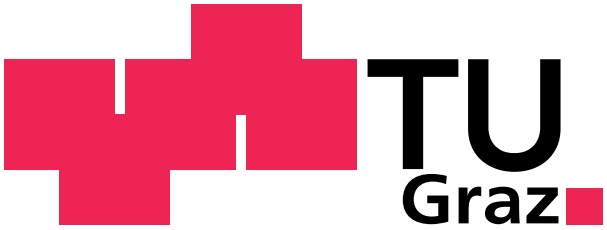

Graz University of Technology

Masterarbeit

Masterstudium Informatik

Spectrum Based Diagnosis

Manuel Riedl

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Zusammenfassung

Das Auffinden und Entfernen von Fehlern ist ein wesentlicher Bestandteil des Softwareentwicklungsprozesses. Um den hierbei entstehenden Aufwand möglichst gering zu halten, existieren unterschiedliche Techniken zur Unterstützung der Entwickler. Eine davon ist die in dieser Arbeit untersuchte Spectrum Based Diagnosis. Dabei handelt es sich um eine statistische Fehlerlokalisierungsmethode, die versucht, fehlerhafte Komponenten aufgrund deren Beteiligung an positiven sowie negativen Testfällen zu identifizieren.

Um die Leistungsfähigkeit dieser Technik zu ermitteln, wurde ein Tool erstellt, das die Anwendung der Spectrum Based Diagnosis auf Javaprogramme ermöglicht. Mit diesem wurden anschließend Testprogramme, die jeweils ein bzw. zwei fehlerhafte Statements enthalten, untersucht. Ebenso wurden die Auswirkungen von verschiedenen Variationen der Technik auf die Genauigkeit der Fehlerlokalisierung ermittelt.

Die Ergebnisse der Analyse zeigen, dass durch Verwendung des Ochiai-Koeffizienten sowohl bei Programmen mit einem als auch mit zwei Fehlern die größte Lokalisierungsgenauigkeit erzielt werden kann. Ebenso kann durch Ausgabe des minimalen Fehlertraces meist eine weitere Verbesserung erzielt werden. Eine Untersuchung der Auswirkungen, die die Anzahl der zur Verfügung stehenden Testfälle hat, ergibt, dass das Ergebnis hauptsächlich von der Verfügbarkeit von negativen Tests abhängt, während der Einfluss der positiven Testfälle meist relativ gering ist.

Abstract

One major part of software development process is the locating and removing of faults. To keep the effort as low as possible, different techniques exist to support the developer. One of them is the Spectrum Based Diagnosis, which was inspected in this paper. This statistical method of fault localization tries to identify faulty components on the basis of their involvement in positive and negative test cases.

To analyze the performance of this technique, a tool was developed to apply the Spectrum Based Diagnosis on Java programs. This tool was used to investigate test programs which contain either one or two faulty statements. Moreover, the effect of different variants of the technique has been determined.

The results of the analysis show that the Ochiai-coefficient provides the best performance for programs with one as well as two faulty statements. Additional improvement has been achieved by using the variant of printing the minimal error-trace. Furthermore, the impact of the number of available test cases has been investigated. It can be concluded that the result mainly depends on the number of negative test cases, whereas the influence of positive tests is relatively low in general.

Inhaltsverzeichnis

Kapitel 1

Einführung

Aktuelle Softwareprojekte unterliegen immer größer werdenden Anforderungen. Vor allem bei kommerzieller Software ist oft eine sehr hohe Funktionsvielfalt nötig, um auf dem Markt erfolgreich zu sein. Dadurch sind Programme mit mehreren Millionen Codezeilen oft keine Seltenheit mehr. Daneben steigen durch den Einsatz von Techniken wie Multithreading oder Verschlüsselungsfunktionen auch die Komplexität und damit verbunden der Entwicklungsaufwand der Software.

Durch diese erhöhten Anforderungen sind meist mehrere Entwickler nötig, um die gewünschte Software in angebrachter Zeit zu entwickeln. Auch die Beauftragung von Fremdfirmen für bestimmte Teile des Programms ist keine Seltenheit. Dies führt zu dem Umstand, dass Entwickler oft auf Funktionen zugreifen müssen, die sie nicht selbst entwickelt haben und über die sie somit keine genauere Kenntnis besitzen. Weiters können durch die parallele Arbeit von mehreren Entwicklern auch unerwünschte Seiteneffekte entstehen. Diese werden von Statements verursacht, die zwar den für sie bestimmten Zweck korrekt erfüllen, jedoch an anderen Stellen im Code zu Problemen führen.

Diese Umstände führen dazu, dass die Wahrscheinlichkeit für Fehler in Softwareprojekten relativ groß ist. Diese Fehler können oft sehr unangenehme Folgen haben. Ein Beispiel hierfür ist die Explosion der Ariane 5 Rakete im Jahre 1996.

1.1 Explosion der Ariane 5 Rakete

Am 4. Juni 1996 startete die unbemannte Ariane 5 Rakete ihren Jungfernflug. 37 Sekunden nach dem Start tauchten erste Unregelmäßigkeiten in der Flugbahn der Rakete auf. Diese wurden so stark, dass nach ungefähr 40 Sekunden eine Selbstzerstörung veranlasst werden musste [1]. Der dadurch entstandene wirtschaftliche Schaden betrug allein für die Rakete 370 Millionen US-Dollar, der erste erfolgreiche Start verzögerte sich um ein Jahr [2].

Als Ursache für die Explosion der Ariane 5 konnte ein Fehler in der Software des inertialen Navigationssystems (SRI) ausgemacht werden. In dieser wurde ein gemessener Wert, der

in einer 64-bit Float Variable gespeichert wurde, konvertiert und in einer 16-bit Integer Variable gespeichert. Dies kann zu einem Problem führen, wenn der Wert der 64-bit Variablen zu groß für die 16-bit Variable ist.

Die Software selbst wurde in Ada geschrieben und wurde direkt vom Vorgänger, der Ariane 4, übernommen, da diese dort fehlerfrei funktionierte. Ada reagiert beim Versuch, eine Zahl in eine zu kleine Variable zu speichern, mit einem Programmabbruch und dem Senden einer Diagnose-Meldung.

Insgesamt besaß die SRI-Software sieben solcher 16-bit Variablen, in denen die Werte von 64-Bit Messungen gespeichert werden mussten. Davon wurde bei vier überprüft, ob die 16-bit Variable groß genug für den gemessenen Wert ist, andernfalls wurden diese entsprechend gerundet. Aus Performancegründen wurde dies jedoch bei den restlichen drei nicht durchgeführt, da angenommen wurde, dass die gemessenen Werte immer klein genug für die 16-bit Variable sind.

Diese Annahme traf für die Ariane 4 zu, wodurch die Software hier problemlos funktionierte. Dies war jedoch bei der Ariane 5 nicht mehr der Fall, sodass einer der ungeprüften Messwerte zu groß wurde und somit die Konvertierung einen Fehler verursachte. Dies führte erst zu einem Ausfall des Haupt-SRIs, wodurch vom Hauptcomputer das Backup-SRI aktiviert wurde. Dieses wurde jedoch kurze Zeit später ebenfalls durch eine fehlerhafte Konvertierung beendet und eine entsprechende Fehlermeldung an den Hauptcomputer geschickt. Dieser war jedoch auf eine derartige Fehlermeldung nicht vorbereitet und interpretierte diese fälschlicherweise als Navigationsdaten, welche letztendlich zum falschen Flugverhalten und somit zur notwendigen Zerstörung der Rakete führten.

1.2 Failures, Errors und Faults

Das Beispiel der Ariane 5 zeigt, dass Fehler im Sourcecode eines Programms nicht automatisch zu fehlerhaftem Verhalten führen müssen. Die bei der Ariane 4 Rakete eingesetzte Software verursachte erst beim Einsatz in der Ariane 5 einen Fehler.

In [3] werden folgende mögliche Fehlerarten definiert.

- Ein Failure bezeichnet die Ausgabe eines Ergebnisses, das vom gewünschten bzw. erwarteten Ergebnis abweicht.
- Ein Error definiert einen fehlerhaften Zustand des Programms. Dieser kann in bestimmten Fällen zu einem Failure führen.
- Ein Fault bezeichnet einen Fehler im Sourcecode des Programms, der zu einem Error führen kann.

Abbildung 1.1: Explosion der Ariane 5 Rakete
(http://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/ariane/website/Ariane.Htm, 15.10.2010)

Das folgende Beispiels dient zur Veranschaulichung der Unterschiede zwischen Failures, Errors und Faults. Es handelt sich hierbei um eine Funktion, die eine Zahl *dividend* durch eine Zahl *divisor* dividieren soll. Die Funktion akzeptiert dabei sowohl positive als auch negative Zahlen. Um diese korrekt behandeln zu können, wird in Zeile 5 und Zeile 10 überprüft, ob es sich bei *dividend* bzw. *divisor* um eine negative Zahl handelt und dementsprechend die Zahl sowie das Vorzeichen mit -1 multipliziert.

In der folgenden, fehlerhaften Version, wird bei beiden if-Abfragen das Vorzeichen mit 1 statt mit -1 multipliziert. Somit sind im Sourcecode zwei Faults enthalten. Das Verhalten des Programms wird nun mit den Inputs <8,3>, <-8,3>und <-8,-3>überprüft.

```
1  public static int divide(int dividend, int divisor) {
2
3          int sign = 1;
4
5          if (dividend < 0){
6                  sign *= 1;
7                  dividend *= -1;
8                  }
9
10         if (divisor < 0){
11                 sign *= 1;
12                 divisor *= -1;
13         }
14
15         int result = 0;
16         while(dividend >= divisor) {
17                 result++;
18                 dividend -= divisor;
19         }
20
21         return (result * sign);
22 }
```

Für Input <8,3>werten beide if-Abfragen in Zeile 5 und 10 zu *false* aus, wodurch der fehlerhafte Code nicht ausgeführt wird. Das Programm verursacht somit mit diesem Input keinen Error.

Bei Input <-8,-3>werten hingegen beide if-Abfragen zu *true* aus. Somit wird in beiden Fällen ein fehlerhaftes Statement ausgeführt, was zu zwei Errors führt. Da jedoch bei der korrekten Version des Programms in diesem Fall das Vorzeichen zweimal mit -1 multipliziert werden und somit wieder 1 ergeben würde, wird hier auch vom fehlerhaften Programm das richtige Ergebnis geliefert. Die beiden Errors heben sich so gesehen gegenseitig auf und verursachen keinen Failure.

Bei Input <-8,3>wertet die if-Abfrage in Zeile 5 zu true und die andere zu false aus. Das Ausführen vom Statement in Zeile 6 verursacht einen Error, welcher wiederum beim Berechnen des Rückgabewertes in Zeile 21 zu einem falschen Ergebnis und damit zu einem Failure führt.

1.3 Vermeiden von Faults

Die Tatsache, dass Faults in Programmen auftreten können, ohne Failures zu verursachen, stellt vor allem bei sicherheitskritischen Anwendungen eine Gefahr dar. Eine vollständige Verifikation der Funktionalität eines Programms durch Eingabe aller möglichen Inputs und entsprechender Verifikation des Outputs ist aufgrund der Größe des Input- und Outputraumes meist nicht möglich. Aus diesem Grund existieren unterschiedliche Techniken, die das Auftreten von Faults vermeiden sollen.

1.3.1 Design by Contract

Design by Contract ist eine Technik, die das Auftreten von Faults vermeiden soll. Die Grundidee dahinter ist, dass eine Klasse und die Clients, die diese Klasse aufrufen, eine Art Vertrag miteinander besitzen. Der Client ist dafür verantwortlich bestimmte Vorbedingungen einzuhalten, bevor er eine Methode der Klasse aufruft. Die Klasse wiederum garantiert, dass bestimmte Nachbedingungen nach Ausführen der Methode eingehalten werden [4].

Durch diese Vorgehensweise soll erreicht werden, dass eine Klasse für einen bestimmten Inputraum ein korrektes Verhalten aufweist. Der Inputraum wird hierbei durch die Vorbedingungen (Preconditions) bestimmt, das Verhalten der Klasse durch die Nachbedingungen (Postconditions).

Diese Vor- und Nachbedingungen werden meist direkt in der gewählten Programmiersprache definiert. Hierzu existieren unterschiedliche Tools für die verschiedensten Sprachen, beispielsweise JML für Java. Bei JML werden die Bedingungen mit Hilfe von speziellen Annotations-Kommentaren eingebaut. Eine Verletzung einer solchen Bedingung wird zur Laufzeit erkannt und führt dazu, dass eine entsprechende Exception geworfen wird.

Design by Contract kann dabei helfen, das Auftreten von Faults in Programmen zu verhindern. Allerdings ist hierbei zu beachten, dass die Bedingungen korrekt und auch genau genug spezifiziert sind. Vor allem zu ungenaue Spezifizierungen können hierbei zu verdeckten Faults führen, die durch die Contracts nicht aufgedeckt werden.

1.3.2 Test Driven Development

Test Driven Development, auch Unit-Testing genannt, ist eine Technik, deren Ursprung aus dem Extreme Programming (XP) stammt. Diese besagt, dass vor dem Implementieren einer Funktion Testfälle konzipiert werden sollen, die das gewünschte Verhalten der

Funktion beschreiben [5]. Mit Hilfe dieser Testfälle kann anschließend überprüft werden, ob die Funktion korrekt ist bzw. ob die gesamte Funktionalität implementiert wurde.

Bei den Testfällen handelt es sich im Grunde um kleine Programme bzw. Funktionen, die das Verhalten einer Funktion überprüfen, indem diese mit bestimmten Inputparametern aufgerufen und anschließend das erwartete mit dem tatsächlichen Ergebnis verglichen wird. Sie sind meist in derselben Sprache wie die eigentliche Funktion geschrieben.

1.3.2.1 Unit-Testing mit JUnit

Ein weit verbreitetes Tool für Unit-Testing ist JUnit (http://www.junit.org). Mit diesem ist es möglich, Unit-Tests für Javaprogramme zu schreiben und auszuwerten. Dabei bietet JUnit sowohl eine graphische Benutzeroberfläche als auch die Möglichkeit, Ergebnisse über die Konsole ausgeben zu lassen. Ein weiterer Grund für die Beliebtheit von JUnit ist die gute Integration in die Eclipse IDE (http://www.eclipse.org).

Für die Implementierung der Unit-Tests wird eine eigene, von der Basisklasse *TestCase* abgeleitete Testklasse erstellt. Das Ableiten von *TestCase* ist nötig, damit in dieser Klasse die *assert*-Methoden zur Verfügung stehen. Mit diesen können unter anderem erwartete und tatsächliche Werte einer Funktion miteinander verglichen und somit kann fehlerhaftes Verhalten erkannt werden.

Eine Testklasse besteht im Allgemeinen aus einer Fixture und den zugehörigen Testfällen. Eine Fixture stellt hierbei eine Konfiguration dar, auf die die Testfälle angewandt werden [6]. Dies kann beispielsweise ein Objekt sein, dessen Methoden getestet werden, oder auch eine Kombination von mehreren, miteinander agierenden Objekten. Die Fixture ist für alle Testfälle innerhalb der Klassen gleich.

Der Aufbau einer Fixture erfolgt in der Methode *setUp()*. Diese wird immer vor dem Ausführen der Testfälle aufgerufen. Abbildung 1.2 zeigt den Aufbau einer JUnit-Testklasse inklusive der Definition einer Fixture. Weiters kann eine Methode *tearDown()* existieren, die nach der Ausführung aller Testfälle aufgerufen wird und für das korrekte Beenden der Fixture zuständig ist. Diese wird meist dann benötigt, wenn bestimmte Ressourcen, beispielsweise Netzwerk-Sockets, freigegeben werden müssen.

Die Testfälle selbst testen unterschiedliche Funktionen der Fixture. Dabei haben alle Testfälle gemeinsam, dass sie keine Übergabeparameter besitzen und ihr Rückgabetyp *void* ist. Abbildung 1.3 zeigt ein Beispiel für einen Testfall, der auf die in Abbildung 1.2 definierte Fixture angewandt wird. Das Beispiel zeigt weiters die Anwendung einer *assert*-Methode, in diesem Fall *assertTrue* am Ende des Testfalles. Diese Methode überprüft, ob der Rückgabewert des übergebenen Parameters *expected.equals(result)* dem boolschen Wert *true* entspricht.

```
public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    @Before public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
}
```

Abbildung 1.2: Junit-Klasse mit Fixture

```
@Test public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

Abbildung 1.3: Junit-Testfall

Das Ausführen einer Testklasse kann direkt aus einem Javaprogramm mittels Aufruf von *org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);* bzw. über die Konsole mit *java org.junit.runner.JUnitCore TestClass1 [...other test classes...]* erfolgen. Die Ergebnisse werden hier direkt in der Konsole angezeigt. Weiters können JUnit-Testklassen in vielen Entwicklungsumgebungen, darunter auch Eclipse, direkt ausgeführt werden. Hierbei werden die Ergebnisse in einer grafischen Benutzeroberfläche ausgegeben. Abbildung 1.4 zeigt hier die Ausgabe einer Testklasse mit jeweils zwei positiven und zwei negativen Testfällen.

1.4 Faultlokalisierung

Die bisher genannten Techniken können dabei helfen die Existenz von Faults in Software zu zeigen. Zur genauen Lokalisierung der Faults ist allerdings eine genauere Untersuchung des Programms nötig. Dies erfolgt meist durch Ausgabe von Zwischenergebnissen mittels printout-Statements oder durch Debugging mit Hilfe von Breakpoints [7].

Diese Vorgehensweisen können jedoch vor allem bei großer und komplexer Software sehr viel Zeit in Anspruch nehmen. Weiters ist eine relativ genaue Kenntnis über das Programm nötig, um an den richtigen Stellen nach den Ursachen des Fehlers suchen zu können. Besonders problematisch kann dies bei Wartungsarbeiten werden. Oft treten Failures erst auf, wenn die Software bereits in Verwendung ist. In solchen Fällen kann es sein, dass

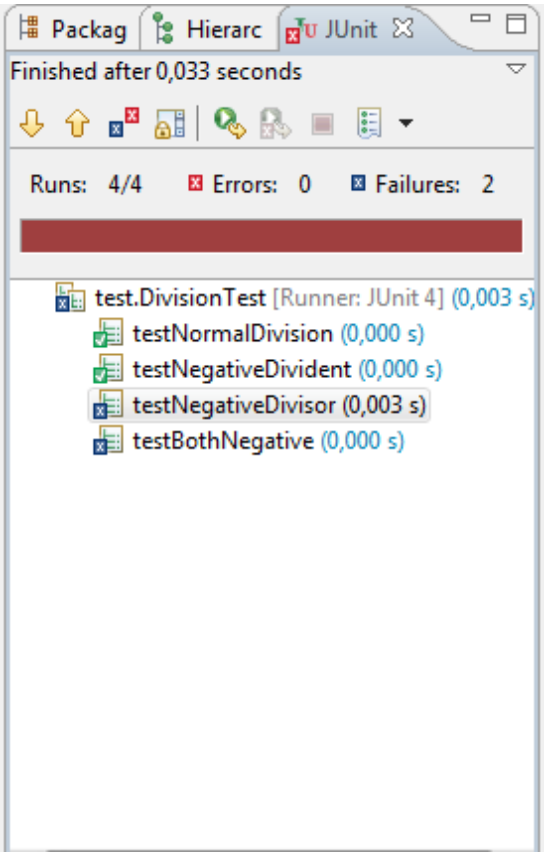

Abbildung 1.4: JUnit GUI in Eclipse

das Entwicklerteam nicht mehr zur Verfügung steht und die Software von neuen Entwicklern gewartet werden muss. Dies führt dazu, dass die Entwickler erst Verständnis über das Programm erlangen müssen, um anschließend den Fehler zu finden und zu beseitigen. So kommt es, dass auch die Entfernung von kleinen und einfachen Fehlern relativ zeitaufwändig sein kann.

Diese Umstände erfordern Techniken, die das Auffinden von Faults in Programmen vereinfachen. Eine davon ist die Reduktion des Programmumfanges durch das Verwenden von so genannten Slices.

1.4.1 Slicing

Ein Slice bezeichnet eine Menge von Statements, die für die Berechnung eines bestimmten Wertes verantwortlich sind. Dabei werden nur die Statements betrachtet, die tatsächlichen Einfluss auf den Wert haben, alle anderen Statements sind nicht im Slice enthalten. Ein Slice verhält sich dabei in Bezug auf diesen Wert gleich wie das ursprüngliche Programm. Dies bedeutet, dass der Slice selbst ausführbar ist und der Wert nach Ausführung des Slices dem des ursprünglichen Programms entspricht.

Mit Hilfe von Slices ist es somit möglich, den Umfang des zu untersuchenden Programms und damit auch den Aufwand zur Fehlerfindung zu reduzieren. So kann beispielsweise für eine Variable, deren Wert nach Ausführen des Programms nicht dem erwarteten Wert entspricht, ein Slice berechnet werden, der nur die Statements enthält, die den Wert der Variablen beeinflussen. Somit werden für die Fehlersuche unwichtige Statements ausgeblendet.

Grundsätzlich gibt es mehrere Möglichkeiten, einen Slice zu berechnen. Eine davon ist das statische Slicing, welches in [8] das erste Mal vorgestellt wurde. Beim statischen Slicing wird im ersten Schritt ein Slicing-Criterion <i,V >definiert. Dabei gibt V an, für welche Variablen der Slice erstellt werden soll. Außerdem wird eine Stelle im Programm definiert, an welcher der Wert der Variablen in V von Interesse ist. Dies kann zum Beispiel die Position des Statements sein, an der alle für den Slice gewählten Variablen einen falschen Wert aufweisen. In vielen Fällen wird auch das Ende des Programms als solche Position gewählt. Die gewählte Stelle wird im Slicing-Criterion durch i definiert.

Der nächste Schritt ist die Berechnung der Statements, die für die im Slicing-Criterion definierten Variablen von Bedeutung sind. Hierzu wird für jedes Statement bestimmt, welche Variablen durch dieses definiert und welche referenziert werden. Dies ist nötig, um die für den Slice benötigten Statements identifizieren zu können.

Anschließend werden die Statements rückwärts, ausgehend von der im Slicing-Criterion definierten Position i, betrachtet. Dabei gilt für ein Statement, dass es genau dann im Slice ist, wenn es eine Variable definiert, die in einem im Slice enthaltenen, darunter liegenden Statement, referenziert wird. Schleifen und if-Abfragen müssen gesondert behandelt

werden, falls diese Statements enthalten, die ebenfalls Teil des Slices sind. In diesem Fall müssen auch die in der Condition referenzierten Variablen betrachtet werden.

Die durch statisches Slicing erhaltenen Slices enthalten oft mehr Statements, als eigentlich notwendig sind. Aus diesem Grund gibt es mehrere Verbesserungen des Algorithmus, mit denen es möglich ist, einen kleineren Slice zu erzeugen. Eine davon ist das dynamische Slicing [9].

Bei diesem wird der Slice für eine bestimmte Ausführung des Programms berechnet. Das bedeutet, dass nur Statements, die die Variablen des Slicing-Criterion für diese eine Ausführung beeinflussen, berücksichtigt werden. Beim statischen Slicing werden hingegen alle Statements berücksichtigt, die in einer beliebigen Ausführung Einfluss auf die Variablen haben können.

Das Beispiel in Abbildung 1.5 soll die Unterschiede zwischen statischem und dynamischem Slicing verdeutlichen. Hierbei wird der Slice mit dem Slicing-Criterion <7,x>für Input <1>berechnet. Da durch den Input 1 die Schleife in Zeile 4 nicht ausgeführt wird, haben auch die Statements in den Zeilen 5 und 6 keinen Einfluss auf das Ergebnis der Variable x. Somit entfallen diese im dynamischen Slice.

