Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Norbert Oster

Erlangen - 2006

Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung: 24.11.2006

Tag der Promotion: 29.01.2007

Dekan: . . . . . Prof. Dr.-Ing. Alfred Leipertz

Berichterstatter: . . . . . Prof. Dr. Francesca Saglietti

Prof. em. Dr. Hans Jürgen Schneider

Zusammenfassung

In dieser Arbeit wird ein Verfahren zur automatisierten Testdatengenerierung vorgestellt, das sowohl auf prozedurale als auch auf objekt-orientierte Programme anwendbar ist. Dabei werden während der Generierung die Testfälle derart optimiert, dass mit einer minimalen Anzahl von Testfällen eine maximale strukturelle Code-Überdeckung erreicht wird. Zur Verfolgung dieser beiden an sich gegenläufigen Ziele werden selbst-adaptive, multi-objektive Metaheuristiken (darunter insbesondere Evolutionäre Algorithmen) eingesetzt. Der Ansatz basiert auf einer vorausgehenden Phase zur automatischen Instrumentierung des Quellcodes, um relevante Informationen über den Kontroll- und Datenfluss während der Laufzeit aufzuzeichnen. Mittels der dadurch gewonnenen Erkenntnisse werden sukzessiv generierte Testdatenmengen schrittweise solange verbessert, bis die vorgegebenen Testziele erreicht sind. Abschließend wird die Effizienz der generierten Testdaten hinsichtlich ihres Fehlererkennungspotenzials durch Mutationstests ermittelt. Darüber hinaus wird der prozentuale Anteil der tatsächlich erzielten Überdeckung mittels statischer Kontroll- und Datenflussanalyse festgestellt. Das hier vorgestellte Verfahren erlaubt damit, den Aufwand zur Verifikation und Validierung komplexer, sicherheitskritischer Software deutlich zu verringern.

Das Dokument ist in sieben Kapitel gegliedert, welche von mehreren Anhängen gefolgt werden. Zunächst wird die Zielsetzung des Forschungsprojektes in Kapitel 1 motiviert, dessen Ergebnis hier vorgestellt wird. Anschließend wird in Kapitel 2 das Verfahren und dessen Einsatz sowohl bezüglich des Software-Entwicklungsprozesses als auch im Kontext vergleichbarer Ansätze eingeordnet. Kapitel 3 erläutert die notwendigen Grundlagen der strukturellen Teststrategien, die mit dem hier vorgestellten Verfahren unterstützt werden, während Kapitel 4 die dabei eingesetzten multi-objektiven Metaheuristiken skizziert. Anschließend werden die beiden Fachgebiete in Kapitel 5 zusammengeführt und das (zweistufige) Verfahren detailliert beschrieben. Kapitel 6 präsentiert und diskutiert experimentelle Ergebnisse, welche mit einer prototypischen Implementierung des Verfahrens namens .gEAr für die Programmiersprache JAVA™ gewonnen wurden. Schließlich bietet Kapitel 7 einen Ausblick auf potentielle Erweiterungen des Ansatzes.

Abstract

In this work a technique for the automated generation of test data is presented, which can be equally applied to both procedural and object-oriented software. During the generation the test cases are optimised in a way to achieve maximised structural code coverage with a minimised number of test cases. In order to cope with these inherently conflictive goals, self-adaptive multi-objective metaheuristics (among others evolutionary algorithms) are applied. The approach is based on a preliminary phase comprising an automatic instrumentation of the source code, aiming at recording relevant information about controlflow and dataflow during runtime. Using the insight gained hereby, the test sets are successively improved until the given testing goals have been reached. In a concluding phase the quality of the generated test data is assessed in terms of its fault detection capability by means of mutation testing. Additionally, the actual coverage (expressed as percentage of the entities to be covered) is determined by means of static analysis of the controlflow and the dataflow. The technique presented here allows to considerably reduce the effort required for verification and validation of complex, safety-critical software.

The document is structured in seven chapters, followed by several appendices. In the beginning the goals of the research project, the result of which is presented here, are motivated in chapter 1. In the following chapter 2 the technique and its intended purpose are classified according to the software development process and in the context of preliminary work. In chapter 3 the basics of structural testing strategies supported by the presented technique are introduced, while in chapter 4 the multi-objective metaheuristics applied are outlined. Subsequently, both topics are brought together in chapter 5, where the (two-step) procedure is described in detail. In chapter 6 experimental results are presented and discussed, as gained by means of a prototypical implementation of the technique in a tool named .gEAr for the programming language JAVA™. Finally, chapter 7 gives an outlook on possible extensions to the approach presented.

Meiner Frau Andrea, die darunter am meisten leiden musste,
sowie meinen Eltern Herta und Hermann, die mir alles ermöglicht haben.

Danksagung

Zur Durchführung und zum Gelingen dieser Arbeit haben viele Menschen beigetragen, die mich während der Jahre der Konzentration auf das Thema mit Rat und Tat unterstützt haben.

In diesem Sinne gilt mein Dank Frau *Prof. Dr. Francesca Saglietti*, welche mir die Ehre zuteil werden ließ, als erster Mitarbeiter in die Geschichte ihres neu gegründeten Lehrstuhls für Software Engineering einzugehen. Insbesondere danke ich ihr dafür, dass sie die Betreuung dieses Themas übernommen hat. Darüber hinaus erkenne ich Frau Prof. Saglietti an, dass das entwickelte Verfahren und damit die gesamte Dissertation erst aufgrund der ausgiebigen Besprechungen und fachlichen Ratschläge ihre abgerundete Form gefunden haben.

Ebenso dankbar bin ich Herrn *Prof. em. Dr. Hans Jürgen Schneider*, der trotz des Umfangs der Arbeit die zweite Begutachtung übernommen und zügig über die wohlverdienten Weihnachtsfeiertage durchgeführt hat. Mein Dank gilt auch Herrn *Prof. em. Dr. Fridolin Hofmann*, welcher den Vorsitz des Promotionsprüfungskollegiums angenommen hat. Bei Herrn *Prof. Dr. Gerd Häusler* bedanke ich mich für die Kooperation als fachfremdes Prüfungsmitglied und entschuldige mich für den kurzfristigen „Überfall“.

Ein herzliches Dankeschön gebührt allen Kollegen am Lehrstuhl für Software Engineering beziehungsweise am Institut für Informatik, die mich während der Erforschung des Themenkomplexes, der Implementierung des Werkzeugs •gEAr und dem Editieren der Dissertation begleitet haben. Danke *Brigid* für den Artikel, welcher den Einstieg ins Thema markiert hat. Danke *Ingeborg* für die „frische Luft“ und das „eine oder andere Komma“. Danke *Herbert* für den aufmunternden „Witz des Tages“ und die stets treffsichere Kritik. Danke *Jens* für die wertvollen Gespräche. Dank auch allen nicht namentlich erwähnten Kollegen, dass sie mir in den Vorträgen zugehört, wichtige Anmerkungen gemacht und wenn es eng wurde, mir auch mal Arbeit abgenommen haben.

