Statische und dynamische Analyse der Bedingungsüberdeckung objektorientierter Java-Programme

Studienarbeit im Fach Informatik

vorgelegt von

Dominik Schindler

geb. 30.06.1980 in Oberviechtach

angefertigt am

Institut für Informatik
Lehrstuhl für Software Engineering (Informatik 11)
Friedrich-Alexander-Universität Erlangen-Nürnberg
(Prof. Dr. Francesca Saglietti)

Prüfer: Prof. Dr. Francesca Saglietti

Beginn der Arbeit: 08.08.2005
Abgabe der Arbeit: 08.05.2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den (Datum)

Dominik Schindler

Zusammenfassung

Moderne Software ist heutzutage sehr komplex. Diese Komplexität ist auch der Grund dafür, dass immer wieder Fehler in größeren Softwareprojekten auftreten bzw. Projekte vollständig scheitern. Ein aktuelles Beispiel ist die ALG-II-Software der Bundesagentur für Arbeit: Anfang 2005 mussten Hunderttausende Arbeitslosengeld-II Empfänger auf ihr Geld warten da Kontonummern, die weniger als 10 Stellen hatten, auf der falschen Seite (rechtsbündig statt linksbündig) mit Nullen aufgefüllt wurden. Alleine bei der Postbank waren 200.000 Empfänger betroffen[2].

In fast allen Software-Lebenszyklusmodellen existiert eine Test-Phase, in der versucht wird möglichst alle derartige Fehler zu finden. Diese Phase stellt eine Art „Qualitätssicherung“ der Software dar. Die in dieser Phase angewandten Prüftechniken können gemäß Liggesmeyer [6] allgemein in *statische* und *dynamische* Testverfahren klassifiziert werden. Statische Testverfahren bestimmen durch statische Analyse die Korrektheit des Programms, ohne es ausführen zu müssen. Ein Beispiel ist das Model-Checking Verfahren, das anhand eines Software-Modells versucht, die Korrektheit gegenüber der Spezifikation zu beweisen. Dieses Verfahren liefert bei einer Verletzung der Spezifikation gleich ein Gegenbeispiel für den Nachweis des Fehlers.

Im Gegensatz dazu wird bei den dynamischen Testverfahren das Programm mit korrekten Eingaben ausgeführt. Da die Eingaben als korrekt vorausgesetzt werden können vorhandene Fehler durch fehlerhafte Ausgaben erkannt werden. Auch ein fehlerhaftes Verhalten des Programms weist auf einen Fehler in der Software hin. Die dynamischen Verfahren verwenden dazu den Kontroll- bzw. Datenfluss des Programms, der zur Laufzeit überwacht wird. Ein Vertreter kontrollflussorientierter Tests ist der Bedigungsüberdeckungstest, der zum adäquaten Testen komplexer Bedingungen verwendet wird. Die verschiedenen Ausprägungen des Bedingungsüberdeckungstests, deren Eigenschaften sowie eine mögliche Implementierung für die Sprache Java ist Gegenstand dieser Arbeit.

Die vorliegende Arbeit gliedert sich wie folgt: In Kapitel 2 werden die fünf Bedingungsüberdeckungskriterien anhand einer Beispielbedingung näher beschrieben und der Überdeckungsgrad definiert, der die Vollständigkeit eines Tests bezogen auf ein bestimmtes Testkriterium angibt. In Kaptitel 3 werden Konzepte vorgestellt, wie die klassischen Bedingungsüberdeckungskriterien auf die Sprache Java übertragen werden können und was dabei speziell beachtet werden muss. Der Hauptteil bildet Kapitel 4, in dem das implementierte Werkzeug näher vorgestellt wird. Das Kapitel 5 schließt durch eine kurze Zusammenfassung und einen Ausblick auf mögliche Verbesserungen und Erweiterungen des Werkzeugs die Arbeit ab.

Abstract

Modern software-systems are nowadays very complex. This complexity is the reason why failures in big software projects often appear, or why such projects completely fail. A contemporary example is the ALG-II-software of the Bundesagentur für Arbeit: At the beginning of 2005, hundred thousands unemployment benefit recipients had to wait for the money because the account numbers with less then 10 digits were zeroized wrong (right-aligned than left-aligned). At the Postbank were 200.000 recipients affected[2].

In most software lifecycles exists a test phase, which tries to find all of those failures. This phase is a kind of „quality assurance“ for the software. The test techniques applied in this phase can be classified according to [6] into *static* and *dynamic* test techniques. Static test techniques determine the correctness of the program by analyzing the source code. An example for such a technique is model checking, which tries to prove the correctness of a software model against the specification. In case of an error, this technique also presents a counter example, which proves the failure.

By way of contrast, the dynamic test techniques run the program with correct input. Because of the correct input, existing failures can be determined by faulty output, or by faulty program behavior. Dynamic test techniques make use of the control- or dataflow, which they observe at runtime. An example for a controlflow based test technique is the condition coverage test, which is used to adequate test complex conditions. The different types of condition coverage tests, the properties and a possible implementation for the Java language are subject of this work.

This paper is divided into five chapters: Chapter 2 introduces the five condition coverage tests and the coverage metrics, which describes the completeness of a test relative to a criteria. The Chapter 3 shows the concepts, how the classic condition coverage tests are transferred to Java. The main part is chapter 4, where the implemented tool is described. Chapter 5 concludes the paper with a short summary and an outlook with possible improvements and extensions of the tool.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 2

1.2 Aufgabenstellung 2

2 Grundlagen **4**

2.1 Einfache Bedingungsüberdeckung 5

2.2 Bedingungs-/Entscheidungsüberdeckung 7

2.3 Minimale Mehrfachbedingungsüberdeckung 7

2.4 Modifizierte Bedingungs-/Entscheidungsüberdeckung 8

2.5 Mehrfachbedingungsüberdeckung 9

2.6 Definition der Bedingungsüberdeckungsmetriken 9

3 Übertragung der Kriterien auf Java **12**

3.1 Explizite Bedingungen 12

3.2 Ternäre Operator (?:) 13

3.3 Switch-Case-Anweisung 14

3.4 Ausnahme(Exception)-Behandlung 16

3.5 Polymorphie und dynamisches Binden 17

3.6 Overloading - Überladen von Methoden 19

4 Das Werkzeug **21**

4.1 Ansätze für die Instrumentierung 21

4.2 Allgemeine Funktionsweise 22

4.3 Der Parser 23

4.4 Der Instrumentierer 24

4.5 Der Schreiber 26

4.6 Der Logger 29
4.7 Der Analysierer 32

5 Ausblick **35**
5.1 Statische Überprüfung der Erfüllbarkeit 35
5.2 Weitere mögliche Erweiterungen 36
5.3 Mögliche Optimierungen und Anpassungen 37

A CountWords.java Beispielquelltext **38**

B Wichtige Klassen **39**
B.1 LoggerTypes.java Klasse 39
B.2 Symbol-Klasse 40

C Die API **41**
C.1 Die Instrumenter-Klasse 41
C.2 Die Analyzer-Klasse 42

Literaturverzeichnis **44**

Index **45**

Abbildungsverzeichnis

4.1 Allgemeine Funktionsweise des Werkzeugs 23
4.2 Beispiel-AST (Auszug) einer for()-Schleife aus dem Quelltext `CountWord.java`, Zeile 15 24
4.3 Instrumentierter Beispiel-AST für die markierte Teilbedingung 25
4.4 Mögliche Arten/Typen eines Eintrags in der Symboltabelle 27
4.5 Symboltabelle für den Quelltext `CountWords.java` 28
4.6 Besondere Ereignisse 30
4.7 Auszug aus einer Logdatei 30

B.1 `LoggerTypes.java`-Klasse 39

Tabellenverzeichnis

2.1 Wahrheitstabelle für die Beispielbedingung AB||CD (vollständige Evaluation) . . 6
2.2 Wahrheitstabelle für die Beispielbedingung AB||CD (unvollständige Evaluation) . 6

Kapitel 1

Einleitung

Moderne Software ist heutzutage sehr komplex. Diese Komplexität ist auch der Grund dafür, dass immer wieder Fehler in größeren Softwareprojekten auftreten bzw. Projekte vollständig scheitern. Ein aktuelles Beispiel ist die ALG-II-Software der Bundesagentur für Arbeit: Anfang 2005 mussten Hunderttausende Arbeitslosengeld-II Empfänger auf ihr Geld warten da Kontonummern, die weniger als 10 Stellen hatten, auf der falschen Seite (rechtsbündig statt linksbündig) mit Nullen aufgefüllt wurden. Alleine bei der Postbank waren 200.000 Empfänger betroffen[2].

Gemäß der Definition von Balzert ist ein Software-Fehler eine „Abweichung der tatsächlichen Ausprägung eines Qualitätsmerkmals von der vorgesehenen Soll-Ausprägung, jede Inkonsistenz zwischen der Spezifikation und der Implementierung und jedes strukturelle Merkmal des Programmablaufs, das ein fehlerhaftes Verhalten des Programms verursacht“[1]. Um solche vorhandenen Fehler erkennen und beheben zu können existieren unterschiedliche Testverfahren, die allgemein in *statische* und *dynamische* Testverfahren klassifiziert werden können. Bei den statischen Testverfahren wird das zu testende Programm nicht ausgeführt sondern nur statisch analysiert. Vertreter dieser Testverfahren sind das Model-Checking, die Inspektion, der Review und das Walk-through.

Im Gegensatz dazu wird bei den dynamischen Tests das zu testende Programm mit korrekten Eingaben ausgeführt. Dadurch können anhand fehlerhafter Ausgaben oder durch ein fehlerhaftes Verhalten vorhandene Fehler erkannt werden. Dazu verwenden diese Verfahren den Kontroll- oder Datenfluss des Programms. Grundsätzlich existieren zwei unterschiedliche Ausprägungen dynamischer Testverfahren: der Black-Box-Test und der White-Box-Test. Der Grey-Box-Test ist eine Mischung aus beiden.

Der Black-Box-Test vergleicht die funktionalen Eigenschaften des Programms mit den in der Spezifikation festgelegten Eigenschaften. Da dieser Test nur die Funktion der Software testet, wird dieser als funktionsorientiert bezeichnet.

Der White-Box-Test verwendet zum Testen der Software den Kontroll- oder Datenfluss des Programms. Ein bekannter Vertreter kontrollflussorientierter Tests und von [6] als minimal gefordertes Testverfahren ist der Zweigüberdeckungstest, der die Ausführung aller Zweige im Kontrollflussgraphen fordert. Dieser Test kann aber unter Umständen nicht ausreichend sein, zum

Beispiel beim Test komplexer Bedingungen. Um diese Bedingungen trotzdem akkurat zu testen wurden verschiedene Bedingungsüberdeckungstests entwickelt, die angemessener für den Test komplexere Bedingungen sind.

1.1 Motivation

Der Zweigüberdeckungstest gilt zwar als Minimalkriterium der kontrollflussorientierten Software-Tests[6], trotzdem bedeutet eine Zweigüberdeckungsrate von 100% nicht notwendigerweise, dass die Software fehlerfrei ist. Ein Grund dafür ist der mangelhafte Test komplexer Bedingungen mit mehreren Schachtelungsebenen. Betrachtet man zum Beispiel den Zweigüberdeckungstest bei der einfachen Anweisung

$$\text{if } (x > 10) \ldots$$

kann die Bedingung gemäß Liggesmeyer [6] als ausreichend getestet betrachtet werden, wenn gegen die beiden Wahrheitswerte Wahr und Falsch getestet wurde. Im Gegensatz dazu ist nur die Zweigüberdeckung der komplexen Bedingung

$$\text{if } ((a < 10) \,\&\&\, (b > 0) \,\|\, (c > 10) \,\&\&\, (d == 0)) \ldots$$

nicht mehr ausreichend, wie die folgenden beiden Testfälle verdeutlichen:

$$a = 5, b = 0, c = 7, d = 1 \rightarrow \text{Falsch}$$
$$a = 5, b = 1, c = 11, d = 1 \rightarrow \text{Wahr}$$

Diese beiden Testfälle erreichen bereits vollständige Zweigüberdeckung der komplexen Bedingung, sie verdeutlichen aber auch das größte Problem der Zweigüberdeckung: Bei unvollständiger Evaluation komplexer Ausdrücke (siehe Kapitel 2) wird die letzte Bedingung (d == 0) von beiden Testfällen überhaupt nicht getestet, so dass ein eventuell vorhandener Fehler maskiert wird. Der Zweigüberdeckungstest beachtet also die Struktur komplexer Bedingungen nur unzureichend.

Beim Zweigüberdeckungstest und der Evaluation von Ausdrücken von links nach rechts gilt grundsätzlich, dass eine Bedingung desto schlechter getestet wird, je weiter rechts sie im Ausdruck steht[6]. Um diese Bedingungen trotzdem akkurat zu testen existieren weitere Testverfahren. In Kapitel 2.1 werden unterschiedliche Verfahren erläutert, die besser für den Test komplexer Bedingungen geeignet sind als der Zweigüberdeckungstest, da diese die Struktur besser berücksichtigen. Dadurch können Fehler gefunden werden, die der Zweigüberdeckungstest nicht beachtet hätte.

1.2 Aufgabenstellung

Das Ziel dieser Arbeit ist die Messung der Bedingungsüberdeckung objekt-orientierter Java-Programme. Dazu sollen bereits existierende Ansätze zur Übertragung der klassischen Bedin-

gungsüberdeckungskriterien auf objekt-orientierte Software vergleichend bewertet und mit eigenen Konzepten geeignet ergänzt werden (vgl. Kapitel 3). Diese Konzepte sollen anschließend in ein Werkzeug implementiert werden, das die Programme bezüglich der bereits ermittelten Kriterien statisch analysiert, sowie, darauf aufbauend, die während der Testausführung erzielten Bedingungsüberdeckungen ermittelt. Das für diese Anforderungen entwickelte Werkzeug wird in Kapitel 4 detailliert vorgestellt.

Kapitel 2

Grundlagen

Es existieren fünf unterschiedliche Ausprägungen des Bedingungsüberdeckungstests, von denen die schwächste die einfache Bedingungsüberdeckung ist. Diese Bedingungsüberdeckung testet nur die atomaren Bedingungen, so dass dieser Test als nicht ausreichend betrachtet werden kann. Die Bedingungs-/Entscheidungsüberdeckung stellt eine Erweiterung der einfachen Bedingungsüberdeckung dar, die zusätzlichen alle Zweige testet und die Zweigüberdeckung mit einschließt. Eine Abwandlung der Bedingungs-/Entscheidungsüberdeckung ist die modifizierte Bedingungs-/Entscheidungsüberdeckung, bei der gezeigt werden soll, dass die atomaren Bedingungen einen Einfluss auf das Ergebnis der Entscheidung haben.