```
1   read(i);

2   x = 2;
3   y = 1;

4   while(y < i) do
5       x = x * 2;
6       y = y + 1;
    end do

7   write(x);


Statischer Slice:  <1,2,3,4,5,6>
Dynamischer Slice: <1,2,3,4>
```

Abbildung 1.5: Statischer und dynamischer Slice für Input <1>

In [10] werden weitere Verfeinerungen und Erweiterungen der Slicingtechnik erläutert. Für alle Ansätze gilt gemeinsam, dass diese zwar eine Reduktion der zu betrachtenden Statements bieten, jedoch keine weiteren Informationen über den Ursprung des Fehlers liefern können. Somit sind diese Verfahren speziell bei unbekanntem Code nur bedingt geeignet.

Eine bessere Methode zur Analyse von unbekanntem Code stellen automatische Fehlerlokalisierungsmethoden dar. Diese erfordern keine Kenntnisse über den zugrunde liegenden

Sourcecode und können im Idealfall das fehlerhafte Statement eindeutig identifizieren. Grundsätzlich wird hierbei zwischen statistischen und modellbasierten Ansätzen unterschieden.

1.4.2 Modellbasierte Fehlerlokalisierung

Die Grundidee der modellbasierten Fehlerlokalisierung basiert auf der Beschreibung von physikalischen Systemen. Hierbei wird erst ein Modell des Systems erstellt, das dessen erwartetes Verhalten beschreibt. Anschließend werden mit Hilfe des Modells Vorhersagen durchgeführt, die mit den Ergebnissen des tatsächlichen Systems verglichen werden. Dieser Vergleich soll dazu beitragen, fehlerhafte Komponenten im System ausfindig zu machen [11].

Ein ähnlicher Ansatz kann zur Analyse von Softwaresystemen herangezogen werden. Hierbei stellt das Modell das gewünschte und korrekte Verhalten des Programms dar, beim System handelt es sich um dessen tatsächliche Implementierung. Dieses Verfahren wird als Model Based Diagnosis bezeichnet [12].

Dabei wird angenommen, dass eine Komponente des Modells entweder korrekt funktioniert oder sich abnormal verhält. Die Aufgabe ist nun, herauszufinden, welche Komponenten sich abnormal verhalten müssen, damit dadurch die Abweichungen der Ergebnisse des Systems zu denen des Modells erklärt werden. Die Menge der abnormalen Komponenten wird als Diagnose bezeichnet. Dabei ist zu beachten, dass es für ein System mehrere Diagnosen geben kann, die dessen Fehlverhalten erklären.

Obwohl Model Based Diagnosis meist relativ gute Ergebnisse liefert, können in der ursprünglichen Form Probleme auftreten. So kann es beispielsweise vorkommen, dass das Modell selbst fehlerhaft ist. Dies kann dazu führen, dass die berechneten Diagnosen falsch sind oder nur relativ wenige fehlerhafte Komponenten gefunden werden [13]. Weiters kann es auch zu unentdeckten Faults kommen, wenn diese sowohl im Modell als auch im System enthalten sind.

Ein weiteres Problem bei Model Based Diagnosis ist die Tatsache, dass die Modelle manuell erstellt werden müssen. Dies bedeutet zusätzlichen Aufwand für die Entwickler und setzt meist näheres Verständnis über die gewünschte Funktionalität des Programms voraus. Weiters ist auch eine gewisse Kenntnis über die eingesetzte Technik selbst vonnöten, um das Modell korrekt zu erstellen.

Eine Möglichkeit, das manuelle Erstellen des Modells zu vermeiden, ist das Model Based Debugging, welches in [11] beschrieben wird. Hierbei stellt die Software selbst das Modell dar, die Beobachtungen stammen aus positiven und negativen Testfällen. Da die Testfälle beim Einsatz von Unit-Testing jederzeit zur Verfügung stehen, ist Model Based Debugging hier meist automatisch durchführbar. Die Berechnung der Diagnose wird hierbei durch eine spezielle Software, den so genannten Model Checker, durchgeführt.

1.4.3 Statistische Fehlerlokalisierung

Bei der statistischen Fehlerlokalisierung wird versucht, mit Hilfe von verschiedenen Programmausführungen Faults in Programmen zu lokalisieren. Dabei werden sowohl positive als auch negative Programmtraces miteinander verglichen, um mögliche fehlerhafte Komponenten zu identifizieren.

Die Identifizierung erfolgt durch statistische Auswertung der Beteiligung der einzelnen Komponenten in den Traces. Dabei gilt, dass eine Beteiligung an negativen Traces die Verdächtigkeit eines Statements erhöht, für einen Failure verantwortlich zu sein. Durch Beteiligung an einem positiven Trace sinkt diese Verdächtigkeit wieder. Im folgenden Kapitel wird mit Spectrum Based Diagnosis eine Technik vorgestellt, die die Methode der statistischen Fehlerlokalisierung anwendet.

Kapitel 2

Spectrum Based Diagnosis

Spectrum Based Diagnosis ist eine automatisch ablaufende, statistische Fehlerlokalisierungsmethode. Der grundsätzliche Ansatz hierbei ist, die während eines Testfalles ausgeführten Programmkomponenten zu speichern und damit ein so genanntes Spektrum aufzubauen. Dieses enthält die Komponenten eines Testfalles sowie das Ergebnis des Testfalles selbst, das heißt, ob es sich um einen positiven oder negativen Testfall handelt. Anschließend werden die im Spektrum enthaltenen Daten analysiert, um den Komponenten Werte zuzuweisen, die deren Verdächtigkeit, für einen Fehler verantwortlich zu sein, widerspiegeln. Die Testfälle können beispielsweise vollständige Ausführungen des Programmes, aber auch Unit-Tests sein.

Der Ablauf bei Spectrum Based Diagnosis besteht aus drei Schritten. Der erste Schritt ist die so genannte Instrumentisierungsphase. Auf diese folgt die Ausführungsphase, in der alle positiven und negativen Testfälle ausgeführt werden. Der dritte Schritt ist die Analyse des Spektrums und die Zuweisung von Verdächtigkeitswerten.

2.1 Instrumentisierung

Um das Spektrum korrekt aufzubauen ist es nötig zu erkennen, welche Komponenten während der Ausführung eines Testfalles aufgerufen werden. Hierzu wird der Sourcecode vor dem Ausführen der Testfälle instrumentiert. Dies bedeutet, dass zusätzliche Statements eingeführt werden, die für den Aufbau des Spektrums verantwortlich sind.

Die Statements werden im Normalfall direkt vor der jeweiligen Komponente eingefügt. Dadurch wird sichergestellt, dass diese genau dann ausgeführt werden, wenn auch die zugehörige Komponente ausgeführt werden soll. Hierbei muss beachtet werden, dass durch das zusätzliche Ausführen dieser Statements die eigentliche Funktionalität des Programms nicht beeinflusst werden darf.

In manchen Fällen ist es auch nötig, neben der Ausführung einer Komponente auch weitere Informationen über diese zu speichern. So wird meist auch eine Angabe der Position von

Komponenten benötigt, um diese anschließend dem Benutzer in geeigneter Weise mitteilen zu können. Auch hierfür sind die in der Instrumentisierungsphase eingeführten Statements verantwortlich.

Bei komplexen Programmen kann der Aufwand für die Instrumentierung oft relativ groß sein. Weiters steigt durch die zusätzlichen Statements auch der Zeitbedarf für die Ausführung der Testfälle. Dies kann speziell bei Umgebungen mit beschränkten Ressourcen, beispielsweise Embeddes Systems oder Smartcards, zu erheblichen Geschwindigkeitseinbußen führen. Ein weiterer limitierender Faktor kann der Speicher werden, da der Speicheraufwand für das Spektrum mit der Anzahl der Testfälle und der Anzahl der Komponenten steigt.

Eine Abhilfe für diese Probleme wird durch die Möglichkeit geschaffen, nur einzelne Teile des Sourcecodes, beispielsweise bestimmte Klassen, zu instrumentieren. Dadurch ist es möglich, sowohl den Zeitaufwand für die Instrumentierung als auch die Erhöhung von Laufzeit und Speicherbedarf zu minimieren. Jedoch ist hier darauf zu achten, dass durch die Teilinstrumentisierung keine der für eine korrekte Auswertung benötigen Informationen verloren gehen. So könnten beispielsweise fehlerhafte Komponenten nicht beachtet werden, wodurch die Qualität der Fehlerlokalisierung stark beeinträchtigt wird. Auch Seiteneffekte, deren Einfluss auf das Ergebnis eines Testfalles oft nicht sofort ersichtlich sind, könnten durch eine zu geringe Instrumentierung leicht übersehen werden.

Eine weitere Möglichkeit, den zusätzlichen Ressourcenbedarf für Spectrum Based Diagnosis so gering wie möglich zu halten, ist die Verwendung von unterschiedlichen Spektren. So kann zuerst ein Spektrum verwendet werden, bei dem das Programm relativ grob in Komponenten aufgeteilt wird. Dadurch können die betroffenen Programmteile identifiziert werden, welche anschließend durch Verwendung eines anderen Spektrums in feinere Komponenten, bis hin zu einzelnen Statements, unterteilt werden. Durch diese Vorgehensweise wird sichergestellt, dass nur betroffene Programmteile betrachtet werden.

2.2 Arten von Spektren

2.2.1 Complete Path Spektrum

Das Complete Path Spektrum bietet die gröbste Unterteilung des Programms in Komponenten. Hier wird jeder Ausführungspfad als eigene Komponente definiert [14]. Dies bedeutet, dass bei jedem Testfall genau eine Komponente, nämlich der ausgeführte Pfad, aktiv ist. Dadurch werden sowohl Laufzeit als auch Speicherbedarf so gering wie möglich gehalten.

Die Genauigkeit der Fehlerlokalisierung ist hierbei jedoch stark beschränkt, da im besten Fall nur der Ausführungspfad, der den für das Fehlverhalten verantwortlichen Teil enthält, identifiziert werden kann. Somit eignet sich diese Art von Spektrum hauptsächlich für Umgebungen mit eingeschränkten Ressourcen bzw. für eine erste Analyse der betroffenen

Programmteile, um diese anschließend durch ein Spektrum mit feiner Komponentenaufteilung weiter zu untersuchen.

2.2.2 Branch-Hit Spektrum

Bei dieser Art von Spektrum wird das Programm in so genannte Basic-Blocks aufgeteilt, wodurch es in der Literatur oft auch als Block-Hit Spektrum bezeichnet wird. Ein Basic-Block ist hierbei ein logisch zusammengehörender Block von einzelnen Statements. Logisch zusammengehörend bedeutet dabei, dass innerhalb des Blocks keine Verzweigungen auftreten und somit entweder alle oder keines der Statements ausgeführt wird. Weiters besitzt ein Basic Block genau ein Anfangs- und ein Endstatement. Beispiel 2.1 zeigt, wie ein einfaches Programm, das die Anzahl der Elemente in der Hauptdiagonalen einer n*m Matrix berechnet, in Basic-Blocks aufgeteilt wird [7].

Das Block-Hit Spektrum bietet einen guten Kompromiss aus Fehlerlokalisierungsgenauigkeit und Ressourcenbedarf. Meist reicht die Identifizierung des betroffenen Blocks aus, um den Fehler zu finden. Ansonsten ist es auch hier möglich, eine weitere Verfeinerung durch andere Spektren durchzuführen.

```
//block 1
read(n)
read(m)
sum = 0

for(i = 1..n) do
  //block 2
  for (j = 1..n) do
    //block 3
    if (i == j)
      //block 4
      sum = sum + 1
  end for
énd for

print(sum)
```

Abbildung 2.1: Branch-Hit Spektrum eines Programms

2.2.3 Execution-Trace Spektrum

Beim Execution-Trace Spektrum handelt es sich um die genaueste Möglichkeit, das gegebene Programm zu analysieren. Hierbei wird jedes einzelne Statement betrachtet, wodurch es im besten Fall möglich ist, das tatsächlich für den Fehler verantwortliche Statement zu identifizieren [14].

Beim Einsatz eines Execution-Trace Spektrums muss beachtet werden, dass dieses unter Umständen die Systemressourcen sehr stark belasten kann. So steigt einerseits der Laufzeitbedarf, da für jedes Statement im Programm ein weiteres zum Sammeln der Daten vonnöten ist. Die bedeutet im ungünstigsten Fall eine Verdoppelung der Programmgröße.

Weiters kann die Größe des Spektrums selbst zu Ressourcenproblemen führen. Die Größe des Spektrums ist im Allgemeinen gegeben durch die Anzahl der Komponenten multipliziert mit der Anzahl der Testfälle. Für große Programme mit mehreren tausend Statements und einer Vielzahl an Testfällen kann somit leicht die Dimension des resultierenden Spektrums erahnt werden.

2.2.4 Count- und Hit-Spektren

Neben der Feinheit der Unterteilung in Komponenten kann bei Spektren auch noch unterschieden werden, ob es sich um Count- oder um Hit-Spektren handelt. Ein Count-Spektrum zählt hierbei die Anzahl der Ausführungen einer Komponente während eines Testfalles. Ein Hit-Spektrum registriert hingegen nur, ob eine Komponente ausgeführt wurde oder nicht, während die Anzahl der Ausführungen keine Rolle spielt. Das Beispiel in Abbildung 2.2 zeigt die Werte des Branch-Count Spektrums für das Programm aus Abbildung 2.1 mit Input <3,5>. Das Branch-Hit Spektrum würde in diesem Fall für jeden Block den Wert 1 ergeben, da für diesen Input jeder Block ausgeführt wird.

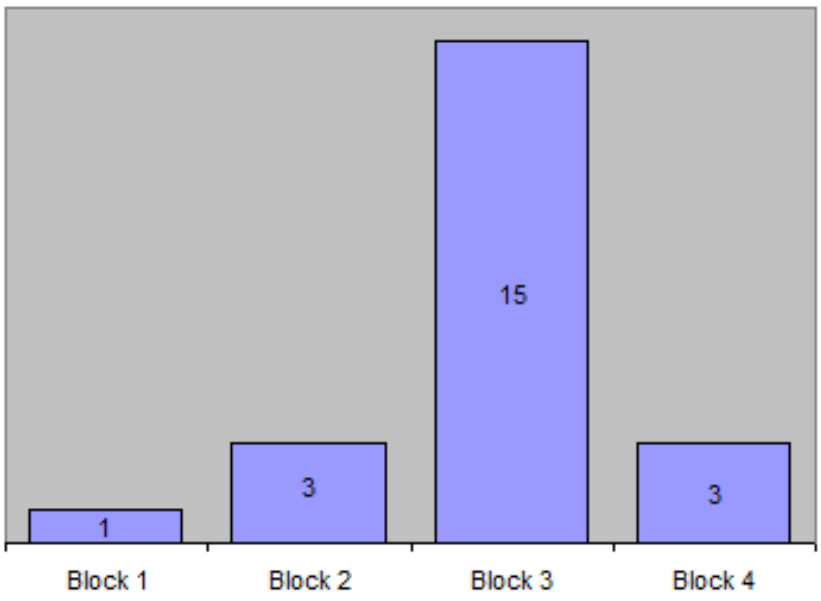

Abbildung 2.2: Branch-Count Spektrums

Kurzform	Name	Beschreibung
BHS	Branch-Hit	Conditional-Branches, die ausgeführt wurden
BCS	Branch-Count	Anzahl, wie oft Conditional-Branches ausgeführt wurden
PHS	Path-Hit	intraprozeduraler, loop-freier Pfad, der ausgeführt wurde
PCS	Path-Count	Anzahl, wie oft ein intrapr., loop-freier Pfad ausgeführt wurde
CPS	Complete-Path	ausgeführter Pfad
DHS	Data-Dependence-Hit	Definition-Use Paare, die ausgeführt wurden
DCS	Data-Dependence-Count	Anzahl, wie oft Definition-Use Paare ausgeführt wurden
OPS	Output	Resultierender Output
ETS	Execution-Trace	Kompletter Trace der Ausführung

Tabelle 2.1: Aufzählung Spektren

2.2.5 Übersicht und Zusammenhang der Spektren

Tabelle 2.1 zeigt eine Aufzählung der in [14] beschriebenen Spektren mitsamt einer Kurzbeschreibung. Der Zusammenhang der einzelnen Spektren ist in Abbildung 2.3 ersichtlich. Dabei signalisiert eine Verbindung A → B, dass Spektrum A eine Verfeinerung von Spektrum B darstellt.

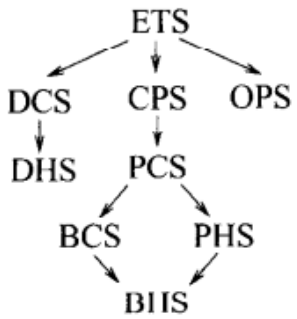

Abbildung 2.3: Zusammenhang der Spektren

2.3 Das Spektrum

Das Spektrum ist, wie der Name bereits andeutet, ein wesentlicher Teil bei Spectrum Based Diagnosis. Anhand der Inhalte des Spektrums wird die eigentliche Analyse zur Auffindung des Fehlers durchgeführt. Für den Aufbau des Spektrums sind die in der Instrumentisierungsphase eingeführten Statements veantwortlich.

Die Speicherung des Spektrums erfolgt in einer so genannten Participation Matrix O. Diese ist eine N x (M + 1) Matrix, wobei M durch die Anzahl der Komponenten und N

	comp1	comp2	comp3	comp4	comp5	e
TC 1	1	1	1	1	0	0
TC 2	1	1	1	1	0	1
TC 3	1	1	1	1	1	0
TC 4	1	1	0	0	1	1

Tabelle 2.2: Beispiel eines Spektrums

durch die Anzahl der zur Verfügung stehenden Testfälle definiert ist. Weiters enthält die Matrix O einen error vector e. Dieser Spaltenvektor stellt die Ergebnisse der ausgeführten Testfälle dar, das heißt, ob es sich um einen positiven oder negativen Testfall handelt. Somit steht jede Zeile der Matrix O für einen Testfall mitsamt dessen Ergebnis und der darin ausgeführten Komponenten.

Die Werte der Matrix können 0 oder 1 sein. Der Wert 1 im Matrixfeld o_{ij} steht hierbei für eine Beteiligung der Komponente j am Testfall i, Wert 0 für keine Beteiligung. Für den error vector bedeutet Wert 0 im Feld e_i, dass es sich bei Testfall i um einen positiven bzw. bei Wert 1 um einen negativen Testfall handelt. Tabelle 2.2 zeigt ein Beispiel für ein Spektrum mit fünf Komponenten und jeweils zwei positiven und zwei negativen Testfällen.

2.4 Analyse und Fehlerlokalisierung

Für die Analyse des Spektrums wird der Spaltenvektor aller Komponenten mit dem error vector verglichen. Dabei gilt, dass eine hohe Ähnlichkeit von Komponentenvektor und error vector auf eine hohe Fehlerverdächtigkeit schließen lässt. Zur Berechnung dieser Ähnlichkeit existieren mehrere unterschiedliche Berechnungsschemen, die so genannten Similarity Koeffizienten.

2.4.1 Beteiligungsterme

Vor der Berechnung des Verdächtigkeitswertes einer Komponente durch einen Similarity Koeffizienten ist es nötig, die Beteiligungsterme aller Komponenten zu ermitteln. Diese Beteiligungsterme geben an, wie oft die Komponenten bei der Ausführung von positiven und negativen Testfällen beteiligt waren. Es existieren insgesamt vier Terme $a_{pq}(j)$, wobei p,q $\in \{0,1\}$ und $a_{pq} = |\{i|o_{ij} = p \wedge e_i = q\}|$. Tabelle 2.4 zeigt die Beteiligungsterme für das in Tabelle 2.2 beschriebe Spektrum.

2.4.2 Similarity Koeffizienten

Aus den Beteiligungstermen kann mit Hilfe eines Similarity Koeffizienten der Verdächtigkeitswert einer Komponente berechnet werden. Hierzu steht eine Vielzahl an unter-

a_{00}(j)	Anzahl der positiven Testfälle, an denen die Komponente nicht beteiligt ist
a_{01}(j)	Anzahl der negativen Testfälle, an denen die Komponente nicht beteiligt ist
a_{10}(j)	Anzahl der positiven Testfälle, an denen die Komponente beteiligt ist
a_{10}(j)	Anzahl der negativen Testfälle, an denen die Komponente beteiligt ist

Tabelle 2.3: Übersicht der Beteiligungsterme

	comp1	comp2	comp3	comp4	comp5
a_{00}	0	0	0	0	1
a_{01}	0	0	1	1	1
a_{10}	2	2	2	2	1
a_{11}	2	2	1	1	1

Tabelle 2.4: Berechnete Beteiligungsterme

schiedlichen Koeffizienten zur Verfügung [15]. Im Folgenden werden drei weit verbreitete Koeffizienten genauer vorgestellt.

2.4.2.1 Tarantula

Dieser Koeffizient stammt aus dem Tarantula Tool, mit dessen Hilfe C-Code analysiert werden kann. Es gilt als eines der ersten Tools, das Spectrum Based Diagnosis einsetzt. Aufgrund dessen wird der Tarantula-Koeffizient oft als Vergleichswert herangezogen, wenn die Performance von Similarity Koeffizienten miteinander verglichen wird. Tarantula selbst liefert meist nur durchschnittliche Resultate.

Die Verdächtigkeit einer Komponente wird bei Tarantula ermittelt, indem der Prozentsatz der negativen Testfälle durch die Summe der Prozentsätze von negativen und positiven Testfällen dividiert wird [16].

$$s_j = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}}$$

2.4.2.2 Jaccard

Jaccard stammt aus dem Jahr 1901 und wurde ursprünglich für Klassifikationsaufgaben in der Biologie eingesetzt [15]. Die Klassifikation erfolgt hierbei durch Berechnung der Ähnlichkeit zweier Mengen, indem die Größe der Schnittmenge der beiden Mengen durch die Größe ihrer Vereinigung dividiert wird. Die resultierenden Werte liegen hierbei zwischen 0 und 1, wobei größere Werte eine höhere Ähnlichkeit signalisieren.

Dieses Verhalten kann bei Spectrum Based Diagnosis genutzt werden, um die Fehlerverdächtigkeit einer Komponente zu berechnen. Die zur Berechnung benötigten Mengen

werden zum einen durch die Menge der negativen Testfälle, zum anderen durch die Menge der Testfälle, in der eine Komponente aktiv war, dargestellt.

Jaccard wird unter anderem als Similarity Koeffizient im Tool Pinpoint eingesetzt [17]. Seine Performance liegt meist im oberen Bereich und über der des Tarantula Koeffizienten [15].

$$s_j = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)}$$

2.4.2.3 Ochiai

Ochiai ist ein aus der Biologie bekannter Similarity Koeffizient. Das Prinzip hinter Ochiai ist die Berechnung des Kosinus des Winkels zwischen dem error vector e und dem Vektor einer Komponente. Dies ergibt einen Wert zwischen 0 und 1, wobei 1 bedeutet, dass der error vector und der Komponentenvektor äquivalent sind.

Die Performance von Ochiai übertrifft im Allgemeinen die anderer Koeffizienten. Ein Grund hierfür ist, dass diese Koeffizienten Komponenten, die ausschließlich in negativen Testfällen vorkommen, automatisch den höchsten Verdächtigkeitswert zuweisen. Dabei wird die Anzahl der Testfälle nicht beachtet.

Ochiai hingegen berücksichtigt zusätzlich diese Anzahl bei der Zuweisung der Verdächtigkeitswerte. Somit wird bei Komponenten, die ausschließlich in negativen Testfällen vorkommen, denjenigen ein höherer Wert zugewiesen, die in einer größeren Anzahl an Testfällen aktiv war [7].

$$s_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}$$

Kapitel 3

Implementierung

3.1 Anforderungen

Das Ziel dieser Diplomarbeit ist die Entwicklung eines Java Programms, welches die Anwendung der Spectrum Based Diagnosis Technik auf Java Programme ermöglicht. Dieses soll neben der Möglichkeit zur Unterstützung bei der Fehlerauffindung auch dazu dienen, die Performance von Spectrum Based Diagnosis zu analysieren und mit anderen Fehlerlokalisierungstechniken vergleichbar zu machen. Weiters soll damit die Leistungsfähigkeit von verschiedenen Similarity Koeffizienten miteinander verglichen werden können.

Das Tool soll standardmäßig die Koeffizienten Ochiai, Jaccard und Tarantula unterstützen. Zusätzliche Koeffizienten sollen später möglichst einfach eingebunden werden können. Um die Vergleichbarkeit mit anderen Fehlerlokalisierungstechniken zu verbessern, sollen während der Instrumentierungsphase auch die für Slicing benötigen Def- und Ref-Mengen der jeweiligen Statements bestimmt werden, sodass diese anschließend ebenfalls aus dem Spektrum ausgelesen werden können.

3.2 Eingesetzte Tools

3.2.1 ANTLR

ANTLR ist ein von Terence Parr in Java entwickelter Parser Generator. Mit diesem ist es möglich, Parser für beliebige Programmiersprachen zu erstellen. Deren Syntax wird in einem Grammatikfile definiert, aus dem ANTLR eine Lexer- sowie eine Parserklasse generiert. Diese können anschließend in Javaprogrammen verwendet werden, um zu überprüfen, ob ein bestimmter Code Teil der definierten Sprache ist. Hierzu wird der Code dem Parser als String oder File übergeben.

Im ersten Schritt wird der Code vom Lexer in Literale aufgeteilt. Ein Literal bezeichnet dabei einen Teil des Codes, der nicht weiter zerlegt werden kann. So werden beispielsweise der Aufbau von Variablennamen oder Zahlen als Literal definiert.

Im nächsten Schritt überprüft der Parser die syntaktische Zusammensetzung der Sprache. Hierzu wurden in der Grammatik Regeln definiert, in welchen Kombinationen die Literale auftreten dürfen. Eine solche Kombination wird als Token bezeichnet. Dabei kann ein Token sowohl Literale als auch weitere Tokens enthalten. Das folgende Beispiel zeigt den Aufbau des Tokens für eine while-Schleife in Java:

```
1 whileLoop
2         :         'while' condition statements
```

Hierbei bezeichnet der String *'while'* ein Literal, während *condition* und *statements* weitere Tokens beschreiben.

Grundsätzlich enthält eine ANTLR Grammatik ein Starttoken, bei dem der Parser startet. Der anschließende Parsvorgang kann durch folgenden Algorithmus beschrieben werden:

```
hole alle Elemente aus der Regel von startToken
while weitere Elemente vorhanden do
  hole nächstes Element
  if Element-Typ == 'Literal' then
    hole nächstes Element vom Lexer
    if Element-Text == Literal-Text then
      return  true
    else
      return  false
    end if
  else if Element-Typ == 'Token' then
    return  parse(Element)
  else
    return  false
  end if
end while
```

Anhand des Algorithmus ist ersichtlich, dass die Elemente innerhalb einer Regel entsprechend einer Tiefensuche aufgelöst werden. Das bedeutet, dass zuerst ein Element vollständig in Literale zerlegt und überprüft wird, bevor mit dem nächsten fortgesetzt wird. Das Parsen wird so lange fortgeführt, bis alle Literale des Codes überprüft wurden oder ein inkorrektes Literal gefunden wurde. In diesem Fall gibt ANTLR eine entsprechende Fehlermeldung aus. Ansonsten ist der Code Teil der definierten Sprache.

Neben dem reinen Parsen von Grammatiken ist es mit ANTLR auch möglich, annotierte Grammatiken zu erstellen. Hierzu kann im Grammatikfile zusätzlicher Javacode angegeben werden, der ausgeführt wird, sobald die Regel eines Tokens überprüft wird. Dabei ist es auch möglich, auf Informationen der Tokens zuzugreifen. So kann beispielsweise der

Text im Code, der durch ein Token dargestellt wird, ausgelesen werden. Mit Hilfe einer annotierten Grammatik ist es somit möglich, Compiler zu erstellen, die Programme einer Sprache in eine andere Sprache übersetzen.

3.2.2 JUnit

Das Spectrum Based Diagnosis Tool verwendet zum Aufbau des Spektrums Testfälle, die mit Hilfe des JUnit Frameworks erstellt wurden. Eine kurze Vorstellung der Funktionsweise von JUnit findet sich unter http://www.junit.org sowie Kapitel 1.3.2

3.3 Programmaufbau

3.3.1 Beschreibung der Klassen

3.3.1.1 Diagnoser

Die Klasse Diagnoser enthält die Main-Funktion des Programms und ist für die Vorbereitung der benötigten Daten sowie die Steuerung des Programmablaufes zuständig.

Im ersten Schritt der Main-Funktion werden die beim Programmstart angegebenen Pfade nach Test-, Source- sowie JAR-Files durchsucht. Als Testfiles gelten hierbei alle Javadateien, die mit *Test.java* enden, während alle anderen Javadateien als Sourcecodefiles interpretiert werden. Die JAR-Files werden unter Umständen für die spätere Neukompilierung des Programms benötigt und daher ebenfalls gespeichert.

Anschließend werden die originalen Sourcecodefiles gesichert, da diese durch die Instrumentierung überschrieben werden. Dies geschieht, indem Sicherungen der Originale angelegt und im selben Verzeichnis wie diese abgespeichert werden. Als Dateiname wird hierbei der Hashwert des jeweiligen Originals gewählt. Dadurch soll sichergestellt werden, dass keine zwei Dateien mit dem gleichen Namen gesichert werden und auch keine bestehenden Dateien überschrieben werden. Zudem kann durch eine erneute Berechnung des Hashwertes eindeutig bestimmt werden, welche Sicherungsdatei bei der Wiederherstellung für ein bestimmtes Sourcecodefile zu verwenden ist.

Zusätzlich wird ein neues Verzeichnis angelegt, in dem später die neu kompilierten Dateien gespeichert werden. Dies soll dazu dienen, dass möglicherweise bereits vorhandene Class-Files nicht überschrieben werden.

Als nächstes werden die entsprechenden Klassen für die Instrumentierung, Recompilierung, Ausführung der Testfälle sowie Berechnung und Ausgabe der Verdächtigkeitswerte aufgerufen. Abschließend werden die originalen Sourcecodefiles wiederhergestellt und das Verzeichnis mit den Class-Files des instrumentierten Sourcecodes gelöscht.

Abbildung 3.1: Klassendiagramm

3.3.1.2 JavaParser

Der JavaParser ist ein von ANTLR erzeugter Parser, weshalb an dieser Stelle das entsprechende Grammatikfile beschrieben wird. Als Grundlage für dieses wurde eine bestehende Grammatik für Java 1.5 verwendet (http://www.antlr.org/grammar/1152141644268/Java.g).

Die Instrumentierung erfolgt, indem durch die annotierte Grammatik ein String mit dem instrumentierten Code erzeugt wird, der anschließend ausgelesen werden kann. Dieser ist beim Aufruf der Startregel des Parsers zunächst leer.

Zur Befüllung des Strings werden die Art der Auflösung von Regeln in ANTLR sowie die Tatsache, dass Regeln auf bestimmte Eigenschaften ihrer Tokens zugreifen können, ausgenutzt. Eine dieser Eigenschaften ist das Text-Attribut, mit dem auf im Programm verwendete Textteile zugegriffen werden kann. Somit ist beispielsweise über das Token, das eine if-Abfrage beschreibt, mit Hilfe des Text-Attributes der Zugriff auf die tatsächlich im zu untersuchenden Programm verwendete if-Abfrage möglich. Weiters erlaubt ANTLR das Überschreiben des Text-Attributes einer Regel mit einem beliebigen Rückgabewert.

Damit ist es möglich, den String mit dem instrumentierten Sourcecode zu erstellen. Hierfür wird für jede Regel überprüft, ob der durch diese beschriebene Codeteil instrumentiert werden soll. Ist dies der Fall, wird das Text-Attribut durch einen String, der das Instrumentierungsstatement sowie den beschriebenen Codeteil selbst enthält, ersetzt. Somit wird an jede übergeordnete Regel beim Auslesen des Text-Attributes dieses Tokens eine instrumentierte Version des Codes zurückgeliefert. Da ANTLR die Regeln entsprechend einer Tiefensuche auflöst, werden somit jeweils die instrumentierten Codeteile weiterpropagiert, wodurch nach Überprüfung aller benötigten Regeln eine instrumentierte Version des Sourcecodes aus den Tokens der Startregel ausgelesen werden kann.

Berechnung der Def- und Ref-Mengen Wie bereits erwähnt, sollen im Spektrum neben Informationen wie Art und Position eines Statements ebenfalls dessen Def- und Ref-Mengen gespeichert werden. Diese werden, ähnlich wie die instrumentierten Codeteile, an geeigneter Stelle berechnet und so lange weiterpropagiert, bis die Information benötigt wird. Allerdings erfolgt dies nicht durch Überschreiben des Text-Attributes eines Tokens, da ansonsten kein Zugriff auf den durch das Token repräsentierten Code und somit auch keine korrekte Instrumentierung mehr möglich ist.

Stattdessen erfolgt die Speicherung der Def- und Ref-Mengen in einer separaten Datenstruktur. Hierzu werden folgende Schritte ausgeführt:

1. Löse alle in der aktuellen Regel enthaltenen Token auf
2. Lies die Def- und Ref- Mengen der Tokens aus der Datenstruktur aus
3. Berechne die Def- und Ref-Mengen für die aktuelle Regel

4. Speichere diese Mengen am Ende der Datenstruktur

Durch das von ANTLR verwendete Prinzip der Tiefensuche zur Regelauflösung kann einerseits sichergestellt werden, dass sich die für die Regeln zutreffenden Mengen jeweils am Ende der Datenstruktur befinden und diese entsprechend der Reihenfolge der Tokens in der Regel angeordnet sind.

3.3.1.3 Recompiler

Der Recompiler ist dafür zuständig, die Test- sowie die instrumentierten Sourcecodefiles zu kompilieren. Dies ist nötig, da ansonsten die Testfälle auf dem nicht instrumentierten Sourcecode ausgeführt und somit keine Informationen in das Spektrum eingefügt werden. Außerdem kann nicht davon ausgegangen werden, dass alle benötigten Class-Files zur Verfügung stehen. Dies ist auch der Grund, warum die Testfiles ebenfalls neu kompiliert werden.

Das Kompilieren selbst geschieht durch Aufrufen des Java-Kompilers *javac*. Diesem werden, ähnlich wie bei einem Aufruf über die Konsole, die zu kompilierenden Dateien sowie eventuell vorhandene JAR-Files als Argumente übergeben. Die resultierenden Class-Files werden im vom Diagnoser erstellten Verzeichnis abgelegt.

3.3.1.4 TestcaseRunner

Diese Klasse ist dafür zuständig, die Testfälle aus den Testklassen zu extrahieren, diese auszuführen und das Ergebnis ins Spektrum einzutragen.

Das Extrahieren der Testfälle geschieht mit Hilfe des Java Reflection Features. Dieses erlaubt es, während der Ausführung eines Programms dynamisch weitere, externe Klassen zu laden, deren Methoden aufzulisten und Objekte dieser Klasse zu instanzieren.

Im konkreten Fall lädt der TestcaseRunner zuerst die Testklasse und überprüft, ob eine *suite()*-Methode vorhanden ist. Ist dies der Fall, wird das Durchsuchen der Methoden gestoppt, ein Objekt der Klasse instanziert und die *suite()*-Methode aufgerufen. Diese retourniert eine Testsuite, aus der alle Testfallobjekte ausgelesen werden können.

Sollte die Testklasse keine *suite()*-Methode enthalten, werden entsprechend der JUnit-Konvention alle Methoden innerhalb der Klasse, die mit *test* beginnen und keine Übergabeparameter besitzen, als Testfälle interpretiert. Um aus diesen Testfallobjekte zu erzeugen, wird der Klassenkonstruktor aufgerufen und der Name des gewünschten Testfalles als Parameter übergeben.

Nachdem alle Testfallobjekte zur Verfügung stehen, werden diese der Reihe nach ausgeführt, wodurch das Spektrum mit den in diesem Testfall ausgeführten Statements befüllt wird. Nach Beendigung des Testfalles kann ebenso dessen Ergebnis ins Spektrum eingefügt werden.

3.3.1.5 ComponentAnalyzer

Der ComponentAnalyzer ist für die Analyse des Spektrums sowie die Ausgabe der Ergebnisse zuständig. Hierzu wird im ersten Schritt das Spektrum durchlaufen und für jedes Statement dessen Beteiligungsterme ermittelt. Mit diesen kann der Verdächtigkeitswert des Statements entsprechend dem verwendeten Similarity Koeffizienten berechnet werden.

Abschließend werden die Ergebnisse im Format
Dateiname - *Zeilennummer*: *Koeffizient* ausgegeben, wobei diese absteigend nach dem Verdächtigkeitswert sortiert sind. Um bei großen Programmen die Übersichtlichkeit zu bewahren, ist die Ausgabe auf 50 Statements beschränkt. Ein Beispiel für die Ausgabe eines Divisionsprogramms in der Windowskonsole ist in Abbildung 3.2 zu sehen. Zusätzlich werden die Optionen geboten, die Ergebnisse des gesamten Traces bzw. die des Traces des minimalen negativen Testfalles in eine Datei auszugeben.

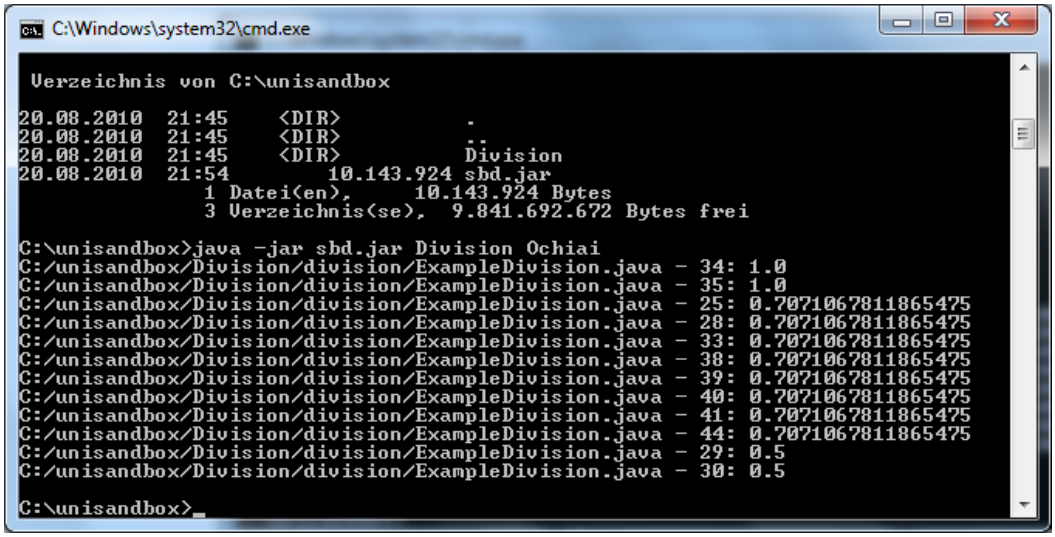

Abbildung 3.2: Output des Tools

3.3.1.6 Spektrum

Im Spektrum werden Informationen über die Statements, die während der Ausführung der Testfälle aufgerufen werden, gespeichert. Jeder Testfall wird dabei innerhalb des Spektrums als Zeile repräsentiert. Eine Zeile enthält die ausgeführten Statements sowie das Ergebnis des jeweiligen Testfalles. Für jedes Statement werden die in Tabelle 3.1 angeführten Informationen gespeichert.

id	Eindeutige ID der Komponente innerhalb des Files
type	Art der Komponente
begin	Position des Zeichens mit dem die Komponente beginnt
end	Position des Zeichens mit dem die Komponente endet
def	Menge der in dieser Komponente definierten Variablen
ref	Menge der in dieser Komponente referenzierten Variablen
fileName	Dateiname
line	Zeilennummer der Komponente

Tabelle 3.1: Aufbau der Spektrumeinträge

3.4 Designentscheidungen

3.4.1 Spektrum als Singleton

Um sicherzustellen, dass alle Komponenten im Spektrum gespeichert werden, ist dieses als Singleton implementiert. Somit existiert zu jeder Zeit maximal eine Instanz der Spektrumklasse. Die Instanzierung erfolgt, sobald der erste Zugriff auf das Spektrum erfolgt.

3.4.2 Speicherung der Komponenten im Spektrum

Die Spectrum Based Diagnosis Technik basiert auf dem Prinzip, dass Statements in mehreren unterschiedlichen Testfällen ausgeführt werden. Da für jedes Statement ein eigenes Objekt angelegt wird, würde dies bei einer direkten Speicherung im Spektrum eine erhebliche Speicherbelastung bedeuten. Aus diesem Grund wird für jedes Statement das zugehörige Objekt nur einmal angelegt und direkt in der Spektrumklasse abgespeichert. In den Zeilen des Spektrums, die die Testfälle repräsentieren, werden anschließend nur noch die IDs der jeweiligen Statements, bestehend aus Dateiname und Zeilennummer des Statements im File, gespeichert.

3.4.3 Kein paralleles Ausführen der Testfälle

Grundsätzlich könnten Testfälle, die voneinander unabhängig sind, parallel ausgeführt werden. Jedoch wäre eine eindeutige Zuordnung, welcher Testfall für die Ausführung eines Statements verantwortlich ist, nicht mehr möglich. Aus diesem Grund wird ein neuer Testfall erst dann gestartet, wenn der vorhergehende beendet wurde und dessen Ergebnis im Spektrum eingefügt ist.

3.4.4 Erzeugen der Koeffizienten via Factory Method

Das Tool soll dazu in der Lage sein, neue Koeffizienten einfach einbinden zu können. Aus diesem Grund implementieren die Koeffizientenklassen ein Interface, das die benötigte

Funktionalität der Koeffizienten definiert. Die Instanzierung der konkreten Koeffizientenklassen erfolgt durch eine Factory, der der Name des gewünschten Koeffizienten übergeben wird und die das entsprechende Objekt zurückliefert. Somit sind zur Einbindung neuer Koeffizienten bis auf eine entsprechende Implementierung des Interfaces sowie die Erweiterung der Factory keine weiteren Eingriffe in den Code nötig.

3.4.5 Einfügen der Komponenten ins Spektrum

Grundsätzlich wird zum Einfügen eines ausgeführten Statements ins Spektrum eine entsprechende Methode der Spektrumklasse aufgerufen, der die benötigten Informationen als Parameter übergeben werden. Diese Methode erzeugt aus den Informationen einen Spektrumeintrag und fügt ihn in die entsprechende Zeile des Spektrums ein.

Allerdings bedeutet ein Methodenaufruf mit einer größeren Anzahl an Übergabeparametern einen entsprechenden Overhead. Dieser ist jedoch nicht immer notwendig, da das ausgeführte Statement möglicherweise bereits im Spektrum vorhanden ist. So ist beispielsweise eine Eintragung von Statements, die in einer Schleife mehrmals aufgerufen werden, nur einmal nötig.

Aus diesem Grund werden bei der Instrumentierung zwei zusätzliche Instruktionen eingefügt. Dabei überprüft die erste, ob sich das Statement bereits im Spektrum befindet. Hierzu ist nur die Übergabe der jeweiligen ID nötig. Die zweite Instruktion ruft die Methode zum Einfügen eines neuen Eintrags im Spektrum auf und übergibt die benötigten Informationen.

Diese Vorgehensweise bringt sowohl Vor- als auch Nachteile. Ein Nachteil ist, dass für neue Statements zwei Methoden aufgerufen werden müssen, obwohl nur die Methode zum Einfügen nötig wäre. Somit bedeutet dies eine Verlangsamung von Testfällen, in denen keine Schleifen ausgeführt werden und jedes Statement nur einmal aufgerufen wird.

Vorteile gibt es jedoch für Testfälle mit Schleifen. Hier erhöht sich die Performance erheblich, da für bereits eingefügte Statements nur noch ein Methodenaufruf mit einem Übergabeparameter nötig ist. Da im Allgemeinen davon auszugehen ist, dass während der Ausführung der Testfälle Schleifen auftreten, ist diese Vorgehensweise zu bevorzugen.

3.5 Ausführung des Tools

3.5.1 Voraussetzungen

Um das Tool korrekt starten zu können, ist es nötig, dass sich eine Kopie der Datei *mrsbd.jar* im Hauptverzeichnis des zu untersuchenden Programms befindet. Dies ist nötig, da ansonsten die Klasse Spektrum nicht bekannt ist und somit kein korrektes Kompilieren des instrumentierten Codes möglich ist.

Zudem ist darauf zu achten, dass bei der Verwendung von relativen Pfaden innerhalb des zu untersuchenden Codes das korrekte *working directory* angegeben wird.

3.5.2 Starten des Tools

Das Tool kann über die Konsole mit dem Aufruf