Meinen tiefsten Dank möchte ich aus ganzem Herzen *meinen Eltern* und meiner *Schwester* sowie ganz besonders meiner Ehefrau *Andrea* aussprechen. Meine Eltern haben buchstäblich alles in der alten Heimat geopfert, um ihren Kindern stets die bestmöglichen Voraussetzungen zu schaffen (ich weiß, sie würden das gleiche jederzeit wieder tun) – Danke für die unermüdliche Unterstützung und den mitgegebenen Ehrgeiz. Am meisten in den fünf Jahren hat wohl Andrea gelitten: Meinen tiefempfundenen Dank für deine starke Schulter, wenn mich so manches Mal der Mut verlassen hat, und verzeihe mir die zeitweilige geistige und (besonders an späten Abenden und etlichen Wochenenden) auch physische Abwesenheit!

Vorwort

Betrachtet man die technologische Entwicklung der letzten Jahre, so stellt man fest, dass Software in immer größere Bereiche unseres Alltags dringt – meist unbemerkt, ab und zu jedoch mit teils katastrophalen Auswirkungen. Durch Software bedingte Unfälle kosten Hersteller oft viel Geld, eine gute Portion Image und nicht selten auch einigen Menschen das Leben. Wirft man einen Blick in die aktuelle Praxis der Softwareentwicklung, bekommt man den Eindruck, die Komplexität heutiger Softwaresysteme wachse zwar gleichmäßig mit der Erforschung rigoroser ingenieurmäßiger Verfahren zur Konstruktion und zum Nachweis der Zuverlässigkeit solcher Systeme, jedoch scheint es, als ob die Komplexität den Fortschritten im Software Engineering stets einen Schritt voraus ist.

Dieser Eindruck ist nur zum Teil gerechtfertigt. Sowohl industrielle als auch universitäre Forschungseinrichtungen haben eine Vielzahl unterschiedlicher Vorgehensmodelle, konstruktive und analytische Maßnahmen zur Verhinderung oder zumindest zur (rechtzeitigen?) Identifizierung von Problemfällen in der Software sowie zahlreiche Empfehlungen, wie man die Qualität des Software enthaltenden Produkts oder zumindest dessen Entwicklungsprozess zu verbessern vermag, entwickelt. Der oben genannte Eindruck entsteht vielmehr deshalb, weil die industrielle Praxis zwar immer größere und komplexere Produkte auf den stets „heiß umkämpften“ Markt bringt, ohne jedoch diese verbesserten Techniken aufzugreifen.

Der meistgenannte Grund hierfür ist der *Return on Investment* (RoI). Zugegeben, viele interessante Verbesserungsvorschläge für die Softwareentwicklung fordern zunächst einen teilweise beachtlichen Mehraufwand, der das Entwicklungsziel *jetzt* und meist *empfindlich* sowohl finanziell als auch zeitlich belastet. Leider sind die Schätzungen des möglichen Zugewinns an Sicherheit und Stabilität der Software noch sehr ungenau und beziehen sich eben auf *irgendwann* in der Zukunft. Verständlich, dass ein Auftraggeber lieber *jetzt* spart.

Die zusätzlich notwendigen Investitionen *vor* der Auslieferung der Software ließen sich oft spürbar senken, wenn die vorgeschlagenen Methoden automatisiert oder zumindest mit maschineller Hilfe abgewickelt werden könnten. Eine der Phasen, die die Softwareentwicklung teuer macht, ist die Testphase ganz gleich welcher Ausprägung. Verschiedene Ansätze, einzelne Aspekte zu automatisieren gibt es, jedoch ist die eigentliche Ermittlung der Testfälle (speziell der dazu notwendigen Testdaten) zur gezielten White-Box-Überdeckung wohl mit dem größten Aufwand verbunden. Die vorliegende Arbeit setzt genau hier an und zielt auf eine vollautomatische Generierung der Testdaten nach soliden Überdeckungskriterien für aktuelle Programmierparadigmen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Kapitel 1

Einleitung

„There's always one more bug."
Lubarsky's Law of Cybernetic Entomology

Ein Nachweis der Korrektheit eines Programmcodes mittels Testen ist nur möglich, indem die Software mit allen möglichen Eingaben in allen möglichen Ausführungsumgebungen ausgeführt wird. Lässt man einen Fall aus, kann man nie mit Verlass sagen, ob sich nicht ausgerechnet unter diesen Umständen ein Versagen offenbaren würde.

Man betrachte im folgenden Gedankenexperiment ein Programm, welches eine einzige Eingabe zu verarbeiten hat, nämlich eine 32 Bit lange Ganzzahl. Nimmt man idealisiert an, dass jede Ausführung des Programms im Mittel nur eine Sekunde dauert, es nur eine mögliche Umgebung gibt und das Ergebnis automatisch validiert werden kann (was in den seltensten Fällen überhaupt möglich ist), dann dauert die „Testphase" über 136 Jahre. Versucht man auf der anderen Seite die Korrektheit formal mittels mathematisch-logischer Beweise zu führen, scheitert man bereits an Modulen geringer Komplexität.

Also bleibt dem durchschnittlichen Tester nur noch ein Mittelweg: So intensiv zu testen wie das Budget es erlaubt. Doch auch hier kann man mit dem gleichen Aufwand mehr oder weniger gründlich vorgehen und damit entscheidend auf die Chance, Fehler aufzudecken, einwirken. So kann es vorkommen, dass man mit einer großen Testfallmenge keinen einzigen Fehler findet, z. B. indem nur ein Teil der Funktionalität angefordert oder nur wenig Programmcode zur Ausführung gebracht wird, jedoch mit einer kleinen Testfallmenge womöglich alle Fehler findet, wenn diese Testfälle so geschickt gewählt werden, dass sie das Programm so gründlich wie möglich ausführen.

1.1 Motivation I: Testaufwand und Testerfolg

Leider gibt es sehr viele, teilweise sehr verschiedene Entscheidungskriterien, wonach eine Testfallmenge unterschiedlich rigoros einzelne Aspekte eines Programms testet. Alle zu erfüllen, ist teilweise aus Zeitgründen, in realen Projekten fast immer aus Kostengründen, nicht möglich. Dass aber in der industriellen Softwareentwicklung offenbar nicht intensiv genug getestet wird,

belegt eine Vielzahl von Unfällen und diverse Statistiken, wie sie in [Lig02] zusammengefasst dargestellt werden. Demnach hat eine Untersuchung an 130 Projekten ergeben, dass eine durchschnittliche Software mit geschätzten 15 Fehlern pro 100 KLOC[1] ausgeliefert wird. Bedenkt man, dass z.B. die ERP[2]-Software SAP R/3 im Jahre 1999 ca. 50 Millionen Codezeilen enthielt, erscheint die Anzahl der verbliebenen Fehler doch beängstigend hoch.