Eine weitere Ausprägung ist die minimale Mehrfachbedingungsüberdeckung, die eine abgeschwächte Version der Mehrfachbedingungsüberdeckung ist. Im Gegensatz zur Mehrfachbedingungsüberdeckung, die den Test aller Wahrheitswertekombinationen der atomaren Bedingungen fordert, ist diese Bedingungsüberdeckung nicht so aufwändig, da weniger als 2^n Testfälle notwendig sind.

Definition: atomare (Teil-)Bedingung Eine atomare (Teil-)Bedingung ist eine Bedingung, die nicht weiter in (Teil-)Bedingungen zerlegt werden kann. Solche Bedingungen enthalten keine bool'schen Operatoren (Und, Oder, Nicht).

Zum Beispiel ist die Bedingung

$$(a < 10)\ \&\&\ (b > 0)\ \|\ (c > 10)\ \&\&\ (d == 0)$$

aus Kapitel 1.1 laut Definition keine atomare Bedingung, die Teilbedingung (a < 10) ist aber eine atomare Bedingung.

Definition: unvollständige Evaluation (engl. „lazy evaluation", in Java „short circuit evaluation") Unter unvollständiger Evaluation versteht man das vorzeitige Beenden der Auswertung, wenn das Endergebnis bereits feststeht. Im Gegensatz dazu wird bei vollständiger Evaluation immer der ganze Ausdrucke ausgewertet.

Zum Beispiel erfolgt die unvollständige Evaluation der Bedingung

$$(a < 10)\ \&\&\ (b > 0)\ ||\ (c > 10)\ \&\&\ (d == 0)$$

mit den beiden erwähnten Testfällen aus Kapitel 1.1 wie folgt: Wird die Bedingung mit dem Testfall 1 ausgeführt wird die Auswertung nach der atomaren Bedingung (c > 10) mit dem Endergebnis Falsch abgebrochen. Mit dem Testfall 2 wird bereits nach der Auswertung der zusammengesetzten Bedingung (a < b) && (b > 0) mit dem Endergebnis Wahr vorzeitig beendet, da die nachfolgenden Bedingungen keinen Einfluss mehr auf das Endergebnis haben.

Anmerkung 1 Bei den folgenden Wahrheitstabellen werden aus Platzgründen für die einzelnen Bedingungen folgende Abkürzungen verwendet:

A = (a < 10), B = (b > 0), C = (c > 10), D = (d == 0)

für die atomaren Bedingungen,

AB = (a < 10) && (b > 0)
CD = (c > 10) && (d == 0)
AB||CD = ((a < 10) && (b > 0) || (c > 10) && (d == 0))

für die zusammengesetzten Bedingungen und „F“ für Falsch bzw. „W“ für Wahr.

Anmerkung 2 Die in den folgenden Abschnitten erwähnten Testfälle dienen nur zur Demonstration der Kriterien. Selbstverständlich sind auch andere Testfallkombinationen möglich.

2.1 Einfache Bedingungsüberdeckung

Die einfache Bedingungsüberdeckung fordert, dass alle atomaren Bedingungen mindestens einmal gegen die beiden Wahrheitswerte Wahr und Falsch getestet werden. Die Tabelle 2.1 zeigt die Wahrheitstabelle für die bereits erwähnte Bedingung bei vollständiger Evaluation. Die beiden Testfälle 6 und 11 in dieser Tabelle erreichen bereits vollständige einfache Bedingungsüberdeckung. In dieser Tabelle ist auch zu erkennen, dass die Auswertung der Entscheidung bei den Testfällen 6 und 11 immer zu Falsch erfolgt, also nur der Falsch-Zweig überdeckt wird. Es gilt, dass bei vollständiger Evaluation der Bedingungen die einfache Bedingungsüberdeckung nicht die Zweigüberdeckung mit einschließt. Deshalb ist sie als alleiniges Kriterium ungeeignet[1].

Bei unvollständiger Evaluation der Bedingungen schließt die einfache Bedingungsüberdeckung die Zweigüberdeckung mit ein. Die unvollständige Evaluation führt zur Bildung von Testfallklassen, die mehrere Testfälle zusammenfassen können. Tabelle 2.2 zeigt die Wahrheitstabelle

	A	B	C	D	AB	CD	AB‖CD
1	f	f	f	f	f	f	f
2	f	f	f	w	f	f	f
3	f	f	w	f	f	f	f
4	f	f	w	w	f	w	t
5	f	w	f	f	f	f	f
6	f	w	f	w	f	f	f
7	f	w	w	f	f	f	f
8	f	w	w	w	f	w	w
9	w	f	f	f	f	f	f
10	w	f	f	w	f	f	f
11	w	f	w	f	f	f	f
12	w	f	w	w	f	w	w
13	w	w	f	f	w	f	w
14	w	w	f	w	w	f	w
15	w	w	w	f	w	f	w
16	w	w	w	w	w	w	w

Tabelle 2.1: Wahrheitstabelle für die Beispielbedingung AB‖CD (vollständige Evaluation)

	Testfall	A	B	C	D	AB	CD	AB‖CD
I	1,2,5,6	f	-	f	-	f	f	f
II	3,7	f	-	w	f	f	f	f
III	4,8	f	-	w	w	f	w	w
IV	9,10	w	f	f	-	f	f	f
V	11	w	f	w	f	f	f	f
VI	12	w	f	w	w	f	w	w
VII	13,14	w	w	f	-	w	-	w
VIII	15	w	w	w	f	w	-	w
IX	16	w	w	w	w	w	-	w

Tabelle 2.2: Wahrheitstabelle für die Beispielbedingung AB‖CD (unvollständige Evaluation)

bei unvollständiger Evaluation und wie die Testfallklassen aus den Testfällen gebildet werden. Die beiden Testfälle 6 und 11 aus der Testfallklasse I bzw. V, die bei vollständiger Evaluation einfache Bedingungsüberdeckung erreicht haben, sind bei unvollständiger Evaluation nicht mehr ausreichend, da B und D nur gegen Falsch getestet wurden. Deshalb muss bei diesem Beispiel eine weitere Testfallklasse hinzugefügt werden, z.B. Testfallklasse IX, so dass die ausgewählten Testfallklassen I, V und XI einfache Bedingungsüberdeckung bei unvollständiger Evaluation erreichen.

2.2 Bedingungs-/Entscheidungsüberdeckung

Die Bedingungs-/Entscheidungsüberdeckung fordert zusätzlich zur einfachen Bedingungsüberdeckung, dass ebenfalls alle Zweige überdeckt werden. Zum Beispiel erreichen die beiden Testfälle 5 und 12 in Tabelle 2.1 bereits vollständige Bedingungs-/Entscheidungsüberdeckung. Diese beiden Testfälle sind auch ein Beispiel dafür, dass die Bedingungs-/Entscheidungsüberdeckung nur die atomaren Bedingungen und die Entscheidung prüft, aber größtenteils die logische Struktur komplexer Bedingungen ignoriert. So wurde die zusammengesetzte Bedingung AB beide Male nur gegen Falsch getestet. Da bei unvollständiger Evaluation die einfache Bedingungsüberdeckung die Zweigüberdeckung subsumiert, ist dieser Test nur bei vollständiger Evaluation der Bedingungen sinnvoll (vgl. [1]).

2.3 Minimale Mehrfachbedingungsüberdeckung

Die minimale Mehrfachbedingungsüberdeckung fordert, dass alle atomaren und nicht atomaren Bedingungen gegen Wahr und Falsch geprüft werden. Dadurch wird die Struktur komplexer Bedingungen besser beachtet als von den bereits vorgestellten Überdeckungskriterien, da alle Schachtelungsebenen einer komplizierten Bedingung gleichermaßen beachtet werden[6].

Die beiden Testfälle 1 und 16 der Wahrheitstabelle 2.1 zeigen exemplarisch, wie die minimale Mehrfachbedingungsüberdeckung bei vollständiger Evaluation erreicht werden kann, da alle (Teil-)Bedingungen und die Entscheidung einmal gegen Wahr und Falsch getestet wurden. Trotzdem wurden diese beiden Testfälle schlecht gewählt, da sie die logische Struktur der Bedingungen nicht sinnvoll prüfen. So können beide Testfälle unter gewissen Umständen eine fehlerhafte Bedingung nicht erkennen. Wurde zum Beispiel statt der Bedingung AB||CD die falsche Bedingung A||BC||D geschrieben, hätten diesen Fehler keiner der beiden Testfälle erkannt, da die Ergebnisse bei beiden Testfällen gleich sind.

Bei unvollständiger Evaluation sind die beiden Testfälle 1 und 16, respektive Testfallklasse I und IX, nicht mehr ausreichend, so dass weitere Testfälle benötigt werden. Wählt man zum Beispiel aus der Wahrheitstabelle 2.2 zusätzlich die Testfallklassen V und VI aus, wird die minimale Mehrfachbedingungsüberdeckung bei unvollständiger Evaluation erreicht.

Bei unvollständiger Evaluation kann im Gegensatz zur vollständigen Evaluation die fehler-

hafte Bedingung A||BC||D erkannt werden, da bei der Auswertung die erhaltenen Ergebnisse von den korrekten Ergebnissen abweichen. Zum Beispiel wird die fehlerhafte Bedingung bei Testfall 11 zu Wahr ausgewertet, obwohl gemäß der Wahrheitstabelle 2.2 der Wahrheitswert Falsch richtig gewesen wäre.

Das Überdecken aller atomaren und nicht atomaren Bedingungen ist zwar sinnvoll, es kann aber unter Umständen schwierig bis unmöglich sein für bestimmte Bedingungen Testfälle zu generieren. Zum Beispiel kann die Bedingung (x > 10) && (x < 5) niemals Wahr werden, da die Wertebereiche disjunkt sind und deshalb x nicht gleichzeitig in beide Wertebereiche fallen kann. Allgemein bezeichnet man eine (Teil)-Bedingung, für die keine Testfälle gefunden werden können um sie gegen Wahr und Falsch zu prüfen, als invariant. Invariante Bedingungen können entfernt werden und weisen auf einen Software-Fehler hin[6].

2.4 Modifizierte Bedingungs-/Entscheidungsüberdeckung

Die modifizierte Bedingungs-/Entscheidungsüberdeckung fordert Testfälle die zeigen, dass jede atomare Bedingung einen Einfluss auf das Endergebnis der Entscheidung hat – unabhängig von den anderen Bedingungen. Dieser Test stellt einen Kompromiss zwischen Testeffizienz und Testaufwand dar, da der Zusammenhang zwischen der Anzahl der atomaren Bedingungen und der Anzahl der erforderlichen Testfälle linear ist. So werden für den Test einer Bedingung mit n atomaren Bedingungen mindestens $n+1$ Testfälle benötigt. „Grundsätzlich zielt diese Technik auf einen möglichst umfassenden Test der Logik von zusammengesetzten Entscheidungen mit einem vertretbaren Testaufwand.“[6].

Die Testfälle 2 und 4 aus der Wahrheitstabelle 2.1 erzielen das Kriterium für die atomare Bedingung C, da die anderen atomaren Bedingungen A, B und D bei beiden Testfällen fest sind (Falsch, Falsch, Wahr), und nur C das Endergebnis beeinflusst. Die Testfälle 5 und 13 erfüllen ebenfalls das Kriterium für die atomare Bedingung A, da die atomaren Bedingungen B, C und D unverändert bleiben (Wahr, Falsch, Falsch), so dass das Endergebnis nur von A abhängt.

Problematisch ist dieses Kriterium bei gekoppelten Bedingungen, weil bestimmte Wahrheitswertkombinationen eventuell nicht generiert werden können. Ein Beispiel einer Entscheidung mit gekoppelten atomaren Bedingungen ist (x == 0) || (x == 10) || (x == 100). Bei dieser Bedingung ist es unmöglich den Wahrheitswert der einzelnen Teilbedingungen unabhängig von den anderen Teilbedingungen zu verändern, da der Wahrheitswert aller atomaren Bedingungen von x abhängig ist. Eine Entscheidung bezeichnet man als schwach gekoppelt, wenn die Änderung einer Teilbedingung die anderen Teilbedingungen beeinflussen *kann*. Stark gekoppelte Bedingungen ändernd stets ihren Wahrheitswert, sobald der Wahrheitswert einer der Teilbedingungen verändert wird. Ein Beispiel einer stark gekoppelten Bedingung ist (X && !Y)|| (!X && Y). Bei dieser Bedingung ist kein Testfall erzeugbar, so dass X und $!X$ gleichzeitig Wahr oder Falsch sind.

Anmerkung: Dieser Test wird vom Standard RTCA DO-178B für kritische Software in der Luftfahrt gefordert[7].

2.5 Mehrfachbedingungsüberdeckung

Die Mehrfachbedingungsüberdeckung fordert, dass alle möglichen Wahrheitswertkombinationen der atomaren Bedingungen einer Entscheidung getestet werden. Dadurch ist sichergestellt, „dass unabhängig von der Verknüpfungslogik zusammengesetzter Bedingungen einer Entscheidung beide Wahrheitswerte berücksichtigt werden“[6]. Das Hauptproblem dieses Tests ist der immense Testaufwand, da für n atomare Bedingungen insgesamt 2^n Testfälle benötigt werden. Dieser Testaufwand ist in der Regel nicht akzeptabel und eine 100%tige Überdeckung ist beispielweise aufgrund invarianter Bedingungen nur recht schwer zu erzielen. Deshalb wurde eine schwächere Version dieser Bedingungsüberdeckung entwickelt, die bereits in Kapitel 2.3 beschriebene minimale Mehrfachbedingungsüberdeckung. Für die Mehrfachbedingungsüberdeckung müssen alle 16 (=2^4) Testfälle aus der Wahrheitstabelle 2.1 geprüft werden.

Bei unvollständiger Evaluation existieren nur die in Tabelle 2.2 gezeigten 9 Testfallklassen. Es können zwar alle 16 Testfälle hergestellt werden, aber nur die in der Wahrheitstabelle dargestellten 9 Testfallklassen können erkannt werden. Außerdem kann das bereits in Kapitel 2.4 beschriebene Problem bei gekoppelten Bedingungen auftreten, so dass bestimmte Testfälle nicht herstellbar sind.