```
java -jar mrsbd.jar ROOTDIR [-i INSTRDIR] [-t TESTDIR]
[-c COEFFICIENT] [-o OUTPUTVARIANT FILENAME]
```

gestartet werden.

3.5.3 Beschreibung der Parameter

ROOTDIR: Gibt den Pfad zum Hauptverzeichnis des zu untersuchenden Programms an. In diesem müssen alle Source- und Testfiles sowie zusätzlich verwendete JAR-Files enthalten sein.

INSTRDIR: Dieser optionale Parameter gibt den Pfad zu dem Verzeichnis an, in dem sich die zu instrumentierenden Dateien befinden. Es werden nur die in diesem Verzeichnis sowie dessen Unterverzeichnissen vorhandenen Javafiles instrumentiert. Wird dieser Parameter nicht angegeben, werden alle im ROOTDIR vorhandenen Javafiles instrumentiert.

TESTDIR: Dieser optionale Parameter gibt den Pfad zu dem Verzeichnis an, in dem sich die Testfiles befinden. Es werden nur die in diesem Verzeichnis sowie dessen Unterverzeichnissen vorhandenen Testfiles ausgeführt. Wird dieser Parameter nicht angegeben, werden alle im ROOTDIR vorhandenen Testfiles ausgeführt.

COEFFICIENT: Dieser optionale Parameter dient zur Auswahl des Similarity Koeffizienten, der zur Berechnung der Verdächtigkeitswerte verwendet werden soll. Mögliche Werte sind hierbei Ochiai, Jaccard und Tarantula. Wird dieser Parameter nicht angegeben, wird der Ochiai Koeffizient verwendet.

OUTPUTVARIANT FILENAME: Diese Option ermöglicht es, die Art der Ausgabe zu definieren. Erlaubt sind hierbei *normal* für eine Ausgabe aller Ergebnisse in der Konsole, *File* für eine Ausgabe aller Ergebnisse in eine Datei sowie *minTrace*, bei der die Ergebnisse aller Statements, die im kleinsten negativen Testfall ausgeführt werden, in eine Datei geschrieben werden. Es gilt zu beachten, dass bei den Varianten *File* und *minTrace* neben dem Namen der Variante ebenso der Name der Ausgabedatei als eigener Parameter anzuführen ist. Wird dieser Parameter nicht angeführt, wird als Standard die Ausgabevariante *normal* gewählt.

3.5.4 Testfälle

Wie bereits erwähnt, verwendet das Tool zum Aufbau des Spektrums Testfälle, die auf dem JUnit Framework basieren. Um vom Tool korrekt erkannt zu werden, müssen diese folgende Spezifikationen erfüllen:

1. Jede Testfallklasse muss mit dem String *Test* enden.
2. Die Testfallklasse muss von *junit.framework.TestCase* abgeleitet sein.
3. Jeder Testfall muss mit dem String *test* beginnen und darf keine Übergabeparameter besitzen. Alternativ ist es auch möglich, Testfälle zu einer Testsuite zusammenzufassen. In solch einem Fall muss die Klasse eine statische Methode *suite()* besitzen.

Durch das Ableiten von *junit.framework.TestCase* ist es möglich, die Testfälle in der für JUnit gewohnten Weise auszuführen. So wird, falls vorhanden, für jeden Testfall vor dessen Ausführung die Methode *setUp()* und nach der Ausführung die Methode *tearDown()* ausgeführt. Weiters kann durch diese Ableitung auf das Ergebnis des Testfalles zugegriffen werden, sprich, ob dessen Resultat positiv oder negativ ist.

3.6 Bekannte Probleme

3.6.1 Initialisierung durch Methodenaufrufe bei Testsuiten

Beim Aufruf der *suite()*-Methode einer Testklasse werden alle Testfälle in der Testsuite konstruiert. Dies bedeutet, dass für jeden Testfall der entsprechende Konstruktor aufgerufen wird. Hierbei kann es zu Problemen kommen, wenn innerhalb des Konstruktors eine Methode einer instrumentierten Klasse aufgerufen wird, beispielsweise um Variablen des Testfalles zu initialisieren. Da in solch einem Fall nicht eruiert werden kann, welcher Testfall innerhalb der Suite eine derartige Methode aufgerufen hat, kann diese auch keinem Testfall zugeordnet werden. Aus diesem Grund werden die ausgeführten Statements eines solchen Methodenaufrufes nicht im Spektrum eingetragen.

Dieses Problem tritt jedoch nur bei der Verwendung von Testsuiten auf. Bei einzelnen Testfällen wird zuerst ein neuer Testfall im Spektrum angelegt, bevor der entsprechende Konstruktor aufgerufen wird. Dadurch steht auch für im Konstruktor aufgerufene Methoden zu jeder Zeit fest, welchem Testfall diese zuzuordnen sind.

Zur Vermeidung dieses Problems empfiehlt es sich somit, auf Methodenaufrufe innerhalb des Konstruktors eines Testfalles zu verzichten und diese stattdessen in die *setUp()*-Methode zu verlagern. Diese wird erst beim Ausführen des Testfalles aufgerufen, wodurch das Problem umgangen werden kann.

Alternativ kann natürlich auch auf eine Einbindung eines solchen Testfalles in eine Testsuite verzichtet werden. Hierbei ist allerdings zu beachten, dass in einem solchen Fall keine *suite()*-Methode in der Testklasse existieren darf, da ansonsten der Testfall ignoriert wird.

3.6.2 Nicht korrekt beendete Threads

Sollten in den zu untersuchenden Programmen Threads verwendet werden, muss beim Design der Testfälle darauf geachtet werden, dass diese Threads bei Beendigung des Testfalles ebenfalls beendet sind. Ansonsten würden diese weiterhin Daten ins Spektrum einfügen, obwohl unter Umständen bereits ein anderer Testfall ausgeführt wird. Dies würde zu einer Verfälschung der Ergebnisse führen.

Kapitel 4

Resultate

4.1 Testumgebung

4.1.1 Testprogramme

Zum Testen des SBD-Tools sowie der Technik selbst standen drei Testprogramme zur Verfügung. Dabei wurde darauf geachtet Programme zu wählen, die für reale Anwendungsbereiche und nicht ausschließlich zum Evaluieren von Fehlerlokalisierungsmethoden entwickelt wurden.

Der Sourcecode der Testprogramme wurde dabei nicht verändert, es wurden lediglich Anpassungen bei den Testfällen durchgeführt, damit diese korrekt vom Tool erkannt und ausgeführt werden konnten. Der Ablauf der Testfälle wurde hierbei ebenfalls nicht verändert.

Als Ausgangsbasis wurden nur positive Testfälle zugelassen. Dadurch soll verhindert werden, dass eine spätere Untersuchung durch reale, im Testprogramm enthaltene Fehler beeinträchtigt wird. Weiters wurden potentiell problematische Testfälle wie solche, die Threads nicht korrekt beenden, nicht berücksichtigt.

4.1.1.1 KickoffTUG

KickoffTUG (http://kickofftug.tugraz.at) ist ein Multi-Agent-System, das für die Teilnahme an der RoboCup Simulation League (http://www.robocup.org/robocup-soccer/simulation/) entwickelt wurde. Bei dieser werden Fußballspiele simuliert, indem die Spieler durch eigenständige Agenten dargestellt werden und über einen eigenen Server gegeneinander spielen. KickoffTUG stellt das größte für die Analyse verwendete Testprogramm dar.

Sourcecodefiles	1182
Lines of Code	177272
Testklassen	70
Testfälle	156

4.1.1.2 JTopas

JTopas ist eine Java Bibliothek, die zum Erstellen von Parsern und Tokenizern für Textdaten verwendet werden kann. Dabei können unter anderem Command Line Parser oder File Parser erstellt werden.

Sourcecodefiles	45
Lines of Code	16354
Testklassen	7
Testfälle	80

4.1.1.3 Reflection-Visitor

Bei Reflection-Vistitor handelt es sich um eine Basisklasse, die das Visitor-Designpattern mit Hilfe von Java-Reflection implementiert. Im Gegensatz zum ursprünglichen Visitor-Pattern ist es dadurch möglich, die für eine bestimmte Elementklasse vorgesehene *visit(. . .)*-Methode aufzufinden, ohne dass diese über eine entsprechende *accept(. . .)*-Methode verfügen muss.

Sourcecodefiles	29
Lines of Code	1140
Testklassen	3
Testfälle	14

4.1.2 Testsystem

Zur Analyse der Laufzeit- und Speicheranforderungen wurde folgendes Testsystem verwendet:

Betriebssystem	Windows 7 Professional
CPU	Intel Core 2 Duo E6750
RAM	3 GB
Entwicklungsumgebung	Eclipse 20100218-1602

4.2 Laufzeit- und Speicheranalyse

4.2.1 Durchschnittlicher Laufzeitbedarf

Die Ausführung des SBD-Tools besteht, wie in Kapitel 3.3.1 beschrieben, aus mehreren Schritten. Der durchschnittliche Zeitbedarf für die Durchführung dieser Schritte wird

	KickoffTUG	JTopas	Reflection-Visitor
Suchen der benötigten Dateien	599 ms	62 ms	172 ms
Sichern der Original Dateien	2066 ms	219 ms	53 ms
Instrumentierung	19557 ms	925 ms	329 ms
Recompilierung	34966 ms	1114 ms	735 ms
Ausführung der Testfälle	253536 ms	244250 ms	235 ms
Wiederherstellung der Original Dateien	15472 ms	368 ms	119 ms
Analyse und Ausgabe der Ergebnisse	27 ms	11 ms	3 ms

Tabelle 4.1: Ausführungszeiten

	KickoffTUG	JTopas	Reflection-Visitor
LoC/Sourcefile	149,976	363,422	39,31
Zeitbedarf/LoC	0,110 ms	0,057 ms	0,289 ms

Tabelle 4.2: Zeitbedarf Instrumentierung

in Tabelle 4.1 angegeben. Die Werte ergeben sich aus dem Durchschnitt von zehn hintereinander durchgeführten Testläufen und sind in Millisekunden angegeben. Dabei ist ersichtlich, dass für jedes Testprogramm die Schritte Instrumentierung, Recompilierung sowie Ausführung der Testfälle den größten Zeitaufwand verursachen.

Für die Instrumentierung sind dabei die Anzahl der zu instrumentierenden Statements sowie deren Verteilung auf einzelne Sourcecodefiles ausschlaggebend. Da jedes File erst eingelesen werden muss, bevor die Instrumentierung begonnen werden kann, steigt der relative Zeitaufwand pro Loc entsprechend der Anzahl der Sourcefiles des jeweiligen Testprogramms (4.2).

Durch die bei der Instrumentierung zusätzlich eingefügten Instruktionen erhöht sich der Zeitbedarf bei der Ausführung der Testfälle gegenüber den nicht instrumentierten Originalen. Der Faktor der Erhöhung für die zugrunde liegenden Testprogramme ist in Tabelle 4.3 ersichtlich.

4.2.2 Vergleich der Instrumentierungsvarianten

In Abschnitt 3.4.5 wurde eine Variante zur Instrumentierung vorgestellt, die den zusätzlichen Zeitaufwand für Statements, die während eines Testfalles mehrmals vorkommen,

	KickoffTUG	JTopas	Reflection-Visitor
original	49369 ms	27699 ms	204 ms
instrumentiert	253536 ms	244250 ms	235 ms
Faktor	5,16	8,81	1,15

Tabelle 4.3: Erhöhung der Testlaufzeiten durch Instrumentierung

	# Statements insgesamt	davon in Schleife	Schleifendurchgänge	# ausgeführte Statements
Szenario 1	500000	125000	1	500000
Szenario 2	500000	125000	5	1125000
Szenario 3	500000	125000	10	1750000
Szenario 4	500000	125000	20	3000000
Szenario 5	500000	125000	50	6750000

Tabelle 4.4: Testfall-Szenarien

minimiert. Dies wird allerdings durch eine Verschlechterung bei erstmalig auftretenden Statements erkauft.

Um den Vorteil dieser Variante zu eruieren, wurden beide Varianten anhand mehrerer unterschiedlicher Szenarien gegenübergestellt. Variante 1 bezieht sich hierbei auf die im Programm verwendete Methode mit zusätzlicher Instruktion zur Überprüfung des Vorkommens eines Statements im Spektrum. Bei Variante 2 wird hingegen direkt die Methode zur Eintragung ins Spektrum aufgerufen.

Als Ausgangsbasis dient ein simulierter Testfall, der aus einer bestimmten Anzahl an unterschiedlichen, instrumentierten Statements besteht. Davon befinden sich 25% in einer Schleife, sodass diese innerhalb des Testfalles keinmal, einmal oder mehrmals ausgeführt werden können. Insgesamt werden fünf Szenarien mit einer unterschiedlichen Anzahl an Schleifendurchgängen analysiert (4.4). Es gilt dabei zu beachten, dass nur die Zeit zum Einfügen in das Spektrum gemessen wird, eine etwaige Ausführung der simulierten Statements selbst wird nicht berücksichtigt.

Der jeweilige Zeitbedarf für die Einfügeoperationen wird aus dem Durchschnitt von zehn Ausführungen ermittelt und kann aus Tabelle 4.5 entnommen werden. Zudem wird die Dauer der Testfallausführung für die drei untersuchten Testprogramme angegeben. Es ist ersichtlich, dass Variante 1 bei Testfällen ohne sich wiederholende Statements noch hinter Variante 2 liegt, jedoch bereits ab einer Wiederholungsanzahl von ca. 50 % bereits zu dieser aufschließen kann und ab 70 % stets eine bessere Performance liefert.

Bei den Testprogrammen wird durch die Verwendung von Instrumentierungsvariante 1 bei KickoffTUG und JTopas ebenfalls eine Performancesteigerung erreicht, während bei Reflection-Visitor beide Varianten ähnliche Ergebnisse liefern. Dies liegt daran, dass sowohl KickoffTUG als auch JTopas komplexere Testfälle verwenden, wodurch die Anzahl der sich wiederholenden Statements pro Testfall steigt. Der durchschnittliche Anteil der sich wiederholenden Statements pro Testfall für die einzelnen Testprogramme wird in Tabelle 4.6 aufgelistet.

	Variante 1	Variante 2
Szenario 1	1905 ms	1485 ms
Szenario 2	2588 ms	2526 ms
Szenario 3	4262 ms	4383 ms
Szenario 4	7089 ms	7494 ms
Szenario 5	15804 ms	16992 ms
KickoffTUG	253536 ms	286237 ms
JTopas	244250 ms	298578 ms
Reflection-Visitor	235 ms	237 ms

Tabelle 4.5: Laufzeitbedarf der unterschiedlichen Instrumentierungsvarianten

	Durchschnittlicher Anteil
KickoffTUG	88,79 %
JTopas	73,02 %
Reflection-Visitor	47,50 %

Tabelle 4.6: Durchschnittlicher Anteil der wiederholten Statements pro Testfall

4.2.3 Speicherbedarf

Zur Ermittlung des Speicherbedarfs, der durch die Ausführung des SBD-Tools entsteht, wird das Monitoring-Tool *jconsole* eingesetzt. Dabei handelt es sich um eine Swing-Anwendung, die seit Version 1.5 Bestandteil des JDK ist und zur graphischen Ausgabe des Ressourcenbedarfs von Javaprogrammen dient.

Jconsole öffnet beim Starten des Tools ein Fenster, in dem alle aktuell laufenden Javaprogramme angezeigt werden. Durch Auswahl des entsprechenden Programms wird dessen Ressourcenbedarf mitprotokolliert und entsprechend ausgegeben. Der Speicherbedarf selbst kann über die Registerkarte Memory angezeigt werden. Hier wird der Verlauf des gesamten, vom Heap benötigten Speichers dargestellt.

Da jconsole nur auf bereits laufende Programme zugreifen kann, ist für eine korrekte Analyse des gesamten Speicherverbrauchs eine Anpassung des SBD-Tools nötig. Aus diesem Grund wird zu Beginn eine zusätzliche Instruktion eingefügt, die das Tool zehn Sekunden warten lässt. Diese Zeit reicht aus, um die Analyse des Speicherverbrauchs durch jconsole zu starten. Anschließend startet dass Tool mit der eigentlichen Überprüfung des Testprogramms.

In Abbildung 4.1 ist der Speicherverbrauch für die Ausführung des Tools auf das KickoffTUG Testprogramm ersichtlich. Der maximale Speicherbedarf beträgt hier ungefähr 250 MB und fällt während der Recompilierungsphase an. Der Grund hierfür ist, dass KickoffTUG eine relativ hohe Anzahl an Klassen sowie zusätzlichen JAR-Files besitzt. Die Speicheranforderungen für JTopas sowie Reflection-Visitor betragen 80 bzw. 2 MB und

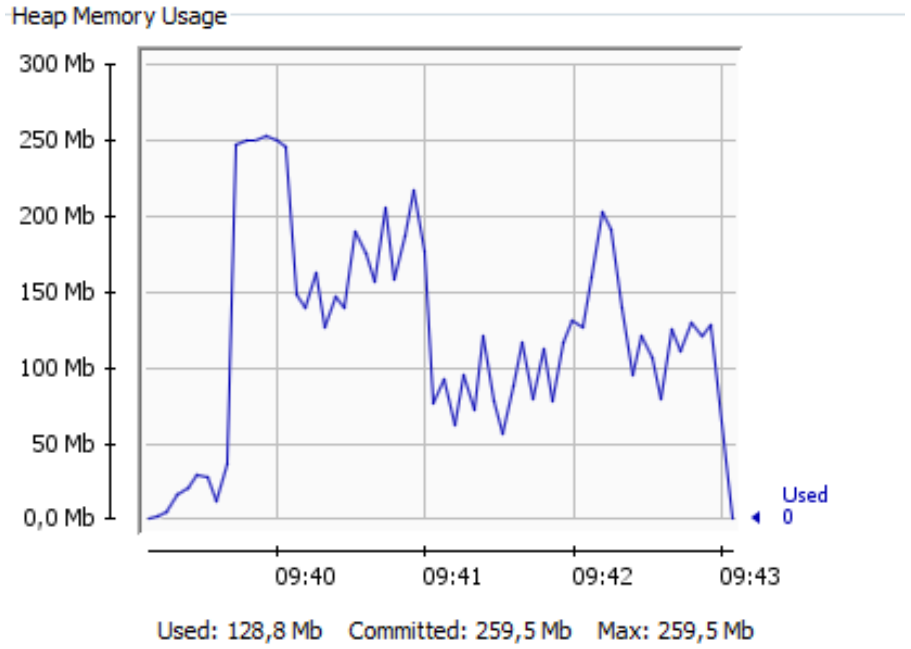

Abbildung 4.1: Ausgabe des Speicherbedarfs für KickoffTUG durch jconsole

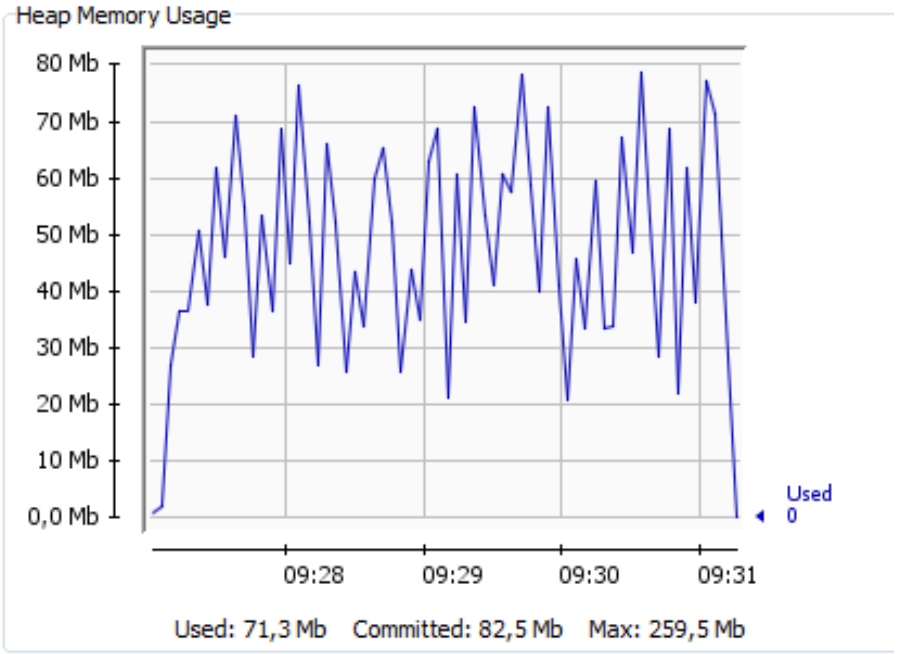

Abbildung 4.2: Ausgabe des Speicherbedarfs für JTopas durch jconsole

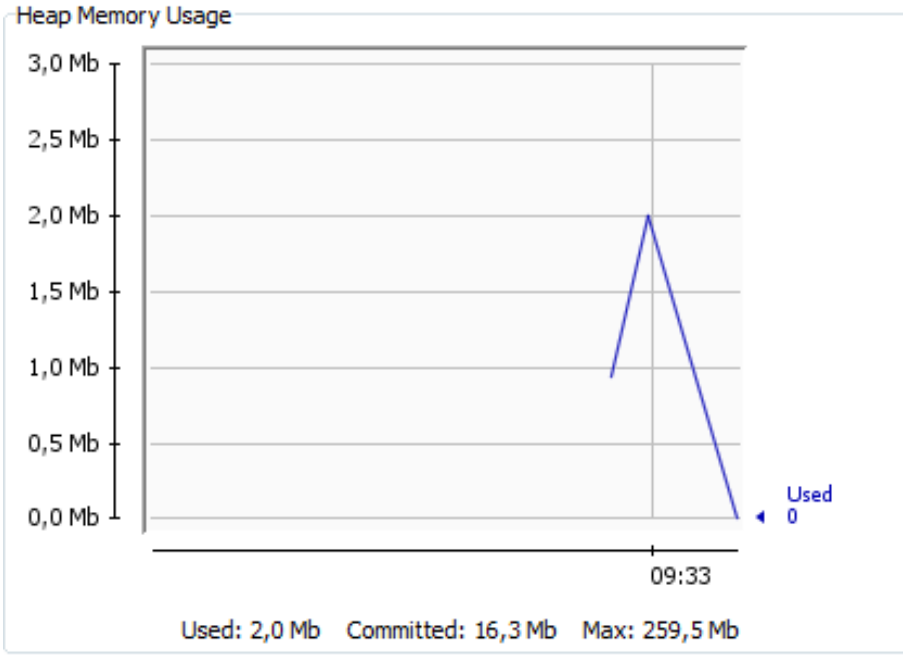

Abbildung 4.3: Ausgabe des Speicherbedarfs für Reflection-Visitor durch jconsole

liegen somit klar unter den von KickoffTUG verursachten (siehe Abbildungen 4.2 und 4.3).

4.3 Technikanalyse

4.3.1 Vorbereitung der Testprogramme

Zur Untersuchung der Spectrum Based Diagnosis Technik werden die in 4.1.1 beschriebenen Testprogramme verwendet. Da diese in der vorliegenden Version keine fehlerhaften Testfälle enthalten, müssen somit im ersten Schritt Statements so verändert werden, dass diese zu fehlschlagenden Testfällen führen.

Grundsätzlich ist davon auszugehen, dass die Anzahl der Testfälle, in denen inkorrekte Statements ausgeführt werden, einen Einfluss auf die anschließenden Ergebnisse der Fehlerlokalisierung hat. Aus diesem Grund wird versucht, die veränderten Statements so zu wählen, dass diese in unterschiedlich vielen Testfällen vorkommen.

Für die nachfolgende Analyse wurden jeweils fünf Statements aus jedem Testprogramm ausgewählt. Diese sind in Tabelle 4.7 angeführt. Zudem wird angegeben, in wie vielen Testfällen das jeweilige Statement ausgeführt wird. Eine Zuordnung der Statements zu den entsprechenden Sourcefiles ist mit Hilfe von Tabelle 4.8 möglich.

KickoffTUG			
Nr.	ursprüngliches Statement	verändertes Statement	# involvierte Testfälle
1-1	*if (polar)*	*if (!polar)*	108
1-2	*setPosition(position);*	*setPosition(velocity);*	75
1-3	*b.writeInto(b_);*	*a.writeInto(b_);*	43
1-4	*cycle_ = cycle;*	*cycle = cycle;*	10
1-5	*if (abstraction_ ¿ 0)*	*if (abstraction_ ¡ 0)*	1
JTopas			
Nr.	ursprüngliches Statement	verändertes Statement	# involvierte Testfälle
2-1	*token.setStartLine(_lineNumber);*	*token.setStartLine(_columnNumber);*	49
2-2	*offset++;*	*offset+=2;*	33
2-3	*if (ex.isWrapper())*	*if (!ex.isWrapper())*	18
2-4	*return true;*	*return false;*	5
2-5	*groups = new String [_matcher.groupCount() + 1];*	*groups = new String [_matcher.groupCount()];*	1
Reflection-Visitor			
Nr.	ursprüngliches Statement	verändertes Statement	# involvierte Testfälle
3-1	*while(!arg1class.equals(Object.class))*	*while(arg1class.equals(Object.class))*	14
3-2	*while(arg1Interfaces.size() ¿ 0)*	*while(arg1Interfaces.size() ¡ 0)*	8