Es ist daher nicht verwunderlich, dass bei größeren Projekten bis zu 37% des gesamten Entwicklungsaufwandes für die analytische Qualitätssicherung aufgebracht wird, während die Codierung andererseits lediglich mit durchschnittlich 12% zu Buche schlägt. Noch verzerrter wird diese Diskrepanz, wenn man den Wartungsaufwand ebenfalls den Anstrengungen der Qualitätssicherung zurechnet – schließlich handelt es sich weitgehend um die Beseitigung der Fehler, die nach der Freigabe noch verblieben sind – denn dieser beläuft sich durchschnittlich auf bis zum Doppelten des ursprünglichen Entwicklungsaufwandes.

Von besonderem Interesse sollten deshalb die Fehleranzahlen und die dadurch verursachten Korrekturkosten sein. So werden durchschnittlich lediglich ca. 25% der Fehler während des Entwicklertests, ca. 50% im Systemtest und weitere ca. 10% jedoch erst beim Kunden im Feld aufgedeckt. Diese Situation wird besonders dadurch verschlimmert, dass die Beseitigung von Produktfehlern, die erst nach Auslieferung zum Versagen der Software führen, im Mittel über 4-mal soviel kostet, als wenn diese während des Systemtests bereinigt worden wären und sogar das 12fache dessen, was deren Feststellung und Korrektur während des Entwicklertests an Kosten verursachen.

1.2 Motivation II: Folgen verbleibender Restfehler

Während das Versagen durchschnittlicher Büroanwendungen wie Textverarbeitung oder Tabellenkalkulation im Allgemeinen lediglich ein Ärgernis darstellt, das sich mit dem Einspielen eines Patches leicht beheben lässt, gibt es leider unzählige Beispiele großer, missionskritischer oder gar sicherheitsrelevanter Systeme, bei denen bereits kleine unentdeckt gebliebene Fehler Milliardenschäden verursachen und oftmals Menschenleben kosten.

Beispielsweise fanden zwei Prestigemissionen der Raumfahrt ein jähes Ende wegen an sich kleiner Nachlässigkeiten bei der Implementierung und insbesondere beim Testen [Vig05]. So verursachte die notwendig gewordene, kontrollierte Sprengung der „Mariner 1“-Venussonde 1962, nur 4,8 Minuten nach ihrem Start und somit lange vor Erfüllung ihrer Mission, einen finanziellen Schaden von ca. 18,5 Mio. $ weil im Fortran-Code der Steuerung ein Punkt anstelle eines Kommas gesetzt wurde. Dass versehentlich „`DO 5 K = 1. 3`“ anstatt „`DO 5 K = 1, 3`“ implementiert wurde, konnte auch der Compiler nicht feststellen, da der fehlerhafte Code syntaktisch richtig ist. Bedauerlicherweise wurde „`DO 5 K = 1. 3`“ korrekt als Zuweisung des Gleitkommawertes `1.3` an die Variable `DO5K` interpretiert und übersetzt, da „`.`“ laut Sprachdefinition ein Dezimaltrennzeichen darstellt. Erwünscht war stattdessen jedoch eine Schleife, welche den der Anweisung folgenden Codeblock bis zur Marke `5` mit Belegungen der Variable `K` mit den

[1] Kilo Lines Of Code, Tausend Programmzeilen
[2] Enterprise Ressource Planning

ganzzahligen Werten von 1 bis 3 durchlaufen sollte. Dass die semantisch deutlich unterschiedliche Interpretation auch in der Testphase nicht identifiziert wurde, ist hingegen unverzeihlich.

Nach einer zehnjährigen Entwicklungszeit, welche Gesamtkosten in Höhe von 5,5 Mrd. € verursacht hat, musste die Ariane 5 im Jahre 1996 nur 37 Sekunden nach ihrem Start zum Jungfernflug ebenfalls wegen eines Programmierfehlers gesprengt werden. In diesem Fall hatte man Teile des in Ada implementierten *Inertial Reference System (SRI)* der Vorgängerrakete Ariane 4 ohne Anpassung an die geänderten Eigenschaften der technischen Neuentwicklung übernommen. Darin wurde eine Gleitkommazahl, welche den Wert der horizontalen Geschwindigkeit in einer internen Darstellung repräsentiert, in eine Ganzzahl umgewandelt. Irrtümlicherweise hatte man die Ausnahmebehandlung für den Fall eines Überlaufs bei der Konversion wenige Programmzeilen vorher ausgeschaltet. Etwa 30 Sekunden nach dem Abheben erreichte die Ariane 5 in einer Höhe von 3,7 km eine Horizontalgeschwindigkeit, welche den Gültigkeitsbereich der Zielvariablen überstieg. Die Kopie des redundant ausgelegten Navigationssystems hatte leider den gleichen Fehler verursacht und sich infolge dessen bereits abgeschaltet. Die nun an den Zentralcomputer übermittelten Fehlerinfomationen wurden als Daten der Fluglage interpretiert, was zu einer unkontrollierbaren Richtungsänderung der Rakete geführt hat.

Während die beiden Beispiele aus der Raumfahrt lediglich von softwarebedingtem Versagen berichten, welche sprichwörtlich riesige Geldsummen zu Staub werden ließen, gibt es Fälle trauriger Berühmtheit, bei denen viele Menschen zu Schaden kamen und teilweise deshalb ihr Leben lassen mussten. So verstrahlte die Therac-25[3] zwischen Juni 1985 und Januar 1987 massiv sechs Personen. Da die Fehler zunächst nicht reproduziert werden konnten, hat es sehr lange gedauert, bis eine Untersuchung die Ursachen zu Tage gefördert hat. Nachdem die bisherigen Varianten dieses Bestrahlungsgerätes mit physikalischen Barrieren ausgestattet waren, die ein versehentliches Vermischen der Strahlungsarten und der -intensitäten verhinderten, konnte dieser nunmehr in Software implementierte Schutz ausgehebelt werden, wenn die das Gerät bedienende Person die vorab gewählten Einstellungen so schnell geändert hat, dass die Zustände des Gerätes und der innere Zustand des Programms nicht mehr übereinstimmten. Dabei kam es vor, dass die, gegenüber der dazu vorgesehenen Elektronenstrahlung, energiereichere Röntgenstrahlung fokussiert eingesetzt wurde.