2.6 Definition der Bedingungsüberdeckungsmetriken

Der Überdeckungsgrad gibt die Vollständigkeit eines Tests bezogen auf ein bestimmtes Testkriterium an [6]. Dieser ist beim strukturorientierten Testen definiert als der Quotient aus der Anzahl der überdeckten Elemente und die Anzahl der von einem Kriterium geforderten Elemente.

$$\text{Ueberdeckungsgrad} = \frac{\text{Anzahl der ueberdeckten Elemente}}{\text{Anzahl vom Kriterium geforderten Elemente}}$$

Ein Überdeckungsgrad von 0.5 bedeutet, dass die Hälfte aller geforderten Elemente durch die Testfallmenge ausgeführt wurde.

Der Überdeckungsgrad beim Bedingungsüberdeckungstest kann allgemein in *grobgranular* und *feingarnular* eingeteilt werden. Die geforderten Bedingungen werden durch das jeweilige Überdeckungskriterium aus Kapitel 2.1 bestimmt. Werden für die Ermittlung des Überdeckungsgrads alle Bedingungen des gesamten Programms und als Testfallmenge alle Testfälle herangezogen, wird der Überdeckungsgrad als grobgranular bezeichnet:

$$\text{UG}_{grob} = \frac{\text{Anzahl von allen Testfällen ueberdeckter Bedingungen}}{\text{Anzahl der von einem Kriterium geforderten Bedingungen des ganzen Programms}}$$

Wird stattdessen für jede Entscheidung e und für jeden Testfall t der Überdeckungsgrad angegeben, bezeichnet man diesen als feingranular:

$$\mathrm{UG}_{fein} = \frac{\text{Anzahl von t ueberdeckter und vom Kriterium geforderter Teilbedingungen von e}}{\text{Anzahl der vom Kritierium geforderten Teilbedingungen der Entscheidung e}}$$

Dazwischen können weitere Abstufungen, z.B. pro Methode, pro Klasse, usw., existieren.

Bei den beiden Überdeckungsmetriken für die modifizierte Bedingungs-/Entscheidungsüberdeckung und die Mehrfachbedingungsüberdeckung muss zusätzlich zwischen vollständiger und unvollständiger Evaluation unterschieden werden. Bei unvollständiger Evaluation der Bedingungen ist es sehr wahrscheinlich, dass gewisse atomare Bedingungen nicht ausgewertet werden (siehe Tabelle 2.2). Dadurch können für eine Entscheidung aus n atomaren Bedingungen weniger als 2^n Testfälle existieren.

$$\mathrm{UG}_{vollstaendig} = \frac{\text{Anzahl von einer Entscheidung überdeckter Testfälle}}{\text{Anzahl aller Testfälle einer Entscheidung}=2^n}$$

Um trotzdem einen aussagekräftigen Überdeckungsgrad angeben zu können, müssen für die nicht ausgewerteten atomaren Bedingungen Annahmen getroffen werden. In dieser Arbeit werden atomare Bedingungen, die nicht ausgewertet wurden, als Wahr und Falsch gleichzeitig betrachtet. Diese als „Wildcard“ bekannte Vorgehen stellt zwar eine Abschwächung der klassischen Definitionen der beiden Kriterien dar, es ist aber bei unvollständiger Evaluation sinnvoll. Würde man beispielsweise die 2^n Testfälle aus Tabelle 2.2 fordern, die Bedingungen aber unvollständig evaluieren, dann würden die Testfälle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 14, 15 und 16 nie überdeckt werden, da jeweils mindestens eine Bedingung nicht ausgewertet wurde.

$$\mathrm{UG}_{unvollstaendig} = \frac{\text{Anzahl von einer Entscheidung überdeckter Testklassen}}{\text{Anzahl aller Testklassen einer Entscheidung}}$$

Der Überdeckungsgrad für die Switch-Case-Anweisung ist wie folgt definiert:

$$UG_{Switch-Case} = \frac{\text{Anzahl ueberdeckter Faelle einer Switch-Case-Anweisung}}{\text{Anzahl vorhandener Faelle einer Switch-Case-Anweisung}}$$

Der Überdeckungsgrad für die Switch-Case-Answeisungen kann auch grobgranular erfolgen. Dazu werden alle überdeckten Fälle aller Switch-Case-Anweisungen eines Programms für die Berechnung verwendet. Ein Fall gilt als überdeckt, sofern er mindestens einmal ausgeführt wurde.

Der Überdeckungsgrad bei der Polymorphie ist wie folgt definiert:

$$UG_{polymorph} = \frac{\text{Anzahl ueberdeckter Klassen einer polymorphen Variable}}{\text{Anzahl möglicher Klassen einer polymorphen Variable}}$$

Eine Klasse gilt als überdeckt, sofern beim Aufruf einer Methode einer polymorphen Variable diese Variable ein Instanz der Klasse zugewiesen wurde (siehe 3.5). Die grobgranulare Definition wendet diese Definition auf alle polymorphen Variablen im gesamten Programm an.

Der Überdeckungsgrad für das Overloading (siehe Kapitel **??**) entspricht dem Überdeckungsgrad der Methodenüberdeckung („method coverage“) mit der Einschränkung, dass eine Berechnung pro Methode erfolgt. Die geforderten Methoden sind alle überladenen Versionen dieser Methode, inkl. der Methode selbst. Eine Methode gilt bereits als überdeckt, sofern sie mindestens einmal aufgerufen wurde.

$$UG_{Overloading} = \frac{\text{Anzahl ueberdeckter und geforderter Methoden}}{\text{Anzahl geforderter und überladener Methoden}}$$

Anmerkung Das in Kapitel 4 vorgestellte Werkzeug unterstützt die vier Metriken: grobgranular, feingranular, pro Methode/Konstruktor und pro Klasse. Weitere Überdeckungsmetriken können aber einfach hinzugefügt werden (siehe Kapitel 5: Ausblick).

Kapitel 3

Übertragung der Kriterien auf Java

In der Literatur werden die klassischen Bedingungsüberdeckungskriterien meistens nur auf explizite Bedingungen, wie beispielsweise bei den bedingten Anweisungen, angewendet. In Java können aber Bedingungen nicht nur explizit (wie beispielsweise bei der if()-Anweisung), sondern auch implizit auftreten. Eine implizite Bedingung stellt zum Beispiel die Auswahl eines Falls bei der Switch-Case-Anweisung dar. Da die klassischen Bedingungsüberdeckungskriterien nur auf explizite Bedingungen angewendet werden können, müssen für implizite Bedingungen explizite Bedingungen konstruiert werden. Davon ist in Java der ternäre Operator (?:), die Switch-Case-Anweisung sowie Polymorphie, dynamisches Binden und das Überladen von Methoden betroffen. Für diese Anweisungen bzw. Konzepte müssen explizite Bedingungen konstruiert werden. Weiterhin müssen für die Behandlung von Ausnahmen (in Java Exceptions genannt) Vorkehrungen getroffen werden, um hervorgerufene Ausnahmen in einem Ausdruck erkennen und behandeln zu können.

3.1 Explizite Bedingungen

Explizite Bedingungen sind Bedingungen, die mit den bedingten Anweisungen If(), For(), While() und Do() zwischen den beiden Klammern angeben werden. Diese werden deshalb als explizit betrachtet, da für sie keine Bedingung konstruiert werden muss und deshalb eine umgehende Analyse durch die Bedingungsüberdeckungskriterien möglich ist.

Für die Analyse expliziter Bedingungen zur Laufzeit werden diese durch die Methode LogB() ersetzt, die als Parameter die ersetzte Bedingung erwartet und diesen Wert wieder zurück liefert. Durch den Aufruf der Methode kann anschließend festgestellt werden, zu welchem Wahrheitswert die überwachte Bedingung ausgewertet wurde. Auch hierarchisch strukturierte Bedingungen können so durch eine Schachtelung dieser Methode überwacht werden (siehe Beispiel).

```
...
if ( a && b || c ) {...}
...
```

```
...
if ( LogB ( LogB (a && b) || c ) ) {...}
...
```

Bei geschachtelten Bedingungen muss außerdem darauf geachtet werden, dass die Reihenfolge der Auswertung erhalten bleibt. Im Beispiel muss (a && b) zusammen analysiert werden, da das logische Und stärker bindet als das Oder. Eine Änderung der Reihenfolge würde eine komplett andere Bedingung ergeben und den Kontrollfluss verfälschen.

Eine Besonderheit von Java sind beiden Operatoren „&“ und „|“, die abhängig vom Typ der beiden Operanden unterschiedliche Bedeutungen besitzen. Sind beide ganzzahlig, bezeichnen die Operatoren das bitweise Und bzw. das bitweise Oder, so dass ein solcher Ausdruck von der Analyse ausgeschlossen werden muss. Sind beide Operanden vom Typ *boolean*, bezeichnen die beiden Operatoren das logische, nicht kurzschließende Und bzw. Oder und der Ausdruck muss bei der Analyse berücksichtigt werden. Das folgende Beispiel verdeutlicht den Unterschied:

```
(1) if ( b != null && b.foo() ) {...}
(2) if ( b != null & b.foo() ) {...}
```

Angenommen, es gilt $b = null$: Dann wird bei der ersten Anweisung die Methode $b.foo()$ wegen dem kurzschließenden Und nicht aufgerufen, so dass die Bedingung unvollständig evaluiert wird. Im zweiten Ausdruck wurde dagegen das nicht kurzschließende Und verwendet, so dass der Ausdruck immer vollständig evaluiert wird. Dadurch wird die Methode $b.foo()$ immer ausgeführt, unabhängig davon, ob b etwas zugewiesen wurde oder nicht. Dieses Beispiel zeigt auch, dass die kurzschließenden und nicht kurzschließenden Operatoren nicht einfach gegenseitig ausgetauscht werden können. Im Kapitel 4.7 über den Analysierer wird auf diese Problematik im Zusammenhang mit der modifizierten Bedingungs-/Entscheidungsüberdeckung und der Mehrfachbedingungsüberdeckung näher eingegangen.

3.2 Ternäre Operator (?:)

Die Syntax des ternären Operators (?:) von Java lautet:

Variable = (*Bedingung*) ? $Ausdruck_1$: $Ausdruck_2$;

Ist die Bedingung vor dem Fragezeichen Wahr, wird $Ausdruck_1$ zurückgeliefert, andernfalls $Ausdruck_2$. Da der ternäre Operator auch dort verwendet werden kann, wo eine Variable erwartet wird – zum Beispiel beim Aufruf einer Methode –, ist *Variable* optional. Außerdem muss der resultierende Typ der beiden alternativen Ausdrücke gleich sein. Da der ternäre Operator leicht in eine If()-Anweisung mit Else-Zweig umgewandelt werden kann, wird die *Bedingung* vor dem Fragezeichen als zu analysierende Bedingung betrachtet. Deshalb muss keine explizite Bedingung konstruiert werden, und es kann wie bei den expliziten Bedingungen verfahren werden:

```
...
File f;
...
String s = "Datei␣kann␣gelesen␣werden:";
s += ( f.canRead() ) ? ( "ja" ) : ( "nein" );
...
```

```
...
String s = "Datei␣kann␣gelesen␣werden:";
if ( f.canRead() ) {
    s += "ja";
} else {
    s += "nein";
}
...
```

Die Bedingung *f.canRead*() aus dem obigen Beispiel wird durch die Methode LogB() ersetzt, die wiederum die ersetzte Bedingung als Parameter erhält:

```
...
String s = "Datei␣kann␣gelesen␣werden:";
s += ( LogB ( f.canRead() ) ) ? ( "ja" ) : ( "nein" );
...
```

3.3 Switch-Case-Anweisung

Die Switch-Case-Anweisung hat die Syntax:

```
switch ( Vergleichsvariable ) {
   case Fallwert1: Anweisung(en); [break;]
   ...
   case Fallwertn: Anweisung(en); [break;]
   [default: Anweisung(en)]
}
```

Die Switch-Case-Anweisung vergleicht nacheinander den Ausdruck der *Vergleichsvariable* mit jedem *Fallwert* (englisch „cases“). Stimmt dieser Ausdruck mit einer Konstanten überein wird der Anweisungsblock hinter der Sprungmarke ausgeführt. Wird keine Übereinstimmung mit einem Fallwert gefunden werden die Anweisungen in einem ggf. vorhandenen *default*-Block ausgeführt. Es gilt, dass die Vergleichsvariable vom Java-Typ *byte*, *char*, *short* oder *int* sein muss. Dieser Vergleich wird für die Konstruktion einer äquivalenten Bedingung verwendet, auf die die Kriterien angewendet werden können. Zur Laufzeit wird die Auswertung dieser Bedingung durch eine Logge()-Methode im entsprechenden Anweisungsblock protokolliert:

```
...
switch ( x ) {
    case y : Anweisung(en);
    case z : Anweisung(en); break;
    default: Anweisung(en);
}
...
```

```
...
switch ( enterSwitch( x ) ) {
    case y : Logge( x == y );
             Anweisung(en);
    case z : Logge( x == z );
             Anweisung(en)2; break;
    default: Logge( x == default );
             Anweisung(en)3;
}
leaveSwitch();
...
```

Um eine eindeutige Zuordnung der einzelnen Fälle zu einem Switch-Case-Block zu ermöglichen, muss außerdem der komplette Switch-Case-Block durch die beiden Methoden *enterSwitch*() und *leaveSwitch*() – wie in Beispielquelltext gezeigt – eingeschlossen werden.

Das Protokollieren der Auswertungen der Bedingungen zur Laufzeit läuft folgendermaßen ab: Wird der Anweisungsblock für den Fallwert ausgeführt, der mit der Vergleichsvariablen übereinstimmt, wird ebenfalls die eingefügte Logge()-Methode ausgeführt. Diese Methode protokolliert für die konstruierte Bedingung den Wahrheitswert Wahr und für alle anderen, nicht ausgeführten Fälle, den Wert Falsch.

Beispiel: Sei (x = y), so dass der Anweisungsblock des ersten Falls ausgeführt und die Bedingung $(x == y)$ als Wahr gespeichert wird. Da am Ende des Anweisungsblocks des ersten Falls kein *break* gesetzt wurde, werden ebenfalls die Anweisungen des zweiten Falls ausgeführt. Dadurch wird die konstruierte Bedingung $(x == z)$ für den zweiten Fall auch als Wahr gespeichert. Anschließend wird durch die *break*-Anweisung der Switch-Case-Block umgehend verlassen und die *leaveSwitch*()-Methode ausgeführt. Später kann anhand den protokollierten Informationen festgestellt werden, dass der Switch-Case-Block zwar betreten und verlassen wurde, die Anweisungen des $default$-Falls aber nicht ausgeführt wurden. Dadurch wird der $default$-Fall für diesen Durchlauf als zu Falsch ausgewertet betrachtet.