3-3	*return visit(v, null);*	*return visit(v);*	5
3-4	*first = false;*	*first = true;*	2
3-5	*return 1*	*return 0*	1

Tabelle 4.7: Inkorrekte Statements in den Testprogrammen

KickoffTUG		
Nr.	File	Zeilennummer
1-1	src/kickofftug/toolkit/math/SafeVector2D.java	34
1-2	src/kickofftug/system/worldmodel/MovingObject.java	42
1-3	src/kickofftug/system/tactics/clang/RegionQuad.java	30
1-4	src/kickofftug/system/server/relative/RelativeWorldState.java	86
1-5	src/kickofftug/system/command/Attentionto.java	64
JTopas		
Nr.	File	Zeilennummer
2-1	src/de/susebox/jtopas/AbstractTokenizer.java	557
2-2	src/de/susebox/jtopas/impl/SequenceStore.java	491
2-3	src/de/susebox/java/lang/ThrowableMessageFormatter.java	99
2-4	src/de/susebox/jtopas/Token.java	587
2-5	src/de/susebox/jtopas/impl/PatternMatcher.java	119
Reflection-Visitor		
Nr.	File	Zeilennummer
3-1	trunk/src/at/tugraz/ist/reflectionvisitor/util/ReflectionHelper.java	51
3-2	trunk/src/at/tugraz/ist/reflectionvisitor/util/ReflectionHelper.java	63
3-3	trunk/src/at/tugraz/ist/reflectionvisitor/visitors/Visitor.java	27
3-4	trunk/src/at/tugraz/ist/reflectionvisitor/visitors/Visitor.java	95
3-5	trunk/src/at/tugraz/ist/reflectionvisitor/visitors/TestElementCounter.java	18

Tabelle 4.8: Filenamen und Zeilennummern der inkorrekten Statements

Nr.	Verdächtigkeitswert	Statments mit höherem Wert	Statements mit gleichem Wert	Statements mit niedrigerem Wert
1-1	0,41944	2	4	10850
1-2	0,30551	65	6	10865
1-3	0,15250	4	12	10922
1-4	0,70711	252	1	10675
1-5	1,00000	0	165	10763
2-1	0,40406	290	214	649
2-2	0,24618	86	3	1059
2-3	1,00000	0	0	1146
2-4	1,00000	0	7	1145
2-5	1,00000	0	34	1081
3-1	0,75593	2	15	74
3-2	0,79057	0	6	132
3-3	1,00000	0	0	115
3-4	1,00000	0	34	118
3-5	1,00000	0	1	153

Tabelle 4.9: Ergebnisse mit dem Ochiai-Koeffizienten

4.3.2 Performance bei Programmen mit einem Fehler

Das Ziel der Spectrum Based Diagnosis Technik ist die Zuweisung eines möglichst hohen Verdächtigkeitswertes für fehlerhafte Statements, während korrekte Statements kleinere Werte erhalten sollen. Im ersten Schritt werden hierzu Programme analysiert, die genau einen Fehler enthalten.

Hierzu wird jeweils eines der in Tabelle 4.7 definierten fehlerhaften Statements im entsprechenden Testprogramm verwendet. Das Ziel der Technik ist hierbei, diesem einen möglichst hohen Verdächtigkeitswert zuzuweisen, während alle anderen Statements entsprechend niedrige Werte erhalten. Dadurch könnte der Fehler bei der anschließenden Ausgabe der Ergebnisse relativ schnell entdeckt werden.

4.3.2.1 Ergebnisse mit Ochiai

Als Standardkoeffizient wird im Tool Ochiai verwendet. Dieser gilt aktuell als der Koeffizient, mit dem die genaueste Lokalisierung des Fehlers möglich ist. Tabelle 4.9 zeigt die Ergebnisse der Ausführung des Tools für die Testprogramme mit jeweils einem Fehler.

Es ist ersichtlich, dass für acht der insgesamt fünfzehn fehlerhaften Statements der höchstmögliche Wert ausgegeben wird. Allerdings ist hierbei zu beachten, dass die Reihenfolge der Ausgabe für Statements mit gleichem Verdächtigkeitswert alphabetisch nach dem Namen des Sourcefiles angeordnet ist.

So wird beispielsweise dem fehlerhaften Statement 1-5 der höchstmögliche Verdächtigkeitswert zugewiesen. Allerdings verfügen für dieses Testprogramm auch 165 weitere, korrekte Instruktionen über denselben Wert. Somit würde im schlechtesten Fall das fehlerhafte Statement erst an 166. Stelle in der Ausgabe auftreten. Aus diesem Grund sollte die Anzahl für korrekte Statements mit höherer sowie gleicher Verdächtigkeit möglichst gering sein.

Grundsätzlich sollte sich der Fehler für eine effiziente Lokalisierung unter den ersten 30 ausgegebenen Statements befinden. Für KickoffTUG sowie JTopas konnte dies bei zwei der fünf Fälle erreicht werden. Bei Reflection-Visitor wurden sogar vier der fünf fehlerhaften Statements jeweils unter den ersten 30 ausgeben. Allerdings ist dieses Testprogramm auch jenes mit den wenigsten Statements insgesamt, wodurch eine bessere Platzierung wahrscheinlicher ist als bei den beiden größeren Programmen.

Zudem fällt auf, dass für die Statements 1-5, 2-5 und 3-5, die im jeweiligen Testprogramm nur in einem Testfall vorkommen, stets der maximale Verdächtigkeitswert zugewiesen wurde. Jedoch existieren sowohl bei KickoffTUG als auch bei JTopas auch mehrere Statements mit gleichem Wert, wodurch eine Ausgabe des Fehlers unter den ersten 30 Statements nicht garantiert werden kann und somit die Fehlerlokalisierung für diese Fälle als nicht genau genug gilt.

4.3.2.2 Vergleich der Koeffizienten

Die Genauigkeit der Fehlerlokalisierung bei Spectrum Based Diagnosis hängt zum einen davon ab, wie die Statements in die Testfälle involviert sind, zum anderen auch vom Similarity-Koeffizienten, der die Verdächtigkeit der Statements berechnet. Ein Vergleich der im Tool implementierten Koeffizienten Ochiai, Jaccard und Tarantula anhand der zur Verfügung stehenden Testprogramme ist in Tabelle 4.10 angeführt.

Dabei sind sowohl der jeweilige Verdächtigkeitswert als auch die Position des fehlerhaften Statements im Output aufgelistet. Die Position gibt dabei an, an welcher Stelle das Statement sich im Output befindet, wenn alle Statements mit höherer bzw. gleicher Verdächtigkeit vor diesem ausgegeben werden. Dadurch soll verhindert werden, dass die Sortierreihenfolge von gleich verdächtigen Statements das Ergebnis des Vergleichs beeinflusst.

Der Vergleich zeigt, dass Ochiai für die verwendeten Beispiele in jedem Fall bessere oder zumindest gleich gute Ergebnisse liefert wie Jaccard und Tarantula. Dies bestätigt die Vermutung, dass Ochiai den beiden anderen Koeffizienten in den meisten Fällen überlegen ist. Weiters ist erkennbar, dass Jaccard in zwölf der fünfzehn Fälle gleiche Ergebnisse wie Ochiai liefert, während Tarantula dieselbe Lokalisierungsgenauigkeit nur in fünf Fällen erreicht.

Dies ist dadurch erklärbar, wie Ochiai und Jaccard auf Statements, die ausschließlich in negativen Testfällen vorkommen, reagieren. Tarantula weist solchen Statements je-

	Ochiai		Jaccard		Tarantula	
Nr.	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position
1-1	0,41944	7	0,17593	11	0,60619	354
1-2	0,30551	72	0,09333	117	0,68664	129
1-3	0,15250	17	0,02326	17	0,78680	17
1-4	0,70711	254	0,50000	254	0,96795	453
1-5	1,00000	166	1,00000	166	1,00000	166
2-1	0,40406	505	0,16327	526	0,63717	580
2-2	0,24618	90	0,06061	90	0,71560	90
2-3	1,00000	1	1,00000	1	1,00000	30
2-4	1,00000	8	1,00000	8	1,00000	34
2-5	1,00000	35	1,00000	35	1,00000	35
3-1	0,75593	18	0,57143	18	0,50000	45
3-2	0,79057	7	0,62500	7	0,75000	17
3-3	1,00000	1	1,00000	1	1,00000	2
3-4	1,00000	35	1,00000	35	1,00000	69
3-5	1,00000	2	1,00000	2	1,00000	2

Tabelle 4.10: Vergleich der Koeffizienten Ochiai, Jaccard und Tarantula

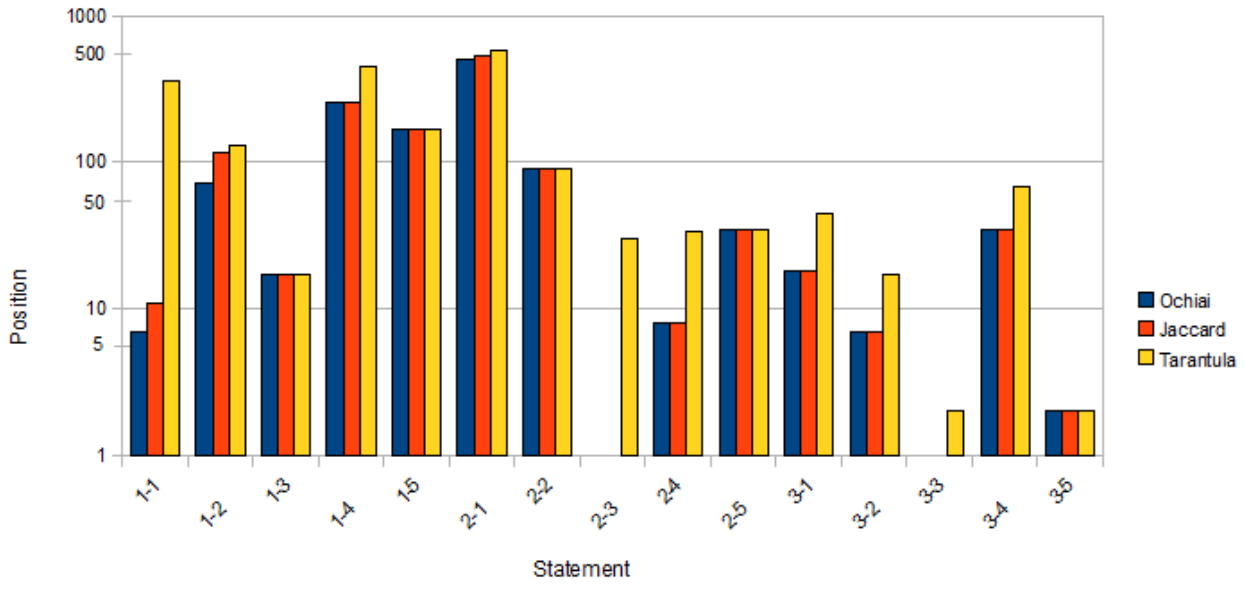

Abbildung 4.4: Position der Statements bei unterschiedlichen Koeffizienten

weils den höchsten Verdächtigkeitswert zu, ohne zu berücksichtigen, in wie vielen unterschiedlichen Testfällen diese vorkommen. Bei Ochiai und Jaccard dagegen erhöht sich die Verdächtigkeit solcher Statements mit deren Beteiligung an negativen Testfällen [7]. Die Auswirkungen dieser Vorgangsweise sind bei Statement 2-3 in Tabelle 4.10 ersichtlich. Während sowohl Ochiai als auch Jaccard dem fehlerhaften Statement den höchsten Verdächtigkeitswert zuweisen, kommen bei Tarantula weitere 29 Statements mit demselben Wert vor.

Der leichte Vorteil von Ochiai gegenüber Jaccard kann hingegen dadurch erklärt werden, dass sich eine Beteiligung bei negativen Testfällen bei Ochiai stärker auswirkt als bei Jaccard und sich somit der Verdächtigkeitswert des entsprechenden Statements verhältnismäßig stärker erhöht [7].

4.3.2.3 Vergleich Hit- und Count-Spektrum

Die bisher beschriebenen Analyseschritte wurden mit Hilfe eines Hit-Spektrums durchgeführt. Bei diesem wird jedes Statement, das während eines Testfalles ausgeführt wird, genau einmal in die entsprechende Zeile des Spektrums eingetragen. Dabei spielt es keine Rolle, wie oft das Statement tatsächlich im entsprechenden Testfall aufgerufen wird.

Im Gegensatz dazu wird beim Aufbau eines Count-Spektrums die Anzahl der Statementausführungen innerhalb eines Testfalles ebenfalls berücksichtigt. Dies hat den Effekt, dass sich ein mehrmaliges Vorkommen, je nach Ergebnis des Testfalles, positiv oder negativ auf den Verdächtigkeitswert der betreffenden Instruktion auswirkt.

Zum Vergleich der Performance des Count-Spektrums bei den verwendeten Testprogrammen wurde dieses den Ergebnissen des Hit-Spektrums unter Verwendung des Ochiai-Koeffizienten gegenübergestellt (Tabelle 4.11). Es ist hierbei ersichtlich, dass die Verwendung des Count-Spektrums bis auf einen Fall gleiche oder schlechtere Ergebnisse liefert.

Der Grund für die teilweise erheblich schlechtere Performance des Count-Spektrums liegt in der Tatsache, dass sich durch die Miteinbeziehung der Ausführungsanzahl einer Instruktion eine relativ starke Beeinflussung auf deren Verdächtigkeit ergibt. So ist beispielsweise ein zehnmaliges Auftreten in einem negativen Testfall gleichbedeutend wie ein einmaliges Auftreten in jeweils zehn negativen Testfällen.

Im Allgemeinen kann jedoch davon ausgegangen werden, dass das Ergebnis eines Testfalles nicht davon beeinflusst wird, ob ein fehlerhaftes Statement ein- oder mehrmals aufgerufen wird. Meist reicht das einmalige Ausführen eines inkorrekten Statements, um einen fehlschlagenden Testfall zu verursachen.

Somit wird durch das Abzählen der Ausführungsanzahl in den meisten Fällen keine zusätzliche Information über die Position des Statements, das zum Fehlschlagen des Testfalles führt, gewonnen. Allerdings steigt dadurch die Wahrscheinlichkeit, dass korrekten Statements, die beispielsweise in einer Schleife ausgeführt werden, eine zu hohe Verdächtigkeit zugewiesen wird.

	Hit-Spektrum		Count-Spektrum	
Nr.	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position
1-1	0,41944	7	0,05119	2078
1-2	0,30551	72	0,02761	1942
1-3	0,15250	17	0,15076	16
1-4	0,70711	254	0,00674	529
1-5	1,00000	166	1,00000	166
2-1	0,40406	505	0,00073	626
2-2	0,24618	90	0,00124	386
2-3	1,00000	1	1,00000	1
2-4	1,00000	8	1,00000	15
2-5	1,00000	35	1,00000	35
3-1	0,75593	18	0,75593	31
3-2	0,79057	7	0,05934	20
3-3	1,00000	1	1,00000	1
3-4	1,00000	35	1,00000	35
3-5	1,00000	2	1,00000	2

Tabelle 4.11: Vergleich von Hit- und Count-Spektrum

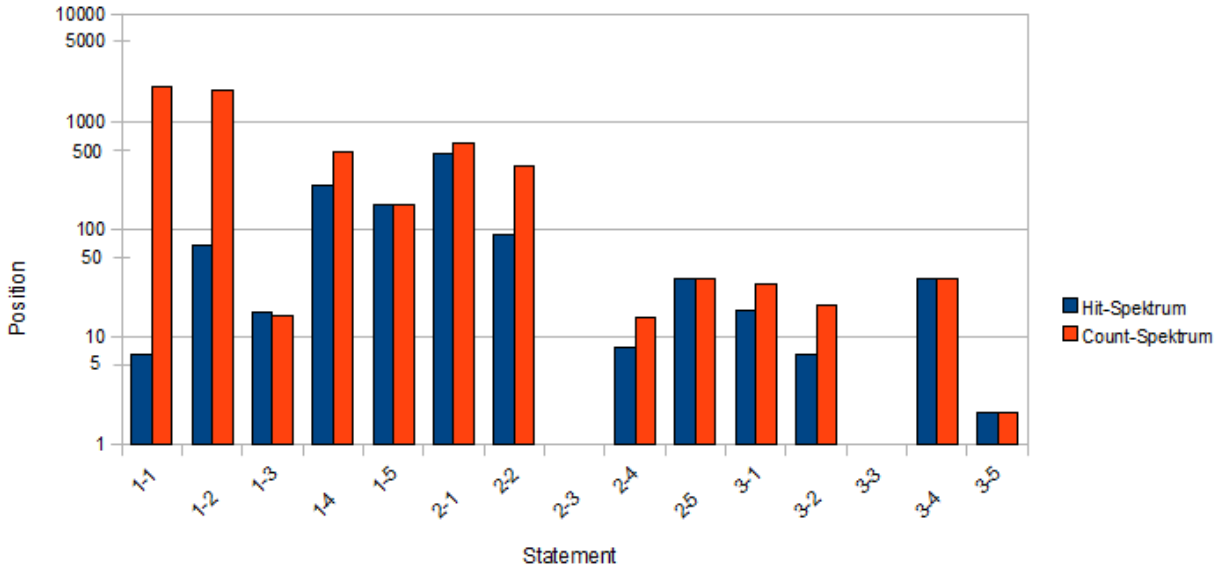

Abbildung 4.5: Position der Statements bei Hit- und Count-Spektrum

	Gesamter Trace		minimaler Fehler-Trace	
Nr.	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position
1-1	0,41944	7	0,41944	5
1-2	0,30551	72	0,30551	37
1-3	0,15250	17	0,15250	17
1-4	0,70711	254	0,70711	254
1-5	1,00000	166	1,00000	166
2-1	0,40406	505	0,40406	485
2-2	0,24618	90	0,24618	90
2-3	1,00000	1	1,00000	1
2-4	1,00000	8	1,00000	8
2-5	1,00000	35	1,00000	35
3-1	0,75593	18	0,75593	18
3-2	0,79057	7	0,79057	7
3-3	1,00000	1	1,00000	1
3-4	1,00000	35	1,00000	35
3-5	1,00000	2	1,00000	2

Tabelle 4.12: Vergleich mit Ausgabe des minimalen Fehler-Traces

4.3.2.4 Variante: Ausgabe des minimalen Fehler-Traces

Eine interessante Variante, die Genauigkeit der Fehlerlokalisierung zu verbessern, wird in [18] vorgestellt. Das darin beschriebene Tool VIDA berücksichtigt für die Ausgabe der Ergebnisse nur Statements, die im kleinsten negativen Testfall ausgeführt werden. Alle anderen Statements werden nicht ausgegeben, ungeachtet der Höhe von deren Verdächtigkeitswerten.

Der Vorteil dieser Vorgangsweise ist, dass dadurch im Idealfall Statements, die zwar in negativen Tests vorkommen, jedoch nicht für dessen Fehlschlagen verantwortlich sind, nicht ausgegeben werden. Dadurch wird das inkorrekte Statement unter Umständen an einer höheren Position ausgegeben, falls dessen Verdächtigkeitswert niedriger ist als der der nicht berücksichtigen Statements.

Durch die Ausgabe des Traces des kleinsten fehlerhaften Testfalles ist zudem sichergestellt, dass bei Programmen mit einem Fehler dieser immer im Trace auftaucht. Der Grund hierfür ist, dass ein Testfall nur durch die Ausführung dieses Statements negativ werden kann. Somit kann garantiert werden, dass durch diese Variante nur korrekte Instruktionen ausgelassen werden.

Ebenso ist eine Verschlechterung der Ausgabeposition nicht möglich, da die Verdächtigkeitswerte selbst nicht verändert werden. Auch hier werden für deren Berechnung wie üblich alle Testfälle sowie die darin vorkommenden Statements berücksichtigt.

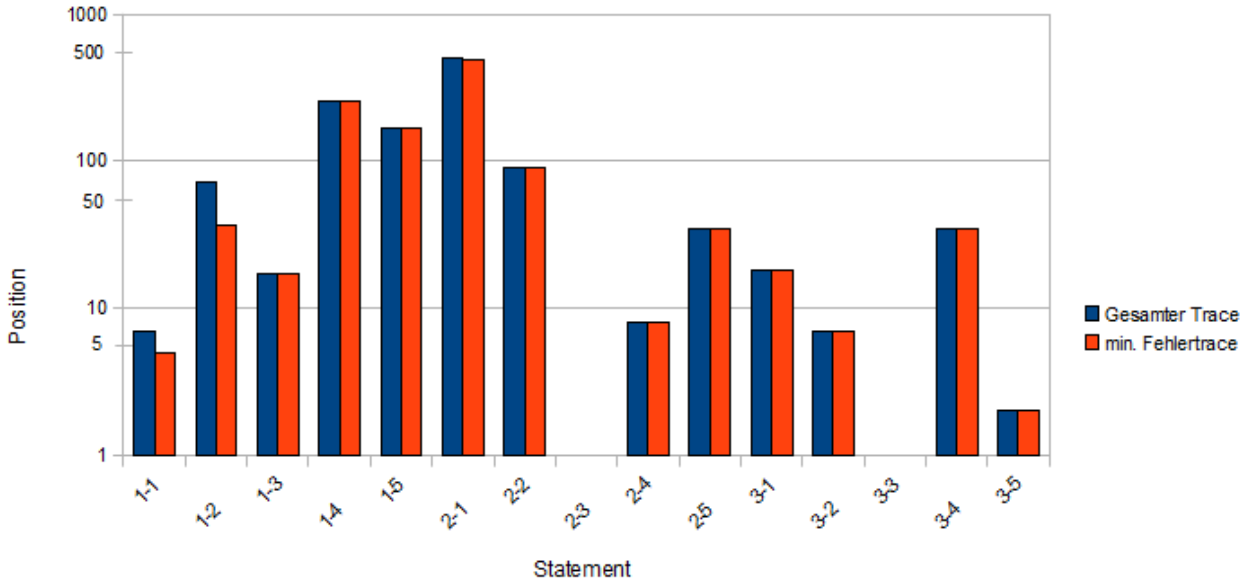

Abbildung 4.6: Position der Statements bei Gesamttrace und min. Trace

Die Verwendung dieser Variante zeigt für die vorhandenen Testprogramme wie erwartet eine teilweise Verbesserung der Ausgabeposition. Insgesamt konnte für drei Fälle eine genauere Lokalisierung erreicht werden. Auffällig ist hierbei, dass die Verbesserung in den drei Fällen auftritt, in denen die Testfallbeteiligung der inkorrekten Statements am größten ist. So ist aus Tabelle 4.7 ersichtlich, dass die fehlerhaften Instruktionen für diese Fälle jeweils an 108, 75 bzw. 49 Testfällen beteiligt sind.

Ein Grund hierfür ist die Tatsache, dass die Beteiligung eines inkorrekten Statements an einem Testfall nicht zwangsläufig zum Fehlschlagen desselben führen muss. Vielmehr kann beobachtet werden, dass das Einfügen eines fehlerhaften Statements meist nur eine relativ geringe Anzahl an negativen Testfällen zur Folge hat (vgl. [7]).

Dieser Umstand kann zu Problemen führen, wenn das betroffene Statement in einer größeren Anzahl an Testfällen vorkommt. Durch die wenigen fehlschlagenden Tests wird die Verdächtigkeit dieses Statements nicht sonderlich erhöht. Allerdings werden korrekten Instruktionen, die zwar an negativen Testfällen beteiligt sind, insgesamt jedoch nur eine geringe Beteiligungsanzahl aufweisen, fälschlicherweise höhere Verdächtigkeitswerte zugewiesen.

Für die verwendeten Testbeispiele ist dies anhand des Falles 1-2 ersichtlich. Das fehlerhafte Statement befindet sich hier im Konstruktor einer Klasse, die als Basis für zwei weitere, relativ häufig benötigte Klassen dient. Dies verursacht die relativ hohe Testfallbeteiligung des betroffenen Statements.

Allerdings führt auch diese hohe Beteiligung nur zu insgesamt sieben fehlschlagenden Tests, wodurch sich ein Verdächtigkeitswert von 0,30551 ergibt. Im Gegenzug dazu exis-

tieren mehrere korrekte Instruktionen, die an insgesamt sechs bzw. sieben Testfällen beteiligt sind, von denen zumindest einer fehlschlägt. Somit wird jedem dieser Statements ein höherer Verdächtigkeitswert zugewiesen als dem für das Fehlschlagen des Testfalles verantwortlichen, fehlerhaften Statement.

Durch die Ausgabe des minimalen Fehlertraces kann dieser Umstand eventuell entschärft werden, da die Wahrscheinlichkeit, dass Statements mit einer geringen Testfallbeteiligung im minimalen Fehlertrace liegen, relativ gering ist. Dadurch können viele dieser Statements bei der Ausgabe ignoriert werden.

4.3.2.5 Testfallsensitivität

Die Qualität der Fehlerlokalisierung basiert bei Spectrum Based Diagnosis zu einem großen Teil auf der Anzahl und Art der zugrunde liegenden Testfälle. Dabei ist jedoch nicht immer eindeutig, wie sich das Vorhandensein von positiven und negativen Testfällen auf das Ergebnis auswirkt.

In [7] wurde hierzu ein Versuch durchgeführt, bei dem jeweils eine bestimmte Anzahl an zufällig gewählten positiven und negativen Tests zur Bestimmung der Fehlerposition verwendet wurde. Das Resultat dieses Versuchs zeigt, dass eine Verminderung der zur Verfügung stehenden negativen Testfälle eine Verschlechterung der Fehlerlokalisierung bewirkt, während das Hinzufügen oder Entfernen von positiven Tests das Ergebnis im Allgemeinen nur geringfügig beeinflusst. Zudem stellt sich heraus, dass ab einem Wert von ungefähr zehn negativen Testfällen keine weite Verbesserung durch eine Erhöhung dieser eintritt.

In der folgenden Analyse wurde ein ähnlicher Ansatz gewählt, um die Testfallsensitivität der zugrunde liegenden Testprogramme zu untersuchen. Als Ausgangsbasis hierzu dienen die drei Testprogramme in ihrer fehlerlosen Version. Um eine möglichst hohe Anzahl an Testfällen zu erzeugen, wurde in jedes Programm jener Fehler aus Tabelle 4.7 eingefügt, für den das Produkt aus den resultierenden positiven und negativen Testfällen den größten Wert besitzt.

Anschließend wurden die Testprogramme mit einer unterschiedlichen Anzahl an positiven und negativen Tests ausgeführt, wobei die Auswahl der jeweiligen Testfälle auf folgendem Algorithmus basiert:

```
for n = 1 bis Anzahl der positiven Tests do
  for m = 1 bis Anzahl der negativen Tests do
    wähle zufällig n positive Tests aus der Testumgebung
    wähle zufällig m negative Tests aus der Testumgebung
    führe die Spectrum Based Diagnosis durch
    berechne die Ausgabeposition des jeweiligen Fehlerstatements
  end for
```

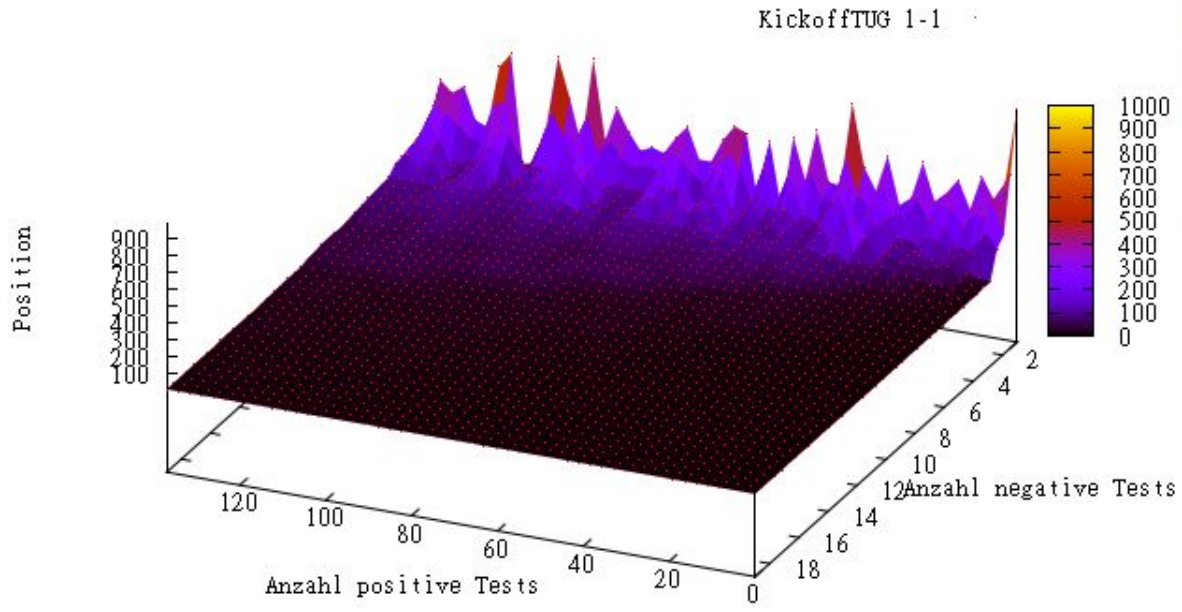

Abbildung 4.7: Testfallsensitivität bei KickoffTUG 1-1

end for

Durch diese Vorgangsweise entstehen bei n positiven und m negativen Testfällen insgesamt n*m unterschiedliche Testumgebungen für jedes Programm. Zusätzlich wurde, um den Einfluss der zufälligen Auswahl der Testfälle möglichst gering zu halten, der Algorithmus insgesamt zehn Mal auf jedes Testprogramm angewendet und anschließend der Durchschnitt aus diesen gebildet.

Die Ergebnisse des Versuchs sind in den Abbildungen 4.7, 4.8 und 4.9 ersichtlich. Es zeigt sich, dass im Falle von KickoffTUG und JTopas eine Variation von positiven Testfällen tatsächlich kaum Einfluss auf das Ergebnis der Fehlerlokalisierung hat, solange ausreichend negative Testfälle vorhanden sind. Ebenso wird die Annahme bestätigt, dass eine Zunahme an negativen Testfällen ab einer bestimmten Anzahl keine Verbesserung mit sich bringt.

Im Falle des Reflection-Visitor Beispiels zeigt sich bei der Variation der negativen Tests ein ähnliches Bild. Allerdings bewirkt hier eine Erhöhung der positiven Tests eine Verschlechterung des Resultats. Der Grund hierfür ist, dass das gesuchte Statement sowohl in allen negativen als auch in allen positiven Testfällen ausgeführt wird, wodurch eine Zunahme an positiven Testfällen eine Senkung des entsprechenden Verdächtigkeitswertes bedeutet.

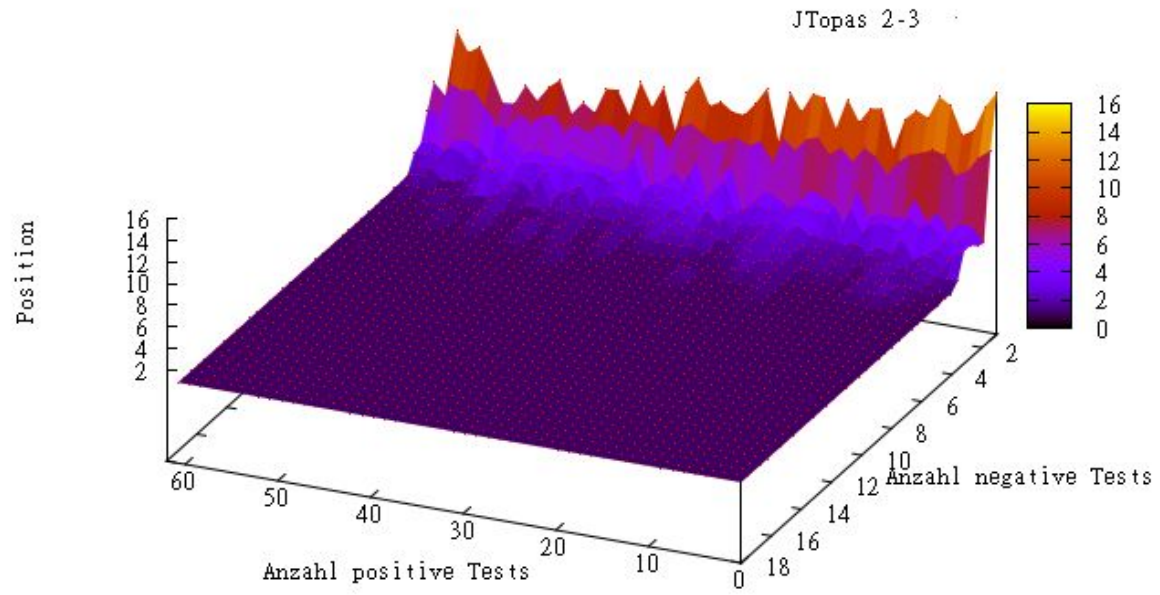

Abbildung 4.8: Testfallsensitivität bei JTopas 2-3

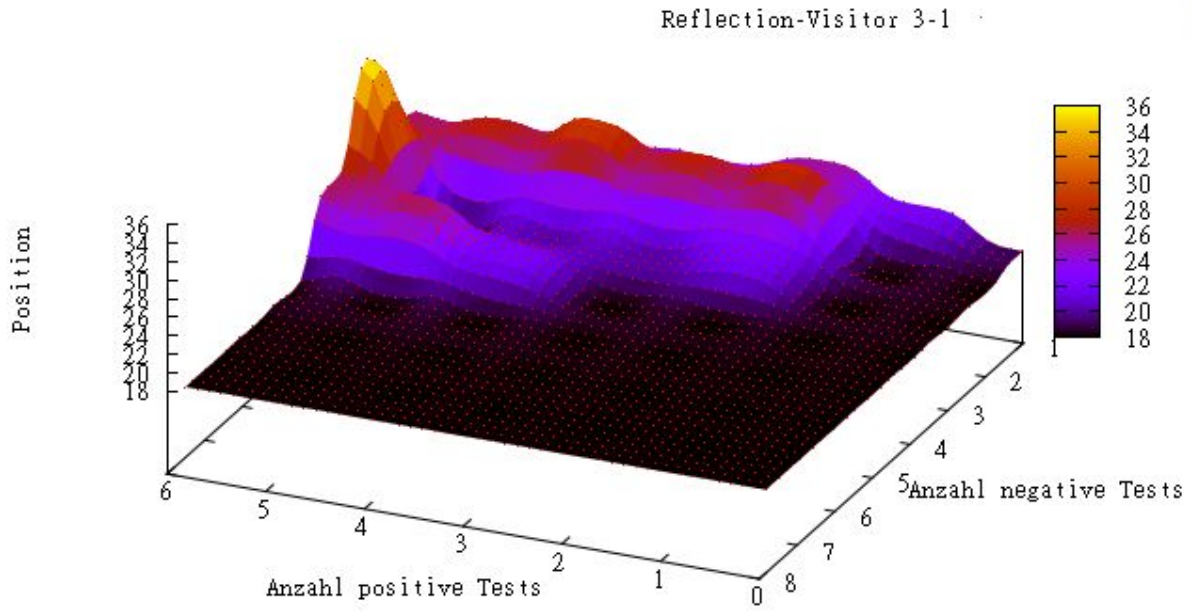

Abbildung 4.9: Testfallsensitivität bei Reflection-Visitor 3-1

4.3.3 Performance bei Programmen mit zwei Fehlern

Bisher wurden ausschließlich Programme betrachtet, in denen genau eine fehlerhafte Instruktion auftritt. Im Allgemeinen ist jedoch die Anzahl der unterschiedlichen Statements, die das Fehlschlagen von Testfällen verursachen, a priori nicht bekannt.

Zudem ist es meist auch nicht möglich, die tatsächliche Fehleranzahl im Vorhinein eindeutig zu bestimmen. Theoretisch besteht zwar die Möglichkeit, das Vorhandensein von mehreren Fehlern zu zeigen, indem Traces von negativen Testfällen gefunden werden, die vollständig disjunkt sind. So kann beispielsweise durch das Auftreten von zwei disjunkten, negativen Testfällen auf mindestens zwei voneinander unabhängige, fehlerhafte Statements geschlossen werden.

Allerdings kann umgekehrt nicht bewiesen werden, dass im Programm nur ein Fehler enthalten ist, wenn keine disjunkten negativen Testfälle gefunden werden. Der Grund hierfür ist, dass das Auftreten eines Statements in unterschiedlichen fehlschlagenden Testfällen nicht zwangsläufig bedeutet, dass dieses Statement auch für das Fehlschlagen des jeweiligen Tests verantwortlich ist.

Vor allem dieser Umstand kann bei der Durchführung von Spectrum Based Diagnosis bei Programmen mit mehreren Fehlern zu Problemen führen. Das grundlegende Prinzip der Technik basiert auf dem Zuweisen von hohen Verdächtigkeitswerten für Komponenten, die an mehreren negativen Testfällen beteiligt sind. Gleichzeitig wird jedoch der Verdächtigkeitswert gesenkt, wenn negative Testfälle existieren, in denen die Komponente nicht auftritt. Somit besteht durch das Vorhandensein von mehreren Fehlern die Möglichkeit, dass inkorrekte Instruktionen an einigen negativen Tests nicht beteiligt sind und somit einen, im Vergleich zu Programmen mit nur einem Fehler, zu niedrigeren Verdächtigkeitswert erhalten.

Um die tatsächliche Auswirkung von mehreren Fehlern auf die Performance von Spectrum Based Diagnosis zu untersuchen, wurden für die folgende Analyse jeweils zwei der in Tabelle 4.7 angeführten Fehler in das jeweilige Testprogramm eingefügt. Dadurch ergeben sich insgesamt 30 unterschiedliche, fehlerhafte Programme.

4.3.3.1 Ergebnisse mit Ochiai

Die Ergebnisse der durchgeführten Analyse mit Berechnung der Verdächtigkeiten durch den Ochiai-Koeffizienten sind in Tabelle 4.13 abgebildet. Die Performance wird durch das fehlerhafte Statement mit der höchsten Ausgabeposition gemessen, da grundsätzlich bereits das Auffinden eines Fehlers als Erfolg gewertet werden kann. Dadurch wäre es möglich, den jeweiligen Fehler zu beheben, wodurch sich wiederum ein Programm mit nur einem fehlerhaften Statement ergibt.

Aus der Tabelle ist ersichtlich, dass sich in insgesamt 20 Fällen eines der fehlerhaften Statements unter den ersten 30 ausgegebenen Werten befindet. Vor allem bei Reflection-Visitor konnte in neun der zehn Fälle eine erfolgreiche Lokalisierung durchgeführt werden.

Nr.	gefundenes Statement	Verdächtig-keitswert	Statments mit höherem Wert	Statements mit gleichem Wert	Statements mit niedrigerem Wert
1-1 & 1-2	1-1	0,47140	0	4	10847
1-1 & 1-3	1-1	0,41944	1	4	10850
1-1 & 1-4	1-1	0,47140	2	4	10838
1-1 & 1-5	1-1	0,43033	2	4	10838
1-2 & 1-3	1-2	0,28577	72	6	10856
1-2 & 1-4	1-4	0,45644	274	1	10649
1-2 & 1-5	1-5	0,35355	40	188	10696
1-3 & 1-4	1-4	0,64550	252	1	10673
1-3 & 1-5	1-5	0,70711	0	169	10757
1-4 & 1-5	1-4	0,64550	252	1	10663
2-1 & 2-2	2-2	0,45437	250	2	896
2-1 & 2-3	2-3	0,83205	0	0	1146
2-1 & 2-4	2-4	0,67420	8	7	1137
2-1 & 2-5	2-1	0,42857	283	214	618
2-2 & 2-3	2-3	0,94868	0	0	1141
2-2 & 2-4	2-4	0,84515	0	7	1140
2-2 & 2-5	2-5	0,57735	33	35	1042
2-3 & 2-4	2-3	0,88465	0	0	1145
2-3 & 2-5	2-3	0,97333	0	0	1108
2-4 & 2-5	2-4	0,91287	0	7	1107
3-1 & 3-2	3-1	0,96362	0	17	27
3-1 & 3-3	3-3	0,70711	0	2	85
3-1 & 3-4	3-1	0,75593	2	15	74
3-1 & 3-5	3-1	0,75593	2	15	74
3-2 & 3-3	3-3	0,74536	0	0	102
3-2 & 3-4	3-2	0,93541	0	6	130
3-2 & 3-5	3-2	0,86603	0	6	132
3-3 & 3-4	3-3	0,91287	0	0	113
3-3 & 3-5	3-3	0,91287	0	0	115
3-4 & 3-5	3-4	1,00000	0	34	118

Tabelle 4.13: Ergebnisse mit dem Ochiai-Koeffizienten

Der tatsächliche Einfluss, den zwei Fehler auf das jeweilige Testprogramm haben, wird in Tabelle 4.14 gezeigt. Hierzu wird die Position des gefundenen Statements mit der Position verglichen, die dieses bei der Variante mit nur einem Fehler einnehmen würde. Ebenso wird angeführt, in wie vielen negativen Testfällen das Statement tatsächlich auftritt.

Es fällt auf, dass sich die Position des Statements durch das Vorhandensein eines weiteren Fehlers in insgesamt neun Fällen verschlechtert. Dies ist vor allem dann der Fall, wenn durch das zusätzliche Fehlerstatement weitere negative Testfälle auftreten, in die das gefundene nicht involviert ist. Hingegen tritt bei einer annähernd gleich bleibenden Anzahl an negativen Testfällen meist kein negativer Einfluss auf.

Neben den Verschlechterungen treten ebenso Fälle auf, in denen sich die Ausgabeposition des Statements durch den zusätzlichen Fehler verbessert. Der Grund hierfür ist, dass durch die zusätzliche fehlerhafte Instruktion Testfälle fehlschlagen, in denen das betrachtete Statement ebenfalls vorkommt, jedoch nicht für das Fehlschlagen verantwortlich ist.

				Programm mit zwei Fehlern		Programm mit einem Fehler	
Nr.	gefundenes Statement	# involvierte neg. Testfälle	# gesamte neg. Testfälle	Verdächtig-keitswert	Position	Verdächtig-keitswert	Position
1-1 & 1-2	1-1	24	24	0,47140	5	0,41944	7
1-1 & 1-3	1-1	19	19	0,41944	6	0,41944	7
1-1 & 1-4	1-1	24	24	0,47140	7	0,41944	7
1-1 & 1-5	1-1	20	20	0,43033	7	0,41944	7
1-2 & 1-3	1-2	7	8	0,28577	79	0,30551	72
1-2 & 1-4	1-4	5	12	0,45644	276	0,70711	254
1-2 & 1-5	1-5	1	8	0,35355	229	1,00000	166
1-3 & 1-4	1-4	5	6	0,64550	254	0,70711	254
1-3 & 1-5	1-5	1	2	0,70711	170	1,00000	166
1-4 & 1-5	1-4	5	6	0,64550	254	0,70711	254
2-1 & 2-2	2-2	8	10	0,45437	253	0,24618	90
2-1 & 2-3	2-3	18	26	0,83205	1	1,00000	1
2-1 & 2-4	2-4	5	11	0,67420	16	1,00000	8
2-1 & 2-5	2-1	9	9	0,42857	498	0,40406	505
2-2 & 2-3	2-3	18	20	0,94868	1	1,00000	1
2-2 & 2-4	2-4	5	7	0,84515	8	1,00000	8
2-2 & 2-5	2-5	1	3	0,57735	69	1,00000	35
2-3 & 2-4	2-3	18	23	0,88465	1	1,00000	1
2-3 & 2-5	2-3	18	19	0,97333	1	1,00000	1
2-4 & 2-5	2-4	5	6	0,91287	8	1,00000	8
3-1 & 3-2	3-1	13	13	0,96362	18	0,75593	18
3-1 & 3-3	3-3	5	10	0,70711	3	1,00000	1
3-1 & 3-4	3-1	8	8	0,75593	18	0,75593	18
3-1 & 3-5	3-1	8	8	0,75593	18	0,75593	18
3-2 & 3-3	3-3	5	9	0,74536	1	1,00000	1
3-2 & 3-4	3-2	7	7	0,93541	7	0,79057	7
3-2 & 3-5	3-2	6	6	0,86603	7	0,79057	7
3-3 & 3-4	3-3	5	6	0,91287	1	1,00000	1
3-3 & 3-5	3-3	5	6	0,91287	1	1,00000	1
3-4 & 3-5	3-4	2	2	1,00000	35	1,00000	35

Tabelle 4.14: Vergleich der Resultate bei einem bzw. zwei Fehlern im Programm

4.3.3.2 Vergleich der Koeffizienten

Wie bereits erwähnt, wird bei den Koeffizienten Ochiai, Jaccard und Tarantula das Auftreten bzw. Nicht-Auftreten von Statements in Testfällen unterschiedlich gewichtet. Durch das Vorhandensein von mehreren Programmfehlern stellt sich somit die Frage, ob dadurch Vorteile für Jaccard bzw. Tarantula gegenüber Ochiai entstehen.

Der Vergleich der Koeffizienten in Tabelle 4.15 bzw. Abbildung 4.10 zeigt, dass Ochiai in insgesamt sechs Fällen eine genauere Lokalisierung möglich ist als bei Verwendung von Jaccard. Im Gegenzug ist Jaccard Ochiai nur einmal überlegen. Der Unterschied ist jedoch in keinem Fall derartig gravierend, dass sich durch den Austausch des Koeffizienten eine Verschiebung des fehlerhaften Statements unter die ersten 30 ausgegebenen Statements ergibt.

Im Gegensatz zu den beiden anderen Koeffizienten scheint die Beeinträchtigung der Lokalisierungsgenauigkeit bei Tarantula durch mehrere fehlerhafte Instruktionen erheblich größer auszufallen. So kann Tarantula weder bei KickoffTUG noch bei JTopas eines der inkorrekten Statements erfolgreich identifizieren. Auch bei Reflection-Visitor befindet sich nur bei drei der insgesamt zehn Testprogramme ein Fehler unter den ersten 30 Ausgabezeilen.