1.3 Zielsetzung dieser Arbeit

Leider ist die Liste der Softwareversagen, die sowohl Hersteller mit hohen Regresskosten als auch Anwender mit teilweise tödlichen Unfällen belasten, lang genug um ganze Bände zu füllen – und täglich kommen weitere Meldungen hinzu. Trotz vielfältiger Verbesserungen des Softwareentwicklungsprozesses, von der Anforderungsspezifikation bis zum Feldtest, verbleiben nach der Testphase dennoch zu viele schwerwiegende Fehler im Programmcode. Zwar könnte Software prinzipiell korrekt erstellt werden, und da sie selbst nicht altert, bliebe sie es auch auf Dauer. Aber solange der Code von Menschen entwickelt wird, und sei es auch mit Hilfe automatischer Codegeneratoren, die zumindest die fehleranfällige manuelle Codierung obsolet machen, kön-

[3] „Medical Devices: The Therac-25“, appendix from Nancy Leveson: „Safeware: System Safety and Computers“

nen Fehler schon in früheren Phasen der Entwicklung auftreten beziehungsweise könnten die Generatoren fehlerhaft sein.

Um Produktfehler in der ausgelieferten Software wirklich sicher auszuschließen, bleibt nur der Weg über einen formalen Korrektheitsbeweis. Wer dies jedoch schon bei kleinen, einfachen Modulen versucht hat, kann bestätigen, dass diese Tätigkeit aufgrund ihrer abstrakten Natur selbst zu Fehlern verleitet und deshalb nicht immer zu einem brauchbaren Ergebnis führt. Viel schlimmer jedoch ist, dass der Weg über mathematische Beweise der Fehlerfreiheit schon für Systeme kleinerer Komplexität einen so hohen Aufwand erfordert, dass diese Lösung praktisch undurchführbar ist.

Da ein deterministischer Nachweis der Korrektheit bei der Komplexität heutiger Systeme nicht gelingt, bleibt nur noch der Gang über das Testen. Um die gleiche Aussagesicherheit wie im Falle eines Beweises zu erreichen, müsste das Programm mit allen möglichen Eingaben ausgeführt werden. Diese Vorgehensweise ist aufgrund der Größe des Eingaberaums im Allgemeinen undurchführbar, wie man schon aus dem einleitenden Beispiel von Kapitel 1 erkennen kann.

Also muss man sich zwangsweise auf eine Teilmenge aller denkbaren Ausführungsszenarien beschränken. Doch dies bedeutet zwangsweise, dass das Programm nicht wirklich in diesem Sinne so vollständig getestet wurde, dass später nicht doch eine Anforderung zu einem Versagen führt. Daher muss bei der Auswahl dieser Teilmenge besondere Sorgfalt aufgewendet werden, um möglichst geeignete Testfälle zu konstruieren – sprich, die vom Programm erwartete und gebotene Funktionalität sollte mit entsprechenden Ausführungen bedacht werden. Was das für die Auswahl der Testfälle bedeutet, hängt demnach vom Testkriterium ab, nach dem der Test durchgeführt werden soll.

In der Literatur gibt es eine Vielzahl solcher Testendekriterien, die im Kapitel 2.2 kurz skizziert werden. Dennoch findet man in der industriellen Praxis lediglich die Anwendung der einfachsten (minimalen) Teststrategien. Dies lässt sich leicht damit begründen, dass Testfallmengen, welche diejenigen Kriterien erfüllen, die einen höheren Testaufwand (mehr Testfälle) erfordern, im Allgemeinen sehr schwer von Hand zu finden sind. Insbesondere für Mitglieder der Qualitätssicherungsabteilung, die zwar die Spezifikation, nicht jedoch die Implementierung kennen, ist es zum Beispiel sehr aufwändig, einen Testfall zu finden, welcher einen ganz bestimmten Pfad im Programm durchläuft.

Dass Testfallmengen, die eine möglichst hohe Überdeckung des Programmcodes erzielen, im Allgemeinen auch eine höhere Fehleraufdeckungsquote aufweisen, können viele empirische Studien belegen. Diese Korrelation gab sogar Anlass zur Definition einer eigenen Teststrategie: dem sogenannten Mutationstesten (siehe Kapitel 3.6). Ein besonders anschauliches Argument liefern die Erfinder der Datenflusskriterien (Kapitel 3.4): *„Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed“* [RW85].

Selbst wenn schließlich die richtigen Testfälle für ein bestimmtes Kriterium gefunden wurden, schließt sich an die Ausführung dieser Testläufe ein weiterer, ebenso aufwändiger Prozess an: Die Überprüfung der tatsächlichen Ergebnisse beziehungsweise des realen Verhaltens der Software gegenüber dem Erwarteten.

Somit ergibt sich für das Qualitätssicherungsteam das Problem, die Software so gründlich

wie möglich zu testen, also nach einem vorgegebenen, möglichst rigorosen Kriterium hinreichend viele Testfälle zu wählen; gleichzeitig jedoch „unnötige“ Testfälle zu vermeiden – wobei *unnötig* hier so zu verstehen ist, dass diese Testfälle keinen zusätzlichen Beitrag zur Erfüllung des gewählten Kriteriums leisten, dennoch einen erheblichen Aufwand für ihre Überprüfung darstellen. Die ideale Lösung für diese widersprüchlichen Anforderungen wäre ein Werkzeug, welches die kleinste (im Weiteren „*optimale*“) Menge an Testfällen automatisch ermittelt, so dass diese das gewünschte Kriterium so vollständig wie möglich erfüllen.

1.4 Übersicht und Gliederung

Das im Rahmen dieser Arbeit vorgestellte Verfahren leistet genau das. Dabei wird die Testdatengenerierung als Such- bzw. Optimierungsproblem behandelt und mittels klassischer und evolutionärer Strategien gelöst. Um die Praxistauglichkeit dieser Methode zu evaluieren, wurde das Verfahren in einem Werkzeug namens •gEAr realisiert, welches exemplarisch Testfälle für objekt-orientierte JAVA™-Programme generiert. Dabei werden die Testdaten so ermittelt, dass sie, je nach Einstellung, ebenso einfache Kontrollfluss- wie insbesondere auch Datenflusskriterien erfüllen.

Nach einer einleitenden Motivation und Beschreibung der Zielsetzung des zu Grunde liegenden Forschungsprojektes in Kapitel 1 widmet sich die vorliegende Arbeit der Einordnung des Verfahrens entsprechend seines Anwendungskontextes sowie der Abgrenzung der beschriebenen Technik von bereits bestehenden Ansätzen ähnlicher Natur in Kapitel 2. Die zum Verständnis und leichteren Nachvollziehen des neuen Ansatzes erforderlichen Kenntnisse bezüglich Teststrategien einerseits und metaheuristischen Such- und Optimierungsverfahren andererseits werden in Kapitel 3 beziehungsweise Kapitel 4 behandelt. Die beiden vorangehend präsentierten Fachgebiete vereint anschließend Kapitel 5 zu diesem neuartigen Verfahren der automatischen Testdatengenerierung und -optimierung. Nachdem die Methodik sowohl generisch als auch im Hinblick auf eine Umsetzung in ein Werkzeug für die Programmiersprache JAVA™ vorgestellt wurde, widmet sich Kapitel 6 den mit diesem Werkzeug •gEAr erzielten Ergebnissen in exemplarischen Projekten. Abschließend bietet Kapitel 7 eine Übersicht des aktuellen Stands des Verfahrens sowie entsprechend anzustrebende Erweiterungen oder Verbesserungen in Form eines Ausblicks.