3.4 Ausnahme(Exception)-Behandlung

In Java erzeugt der Zugriff auf eine nicht initialisierte Variable bzw. der Aufruf einer Methode in einem Ausdruck eine Ausnahme (Exception). Wird eine Exception geworfen, ist der Wert der Entscheidung undefiniert und darf deshalb bei der Auswertung der Ergebnisse nicht mit einfließen. Damit Exceptions erkannt und behandelt werden können, ohne die Weiterverarbeitung durch eventuell bereits vorhandene Try-Catch-Finally-Blöcke zu beeinflussen, müssen bestimmte Vorkehrungen getroffen werden: So wird die zu überwachende Entscheidung durch die Methode *catchException*() ersetzt, die die zu überwachende Bedingung erwartet und das Ergebnis der Auswertung auch wieder zurück liefert. Die zu überwachende Bedingung wird außerdem durch die beiden Methoden *startExpression*() und *endExpression*() eingeschlossen, so dass diese zu Beginn bzw. am Ende der Auswertung des Ausdrucks aufgerufen werden. Damit das Auftreten einer Exception erkannt werden kann, wird außerdem die gesamte Methode durch einen allumfassenden Try-Finally-Block eingeschlossen. Dabei ist aber zu beachten, dass bei den Konstruktoren eine eventuell vorhandene *super*()- oder *this*()-Anweisung nicht mit einbezogen wird, da diese Anweisungen – wenn vorhanden – immer umgehend nach dem Kopf des Konstruktors stehen müssen. Im Finally-Block wird schließlich noch die Methode *handleException*() eingefügt, die **immer** beim Verlassen der Methode aufgerufen wird. Der folgende Quelltextbeispiel zeigt die nötigen Vorkehrungen anhand einer Beispiel-Methode:

```
...
public static void main () {
    if ( o.getValue() ) {
        Anweisungen;
    }
}
...
...
public static void main () {
    try {
        ...
        if ( catchException( startExpression( 5 ), LogB
            ( o.getValue(), 5 ) , endExpression( 5 )) {
            ... }
        ...
    } finally {
        handleException('ID␣von␣main()');
    }
}
...
```

Diese „Kapselung“ der beiden Methoden *startExpression*() und *endEpression*() durch die Methode *catchException*() ist notwendig, da dort wo ein Ausdruck erwartet wird keine Anweisungsfolge stehen darf. Das Beispiel zeigt außerdem, dass den Methoden startExpression(),

LogB() und endExpression() die Identifikation der Entscheidung übergeben wird, damit eine Exception eindeutig einer Bedingung zugeordnet werden kann.

Im Falle einer Exception werden die Anweisungen wie folgt abgearbeitet: Tritt eine Exception bei der Auswertung innerhalb einer Bedingung auf, wird zwar die Methode $startExpression(ID)$ aufgerufen, aber durch die Exception umgehend in den Finally-Block verzweigt, so dass die Methode $handleException()$ aufgerufen wird. Dadurch wird die Methode $endExpression(ID)$ nicht ausgeführt und eine geworfene Exception erkannt, ohne die Weiterverarbeitung der Exception zu behindern.

Das Ergebnis einer Entscheidung kann also nicht nur den Wert Wahr oder Falsch annehmen, sondern auch undefiniert sein. Eine Bedingung mit undefinierten Wert darf nicht bei der Berechnung mit einfließen. Ob der Wert einer Bedingung unbekannt ist wird nicht zur Laufzeit, sondern erst bei der Auswertung der protokollierten Informationen ermittelt.

3.5 Polymorphie und dynamisches Binden

Die Polymorphie und das damit zusammenhängende dynamische Binden sind spezielle Konzepte objekt-orientierter Programmiersprachen. Unter Polymorphie (griechisch, „Vielgestaltigkeit“) versteht man in Java die Eigenschaft einer Variablen, Instanzen unterschiedlicher Klassen aufnehmen zu können. Dabei gilt die Einschränkung, dass der Variablen nur eine Instanz der deklarierten Klasse, oder einer davon abgeleiteten Klasse zugewiesen werden kann. Wird eine Methode einer polymorphen Variablen zur Laufzeit aufgerufen, wird abhängig von der zugewiesenen Klasseninstanz eine andere Methode ausgeführt. Die Wahl der richtigen Methode erfolgt zur Laufzeit, was als dynamisches Binden bezeichnet wird. Der folgende Beispielquelltext verdeutlicht diese beiden Konzepte:

```
...
public class A {
    public String toString() {
       return "Klasse A";
    }
}
public class B extends A {
    public String toString() {
        return "Klasse B";
    }
}
public class C extends A {
    public String toString() {
        return "Klasse C";
    }
}
public class TestClass {
```

```
    public static void main( String[] args ) {
          A obj; // obj ist polymorph
          obj = new A();
          println( obj.toString() );  // "Klasse A"
          obj = new B();
          println( obj.toString() );  // "Klasse B"
          obj = new C();
          println( obj.toString() );  // "Klasse C"
    }
}
...
```

In diesem Quelltext-Auszug sind vier Klassen dargestellt: eine Klasse A, die eine Methode toString() implementiert, die beiden Klassen B und C, die von Klasse A abgeleitet sind und die Methode toString() überschreiben, und eine Test-Klasse, die die Klassen A, B und C verwendet. In der Test-Klasse wird der Variable *obj* zuerst eine Instanz der Klasse A zugewiesen. Anschließend wird die Methode *obj.toString*() aufgerufen, was zum Aufruf der Methode toString() der Klasse A führt – erkennbar an der Ausgabe, die als Kommentar angegeben ist. In den darauf folgenden Zeile wird der Variablen *obj* einmal eine Instanz der Klasse B bzw. eine Instanz der Klasse C zugewiesen und nochmals die Methode *obj.toString*() aufgerufen. Die dargestellten Ausgaben zeigen, dass abhängig von der zugewiesenen Klasseninstanz unterschiedliche Methoden aufgerufen werden. Die Variable *obj* ist per Definition polymorph und die an *obj* zugewiesene Klasseninstanz bestimmt, welche Methode aufgerufen wird. Deshalb kann die zugewiesene Klasseninstanz als eine Art Bedingung angesehen werden kann.

Um zur Laufzeit bestimmen zu können, zu welcher Klasse eine Instanz gehört, die in einer Variablen gespeichert ist, existiert in Java der *instancOf*-Operator. Dieser Operator erwartet zwei Operanden: Der erste Operand ist die Variable die mit dem zweiten Operanden, der Klasse, verglichen werden soll. Wurde der Variablen eine Instanz der angegebene Klasse zugewiesen, liefert der *instanceOf*-Operator Wahr zurück, andernfalls Falsch.

Mit diesem Operator ist es zur Laufzeit möglich den Klassentyp zu bestimmen, der für die Wahl der richtigen Methode herangezogen wurde. Da der *instanceOf*-Operator nur Wahr oder Falsch und nicht die Klasse der Instanz zurück liefert, muss für jede in Frage kommende Klasse ein instanceOf-Operator eingefügt werden. Dazu muss zuerst eine Klassenhierarchie erstellt werden, die die statischen Beziehungen zwischen den beteiligten Klassen abgebildet. Dadurch können – ausgehend vom deklarierten Typ der Variablen – alle von dieser Klasse abgeleiteten Klassen bestimmt und anschließend mit dem entsprechenden *instanceOf*-Operator eingefügt werden.

Der folgende Beispielquelltext zeigt, wie mit einer vorhandenen Klassenhierarchie Proben in den Quelltexte eingefügt werden können, so dass zur Laufzeit mit dem *instanceOf*-Operator der Klassentyp der Instanz protokolliert werden kann:

```
...
public class TestClass {
```

```
    public static void main(String[] args) {
        A obj; // obj ist polymorph
        ....
        obj = new B();
        ...
        if ( obj instanceOf A ) Logge (A) else
        if ( obj instanceOf B ) Logge (B) else
        if ( obj instanceOf C ) Logge (C);

        println( obj.toString() );  // "Klasse B"
        ...
    }
}
```

Im Gegensatz zu anderen Programmiersprachen bietet Java durch das Reflection-Modell bereits reichhaltige Funktionen an, mit denen detaillierte Metainformationen über Klassen, Schnittstellen und Methoden eruiert werden können. Dadurch ist es recht einfach die benötigte Klassenhierarchie zu erzeugen.

3.6 Overloading - Überladen von Methoden

Das Überladen (englisch „Overloading“) bezeichnet die Definition mehrerer Methoden mit gleichem Bezeichner, aber unterschiedlichen Signaturen. Da die Methodenbezeichner identisch sind, erfolgt die Wahl der richtigen Methode anhand der übergebenen Parameter. Zur Veranschaulichung dient der folgende Beispielquelltext, in dem die zweimal überladene Methode *Quadrat*() definiert wurde:

```
public static class TestClass {
    public static void Quadrat( int i ) {
        println( "Ergebnis(int): " + i^2 );
    }
    public static float Quadrat( float f ) {
        println( "Ergebnis(float): " + f^2 );
    }
    public static void main( String[] args ) {
        Quadrat( 2 );      // Ergebnis(int): 4
        Quadrat( 2.0 );    // Ergebnis(float): 4.0
    }
}
```

Beim Overloading wird die Wahl der richtigen Methode anhand der übergebenen Parameterkonfiguration und zur Kompilationszeit getroffen (statisches Binden). Eine Ausnahme stellen überladene Methoden bei polymorphen Variablen dar, die dynamisch gebunden werden. Da die

Wahl der richtigen Methode durch die übergebenen Parameter erfolgt, stellt die Parameterkonfiguration ebenfalls eine Art Bedingung dar.

Kapitel 4

Das Werkzeug

Dieses Kapitel beschäftigt sich mit den verschiedenen Ansätzen, um die zusätzlich benötigten Informationen über das PUT („program under test“) zu erhalten. Außerdem wird in diesem Kapitel das Werkzeug genauer vorgestellt, das eine mögliche Implementierung der im vorausgehenden Kapitel genannten Konzepte zur Übertragung der klassischen Bedingungsüberdeckungskriterien auf die Programmiersprache Java darstellt.

4.1 Ansätze für die Instrumentierung

In diesem Abschnitt werden die drei untersuchten Herangehensweisen vorgestellt, um die zusätzlich benötigten Informationen über das PUT zu ermitteln: die explizite Anreicherung des Quelltextes, die Anpassung der Laufzeitumgebung und die automatische Instrumentierung des Quelltextes. Diese Informationen werden für die Ermittlung der Überdeckungskriterien und des Überdeckungsgrads benötigt.

Bei der expliziten Anreicherung fügt der Programmierer beim Schreiben selbstständig Proben in den Quelltext ein. Das bedeutet auch, dass der Programmierer verantwortlich ist für die korrekte und vollständige Instrumentierung des Quelltextes. Bei dieser Vorgehensweise ist aber nicht gewährleistet, dass die Ergebnisse korrekt sind. So kann beispielsweise der Programmierer eine Bedingung vergessen oder falsch instrumentieren. Bei komplexen Bedingungen mit sehr vielen geschachtelten (Teil-)Bedingungen besteht außerdem eine große Diskrepanz zwischen Aufwand und Nutzen. Deshalb scheidet dieses Verfahren von Grund auf aus.

Eine weitere Möglichkeit, an die benötigten Informationen zu gelangen, ist die Anpassung der Laufzeitumgebung. So bietet die JVM („Java Virtual Machine“) eine Schnittstelle, über die externe Programme und Klassen den Programmablauf verfolgen können. Mit diesem Ansatz wäre es möglich, durch das Verfolgen des Programmablaufs – und damit auch der Auswertung der Bedingungen – an die benötigten Informationen zu gelangen. Leider weist die zur Verfügung gestellte Schnittstelle einen gravierenden Nachteil auf: Sie ist zu grobgranular. So bietet die so genannte JVM-TI („JVM Tool Interface“) zwar verschiedene Methoden an, um den Status

des Programms und die Ausführung der Anweisungen überwachen und kontrollieren zu können, auf die einzelnen Teilbedingungen einer komplexen Bedingung kann aber nicht zugegriffen werden[5]. Da die Teilbedingungen benötigt werden – z.B. für die minimale Mehrfachbedingungsüberdeckung –, scheidet dieser Ansatz ebenfalls aus. Zu erwähnen ist außerdem, dass dieses Verfahren nur funktioniert, sofern der resultierende Programmcode durch eine Laufzeitumgebung interpretiert und nicht in maschinennahen Code compiliert wird.

Der letzte betrachtete Ansatz ist die automatische Instrumentierung des Quelltextes durch einen Parser. Durch die automatische Instrumentierung werden die Mängel der expliziten Anreicherung beseitigt und die Instrumentierung beliebig komplexer Bedingungen ermöglicht. Dazu wird der Quelltext in einen AST (engl. „Abstract Syntax Tree“, abstrakter Syntaxbaum) überführt, anschließend instrumentiert und der instrumentierte AST als instrumentierter Quelletext zurück geschrieben. Ein abstrakter Syntaxbaum ist die abstrakte Darstellung der Hierarchie der Anweisungen – und damit auch der Bedingungen – im Quelltext durch eine baumartige Struktur. Die Abbildung 4.2 im Kapitel 4.3 zeigt beispielhaft einen Ausschnitt eines abstrakten Syntaxbaums für das Beispiel `CountWords.java` aus dem Anhang A.

Da dieses Verfahren auf Quelltextebene ansetzt, kann nicht nur interpretierter Code instrumentiert werden, sondern auch compilierter Code. Außerdem ist dieses Verfahren sehr flexibel, da genau spezifiziert werden kann, welche Informationen wie genau benötigt werden. Aufgrund dieser Flexibilität, und weil das dafür benutze Werkzeug kostenlos und frei verfügbar ist, wurde dieses Verfahren für das Werkzeug gewählt.

4.2 Allgemeine Funktionsweise

Zur Überführung des Java-Quelltextes in einen AST und anschließender Instrumentierung wurde das Tool ANTLR („ANother Tool For Language Recognition“) gewählt. Dieses Tool stellt ein Framework zum Erzeugen von Parsern und Compilern zu Verfügung. Dazu benutzt es eine Mischung aus formaler Grammatik (eine sogenannte LL(k)-Grammatik) und Elemente objektorientierter Sprachen. Eine LL(k)-Grammatik ist eine kontextfreie Grammatik, bei der jeder Ableitungsschritt durch k Symbole der Eingabe eindeutig bestimmt ist. Für weitere Informationen verweise ich auf [3].