Auffällig ist weiterhin, dass Tarantula insgesamt fünfmal bessere Ergebnisse liefert als Ochiai und Jaccard, während dieser bei Programmen mit einem Fehler konstant gleiche oder schlechtere Werte aufweist. Der Grund hierfür liegt in der Tatsache, dass durch die Zuweisung des maximalen Verdächtigkeitswertes für Statements, die ausnahmslos in fehlschlagenden Tests vorkommen, dieser durch weitere negative Testfälle nicht beeinflusst wird. Dieser Vorteil kann jedoch zumeist nicht genutzt werden, da durch diese Art der Berechnung in der Regel auch eine größere Anzahl an korrekten Statements mit maximaler Verdächtigkeit existiert.

4.3.3.3 Variante: Ausgabe des minimalen Fehler-Traces

In Abschnitt 4.3.2.4 wurde die Lokalisierungsvariante mit Ausgabe des minimalen Fehlertraces vorgestellt. Diese bietet für Programme mit einem Fehler den Vorteil, dass sich die Position des fehlerhaften Statements im besten Fall verbessert. Eine Verschlechterung ist im Gegenzug nicht möglich. Zudem wird garantiert, dass der Fehler in jedem Fall ausgegeben wird.

Bei Programmen, die zwei oder mehr Fehler enthalten, kann diese Ausgabevariante ebenfalls zu einer besseren Fehlerlokalisierung führen. Wiederum wird hierbei garantiert, dass sich zumindest eines der für das Fehlschlagen des Testfalles verantwortlichen Statements in der Ausgabe aufscheint.

Allerdings muss hierbei beachtet werden, dass im Vorhinein nicht bekannt ist, welche und wie viele Statements tatsächlich ausgegeben werden. Dadurch besteht im Gegensatz zu

	Ochiai		Jaccard		Tarantula	
Nr.	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position	Verdächtigkeits-wert	Position
1-1 & 1-2	0,47140	5	0,22222	12	0,61111	341
1-1 & 1-3	0,41944	6	0,17593	11	0,60619	353
1-1 & 1-4	0,47140	7	0,22222	7	0,84615	579
1-1 & 1-5	0,43033	7	0,18519	11	1,00000	216
1-2 & 1-3	0,28577	79	0,09211	115	0,65570	136
1-2 & 1-4	0,45644	276	0,29412	276	0,92308	543
1-2 & 1-5	0,35355	229	0,12500	229	1,00000	205
1-3 & 1-4	0,64550	254	0,45455	254	0,96154	457
1-3 & 1-5	0,70711	170	0,50000	170	1,00000	170
1-4 & 1-5	0,64550	254	0,45455	254	1,00000	178
2-1 & 2-2	0,45437	253	0,24242	129	0,70886	184
2-1 & 2-3	0,83205	1	0,69231	1	1,00000	39
2-1 & 2-4	0,67420	16	0,45455	25	1,00000	40
2-1 & 2-5	0,42857	498	0,18367	535	1,00000	44
2-2 & 2-3	0,94868	1	0,90000	1	1,00000	58
2-2 & 2-4	0,84515	8	0,71429	8	1,00000	62
2-2 & 2-5	0,57735	69	0,33333	69	1,00000	63
2-3 & 2-4	0,88465	1	0,78261	1	1,00000	64
2-3 & 2-5	0,97333	1	0,94737	1	1,00000	65
2-4 & 2-5	0,91287	8	0,83333	8	1,00000	69
3-1 & 3-2	0,96362	18	0,92857	18	0,50000	43
3-1 & 3-3	0,70711	3	0,50000	3	1,00000	28
3-1 & 3-4	0,75593	18	0,57143	18	0,50000	45
3-1 & 3-5	0,75593	18	0,57143	18	0,50000	45
3-2 & 3-3	0,74536	1	0,55556	1	1,00000	5
3-2 & 3-4	0,93541	7	0,87500	7	1,00000	71
3-2 & 3-5	0,86603	7	0,75000	7	1,00000	4
3-3 & 3-4	0,91287	1	0,83333	1	1,00000	41
3-3 & 3-5	0,91287	1	0,83333	1	1,00000	43
3-4 & 3-5	1,00000	35	1,00000	35	1,00000	69

Tabelle 4.15: Vergleich der Koeffizienten Ochiai, Jaccard und Tarantula

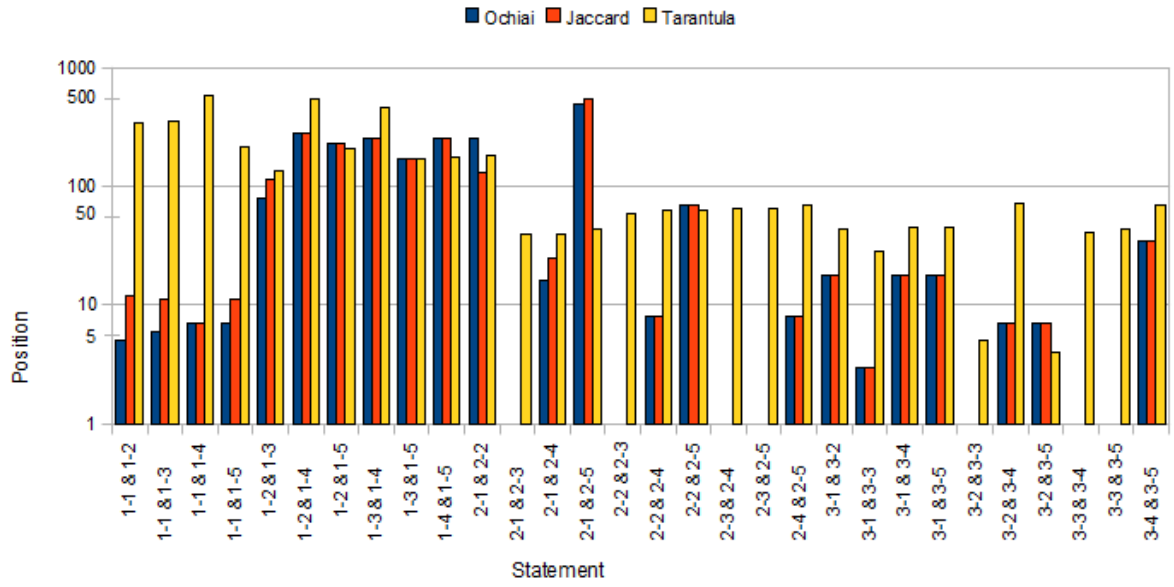

Abbildung 4.10: Position der Statements bei unterschiedlichen Koeffizienten

Programmen mit nur einem Fehler die Gefahr, dass sich die Position des ersten ausgegebenen Fehlerstatements verschlechtert. Dies wäre dann der Fall, wenn das Statement im minimalen Fehlertrace an einer Position liegt, die niedriger ist als die eines anderen Fehlers im gesamten Trace.

Die Performance der genannten Variante wird in Tabelle 4.16 gezeigt. Es ist ersichtlich, dass sich vor allem bei KickoffTUG die Genauigkeit der Lokalisierung erheblich erhöht. So kann dadurch bei sieben der zehn fehlerhaften Testprogramme ein Fehler innerhalb der ersten 30 Ausgabezeilen ausfindig gemacht werden. Insgesamt ist bei elf Testprogrammen eine Verbesserung zu vermerken, während in einem Fall eine niedrigere Ausgabeposition auftritt.

Auffällig ist hierbei, dass bei den Testprogrammen mit der größten Verbesserung der Lokalisierung jeweils ein anderes Statement gefunden wird als bei der Ausgabe des gesamten Traces. In diesen Fällen kann die bessere Lokalisierung dadurch erklärt werden, dass das gefundene Statement selbst zwar eine niedrigere Verdächtigkeit aufweist, diese jedoch durch die Reduzierung der Ausgabe kompensiert werden kann. Es muss jedoch angemerkt werden, dass diese Reduzierung nur dann das Ergebnis verbessern kann, wenn der Trace des kleinsten fehlerhaften Testfalles nur eine relativ geringe Anzahl an Komponenten enthält.

4.3.3.4 Testfallsensitivität

Wie im Falle der Programme mit einem Fehler stellt sich auch hier die Frage, wie sich eine Variation der Testfälle auf das Ergebnis der Fehlerlokalisierung auswirkt. Vor allem der Einfluss einer Verminderung der negativen Tests ist hier von Interesse, da im Gegensatz zu

	Gesamter Trace			minimaler Fehler-Trace		
Nr.	gefundenes Statement	Verdächtig-keitswert	Position	gefundenes Statement	Verdächtig-keitswert	Position
1-1 & 1-2	1-1	0,47140	5	1-1	0,47140	5
1-1 & 1-3	1-1	0,41944	6	1-1	0,41944	5
1-1 & 1-4	1-1	0,47140	7	1-1	0,47140	5
1-1 & 1-5	1-1	0,43033	7	1-1	0,43033	5
1-2 & 1-3	1-2	0,28577	79	1-3	0,10783	24
1-2 & 1-4	1-4	0,45644	276	1-2	0,23333	41
1-2 & 1-5	1-5	0,35355	229	1-2	0,32660	37
1-3 & 1-4	1-4	0,64550	254	1-3	0,06226	21
1-3 & 1-5	1-5	0,70711	170	1-3	0,10783	24
1-4 & 1-5	1-4	0,64550	254	1-4	0,64550	254
2-1 & 2-2	2-2	0,45437	253	2-2	0,45437	206
2-1 & 2-3	2-3	0,83205	1	2-3	0,83205	1
2-1 & 2-4	2-4	0,67420	16	2-4	0,67420	12
2-1 & 2-5	2-1	0,42857	498	2-1	0,42857	489
2-2 & 2-3	2-3	0,94868	1	2-3	0,94868	1
2-2 & 2-4	2-4	0,84515	8	2-4	0,84515	8
2-2 & 2-5	2-5	0,57735	69	2-2	0,30151	71
2-3 & 2-4	2-3	0,88465	1	2-3	0,88465	1
2-3 & 2-5	2-3	0,97333	1	2-3	0,97333	1
2-4 & 2-5	2-4	0,91287	8	2-4	0,91287	8
3-1 & 3-2	3-1	0,96362	18	3-1	0,96362	18
3-1 & 3-3	3-3	0,70711	3	3-3	0,70711	1
3-1 & 3-4	3-1	0,75593	18	3-1	0,75593	18
3-1 & 3-5	3-1	0,75593	18	3-1	0,75593	18
3-2 & 3-3	3-3	0,74536	1	3-3	0,74536	1
3-2 & 3-4	3-2	0,93541	7	3-2	0,93541	7
3-2 & 3-5	3-2	0,86603	7	3-2	0,86603	7
3-3 & 3-4	3-3	0,91287	1	3-3	0,91287	1
3-3 & 3-5	3-3	0,91287	1	3-3	0,91287	1
3-4 & 3-5	3-4	1,00000	35	3-4	1,00000	35

Tabelle 4.16: Vergleich mit Ausgabe des minimalen Fehler-Traces

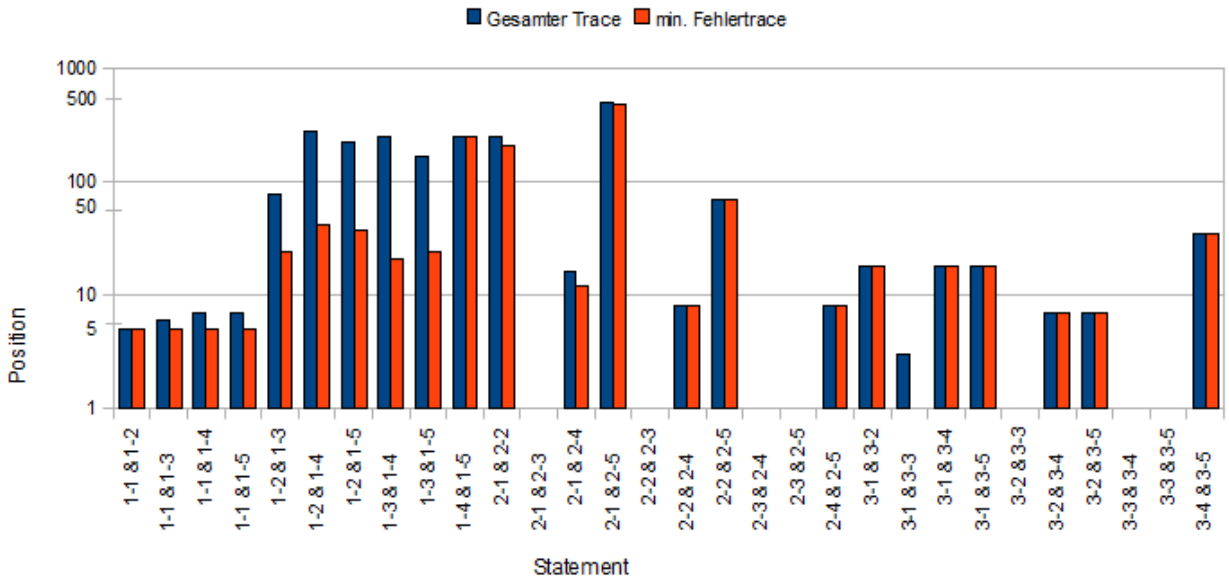

Abbildung 4.11: Position der Statements bei Gesamttrace und min. Trace

der in Abschnitt 4.3.2.5 durchgeführten Untersuchung auch negative Testfälle existieren können, die nicht direkt durch das gefundene fehlerhafte Statement verursacht wurden.

Die grundsätzliche Vorgangsweise zur Variation der Testfälle entspricht der in Abschnitt 4.3.2.5 beschriebenen. Ebenso wird wiederum für jede Variante das Fehlerstatement mit der höchsten Ausgabeposition betrachtet. Dabei gilt es zu beachten, dass sich diese, je nachdem, welche Testfälle ausgewählt wurden, innerhalb der insgesamt zehn berechneten Versionen einer bestimmten Testumgebungsvariante unterscheiden können.

Die Ergebnisse der Testfallvariation zeigen, dass sich ähnlich wie bei Programmen mit jeweils nur einem Fehler ab einer gewissen Anzahl an negativen Tests keine Verbesserung der Lokalisierung durch Hinzufügen von weiteren fehlschlagenden Testfällen erreichen lässt. Im Gegensatz dazu zeigt sich jedoch, dass der Einfluss der positiven Testfälle hierbei größer zu sein scheint.

So wirkt sich sowohl bei JTopas (Abbildung 4.13) als auch bei Reflection-Visitor (Abbildung 4.14) ein Wegfall aller positiven Testfälle negativ aus, während dies bei Versionen mit einem Fehler nicht der Fall war. Zudem steigert ein Hinzufügen von positiven Tests bei beiden Programmen die Genauigkeit der Fehlerlokalisierung

Für KickoffTUG ist aus Abbildung 4.12 ersichtlich, dass sich dieses bei Vorhandensein von zwei Fehlern ähnlich verhält wie bei nur einem Fehler. So bewirkt auch hier eine Steigerung der positiven Tests eine leichte Verschlechterung der Resultate, wenn die Anzahl der zur Verfügung stehenden negativen Tests kleiner als zehn ist.

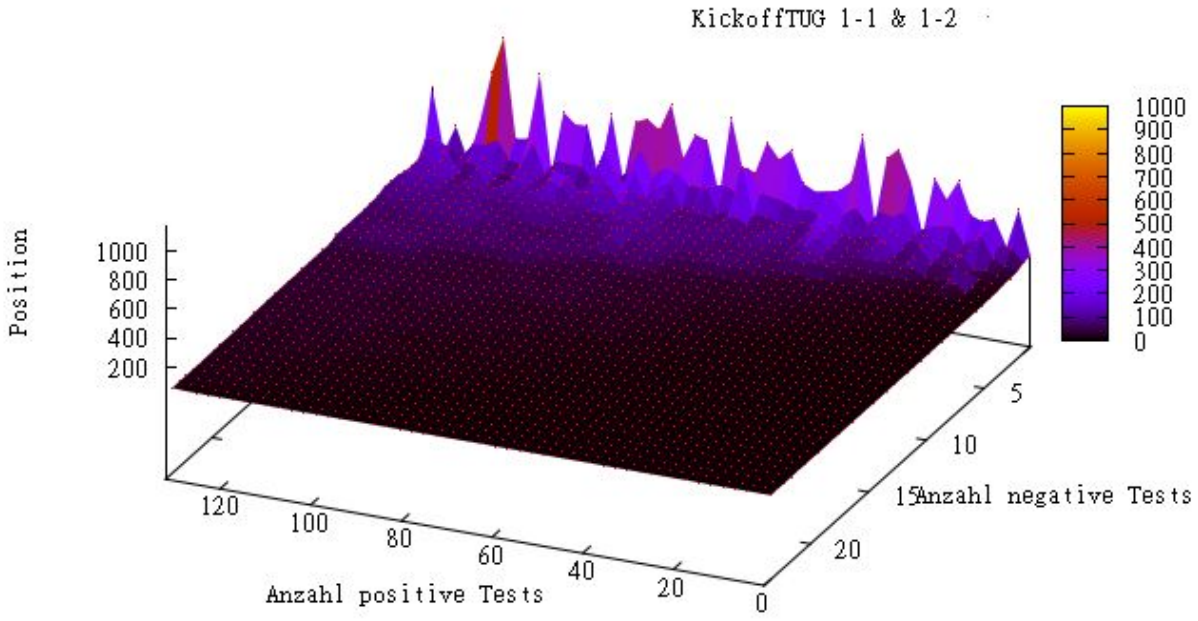

Abbildung 4.12: Testfallsensitivität bei KickoffTUG 1-1 & 1-2

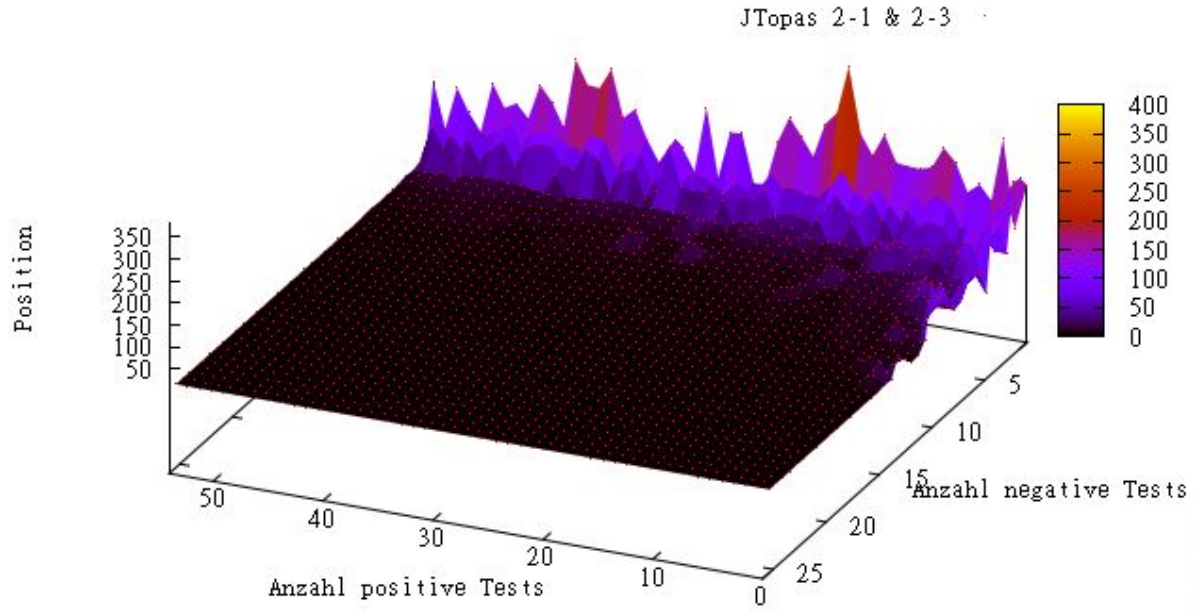

Abbildung 4.13: Testfallsensitivität bei JTopas 2-1 & 2-3

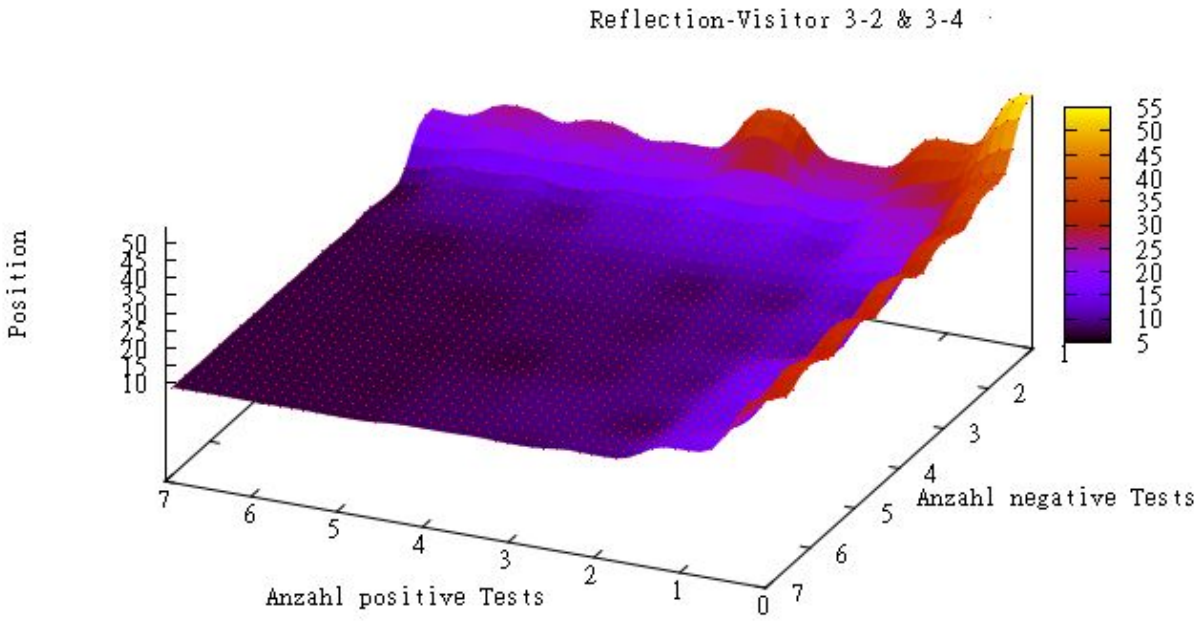

Abbildung 4.14: Testfallsensitivität bei Reflection-Visitor 3-2 & 3-4

Kapitel 5

Fazit

Die Verwendung von automatischen Fehlerlokalisierungsmethoden soll dabei helfen, das Auffinden von Fehlern in Softwareprojekten zu beschleunigen sowie die Wartung von Programmen zu vereinfachen, falls keine genaue Kenntnis des Sourcecodes vorhanden ist. Dabei wird grundsätzlich zwischen modellbasierten und statistischen Ansätzen unterschieden.

Modellbasierte Methoden beruhen hierbei auf der Erstellung eines Modells, das das gewünschte Verhalten eines Programms definiert. Dieses dient zur Überprüfung der vom Programm gelieferten Resultate sowie zur Identifizierung von Komponenten, die im Falle von Abweichungen als mögliche Fehlerursachen in Betracht kommen.

Im Gegensatz dazu basieren statistische Ansätze auf dem Vergleich von positiven und negativen Programmtraces. Bei diesen wird versucht, anhand des Vorkommens der Komponenten innerhalb dieser Traces zu bestimmen, welche Komponente die unerwünschten Abweichungen der Ergebnisse verursacht.

Eine statistische Methode zur Fehlerlokalisierung ist die in dieser Arbeit untersuchte Spectrum Based Diagnosis. Bei dieser wird mit Hilfe von positiven und negativen Tests ein Spektrum aufgebaut, das eine Zuordnung der Komponenten zu den Tests, in die diese involviert sind, darstellt. Damit können im Anschluss durch Verwendung eines Similarity-Koeffizienten Werte berechnet werden, die die Verdächtigkeit einer Komponente, inkorrekt zu sein, angibt.