Kapitel 2

Einordnung und Abgrenzung

„If debugging is the process of removing bugs,
then programming must be the process of putting them in.“
Edsger Wybe Dijkstra (1930-2002), University of Texas

Die vorliegende Arbeit beschäftigt sich mit der automatischen Generierung optimaler Testdatenmengen zur Erfüllung vorgegebener struktureller Überdeckungskriterien. Um dieses Ziel zu erreichen, werden Such- und Optimierungsheuristiken eingesetzt, welche die Ausführung oder zumindest die Simulation des Testobjektes mit den jeweils ermittelten Testfällen erfordern. Dies bedeutet somit, dass entweder der Programmcode in einer ausführbaren Form oder zumindest ein ausreichend detailliertes Modell des zu entwickelnden Systems vorliegen muss. Deshalb kann die hier vorgestellte Art der Testfallerzeugung typischerweise erst nach Abschluss der Implementierungsphase genutzt werden. Da bei der Komplexität heutiger Softwaremodule bzw. -systeme ein relativ großer Eingaberaum zu untersuchen ist, somit auch sehr viele Eingabekombinationen bewertet werden müssen, deren Ausführungszeit entscheidend von der Größe des zu testenden Moduls abhängt, ist dieses Verfahren im Wesentlichen eher für den sogenannten Modultest oder den Integrationstest geeignet – eine Eigenschaft die allen sogenannten white-box-Testverfahren inhärent ist.

In diesem Kapitel werden zunächst typische Softwareentwicklungs- und Testprozesse vorgestellt, um die Zuordnung der in dieser Arbeit beschriebenen Methode nachvollziehbar einzugrenzen. Abschließend wird das hier vorgestellte Verfahren zur automatischen Testdatengenerierung im Lichte bestehender Ansätze eingeordnet und von diesen abgegrenzt.

2.1 Softwareentwicklungsprozesse

Während kleine Programme oder Komponenten, meist von einem einzelnen Programmierer geschrieben, unter Umständen noch ohne einen dedizierten Prozess entwickelt werden können, hat die „Software-Krise“ in der zweiten Hälfte des 20. Jahrhunderts die Notwendigkeit einer strukturierten Herangehensweise an die Entwicklung moderner hochkomplexer Systeme ins Bewusstsein gerufen. Seither wurde für die verschiedensten Arten von Softwaresystemen, von der

einfachen Textverarbeitung bis hin zu eingebetteten reaktiven Steuerungs- und Überwachungssystemen, eine Vielzahl entsprechender Entwicklungsprozesse, Verifikation- und Validierungsverfahren sowie geeignete Werkzeuge definiert, beziehungsweise umgesetzt. Aufgrund dieser Veränderung der Softwareentwicklung von einer rein handwerklichen Programmierung hin zur Nutzung wissenschaftlicher Erkenntnisse ist eine eigene Disziplin namens *Software Engineering* entstanden mit dem Ziel, moderne Softwaresysteme mittels ingenieurwissenschaftlicher Methoden schneller und kostengünstiger zu entwickeln und zugleich sicherer zu machen.

Die Ursachen für die zunehmende Komplexität, mit der sich heutige Softwareunternehmen konfrontiert sehen, sind vielfältig. Einerseits hat der Fortschritt in der Hardware-Entwicklung den Weg für umfangreiche Software mit großer Funktionsvielfalt und breitem Einsatzzweck freigemacht; dementsprechend sind auch die Anforderungen und Erwartungen der zukünftigen Benutzer gestiegen. Andererseits erzwingt auch der gestiegene Wettbewerb unter den Beratungs- und Entwicklungsunternehmen jedes einzelne Softwarehaus zu einem stark wirtschaftlich orientierten Handeln, was sich nicht selten in große, meist weltweit verteilte und sowohl sprachlich als auch fachlich inhomogene Teams niederschlägt. Einen Beitrag dazu, mit dieser schwierigen Situation einfacher zurechtzukommen, stellt die Ausrichtung der Softwareentwicklung an standardisierte, bei Bedarf dediziert angepasste Prozessemodelle dar [Bal97].

Zu den wohl bekanntesten und sich früh stark verbreitenden Varianten gehört das sogenannte *Wasserfallmodell*. Es gliedert sich grob in die aufeinanderfolgenden Phasen *Anforderungsermittlung/Spezifikation*, *Design*, *Implementierung* und *Integration* (allesamt vor der Auslieferung des Softwareproduktes) sowie die sich daran anschließenden Phasen *Wartung* und *Ablösung*, welche im Folgenden noch genauer beschrieben werden. Jede Phase wird dabei mit einem Verifikations- beziehungsweise Validierungsschritt abgeschlossen, um die Vollständigkeit und Korrektheit der in der entsprechenden Phase erstellten Dokumente zu sichern.

Da den für die Entwicklung sicherheitsrelevanter Softwaresysteme zentralen Aspekten der Verifikation und Validierung (kurz *V&V*) im Wasserfallmodell nicht der angemessene Stellenwert eingeräumt wird, entstand Ende der 70er Jahre das von Barry Boehm 1979 veröffentliche „*V-chart*“ [Boe79]. Zu einem ähnlichen Ergebnis kamen auch die vom Bundesministerium für Verteidigung 1986 gestarteten Projekte zur Ermittlung eines geeigneten Entwicklungsprozesses, welche schließlich 1993 unter der Federführung der *Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung* eine einheitliche Version des sogenannten *V-Modell (des Bundes)* hervorgebracht haben. Um neuen Softwareentwicklungsansätzen Rechnung zu tragen, darunter zum Beispiel dem objekt-orientierten Paradigma, wurde dieses Prozessmodell überarbeitet und 1997 in das *V-Modell 97* überführt. Im Februar 2005 wurde schließlich dieses V-Modell 97 durch eine erneut stark überarbeitete Variante namens *V-Modell XT* (eXtreme Tailoring) ersetzt. Das V-Modell XT ist seit dem 04.11.2004 im militärischen Bereich, im Bereich der Bundesverwaltung und einiger Länderverwaltungen verbindlich, was aus einer Empfehlung des Interministeriellen Koordinierungsausschusses (IMKA) an die Behörden der Bundesverwaltung hervorgeht, die die Anwendung des *V-Modell XT* für neu zu entwickelnde Systeme vorschreibt.