Die Abbildung 4.1 gibt einen Überblick über die allgemeine Funktionsweise, den Aufbau des Werkzeugs und das Zusammenspiel der einzelnen Komponenten. Das Werkzeug gliedert sich in drei Teile: einen statischen Teil, einen dynamischen Teil und einen analytischen Teil. Der statische Teil besteht aus den Komponenten *Parser*, die aus dem Quelltext einen AST erzeugt, dem *Instrumentierer*, der den erzeugten AST instrumentiert, und aus dem *Schreiber*, der den instrumentierten Quelltext zurückschreibt und die Symboltabelle erstellt. In dieser Symboltabelle werden alle benötigten, aus dem Quelltext extrahierten Informationen gespeichert. Der dynamische Teil besteht aus der Komponente *Logger*, die die bei der Ausführung des instrumentierten Quelltextes notwendigen Aktivitäten aufzeichnet und in einer Logdatei speichert. Der analytische Teil, bestehend aus dem *Analysierer*, generiert schließlich anhand den Logdateien und der Symboltabelle den Report mit den in Kapitel 2.6 definierten Überdeckungsmetriken.

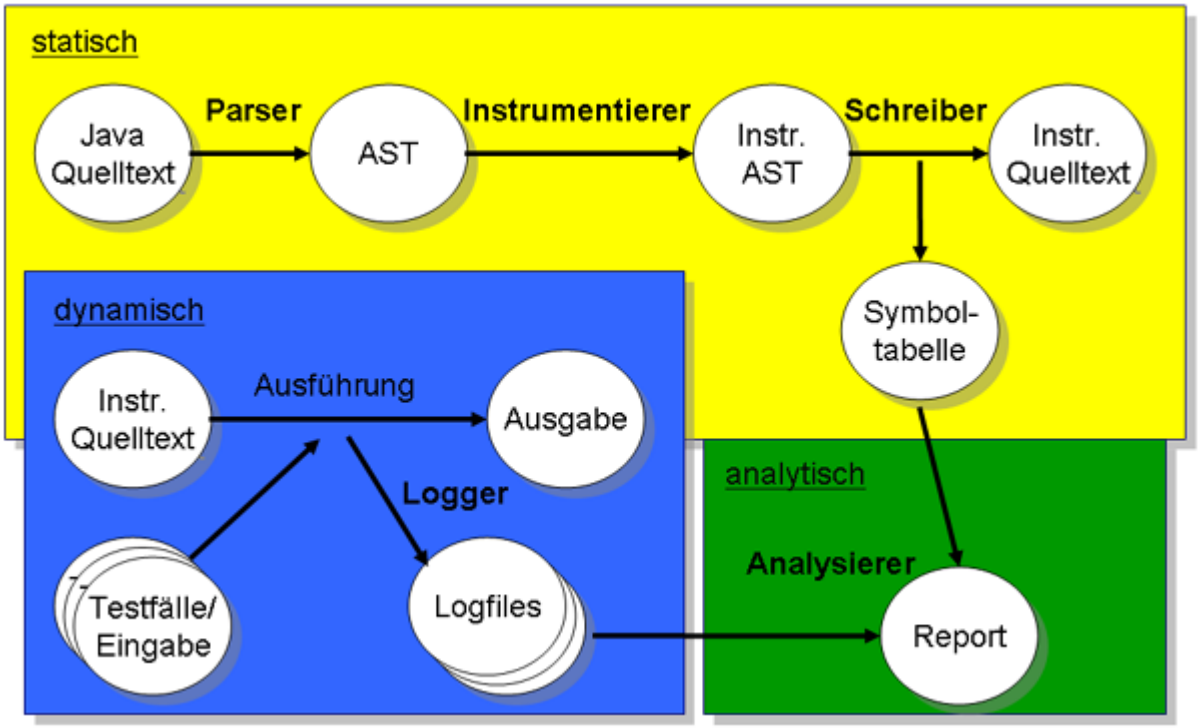

Abbildung 4.1: Allgemeine Funktionsweise des Werkzeugs

Im Folgenden werden die einzelnen Komponenten und der Datenfluss zwischen diesen Komponenten mithilfe des Beispielquelltexts `CountWords.java` aus dem Anhang A genauer beschrieben.

4.3 Der Parser

Der Parser besteht aus den beiden Klassen JavaTokenizer und JavaRecognizer, die mittels dem bereits vorgestellten Tools ANTLR aus der Grammatik java15.g erstellt werden. Eine Instanz der Klasse JavaTokenizer liest den zu instrumentierenden Java-Quelltext ein und bestimmt daraus die Tokens. Wie diese Tokens aus der Eingabezeichenfolge gebildet werden, ist ebenfalls in der Grammatik java15.g spezifiziert. Sobald ein Token vollständig erkannt wurde, wird dieses an eine Instanz der Klasse JavaRecognizer übergeben, die anhand der in der Grammatik definierten Regeln einen AST im Speicher erzeugt. Dieser AST stellt eine abstrakte Repräsentation des eingelesenen Java-Quelltexts dar. Die Abbildung 4.2 zeigt einen Ausschnitt des abstrakten Syntaxbaums (AST) für das Beispiel `CountWords.java` aus dem Anhang A.

Die Grammatik *java*15.*g* für Java Version 1.5 existierte bereits und wurde nur geringfügig geändert. So wurde das Pseudotoken „LOGB“ eingefügt, das zur Instrumentierung beliebig geschachtelter, expliziter Bedingungen, wie im Kapitel 3.1 beschrieben, verwendet wird.

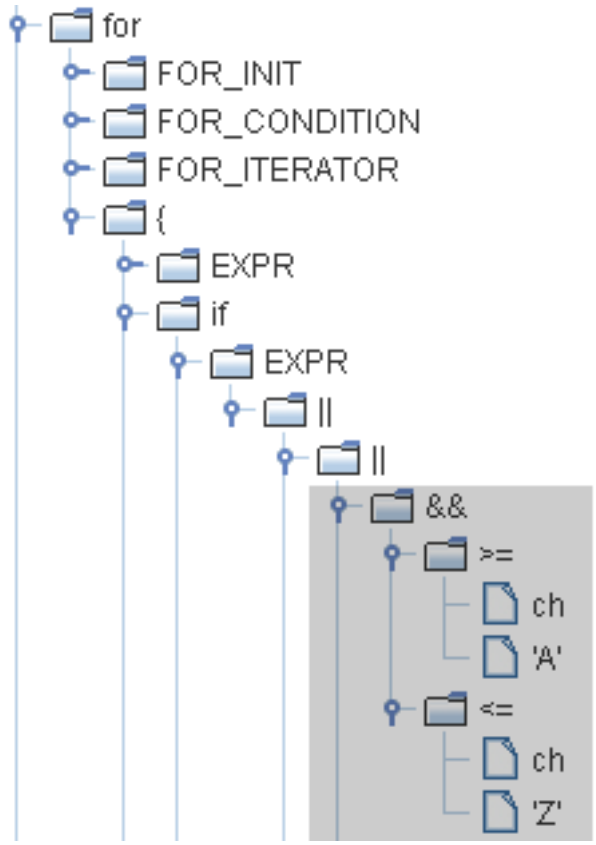

Abbildung 4.2: Beispiel-AST (Auszug) einer for()-Schleife aus dem Quelltext `CountWord.java`, Zeile 15

4.4 Der Instrumentierer

Nachdem vom Parser der AST erstellt wurde, wird er an eine Instanz der Klasse JavaTreeInstrumenter zur Instrumentierung übergeben. Diese Klasse wird ebenfalls aus einer Grammatik (java15.tree.instrumenter.g) durch ANTLR erzeugt. Diese Grammatik wird nicht auf Tokens, wie beim Parser, sondern auf den AST angewendet. Dazu wird in der Grammatik ein besonderes Feature von ANTLR verwendet: Bei Grammatiken für ANTLR ist es möglich, eine Regel mit einer Aktion zu versehen, die bei einer Übereinstimmung mit der Eingabe ausgelöst wird. Für die Instrumentierung des ASTs wurden Aktionen in den Regeln für die logischen Operatoren „&&“, „||“ und „!“ definiert, die in den AST zusätzliche Knoten einfügen. Die Abbildung 4.3 zeigt einen Ausschnitt aus den Beispiel-AST, nachdem er instrumentiert wurde.

Die Instrumentierung expliziter Bedingungen erfolgt wie im Kapitel 3.1 beschrieben wurde. Dazu werden so genannte „LOGB“-Knoten eingefügt, sobald eine Bedingung beim Traversieren des AST vorgefunden wurde. Dadurch können beliebig geschachtelte Bedingungen instrumentiert werden. Damit der Analysierer die benötigten Bedingungen aus der Symboltabelle selektieren kann, besitzen die eingefügten „LOGB“-Knoten weitere Informationen (= die Blätter des

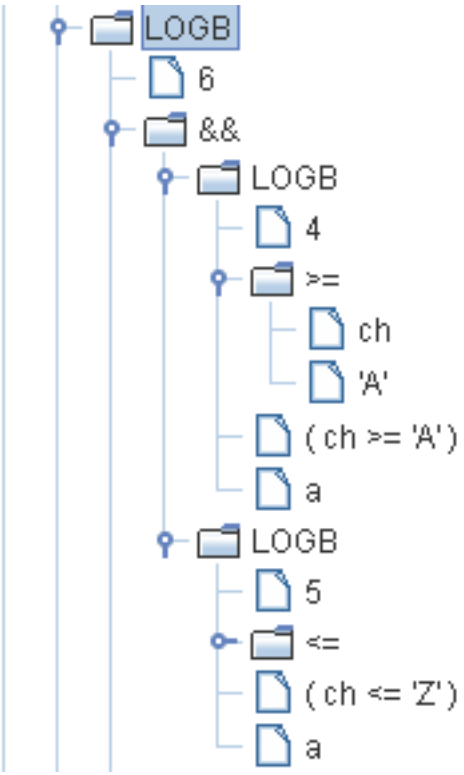

Abbildung 4.3: Instrumentierter Beispiel-AST für die markierte Teilbedingung

Knotens, siehe Abbildung 4.3):

Das erste Blatt des „LogB“-Kontens ist die laufend durchnummerierte, nummerische Identifikation, die eine eindeutige Zuordnung zu einem Eintrag in der Symboltabelle ermöglicht. Das zweite Blatt, eigentlich ein Konten, repräsentiert bei geschachtelten Bedingungen den AST der untergeordneten Bedingung. Das dritte Blatt speichert die Bedingung als Zeichenkette, die aus den bis zu diesem Knoten gefundenen Unterbedingungen zusammengesetzt wird. Und das letzte Blatt spezifiziert die Art des Knotens. Welche verschiedenen Arten möglich sind, wird im Kapitel 4.6 über den Logger genauer erläutert.

Wie bereits in Kapitel 3.1 beschrieben, besitzen die beiden Operatoren „&“ und „|“ in Java zwei Rollen: entweder die Rolle eines bitweisen Operators bei ganzzahligen Operanden, oder die eines logischen, nicht kurzschließenden Operators bei bool’schen Operanden. Liegt ein logischer Operator vor, muss der Ausdruck instrumentiert werden.

Da die Operanden auch Variablen und Methoden sein können, kann während der Instrumentierung nicht statisch zwischen den beiden Bedeutungen unterschieden werden. Damit trotzdem die Semantik des Operators bestimmt werden kann, muss der Typ einer Variablen bzw. Methoden ermittelt werden. Dazu existieren zwei Vorgehensweisen, die je nach Gültigkeit der Variable bzw. Methode angewendet werden: Ist die zu untersuchende Variable bzw. Methode lokal definiert, d.h innerhalb der Klasse, in der sie verwendet wird, erfolgt die Bestimmung des Typs anhand einer Variablen- bzw. Methodentabelle. Diese Tabellen werden beim Lesen jeder Variablen-

bzw. Methodendeklaration in der aktuellen Klasse aktualisiert. Dazu wurden in der Grammatik java15.tree.instrumenter.g Regeln definiert, die bei einer Übereinstimmung mit einer Variablen- bzw. Methodendeklaration aufgerufen werden. Diese Regeln speichern den Variablen- bzw. Methodenbezeichner zusammen mit dem Typ in der entsprechenden Tabelle ab.

Ist stattdessen die Variable bzw. Methode nicht lokal definiert, sondern innerhalb einer externen Klasse oder einer übergeordneten Klasse, wird die dazugehörende Klasse zuerst mittels Reflection dynamisch geladen. Anschließend wird bei einer Methode die Methode `getMethod( String name, Class[] parameterTypes )`, und bei einer Variablen die Methode `getField( String name)` verwendet, um den Typ zu besimmen. Abhängig vom Ergebnis dieser Abfrage wird der Ausdruck instrumentiert oder nicht.

4.5 Der Schreiber

Ein Instanz der Klasse JavaTreeWriter erhält vom Instrumentierer den instrumentierten AST und schreibt diesen als instrumentierten Java-Quelltext zurück. Diese Klasse wird aus der Grammatik java15.treewriter.g durch ANTLR erzeugt. Die Instrumentierung expliziter Bedingungen geschieht wie folgt: Wird beim Durchlaufen des instrumentierten AST ein LOGB-Knoten gelesen, wird die statische Methode `boolean _LogB( boolean, long )` des Loggers in den instrumentierten Quelltext eingefügt. Diese Methode erwartet als Parameter eine Bedingung und die ID der Bedingung. Beide Informationen werden den entsprechenden Blättern des LOGB-Knotens entnommen.

Weiterhin erstellt diese Komponente eine Symboltabelle aus dem AST. In dieser Symboltabelle werden neben den extrahierten Bedingungen ebenfalls die Switch-Case-Anweisungen, die Bezeichner der definierten Methoden, Konstruktoren, polymorphen Variablen und Klassen gespeichert. Ein Eintrag in dieser Tabelle ist wie folgt aufgebaut: In der ersten Spalte steht eine eindeutige ID, die eine Zuordnung des Eintrags zu einem Ausdruck oder einer Anweisung im instrumentierten Quelltext ermöglicht. In der zweiten Spalte ist die Zeilennummer des Ausdrucks bzw. der Anweisung im instrumentieren Quelltext gespeichert, und in der dritten Spalte steht die Art des Eintrags (siehe Tabelle 4.4). In den letzten beiden Spalten ist die vom Quelltext extrahierte (Teil-)Bedingung bzw. der extrahierte Bezeichner und eine optionale Notiz gespeichert. Die Abbildung 4.5 zeigt die Symboltabelle für den Beispiel-Quelltext `CountWords.java` aus dem Anhang A.

Die Schreiber-Komponente ist weiterhin für die Instrumentierung des ternären Operators, der Switch-Case-Anweisung, der Methoden und Konstruktoren und polymorpher Variablen zuständig. Diese Auslagerung ist erfolgt, da eine Instrumentierung über den AST nicht notwendig ist. Die Instrumentierung selbst erfolgt gemäß den in Kapitel 3 erörterten Konzepten.