Das Ziel dieser Arbeit war die Entwicklung eines Tools zur Durchführung der Spectrum Based Diagnosis auf Javaprogrammen sowie die Analyse der Leistungsfähigkeit der Technik anhand unterschiedlicher Testprogramme. Zudem sollte der zusätzliche Ressourcenbedarf durch die Anwendung des Tools erhoben werden.

Die Anforderung an das Tool war hierbei, ein Spektrum auf Basis von Statements aufzubauen und die Ergebnisse der Analyse des Spektrums entsprechend der Verdächtigkeiten der Statements sortiert auszugeben. Zusätzlich sollte die Möglichkeit bestehen, die Def-

und Ref-Mengen der ausgeführten Statements zu erheben, um diese zur Erstellung von Slices einzusetzen.

Die Ausführung des Tools besteht aus mehreren Phasen, wobei eine Analyse der Laufzeit zeigt, dass der größte zusätzliche Zeitbedarf durch Instrumentierung, Recompilierung sowie in manchen Fällen die durch die Instrumentierung verursachte Erhöhung der Testlaufzeiten entsteht. Diese Erhöhung fällt stärker aus, wenn Statements und somit auch die für den Spektrumaufbau zuständigen Instruktionen während der Ausführung eines Testfalles mehrmals ausgeführt werden.

Der Grund hierfür ist, dass jede Statementausführung eine entsprechende Methode aufruft, die die Informationen über dieses Statement in das Spektrum einfügt. Diese wird auch dann aufgerufen, wenn das Statement bereits im Spektrum vorhanden ist und somit eine erneute Eintragung nicht nötig ist. Um diese negativen Auswirkungen auf die Laufzeit zu vermindern, wurde eine Methode vorgestellt, bei der eine zusätzliche Instruktion überprüft, ob das Statement bereits im Spektrum aufscheint. Diese Vorgangsweise bietet den Vorteil, dass die Überprüfung des Vorkommens im Spektrum im Vergleich zur eigentlichen Einfügeoperation mit weniger Übergabeparametern durchgeführt werden kann und somit einen kürzeren Zeitbedarf mit sich bringt.

Neben der Laufzeit wurde ebenso der Speicherbedarf, den die Ausführung des Tools mit sich bringt, untersucht. Dabei ist ersichtlich, dass der maximale Speicherbedarf bei der Recompilierung der instrumentierten Sourcefiles auftritt. Dieser ist hierbei abhängig von der Anzahl der zu kompilierenden Files sowie der vom Testprogramm zusätzlich benötigten JAR-Files.

Um die Leistungsfähigkeit der Spectrum Based Diagnosis Technik zu untersuchen, standen drei Testprogramme zur Verfügung, die sich jeweils in der Anzahl der Sourcefiles sowie der verfügbaren Testfälle unterschieden. In diese wurden fehlerhafte Statements eingefügt, wodurch eine bestimmte Anzahl der Testfälle negativ wurde. Anschließend wurden unterschiedliche Varianten der Spectrum Based Diagnosis durchgeführt und deren Fähigkeit, den Fehler zu lokalisieren, überprüft.

Im ersten Schritt wurden Programme verglichen, die jeweils ein inkorrektes Statement enthalten. Ein Vergleich der Similarity-Koeffizienten Ochiai, Jaccard und Tarantula zeigt hier, dass die Wahl eines bestimmten Koeffizienten einen relativ starken Einfluss auf die Genauigkeit der Fehlerlokalisierung haben kann. Zudem konnte die in mehreren Arbeiten aufgestellte Vermutung bestätigt werden, dass der Einsatz des Ochiai-Koeffizienten die besten Ergebnisse liefert. Dieser lieferte für jedes der verfügbaren Testprogramme bessere oder zumindest gleiche Ergebnisse wie Jaccard und Tarantula.

Neben der Wahl des Similarity-Koeffizienten wird die Fehlerauffindung auch von der Art des Spektrums beeinflusst. Dabei ist ersichtlich, dass Hit-Spektren den ebenfalls möglichen Count-Spektren im Allgemeinen überlegen sind. Der Grund hierfür liegt in der starken Beeinflussung der Verdächtigkeit, den das Count-Spektrum auf Statements, die während

der Ausführung eines Testfalles mehrfach aufgerufen werden, ausübt. Diese steht jedoch meist in keinem Verhältnis zum tatsächlichen Einfluss, den das Statement auf das Ergebnis des Tests hat, wodurch sich eine Verfälschung des Verdächtigkeitswertes der betroffenen Instruktionen ergibt.

Im Gegensatz dazu konnte durch die Ausgabe des Traces des kleinsten negativen Testfalles eine Steigerung der Performance erreicht werden. Zudem bietet diese Variante den Vorteil, dass sich im Falle von Programmen, die genau eine fehlerhafte Instruktion enthalten, keine Verschlechterung der Lokalisierungsgenauigkeit ergeben kann.

Neben den unterschiedlichen Varianten, mit denen die Spectrum Based Diagnosis durchgeführt werden kann, wurde auch der Einfluss der positiven und negativen Testfälle auf die Qualität der Fehlersuche überprüft. Hierzu wurde die Menge der verwendeten Tests schrittweise reduziert, indem eine bestimmte Anzahl an zufällig gewählten positiven und negativen Testfällen entfernt wurde. Das Resultat dieser Untersuchung zeigt, dass eine Variation der positiven Tests kaum Einfluss auf das Ergebnis der Diagnose hat, während eine Verminderung der verfügbaren negativen Testfälle zu ungenaueren Resultaten führt. Ebenso führt eine Erhöhung der Anzahl der negativen Tests ab einem bestimmten Grenzwert zu keiner weiteren Verbesserung der Fehlerlokalisierung.

Neben den Testprogrammen mit einem Fehler wurde auch die Auswirkung des Vorkommens von zwei Fehlern untersucht. Dadurch ergeben sich im Gegensatz zu den bisher untersuchten Programmen Situationen, in denen ein fehlerhaftes Statement in einigen fehlschlagenden Testfällen nicht auftritt, wodurch dessen Verdächtigkeitswert reduziert wird.

Bei den untersuchten Programmen handelt es sich wiederum um die drei verfügbaren Testprogramme, in die jeweils zwei fehlerhafte Statements eingefügt wurden. Dabei wurden dieselben Statements wie bei der Analyse mit einem Fehler verwendet. Dadurch wird eine direkte Vergleichbarkeit zu den Ergebnissen mit diesen Programmen ermöglicht.

Eine erste Durchführung der Diagnose mit Hilfe des Ochiai-Koeffizienten zeigt, dass Spectrum Based Diagnosis grundsätzlich auch bei derartigen Programmen dazu in der Lage ist, zumindest eines der fehlerhaften Statements zu finden. Allerdings treten im Vergleich zu den Ergebnissen mit einem Fehler in manchen Fällen tatsächlich Einbußen bei der Genauigkeit der Fehlerlokalisierung auf.

Weiters wurden auch hier die Koeffizienten Ochiai, Jaccard und Tarantula miteinander verglichen. Wiederum wurde mit Ochiai die allgemein beste Performance erzielt, allerdings konnten sowohl Jaccard als auch Tarantula diese in bestimmten Fällen übertreffen.

Ebenso von Interesse ist der Effekt, den zwei Fehler auf die Variante mit Ausgabe des minimalen Fehlertraces haben. Während bei dieser bei der Untersuchung von Programmen mit jeweils nur einem Fehler garantiert werden kann, dass sich die Ausgabeposition des betroffenen Statements nicht verschlechtert, ist dies hier nicht der Fall. Der Grund für

diesen Umstand ist die Tatsache, dass durch die Existenz von mehreren Fehlern nicht eindeutig feststeht, welche davon im kleinsten negativen Testfall tatsächlich auftreten. Somit kann durch diese Ausgabevariante ein möglicherweise besser gereihter Fehler entfallen, wodurch die angesprochene Verschlechterung der Fehlerlokalisierung auftreten kann.

Dieser Effekt trat bei den untersuchten Programmen insgesamt nur einmal auf, wobei die tatsächliche Verschlechterung der Ausgabeposition jedoch relativ gering ausfällt. Im Gegenzug konnte bei anderen Fällen eine deutliche Steigerung der Performance ausgemacht werden. Somit zeigt sich, dass diese Ausgabevariante auch hier einen positiven Einfluss auf die Ergebnisse haben kann.

Als letzter Teil der Technikanalyse wurde wiederum eine Variation der Testumgebung durchgeführt, bei der die Anzahl der positiven und negativen Tests schrittweise reduziert wurde. Dabei zeigt sich, dass die Auswirkungen des Wegfallens von positiven Testfällen stärker zu sein scheinen als dies bei Programmen mit einem Fehler der Fall ist. So wurden bei zwei der drei Testprogramme schlechtere Ergebnisse erzielt, wenn die Anzahl der verwendeten positiven Tests in der Testumgebung auf ein Minimum reduziert wurde.

Literaturverzeichnis

[1] M. Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[2] Rseaux Et Systmes, Gérard Le Lann, Grard Le Lann, Thme Rseaux Et Systmes, and Projet Reflecs. The ariane 5 flight 501 failure - a case study in system engineering for computing systems, 1996.

[3] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[4] G.T. Leavens and Y. Cheon. Design by Contract with JML. *Draft, available from jmlspecs. org*, 1:4, 2005.

[5] M. Olan. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19(2):319–328, 2003.

[6] K. Beck and E. Gamma. JUnit Cookbook. http://junit.sourceforge.net/doc/cookbook/cookbook.htm, 05.10.2010.

[7] R. Abreu, AJC Van Gemund, and D. TU. Spectrum-based fault localization in embedded software.

[8] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[9] H. Agrawal and J.R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990.

[10] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[11] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Volume 00*, pages 128–137. IEEE Computer Society, 2008.

[12] R. Abreu, W. Mayer, M. Stumptner, and A.J.C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 409–414. ACM, 2009.

[13] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. *Electronic Notes in Theoretical Computer Science*, 89(3):378–404, 2003.

[14] M.J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90. ACM, 1998.

[15] L. Naish, HJ Lee, and K. Ramamohanarao. A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology*.

[16] J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, page 282. ACM, 2005.

[17] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. 2002.

[18] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. Vida: Visual interactive debugging. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 583–586, Washington, DC, USA, 2009. IEEE Computer Society.