Abbildung 2.1 zeigt die Grundstruktur eines typischen *V-Modells* und die wichtigsten Phasen beziehungsweise die dabei entstehenden Dokumente, die bei allen Prozessmodellen vom *V-chart* bis zum *V-Modell XT* im Wesentlichen vergleichbar sind. Die Besonderheit des V-Modells ist

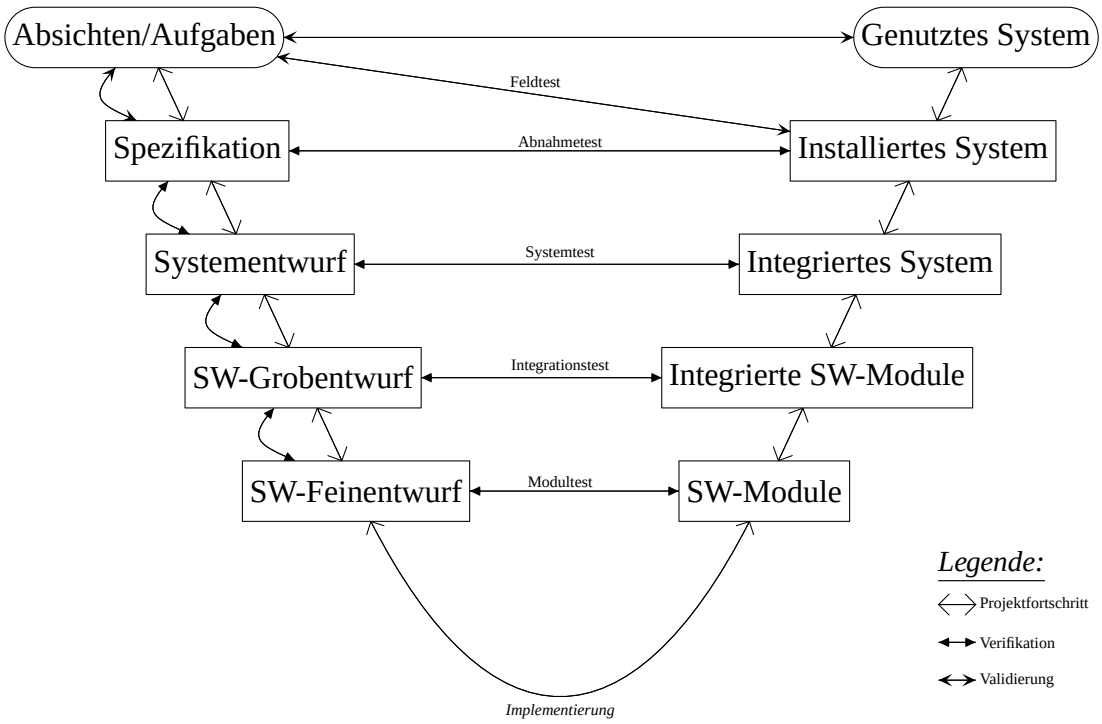


Abbildung 2.1: V-Modell (Hauptknoten: entstehende Produkte)

die der V&V zugestandenen Bedeutung, welche durch den rechten Ast (oder wie hier als Verbindungslinien zwischen den Produkten entlang der beiden Äste) explizit dargestellt ist. Da die Unterstützung dieser Phasen im Fokus dieser Arbeit liegt, geht das Kapitel 2.2 genauer darauf ein. Der linke Ast des V-Modells repräsentiert die einzelnen Schritte zunehmender Verfeinerung im Verlauf der Softwareentwicklung und stimmt weitgehend mit dem Wasserfallmodell überein:

1. *Spezifikation:* Jede strukturierte Softwareentwicklung beginnt mit der sogenannten *„Requirements Engineering“*-Phase. Diese lässt sich in die folgenden Teilschritte untergliedern, welche iterativ solange ausgeführt werden, bis ein Spezifikationsdokument (Pflichtenheft nach DIN 69905) ermittelt wurde, welches sowohl dem Auftraggeber als auch dem Softwareentwickler als Vertragsgrundlage angemessen erscheint:

   - *Requirements elicitation:* Ziel dieses Schrittes ist zunächst die Ermittlung einer Problemdefinition als Antwort auf die Frage *„wozu?“*, also das zu lösende Problem kennenzulernen, um eine gemeinsame Diskussionsbasis und einen gemeinsamen Wortschatz zwischen Kunde und Entwickler zu etablieren. Darauf aufbauend kann anschließend eine umfassende Beschreibung von Anforderungen an die Entwicklung des Softwareproduktes (*„was?“*) zusammengestellt werden, welche im Wesentlichen *funktionaler* (z.B. Dienste/Funktionalität), *qualitativer* (z.B. Zuverlässigkeit, Effizienz, Portabilität), *systembezogener* (z.B. bestehende Hardwareumgebung) oder *prozessbezogener* (z.B. Zeit, Kosten) Natur sind.

- *Requirements analysis:* Stehen die Anforderungen soweit fest, sollten sie einer möglichst umfassenden Analyse unterzogen werden. Dabei sollten sie insbesondere hinsichtlich Korrektheit (im Sinne des Auftraggebers), Vollständigkeit (z.B. ob alle möglichen Zustände/Aktionen erfasst wurden), Sachgerechtheit, Konsistenz (keine widersprüchlichen Eigenschaften gefordert) und Machbarkeit untersucht werden.
- *Requirements specification:* Das Endergebnis der Problemdefinition ist eine „Spezifikation“, also eine vollständige Beschreibung des zu entwickelnden Systems. Da sie einerseits die Basis für die Design-Phase, andererseits aber auch die Verhandlungsgrundlage für den Vertrag zwischen Auftraggeber und Softwareentwickler ist, sollte sie idealerweise möglichst formal (z.B. mittels mathematisch begründeter Sprachen wie Lotos oder Petri-Netze) oder zumindest semi-formal (z.B. UML-use-cases, Attempto Controlled English) erstellt sein, da informale Sprachen (wie alle natürlichen Sprachen) meist zuviel Raum für Mehrdeutigkeiten erlauben.

2. *Systementwurf:* Nachdem die Anforderungen in der Spezifikation vertraglich festgehalten wurden, kann das ausführende Unternehmen im Rahmen des Systementwurfs die Systemarchitektur entwickeln. Diese legt fest, welche bereits beim Auftraggeber vorhandenen Komponenten auf welche Weise weiterverwendet werden können sowie welche neuen Funktionalitäten in Hardware beziehungsweise in Software zu entwickeln sind.

3. *Softwareentwurf:* Ist die Systemarchitektur aufgestellt, kann die Softwareerstellung von der Hardwareentwicklung abgekoppelt und getrennt verfolgt werden.