Instrumentierung des ternären Operators
Beim Matching einer Bedingung des ternären Operators wird diese mit dem entsprechenden Typ in der Symboltabelle eingefügt. Außerdem wird die Bedingung durch das Einfügen der Methode `_LogB( boolean, long )` instrumentiert.

- „a“: die Bedingung ist atomar
- “p“: die Bedingung ist primär, also eine bool’sche Variable bzw. Methode
- „c“: die Bedingung ist zusammengesetzt („combined“)
- „b“: die Bedingung ist eine Entscheidung („decition/branch“)
- „t“: die Bedingung stammt vom ternären Operator
- „s“: die Bedingung stammt von einem Fall der `switch-case`-Anweisung
- „w“: bezeichnet eine Switch-Case-Anweisung
- „m“: bezeichnet eine Methode
- „o“: bezeichnet einen Konstruktor
- „k“: bezeichnet eine Klasse
- „v“: bezeichnet eine polymorphe Variable
- „h“: bezeichnet einen möglichen Typ der polymorphen Variable

Abbildung 4.4: Mögliche Arten/Typen eines Eintrags in der Symboltabelle

Instrumentierung der Switch-Case-Anweisung
Weiterhin wird durch den Schreiber jede Switch-Case-Anweisung nach den in Kapitel 3.3 genannten Konzepten instrumentiert. Dazu wird beim Zurückschreiben des instrumentierten AST die Vergleichvariable durch die statische, und für jeden möglichen Vergleichstyp überladene Methode `_enterSwitch()` des Loggers ersetzt, die als Parameter die ersetzte Vergleichsvariable erwartet und diese auch wieder für den Vergleich zurück liefert. Außerdem fügt der Schreiber in den Anweisungsblock eines jeden Falls die statische Methode `_LogC( long )` ein, die für das Protokollieren der für jeden Fallwert konstruierten Bedingung zuständig ist. Auch das Einfügen der statischen Methode `_leaveSwitch()` umgehend nach der Anweisung erfolgt durch den Schreiber.

Instrumentierung der Methoden-/Konstruktoren
Es werden alle Methoden und Konstruktoren mit einem allumfassenden try/finally-Block instrumentiert, damit (wie in Kapitel 3.4) eine Exception bei der Auswertung einer Bedingung erkannt und behandelt werden kann. Dabei wird darauf geachtet, dass die eventuell vorhandenen Anweisungen *this* und *super* in einem Konstruktor nicht mit eingeschlossen werden, da der Aufruf von *this* oder *super* die erste Anweisung in einem Konstruktor sein muss. Zur Behandlung der Exceptions fügt der Schreiber die statischen Methoden `_catchException( int, boolean, int`

```
ID Row Type Expression                                                        Notes
1  1   k    CountWords
4  7   m    main ( String[] args )
2  8   a    ( args.length > 0 )
3  8   b    ( args.length > 0 )
21 20  m    countWords ( String text )
5  31  a    ( ch >= 'A' )
6  31  a    ( ch <= 'Z' )
7  31  c    ( ch >= 'A' ) && ( ch <= 'Z' )
8  31  a    ( ch >= 'a' )
9  31  a    ( ch <= 'z' )
10 31  c    ( ch >= 'a' ) && ( ch <= 'z' )
11 31  c    ( ch >= 'A' ) && ( ch <= 'Z' ) || ( ch >= 'a' ) && ( ch <= 'z' )
12 31  a    ( ch == ' ' )
13 31  c    ( ch >= 'A' ) && ( ch <= 'Z' ) || ( ch >= 'a' ) && ( ch <= 'z' )
                                                    || ( ch == ' ' )
14 31  b    ( ch >= 'A' ) && ( ch <= 'Z' ) || ( ch >= 'a' ) && ( ch <= 'z' )
                                                   || ( ch  == ' ' )
15 37  a    ( ch == ' ' )
16 37  a    ( ch == '\t' )
17 37  c    ( ch == ' ' ) || ( ch == '\t' )
18 37  a    ( ch == '\n' )
19 37  c    ( ch == ' ' ) || ( ch == '\t' ) || ( ch == '\n' )
20 37  b    ( ch == ' ' ) || ( ch == '\t' ) || ( ch == '\n' )
```

Abbildung 4.5: Symboltabelle für den Quelltext `CountWords.java`

`),` `_startExpression( long )` und `_endExpression( long )` bei Entscheidungen ein und ergänzt den Finally-Block durch die statische Methode `_handleException( long )`. Außerdem fügt der Schreiber bei einer Methoden- bzw. Konstruktordefinition die statische Methode `_enterMethod( long )` als erste, in Ausnahmefällen auch als zweite Anweisung (siehe oben) ein. Die Methode `_leaveMethod( long )` wird schließlich als letzte Anweisung der Definition hinzugefügt. Dadurch werden die in Kapitel 3.6 erörterten Konzepte für das Overloading realisiert.

Instrumentierung polymorpher Variablen

Die Instrumentierung polymorpher Variablen erfolgt, wie in Kapitel 3.5 beschrieben, durch das Einfügen eines If-Else-Blocks vor dem Aufruf einer Methoden. Damit für jeden möglichen Klassentyp ein If-Else-Block und der *instaceOf*-Operator eingefügt werden kann, erstellt der Schreiber für jede zu instrumentierende Klasse eine Klassenhierarchie, in der alle bekannten Klassen und deren statische Beziehungen (Ober-/Unterklasse) zueinander gespeichert werden. Die Klassenhierarchie wird in einer Hashtabelle gespeichert, in der als Schlüssel der Name der Klasse und

als Wert eine nicht initialisierte Instanz verwendet wird. Zum Aufbau der Hierarchie werden zuerst die Wrapper-Klassen der einfachen Datentypen in einer Hashtabelle gespeichert. Anschließend wird an allen bekannten Stellen nach weiteren Standard- oder benutzerdefinierten Klassen gesucht, dazu zählen das Verzeichnis der Klasse, alle Unterverzeichnisse sowie alle Verzeichnisse und Packages im Classpath. Nachdem alle bekannten Klassen in die Hashtabelle eingefügt wurden, wird die Hashtabelle durchlaufen und für jede Klasse die dazugehörende Oberklasse gespeichert. Dazu wird die in der Hashtabelle gespeicherte Instanz der Klasse und die Methode `Class.getSuperClass()` verwendet. Außerdem wird die Klasse selbst zur Unterklassenliste der Oberklasse hinzugefügt. Bei jeder gelesenen Variablendeklaration wird der Bezeichner der Variablen und alle möglichen Klassentypen dieser Variablen in der Symboltabelle mit einer eindeutigen ID abgelegt.

Die Klasse Instrumenter stellt eine Schnittstelle zu den Klassen, die für die Instrumentierung notwendig sind, dar, so dass die vorgestellten Klassen nicht direkt verwendet werden müssen. Dazu bindet diese Klasse die Klassen JavaTokenizer, JavaRecognizer, JavaTreeInstrumenter und JavaTreeWriter ein. Einen Überblick über die zu Verfügung gestellten Funktionen gibt Anhang C.1.

4.6 Der Logger

Der dynamische Teil des Werkzeugs besteht aus dem Logger, der in der gleichnamigen Klasse implementiert ist. Nachdem der Java-Quelltext durch den statischen Teil instrumentiert wurde, wird der Quelltext mit den Testfällen (= Eingaben) ausgeführt. Dabei wird bei jeder Ausführung mit einer Eingabe eine separate Logdatei erstellt. In dieser Logdatei werden durch den Logger alle relevanten Ereignisse (z.B. die Evaluation einer Bedingung, das Betreten und Verlassen einer Methode, usw.) in den einzelnen Einträgen gespeichert. Ein solcher Eintrag ist ein Tripel aus Long-Werten: Der erste Long-Wert bezeichnet die ID, die entweder auf eine Bedingung in der Symboltabelle, oder auf eine polymorphe Variablen verweist, oder eine besonderes Ereignis beschreibt. Ist der Wert der ID größer 0, bezeichnet die ID eine Bedingung oder eine polymorphe Variable in der Symboltabelle, ist der Wert kleiner oder gleich -2, bezeichnet die ID ein besonderes Ereignis. Die möglichen besonderen Ereignisse sind in der Klasse `LoggerTypes.java` als Konstanten definiert und in der Tabelle 4.6 aufgelistet.

Je nach Art des Eintrags in der Symboltabelle hat der zweite Long-Wert in der Logdatei eine andere Bedeutung: Bei einer Switch-Case-Anweisung bezeichnet der zweite Long-Wert den Wert der Vergleichsvariable beim Betreten der Anweisung. Bei einer (Teil-)Bedingung stellt der zweite Long-Wert den bool'schen Wert der (Teil-)Bedingung dar: Der Wert 0 bedeutet Falsch, eine 1 bedeutet Wahr, eine 2 bedeutet Unbekannt und eine -1 bedeutet Null. Beim Betreten bzw. Verlassen einer Methode/eines Konstruktors ist der zweite Long-Wert die ID der Methode bzw. des Konstruktors in der Symboltabelle. Bei einer polymorphen Variablen bezeichnet der zweite Wert die ID der Klasse, von der eine Instanz in der polymorphen Variablen gespeichert ist. Im dritten Long-Wert wird schließlich die Thread-Id gespeichert, die zur Unterscheidung in nebenläufigen Threads notwendig ist. Die Abbildung 4.7 zeigt den Anfang der Logdatei, die bei der

- -2: Eine Switch-Case-Anweisung wurde betreten
- -3: Eine Switch-Case-Anweisung wurde verlassen
- -4: Die `catchException()`-Methode wurde aufgerufen
- -5: Die Evaluation einer Entscheidung wurde gestartet (`startExpression()`)
- -6: Die Evaluation einer Entscheidung wurde erfolgreich beendet (`endExpression()`)
- -7: Ein `finally`-Block wurde betreten (`handleException()`)
- -8: Eine Methode bzw. ein Konstruktor wurde betreten (`enterMethod()`)
- -9: Eine Methode bzw. ein Konstruktor wurde verlassen (`leaveMethod()`)

Abbildung 4.6: Besondere Ereignisse

Ausführung des instrumentierten Java-Beispielquelltexts `CountWords.java` aus dem Anhang A mit dem Testfall „Test“ als Eingabe erzeugt wurde. In dieser Abbildung wurde ein Eintrag (= Long-Tripel) durch gleichfarbige Flächen markiert.

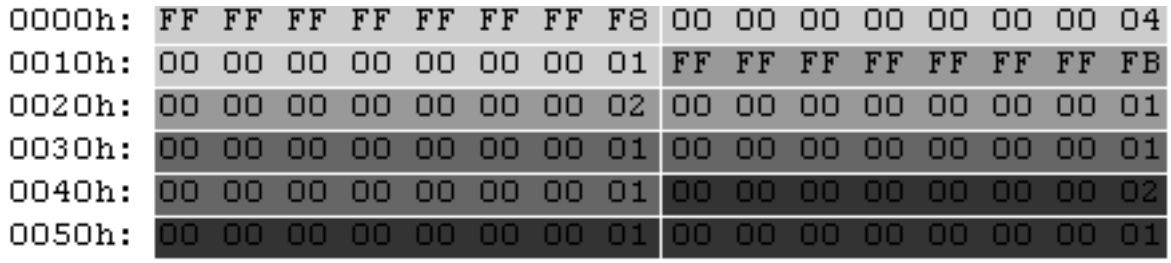

Abbildung 4.7: Auszug aus einer Logdatei

Der erste Long-Wert im ersten Block der Logdatei beschreibt das Betreten einer Methode, gekennzeichnet durch die besondere ID -8 (entspricht FF FF FF FF FF FF FF F8 hexadezimal). Der zweite Long-Wert (4) im ersten Block bezeichnet die ID der Methode (4 = void main) in der Symboltabelle. Im dritten und letzten Long-Wert des Blocks ist schließlich die Thread-ID gespeichert (1). Da im Beispielquelltext keine Nebenläufigkeit auftreten kann, ist dieser Wert immer 1 und wird deshalb in der weiteren Beschreibung nicht mehr erwähnt. Der zweite Block beschreibt den Start der Auswertung (-5) der Bedingung mit der ID 2, der dritte Block die Auswertung der Bedingung mit der ID 1 zu Wahr (1), der vierte Block die Auswertung der Bedingung 2 zu Wahr (1), usw.

Der Logger implementiert außerdem die in Kapitel 3 genannten Methoden zur Verfolgung des Programmablaufs: Die statische Methode `boolean _logB( boolean, long )`, die eine Bedingung und die ID der Bedingung als Parameter erwartet, die ID zusammen mit dem bool'schen Wert der Bedingung als Tripel (ID, bool'scher Wert als Ganzzahl, ThreadID) in die Logdatei

schreibt und diesen Wert wieder zurück liefert. Weiterhin die statische, und für jeden bei der Switch-Case-Anweisung erlaubten Typ überladene Methode `_switchEnter( x, long )`, die den Wert *x* ebenfalls wieder zurück liefert, und die statische Methode `_switchLeave()`, die vom Schreiber umgehend nach der Switch-Case-Anweisung eingefügt wird. Diese beiden Methoden protokollieren das Betreten und Verlassen einer Switch-Case-Anweisung, indem sie das Tripel (-2 bzw. -3, ID der Switch-Case-Anweisung, ThreadID) in der Logdatei speichern. Um die Auswertung der konstruierten Bedingungen für jeden Fall der Switch-Case-Anweisung (wie in Kapitel 3.3 beschrieben) protokollieren zu können, wurde die statische Methode `_LogC( long )` implementiert, die vom Schreiber im Anweisungsblock eines jeden Fallwerts eingefügt wird und bei einem Aufruf die unter der ID referenzierte Bedingung zu Wahr ausgewertet speichert.