   - *Grobentwurf:* Im Rahmen des Softwaregrobentwurfs wird eine Softwarearchitektur entwickelt, die eine geeignete Zerlegung des Ganzen in kleinere, nicht (sinnvoll) weiter zerlegbare Einheiten (den Komponenten) widerspiegelt. Bei dieser funktionalen Dekomposition geht es um die Beantwortung der Frage, *„wie?“* die Software aufgebaut sein soll. Entscheidend ist hier eine möglichst präzise Beschreibung der Schnittstellen zwischen den Komponenten und deren Abhängigkeiten, um den späteren Integrationstest geeignet zu unterstützen.
   - *Feinentwurf:* Stehen die Dienste fest, die jede Komponente jeweils zu erbringen haben, so können diese im Rahmen des Feinentwurfs nun detailliert beschrieben werden.

4. *Implementierung:* Ist der Feinentwurf vollständig und formalisiert, so können moderne Entwicklungswerkzeuge zum Teil bereits ganze ausführbare Codeabschnitte oder zumindest manuell zu ergänzende Gerüste automatisch generieren.

Ist die Implementierung der einzelnen Module abgeschlossen, können sie nach und nach zu einem kompletten Softwarepaket (*Integrierte SW-Module*) zusammengesetzt werden. Anschließend wird die parallel entwickelte Hardware ebenfalls einbezogen und das gesamte Softwaresystem (*Integriertes System*) getestet. Nach Bestehen dieses Tests beim Entwickler wird das System beim Auftraggeber installiert, einem *Abnahmetest* unterzogen und im Falle des Bestehens in den

Produktivbetrieb überführt. Im Laufe des Betriebes werden im Rahmen der *Wartung* Modifikationen am System vorgenommen. Dabei unterscheidet man folgende Arten der Wartungseingriffe:

- *corrective maintenance:* Behebung von in der Anwendung gefundenen Fehlern
- *adaptive maintenance:* Anpassung an neue Betriebsumgebung (z.B. bei Einführung neuer Hardware oder eines neuen Betriebssystems)
- *perfective maintenance:* Anpassung an neue Benutzeranforderungen (z.B. durch Hinzufügen neuer Funktionalität)
- *preventive maintenance:* Vorbeugende Maßnahmen zur Erleichterung späterer Wartungseingriffe (z.B. Aktualisierung der Dokumentation)

Der Softwarelebenszyklus endet schließlich mit der *Ablösung*, bei der das entsprechende Softwaresystem außer Betrieb genommen und eventuell durch ein neues System ersetzt wird. Ein wichtiger Schritt dabei ist die Überführung der mit der alten Software erstellten, verarbeiteten und gespeicherten Daten ins neue System. Dabei ist es durchaus denkbar, dass die alte und die neue Software in einem Parallelbetrieb nebeneinander eingesetzt werden, zum Beispiel solange die neue Applikation noch nicht ausreichend zuverlässig oder vollständig ist, beziehungsweise noch nicht alle Daten übertragen wurden.

Neben Wasserfall- und V-Modell gibt es eine Reihe weiterer Prozessmodelle beziehungsweise Modellvarianten. Darunter sind das *Inkrementelle Modell* und das *Spiralmodell* als die wichtigsten Vertreter zu nennen. Ersteres ermöglicht das frühzeitige Bereitstellen eines funktionsfähigen und vorzeigbaren Prototyps des zu entwickelnden Softwaresystems, indem einzelne und voneinander unabhängige Funktionalitäten nach und nach vollständig entwickelt und integriert werden; wobei die Umsetzung einer Funktionalität wiederum einem der vielfältigen Prozessmodellen folgen kann. Das *Spiralmodell* erfordert jeweils eine Risikoanalyse des gesamten Entwicklungsprojektes nach Abschluss jeder Phase beziehungsweise nach Fertigstellung einzelner Inkremente, womit Fehlentwicklungen wie Zeit-, Funktions- und Budgetuntreue rechtzeitig erkannt werden können.

2.2 Analytische Qualitätssicherungsverfahren

Um Software hoher Qualität zu erstellen, bedarf es einer kontinuierlichen, den gesamten Entwicklungsprozess begleitenden Überprüfung der in jedem Schritt erzeugten Zwischenprodukte. Dabei unterscheidet man die verschiedenen Prüfstrategien zunächst grob in zwei Klassen:

- **Verifikation:** Unter *Verifikation* versteht man *„die Überprüfung der Übereinstimmung zwischen einem Software-Produkt und seiner Spezifikation“* [Bal97]; allgemeiner ausgedrückt handelt es sich dabei um die *„Überprüfung, ob die Ergebnisse einer Entwicklungsphase die Anforderungen zu Beginn der Phase erfüllen“* [Lig02]. Wird also bei einem Modultest überprüft, ob die tatsächliche Implementierung die vom Softwarefeinentwurf geforderten Eigenschaften erfüllt, so handelt es sich dabei um eine Verifikation.

- **Validierung:** Im Rahmen der *Validierung*, oft auch *Validation* genannt, wird überprüft, *„ob ein Software-Produkt die Anforderungen und Bedürfnisse des Benutzers befriedigt„* [Lig02]. Allgemeiner ausgedrückt, wird dabei *„die Eignung bzw. der Wert eines Produktes bezogen auf seinen Einsatzzweck“* [Bal97] untersucht. Überprüft der Auftraggeber zum Ende der Spezifikationsphase das Spezifikationsdokument hinsichtlich funktionaler Vollständigkeit, so handelt es sich dabei um eine der Aktivitäten der Validierung.

Demnach geht die Verifikation der Frage nach, „ob ein Produkt korrekt im Sinne seiner Spezifikation entwickelt wurde“, während die Validierung eine Antwort auf die Frage sucht, „ob das richtige Produkt im Sinne des zukünftigen Benutzers entwickelt wurde“.

2.2.1 Klassifikation nach Phasen

Durchläuft man den Entwicklungsprozess in chronologischer Reihenfolge, so begegnet man der ersten Qualitätssicherungsmaßnahme bereits während der Anforderungsanalyse. Ziel dieser Validierungsaktivitäten ist es, eine vollständige, widerspruchsfreie und im Sinne des zukünftigen Nutzers der Software korrekte Zusammenstellung der Anforderungen zu erreichen. Jede Abweichung des in der Spezifikation tatsächlich beschriebenen Produktes von dem vom Auftraggeber eigentlich gewünschten und benötigten Produkt führt unweigerlich zu Fehlern, welche typischerweise erst nach der vollständigen Entwicklung und der Auslieferung an den Kunden entdeckt werden und daher sehr kostspielig sind. Gelingt es in der Anforderungsphase, formale Modelle (zum Beispiel in Form von Petri-Netzen) wichtiger Aspekte des zukünftigen Softwaresystems zu entwickeln, so können diese bereits hinsichtlich wichtiger Eigenschaften automatisiert überprüft werden. So können *Model Checking*-Werkzeuge auch sehr große Modelle mühelos auf Verklemmungsfreiheit und Erreichbarkeit aller erwünschten Zustände überprüfen. Wichtig in diesem Zusammenhang ist, dass auch die Untersuchung der Modelle auf Erfüllung problemspezifischer Eigenschaften möglich ist, welche orthogonal zu den funktionalen Anforderungen (z.B. in temporaler Logik) formuliert sind, was daher eine echte Validierung darstellt. Der besondere Vorteil solcher Model Checking Verfahren ist, dass im Falle eines entdeckten Fehlers die Lokalisierung der Ursache stark unterstützt wird.