Für die in Kapitel 3.4 beschriebenen Vorkehrungen für die Behandlung der Ausnahmen (Exceptions) besitzt die Logger-Klasse die beiden statischen Methoden `int _startExpression( long )` und `int _endExpression( long )`, die als Parameter die ID der überwachten Entscheidung in der Symboltabelle erwarten. Die ebenfalls statische Methode `boolean _catch Exception( int, boolean, int )` dient als Container für die beiden gerade erwähnten Methoden und die zu überwachende Entscheidung. Diese Methode liefert den Wert der Entscheidung auch wieder zurück, damit der ursprüngliche Kontrollfluss nicht beeinflusst wird. Außerdem implementiert die Klasse Logger die statische Methode `_handleException( long )`, die vom Schreiber in den alles umfassenden Try-Finally-Block eingefügt wird, und als Parameter die ID der Methode bzw. des Konstruktors in der Symboltabelle erwartet. Die beiden Methoden für die Überwachung der Auswertung (startExpression() und endExpression()) protokollieren ihren Aufruf mit dem entsprechenden Tripel (-5 bzw. -6, ID der überwachten Entscheidung, ThreadID) in der Logdatei. Die `_handleException( long )`-Methode schreibt in die Logdatei das Tripel (-7, ID der Methode bzw. des Konstruktors, ThreadID), so dass das Auftreten und der Ort einer Exception erkannt werden kann. Durch diese Vorkehrungen kann der Analysierer die Bedingungen mit unbekanntem Wert bei der Berechnung der Metriken herausfiltern.

Für die Behandlung von Polymorphie und das dynamische Binden wurde vom Schreiber vor dem Aufruf einer Methode und für jeden möglichen Klassentyp ein If-Else-Block eingefügt. In diesen Blöcken wurde ebenfalls die Methode `_LogP( long, long )` eingefügt, die als ersten Parameter die ID der polymorphen Variablen und als zweiten Parameter die ID der Klasse erwartet. Beide Informationen wurden vom Schreiber in der Symboltabelle anhand der erstellen Klassenhierarchie gespeichert. Die Methode LogB speichert die übergebenen Informationen als Tripel (ID der Variablen, ID der Klasse, ThreadId) in der Logdatei.

Damit das Betreten und Verlassen einer Methode bzw. eines Konstruktors verfolgt werden kann, um die in Kapitel 3.6 genannten Konzepte für das Overloading anwenden zu können, wurden die beiden statischen Methoden `_enterMethod( long )` und `_leaveMethod( long )` in der Klasse Logger implementiert. Diese Methoden werden von der Schreiber-Komponente beim Zurückschreiben des instrumentierten AST an den relevanten Stellen jeder Methoden- bzw. Konstruktordefinition eingefügt, und protokollieren jedes Betreten und Verlassen in der Logdatei. Dazu wird das Tripel (-8 bzw. -9, ID der Methode bzw. des Konstruktors, ThreadID), eingefügt.

Durch die beschriebenen Methoden des Loggers kann der Analysierer die genannten Konzep-

te zur Übertragung der klassischen Bedingungsüberdeckung auf Java anwenden und die Überdeckungsmetriken berechnen.

4.7 Der Analysierer

Der analytische Teil des Werkzeugs besteht aus dem Analysierer, der in der Klasse Analyzer implementiert ist. Diese Komponente bestimmt aus der Symboltabelle und den Logdateien, welche Bedigungsüberdeckungskriterien erfüllt sind. Außerdem ermittelt der Analysierer die in Kapitel 2.6 genannten Überdeckungsmetriken. Dazu bietet die Analyzer-Klasse unterschiedliche Methoden an, mit denen Informationen über die gewünschten Kriterien und Metriken erfragt werden können (siehe Anhang C.2).

Für die Berechnung der grobgranularen Überdeckungsmetriken werden alle Logdateien und alle Bedingungen in der Symboltabelle herangezogen. Eine Bedingung gilt als überdeckt, wenn Logdateien existieren, in denen die Bedingung mindestens einmal zu Wahr und mindestens einmal zu Falsch ausgewertet wurde. Dazu existiert die Klasse `Symbol` mit den drei öffentlichen Attributen `int trueCount`, `int falseCount` und `int unknownCount` (siehe Anhang B.2), die ein Symbol in der Symboltabelle repräsentiert. In den drei Attributen eines jeden Symbols wird gespeichert, wie oft ein Symbol (in diesem Fall eine Bedingung) zu Wahr, Falsch und Unbekannt ausgewertet wurde.

Um die feingranularen Überdeckungsmetriken zu berechnen, müssen die Bedingungen zu zusammengesetzten Bedingungen und Entscheidungen in der Symboltabelle zugeordnet werden können. Diese Zuordnung erfolgt folgendermaßen: Wird eine atomare oder primäre Bedingung gelesen, werden diese in einer temporären Liste gespeichert, bis eine zusammengesetzte Bedingung oder Entscheidung gelesen wird. Beim Lesen einer zusammengesetzten Bedingung oder einer Entscheidung werden die entsprechenden IDs der Bedingungen in der temporären Liste auf die ID der zusammengesetzten Bedingung (Attribut *combinedID*) oder Entscheidung (Attribut *belongstoID*) gesetzt. Auf ähnliche Art und Weise erfolgt das Setzen des Attributs *belongstoID* bei Switch-Case-Anweisungen und den polymorphen Variablen.

Um den Überdeckungsgrad pro Methode bzw. pro Konstruktor und pro Klasse berechnen zu können, müssen außerdem die Attribute *methodId* und *classId* jedes relevanten Symbols gesetzt werden. Dazu wird das Attribut *methodId* der Symbole, die in der Symboltabelle zwischen zwei Methodensymbolen bzw. zwischen einem Methodensymbol und dem Ende der Symboltabelle stehen, auf die ID des oberen Methodensymbols gesetzt. Das Genannte gilt ebenfalls für Konstruktorsymbole und in ähnlicher Form für das Attribut *classId* bei Klassensymbole.

In der Symbol-Klasse ist außerdem das öffentliche Attribut *type* definiert, dass den Typ des Symbols angibt. Die möglichen Typen sind in Tabelle 4.4 im Kapitel 4.5 dargestellt. Über diesen Typ kann der Analysierer die für ein Kriterium benötigten Symbole auswählen. Zum Beispiel verwendet der Analysierer für die einfache Bedingungsüberdeckung nur die Symbole mit dem Typ „a“ (atomare Bedingungen) und „p“ (primäre Bedingungen, bool’sche Variablen). Für den Überdeckungsgrad der Bedingungs-/Entscheidungsüberdeckung werden die Typen „a“, „p“ und

„b“ (atomare Bedingungen und Entscheidungen), und für die minimale Mehrfachbedingungsüberdeckung die Typen „a“, „p“, „c“ und „b“ (alle Bedingungen) verwendet. Dazu werden die vom Logger für jede Eingabe erstellten Logdateien ausgelesen, um die überdeckten Bedingungen zu bestimmen, und mit den von jeweiligen Kriterium geforderten Bedingungen zu vergleichen.

Modifizierte Bedingungs-/Entscheidungsüberdeckung und Mehrfachbedingungsüberdeckung
Für die beiden verbleibenden Kriterien (die modifizierte Bedingungs-/Entscheidungsüberdeckung und die Mehrfachbedingungsüberdeckung) werden andere Techniken verwendet, um den Überdeckungsgrad zu messen. Bei der Bedingungs-/Entscheidungsüberdeckung muss anhand der Logdateien festgestellt werden, ob eine atomare Bedingung das Ergebnis der Entscheidung beeinflusst. Dazu wählt der Analysierer eine Auswertung einer Entscheidung aus und vergleicht diese mit den verbleibenden Auswertungen dieser Entscheidung. Diese Auswertungen einer Entscheidung sind in den Logdateien gespeichert. Wurden zwei Auswertungen einer Entscheidung gefunden, bei denen die nicht betrachteten atomaren Bedingungen gleich sind, vergleicht der Analysierer das Ergebnis der Entscheidung mit der betrachten atomaren Bedingung. Ist beides mal der Wahrheitswert der betrachteten atomaren Bedingung und der Wahrheitswert der Entscheidung identisch, so wurde ein Auswertungspaar gefunden, das das Kriterium erfüllt. Danach wird der Algorithmus mit der nächsten atomaren Bedingung der Entscheidung ausgeführt. Wird kein passendes Auswertungspaar gefunden, wird die nächste Auswertung als fest angenommen und wieder mit den restlichen Auswertungen verglichen. Dies wird solange wiederholt, bis keine Auswertungen einer Entscheidung mehr vorhanden sind, oder ein Auswertungspaar für alle atomaren Bedingungen gefunden wurde. Folgendes Beispiel soll den Algorithmus verdeutlichen:

Angenommen, die betrachtete Entscheidung ist aus den drei atomaren Bedingungen (A, B, C) zusammengesetzt, und es existieren in den zu analysierenden Logdateien die folgenden Auswertungen (0 = Falsch, 1 = Wahr):

(0, 1, **0**) = **0**
...
(0, 1, **1**) = **1**

Dann wählt der Algorithmus die erste Auswertung für den ersten Durchlauf aus und vergleicht sie mit den andern Auswertungen der gleichen Entscheidung. Wird die zweite Auswertung gelesen, bei der die nicht betrachtetet atomaren Bedingungen beider Auswertungen identisch sind, vergleicht er das Ergebnis der Entscheidung. In diesem Fall stellt er fest, dass die atomare Bedingung C unabhängig von den anderen atomaren Bedingungen das Ergebnis der Entscheidung beeinflusst. Die beiden Auswertungen bilden ein Auswertungspaar, das das Kriterium für die atomare Bedingung C erfüllt.

Bei unvollständiger Auswertung wird die Anforderung, dass alle nicht betrachteten Bedingungen einer Auswertung identisch sein müssen, abgeschwächt. So werden auch Auswertungen akzeptiert, bei denen die nicht betrachteten atomaren Bedingungen ungleich sein können, sofern diese nicht evaluiert wurden. Beispiel: Angenommen, die betrachtete Entscheidung ist die gleiche wie oben und es existieren in den zu analysierenden Logdateien die folgenden Auswertungen

(0 = Falsch, 1 = Wahr, x = nicht ausgewertet):

(0, 1, 0) = 0
...
(0, x, 1) = 1

Dann wird trotzdem davon ausgegangen, dass beide Auswertungen ein Auswertungspaar bilden, obwohl B im zweiten Fall nicht ausgewertet wurde.

Zusammengefasst kann der Überdeckungsgrad der modifizierten Bedingungs-/Entscheidungsüberdeckung für eine Entscheidung definiert werden als der Quotient aus der Anzahl der gefunden Auswertungspaare und der Anzahl der atomaren Bedingungen.

$$\mathrm{UG}_{MCDC} = \frac{\text{Anzahl der gefundenen Auswertungspaare für eine Entscheidung}}{\text{Anzahl aller atomarer Bedingungen einer Entscheidung}}$$

Für die Mehrfachbedingungsüberdeckung wird für jede Entscheidung eine Wahrheitstabelle mit allen möglichen Wahrheitswertkombinationen der atomaren Bedingungen erstellt (vgl. Tabelle 2.1), aus denen sie zu zusammengesetzt ist. Die in den Logdateien gespeicherten Auswertungen werden anschließend mit den Einträgen in der Tabelle verglichen und bei einer Übereinstimmung als überdeckt markiert.

Der Überdeckungsgrad der Mehrfachbedingungsüberdeckung lässt sich damit leicht durch den Quotienten aus der Anzahl der markierten (= überdeckten) Einträge und der Anzahl der geforderten Einträgen berechnen. Bei vollständiger Evaluation der Bedingungen entspricht die geforderte Anzahl 2^n, und bei unvollständiger Evaluation entspricht die geforderte Anzahl der Anzahl der tatsächlichen Testklassen. Außerdem werden nicht ausgewertete atomare Bedingungen als Wahr und Falsch (Wildcard) gleichzeitig angenommen.

Methodenüberdeckung und Overloading
Wird eine Methode aufgerufen, werden anhand des gleichen Bezeichners ebenfalls alle überladenen Methoden erkannt. Dazu wurden vom Schreiber die Methoden `enterMethod( int )` und `leaveMethod( int )` an der entsprechenden Stelle der Methodendefinition eingefügt, die jedes Betreten und Verlassen einer Methode protokollieren. Das Genannte gilt ebenfalls für alle Konstruktoren. Die Berechnung des Überdeckungsgrads für die Methodenüberdeckung erfolgt gemäß Kapitel 2.6.

Kapitel 5

Ausblick

In den vorausgehenden Kapiteln dieser Arbeit wurden verschiedene Konzepte zur Übertragung der klassischen Bedingungsüberdeckungskriterien auf die Programmiersprache Java vorgestellt. Außerdem wurde eine mögliche Implementierung dieser Konzepte in einem Werkzeug gezeigt und welche Techniken dazu angewendet wurden. Obwohl das Werkzeug den Anforderungen der Aufgabenstellung genügt, existieren noch weitere Optimierungs- und Erweiterungspotenziale. Davon möchte ich einige hier ansatzweise vorstellen.

5.1 Statische Überprüfung der Erfüllbarkeit

Eine nützliche Erweiterung des Werkzeugs wäre die statische Überprüfung der Erfüllbarkeit einer Bedingung. Ein naiver Ansatz wäre, alle möglichen Wahrheitswertkombinationen der atomaren Bedingungen einer zusammengesetzten Bedingung durchprobieren und das Ergebnis der Entscheidung zu beachten. Liegt eine Kontradiktion vor, würde die Entscheidung unabhängig von den atomaren Bedingungen immer zu Falsch ausgewertet werden. Bei einer Tautologie würde immer das Ergebnis Wahr lauten. Dazu könnte zum Beispiel die bereits für die modifizierte Bedingungs-/Entscheidungsüberdeckung vom Analysierer erstellte Wahrheitstabelle herangezogen werden, in der alle Kombinationen der atomaren Bedingungen und das Ergebnis der Entscheidung gespeichert wurden. Dieser „Bruteforce„-Ansatz wäre zwar leicht zu implementieren, aber die Laufzeit würde exponentiell (2^n) zur Anzahl der atomaren Bedingungen steigen, was den Einsatz auf Bedingungen mit wenigen atomaren Bedingungen beschränkt.

Ein weiterer Ansatz wäre die Verwendung eines Entscheidungsdiagramms („Reduced Ordered Binary Decition Diagram“), durch das die Auswertung einer Bedingung grafisch veranschaulicht werden kann. Ein ROBBD ist ein geordneter, gerichteter azyklischer Graph mit den zwei Senken (= Blätter) 1 und 0, die einer Auswertung der Entscheidung zu Wahr und Falsch entsprechen. Eine Tautologie entspräche einem ROBBD, bei dem alle Kanten auf die 1-Senke gerichtet sind. Im Gegensatz dazu würden bei einer Kontradiktion alle Kanten zur 0-Senke zeigen. Für die Darstellung eines ROBBD existiert für Java das JADE-Package („Java Decision Diagram“) von Jochen Römmler und Rolf Drechsler von der Universität Bremen. Dieses Package erzeugt im

Speicher einen ROBBD, der es ermöglicht, eine Bedingung auf Tautologie und Kontradiktion zu prüfen. Für weiterführende Information verweise ich auf [4].

Der letzte Ansatz, den ich hier vorstellen möchte, ist die Umformung der Bedingung in die konjunktive Normalform zur Überprüfung auf eine Tautologie bzw. die Umformung in die disjunktive Normalform zur Überprüfung auf eine Kontradiktion. Diese Umformung ist zweckmäßig, da aufgrund der Struktur der Normalformen eine Erkennung erleichtert wird. Für diese Umformung könnte man ebenfalls das Tool ANTLR mit einer Grammatik für bool'sche Ausdrücke verwenden, mit der ein AST für eine Bedingung erstellt werden kann. Dieser AST kann anschließend in der entsprechenden Normalform zurückgeschrieben werden.

5.2 Weitere mögliche Erweiterungen

Möchte man noch mehr Informationen über die Bedingungen eines Java-Programms erfahren, wäre es möglich, das Werkzeug durch weitere Überdeckungsgrade zu erweitern. Zum Beispiel kann der Überdeckungsgrad pro Package in das Werkzeug eingebunden werden: Zuerst müssen die Bezeichner der Packages und eine eindeutige ID für die spätere Verwendung durch den Analysierer in der Symboltabelle gespeichert werden. Das Speichern selbst erfolgt auf die gleiche Weise, wie es bereits bei den Methoden/Konstruktoren und Klassen beschrieben wurde, d.h., dass die Bedingungen die zwischen zwei Package-Bezeichnern bzw. zwischen einen Package-Bezeichner und dem Ende der Symboltabelle stehen, eindeutig dem oberen Package zugeordnet werden. Anschließend kann der Analysierer so angepasst werden, dass er den Überdeckungsgrad pro Package berechnen kann. Dazu können die gleichen Techniken verwendet werden, die für die Berechnung pro Methode/Konstruktor bzw. pro Klassen benutzt werden.

Weiterhin nimmt das Werkzeug keine Vorauswertung der Bedingungen vor. So wird im folgenden Beispiel nur das Ergebnis der bool'schen Variable *e* überprüft, aber nicht die einzelnen Teilbedingungen, aus denen sie zusammengesetzt ist:

```
...
boolean a, b, c, d;
...
boolean e = ( a || b ) && ( c || d );
if ( e ) {...}
```

Um diese Vorauswertung vorzunehmen, wäre ein naiver Ansatz, die Variable *e* durch den zugewiesenen Ausdruck zu ersetzen und dann wie gewohnt zu Instrumentieren:

```
...
boolean a, b, c, d;
...
if ( ( a || b ) && ( c || d ) ) {...}
```

Diese Anpassung wäre aber nicht im Sinne des Programmierers, da er wahrscheinlich einen guten Grund hatte, für den Vergleich eine eigene Variable zu verwenden, statt die Bedingungen

selbst einzusetzen.

Eine weitere Ansatz wäre deshalb, jede Bedingung, die an eine Variable zugewiesen wird, als explizite Bedingung zu betrachten (siehe Kapitel 3.1) und dementsprechend zu behandeln. Zum Beispiel:

```
...
boolean a, b, c, d;
...
boolean e = LogB( ( a || b ) && ( c || d ) );
if ( e ) {...}
```

Dieses Vorgehen ändert den Code nicht so radikal wie der bereits vorgestellte, ersetzende Ansatz, so dass dieses Vorgehen besser geeignet ist.

5.3 Mögliche Optimierungen und Anpassungen

Bei der Entwicklung der Algorithmen für die modifizierte Bedingungs-/Entscheidungsüberdeckung und die Mehrfachbedingungsüberdeckung der Analysierer-Komponente wurde der Fokus auf Zweckmäßigkeit und einfache Implementierung statt auf Performanz gelegt. So könnte die Laufzeit des Algorithmus für die Bestimmung der Auswertungspaare der modifizierten Bedingungs-/Entscheidungsüberdeckung weiter optimiert werden, z.B. durch die Verwendung eines binären Suchalgorithmus.

Ein weitere nennenswerte Anpassung des Werkzeugs wäre die Beibehaltung der Kommentare aus dem Ausgangsquelltext. Die verwendete Grammatik verwirft beim Erzeugen des ASTs gänzlich die Kommentare, da diese ursprünglich für das Erzeugen von (Cross-)Compilern erstellt wurde. In der Grammatik existieren bereits Regeln für die Behandlung der Kommentare, es fehlen aber die Aktionen die dafür sorgen, dass die Kommentare in den AST aufgenommen werden. Diese Regeln könnten verändert werden, damit die Kommentare in den AST übernommen werden. Dann muss ebenfalls die Grammatik für den Schreiber angepasst werden, so dass die Kommentare auch ein in den instrumentierten Quelltext geschrieben werden. Geht man davon aus, dass der instrumentierte Quelltext jedes Mal nur zu Testzwecken aus dem Ausgang-Quelltext erzeugt wird, ist diese Erweiterung nicht unbedingt notwendig.

Anhang A

CountWords.java Beispielquelltext

```
public class TestCountWords {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("#words␣in␣'" + args[0] + "
               ':␣" + countWords(args[0]));
        }
    }
    public static int countWords(String text) {
        int words = 0, chars = 0;
        char ch;
        text = text.trim();
        if (text.length() > 0) words++;
        for (int i = 0; i < text.length(); i++) {
            ch = text.charAt(i);
            if ((ch >= 'A') && (ch <= 'Z') || (ch >= 'a')
              && (ch <= 'z') || (ch == '␣')) {
                chars++;
            } else {
                return -1;
            }
            if ((ch == '␣') || (ch == '\t') || (ch == '\n')
              ) {
                words++;
            }
        }
        return words;
    }
}
```

Anhang B

Wichtige Klassen

B.1 LoggerTypes.java Klasse

Der folgende Quelltext zeigt die LoggerTypes-Klasse, in der die IDs der besonderen Ereignisse als Konstanten definiert wurden:

```
public class LoggerTypes {
    // define constants for truth values
    public static final long UNKNOWN_VALUE     = 2;
    public static final long TRUE_VALUE        = 1;
    public static final long FALSE_VALUE       = 0;
    public static final long NULL_VALUE        = -1;

    // define special ids
    public static final long ENTER_SWITCH      = -2;
    public static final long LEFT_SWITCH       = -3;
    public static final long CATCH_EXCEPTION   = -4;
    public static final long START_EXPRESSION  = -5;
    public static final long END_EXPRESSION    = -6;
    public static final long HANDLE_EXCEPTION  = -7;
    public static final long ENTER_METHOD      = -8;
    public static final long LEFT_METHOD       = -9;
}
```

Abbildung B.1: `LoggerTypes.java`-Klasse

B.2 Symbol-Klasse

Der folgende Quelltext zeigt die Symbol-Klasse, die ein Symbol in der Symboltabelle repräsentiert:

```
public class Symbol {
    public long id;
    public long row;
    public String type;
    public String exp;
    public String notes;
    public long methodId;        // the Id of the method, to
        which the symbol belongs to
    public long classId;         // the Id of the class, to
       which the symbol belongs to
    public Vector conditionId;   // the Ids of the combined
       condition the symbol/condition belongs to
    public long belongsToId;     // to which overall
       condition, switch or polymorphic variable belongs the
        symbol
    public int trueCount = 0;    // only used temporary by
       the critera-methods
    public int falseCount = 0;   // -"-
    public int unknownCount = 0;// -"-
    ...
}
```

Anhang C

Die API

In diesem Abschnitt erfolgt eine kurze Beschreibung der Programmierschnittstelle (API).

C.1 Die Instrumenter-Klasse

Die Instrumenter Klasse stellt Funktionen für die Instrumentierung des Java-Quelltexts bereit.

```
public void instrumentFile( File inFile, File outFile ) throws IOException,
FileNotFoundException, TokenStreamException, RecognitionException;
```

Instrumentiert den in *inFile* angegebenen Quelltext und schreibt ihn nach *outFile*. Dabei wird auch die Symboltabelle erstellt.

```
public Instrumenter( FileOutputStream symbolTable );
```

Einziger Konstruktor der Instrumenter-Klasse. Erwartet einen FileOutputStream, in den die Symboltabelle geschrieben wird.

```
public AST getAST();
```

Liefert den AST des Quelltexts.

```
public AST getInstrAST();
```

Liefert den instrumentierten AST des Quelltexts.

```
public static void showAST ( File f ) throws FileNotFoundException,
TokenStreamException, RecognitionException;
```

Stellt den AST zum Quelltext, spezifiziert durch f, grafisch dar.

```
public static void showAST ( AST a ) throws FileNotFoundException,
TokenStreamException, RecognitionException;
```

Stellt den AST a grafisch dar.

C.2 Die Analyzer-Klasse

Die Analyzer Klasse stellt Funktionen für die Analyse des instrumentierten Java-Quelltexts bereit. Dazu verwendet diese Klasse die Symboltabelle und eine oder mehrere Logdateien.

```
public void loadSymbolTable( String Filename ) throws IOException,
AnalyzerException;
```

Lädt die Symboltabelle, die für die Analyse benötigt wird.

```
public void printSymbolTable();
```

Gibt die Symboltabelle nach *stdout* aus.

```
public HashMap getSymbolTable();
```

Gibt die HashTabelle zurück, in der die Symbole aus der Symboltabelle gespeichert sind.

```
public void addLogfile( File f );
```

Fügt eine Logdatei zur Analyse hinzu.

```
public Vector getLogfiles();
```

Liefert eine Liste aller Logdateien, die bei der Analyse berücksichtigt werden.

```
public void removeLogfile( int index );
```
und
```
public void removeLogfile( File f );
```

Entfernt eine Logdatei.

```
public void clear();
```

Entfernt die Symboltabelle und alle Logdateien.

```
public CoverageDetails getOverallSCC() throws Exception;
```

Liefert Informationen über die einfache Bedingungsüberdeckung des ganzen Programms.

```
public CoverageDetails getOverallCDC() throws Exception;
```

Liefert Informationen über die Bedingungs-/Entscheidungsüberdeckung des ganzen Programms.

```
public CoverageDetails getOverallMMCC() throws Exception;
```

Liefert Informationen über die minimale Mehrfachbedingungsüberdeckung des ganzen Programms.

```
public MCDCCoverageDetails getOverallMCDC() throws Exception;
```
Liefert Informationen über die modifizierten Bedingungs-/Entscheidungsüberdeckung des ganzen Programms.

```
public MMCoverageDetails getOverallMCC() throws Exception;
```

Liefert Informationen über die Mehrfachbedingungsüberdeckung des ganzen Programms.

```
public CoverageDetails getOverallMC() throws Exception;
```

Liefert Informationen über die Methodenüberdeckung des ganzen Programms.

```
public CoverageDetails getOverallSC() throws Exception;
```

Liefert Informationen über die Switch-Case-Überdeckung des ganzen Programms.

```
public CoverageDetails getSCC( Symbol sym ) throws Exception;
```

Liefert Informationen über die einfache Bedingungsüberdeckung einer kombinierten Bedingung oder Entscheidung.

```
public CoverageDetails getCDC( Symbol sym ) throws Exception;
```

Liefert Informationen über die Bedingungs-/Enscheidungsüberdeckung einer kombinierten Bedingung oder Entscheidung.

```
public CoverageDetails getMMCC( Symbol sym ) throws Exception;
```

Liefert Informationen über die minimale Mehrfachbedingungsüberdeckung einer kombinierten Bedingung oder Entscheidung.

```
public MCDCCoverageDetails getMCDC( Symbol sym ) throws Exception;
```

Liefert Informationen über die modifizierte Bedingungs-/Entscheidungsüberdeckung einer Entscheidung.

```
public MMCoverageDetails getMCC( Symbol sym ) throws Exception;
```

Liefert Informationen über die Mehrfachbedingungsüberdeckung einer Entscheidung.

```
public CoverageDetails getMC( Symbol sym ) throws Exception;
```

Liefert Informationen über die Methodenüberdeckung einer Methode.

```
public CoverageDetails getSC( Symbol sym ) throws Exception;
```

Liefert Informationen über die Switch-Case-Überdeckung einer Switch-Case-Anweisung.

```
public CoverageDetails getCoverage( Vector symbols ) throws Exception;
```

Generische Berechnung der Überdeckung für eine Liste von Bedingungen. Wird von Methoden für die Bedingungsüberdeckungskriterien verwendet.

```
public MCDCCoverageDetails getMCDCCoverage( Vector symbols, Symbol sym )
throws Exception;
```

Generische Berechnung der modifizierten Bedingungs-/Entscheidungsüberdeckung für eine Liste von atomaren Bedingungen.

```
public MMCoverageDetails getMCCoverage( Vector symbols, Symbol sym )
throws Exception;
```

Generische Berechnung der Mehrfachbedingungsüberdeckung für eine Liste von atomaren Bedingungen.

```
public Symbol getSymbolById ( long id );
```

Liefert eine Symbol-Datenstruktur zurück, in der Informationen über das Symbol mit der Iddentifiaktion *id* gespeichert ist.

```
public Analyzer();
```

Der Konstruktor der Analyzer-Klasse.

Literaturverzeichnis

[1] Helmut Balzert, *Lehrbuch der Softwaretechnik 2*, Spektrum Akademischer Verlag, 1998

[2] http://www.heise.de/newsticker/meldung/54690, Stand 23.2.2006

[3] http://www.antlr.org

[4] http://www.informatik.uni-bremen.de/agra/doc/software/manual/, Stand 14.2.2006

[5] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, Stand 22.1.2006

[6] Peter Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, Spektrum Akademischer Verlag, 2002

[7] RTCA, Inc. , *Software Considerations in Airborne Systems and Equipment Certification*, 1992 Januar 2006

Index

Überdeckungsgrad, 9
    feingranular, 10
    grobgranular, 9
Überladen von Methoden, 19

Anreicherung
    automatische, 21
    explizite, 20

Bedingung
    atomare, 4
    explizite, 12
    implizite, 12
Bedingungs-/Entscheidungsüberdeckung, 7

Einfache Bedingungsüberdeckung, 5
Evaluation
    unvollständige, 4
    vollständige, 5

Gekoppelte Bedingungen, 8

Instrumentierung
    Anpassung der Laufzeitumgebung, 20
    durch Parser, 21
    Exception-Behandlung, 15
    expliziter Bedingungen, 12
    manuelle, 20
    Overloading, 19
    Polymorphie, 17
    Switch-Case-Anweisung, 14
    ternärer Operator, 13

LL(k)-Grammatik, 21

Mehrfachbedingungsüberdeckung, 9
Minimale Mehrfachbedingungsüberdeckung, 7
Modifizierte Bedingungs-/Entscheidungsüberdeckung, 8

Zweigüberdeckungstest, 2