Folgt man den weiteren Schritten im Softwareerstellungsprozess nach dem V-Modell aus Abbildung 2.1, so gibt es bis zur Implementierung an der unteren Spitze des Vs lediglich Verifikationsaktivitäten, da der Auftraggeber diese Schritte typischerweise nicht mehr begleitet. Ehe eine der Phasen als abgeschlossen betrachtet werden kann, sollten die in der entsprechenden Phase erstellten Dokumente gegenüber den Anforderungen aus der vorhergehenden Phase geprüft werden – gegebenenfalls ist auch eine Rückkehr zur vorangegangenen Phase denkbar, um eventuell das dortige Produkt zu überarbeiten, falls sich die von Phase zu Phase erfolgenden Verfeinerungen der Entwurfsdokumente als ungeeignet herausstellen sollten. Da bis zur Implementierung noch kein Code vorliegt, eignen sich hier insbesondere manuelle statische Prüfmethoden wie *Inspektionen*, *Reviews* oder *Walkthroughs* [Bal97].

Wird das Softwaresystem strukturiert anhand eines phasengetriebenen Vorgehens wie dem V-Modell entwickelt, so vereinfachen sich die einzelnen Prüfschritte erheblich. Ist die Implementierungsphase angelaufen, so entstehen nach und nach die einzelnen Einheiten und Module. Eine

Einheit (Unit) ist dabei „die kleinste sinnvoll unabhängig testbare Einheit eines Programms“; also im Falle funktionaler Programmiersprachen stellen einzelne Funktionen oder Prozeduren eine Einheit dar, während bei der objekt-orientierten Entwicklung in der Regel eine Klasse ein Modul repräsentiert [Lig02, AO00]. Ist ein Modul fertiggestellt, kann es dem sogenannten *Modultest* unterzogen werden. Grundlage für den Modultest ist die Modulbeschreibung, welche als Teil des Software-Feinentwurfs erstellt wurde. Um einen Modultest tatsächlich durchzuführen, werden in den meisten Fällen sogenannte Treiber (*driver*) und Platzhalter (*stub*, *dummy*, *mock*) benötigt, welche anstelle der noch nicht implementierten Module aus der Umgebung des Testobjektes treten und die Dienste des zu testenden Moduls in Anspruch nehmen (Treiber) oder die Dienste, die das Testobjekt von anderen Modulen benötigt, simulieren (Platzhalter).

Natürlich muss die Modulprüfung nicht nur dynamisch in Form des Modultests erfolgen. Ebenso kann ein einzelnes Modul ergänzend einer statischen Untersuchung unterzogen werden. Dazu kann der Code auf die Einhaltung von Codierregeln (beispielsweise nach MISRA-C:2004[1]) hin überprüft werden. Bei besonders sicherheits- oder risikokritischen Modulen ist auch eine formale Verifikation in Form eines Korrektheitsbeweises denkbar.

Sind die Module erstellt und haben sie den Modultest bestanden, so können sie nun zu einem Gesamtprogramm zusammengesetzt werden. Während im Modultest nur die Funktionsfähigkeit des einzelnen Moduls überprüft wurde, muss nun das Zusammenspiel der verschiedenen Module getestet werden. Demnach liegt der Fokus des sogenannten *Integrationstests* auf der Verifikation der Schnittstellen zwischen den Modulen und dem Zusammenspiel der einzelnen Komponenten. Als Referenz dient dabei die im Software-Grobentwurf definierte Architektur. Zur Durchführung dieser Testphase gibt es eine Reihe unterschiedlicher Strategien, die je nach zu entwickelndem System und den bereitgestellten Ressourcen in dieser Phase verschieden gut geeignet sind [Bal97]. Zu den bekanntesten Verfahren zählen *top-down*, *bottom-up*, *outside-in*, *inside-out* und *sandwich* aus der Klasse der inkrementellen Integrationsteststrategien sowie *geschäftsprozessorientiert* und das nicht zu empfehlende *big-bang* als Repräsentanten der nicht-inkrementellen Verfahren.

Beim *big-bang*-Prinzip werden alle Module auf einen Schlag zusammengeführt und die vollständige Software getestet. Da anfangs erwartungsgemäß noch sehr viele Fehler das reibungslose Zusammenspielen der Module verhindern und die Lokalisierung dieser Fehler im Gesamtsystem nur schwer gelingt, sollte diese Methode der Integration eher vermieden werden. Im Allgemeinen eignet sich das *sandwich*-Vorgehen am Besten, bei dem beginnend mit den untersten operationalen Modulen (diejenigen, die keine anderen Module verwenden) nach und nach die darüber liegenden hinzugefügt werden, während gleichzeitig von den obersten Logikmodulen (diejenigen, die von keinen anderen Modulen verwendet werden) die jeweils darunter liegenden integriert werden, bis die gesamte Software zusammengestellt ist.

Konnten die Software-Module entsprechend dem Grob-Entwurf zu einem funktionsfähigen Verbund zusammengestellt werden, so kann nun im Rahmen des *Systemtests* auch die Hardware-Umgebung integriert werden. Im Wesentlichen dient dieser Systemtest einer Vorbereitung auf den folgenden Abnahmetest. Ideal dabei wäre eine annähernde Antizipation des Abnahmetests,

[1] Motor Industry Software Reliability Association: „Guidelines for the Use of the C Language in Vehicle Based Software“, `http://www.misra.org.uk/`

um beim Softwareentwickler sicherstellen, dass das System den Kunden überzeugen wird. Hierbei sind nicht nur funktionale Aspekte zu untersuchen, sondern ebenso die nicht-funktionalen Eigenschaften des Systems, wie man sie typischerweise in Stress- und Volumentests bewertet.

Hat die Software den Systemtest bestanden, so wird sie im nächsten Schritt beim Auftraggeber installiert und von diesem einem *Abnahmetest* unterzogen. Als Referenz zur Identifikation relevanter Einsatzszenarien und der Bewertung des Verhaltens der Software unter diesen Szenarien dient die Anforderungsspezifikation. Hier hat der Auftraggeber die Möglichkeit, Abweichungen zwischen dem in der Spezifikation beschriebenen und dem tatsächlich realisierten Produkt aufzudecken.

Entspricht das entwickelte System der Anforderungsspezifikation und hat deshalb den Abnahmetest bestanden, bedeutet dies nicht zwangsweise, dass der zukünftige Benutzer mit dem Produkt auch zufrieden ist. Erst ein *Feldtest* zeigt, ob das fertige System auch die tatsächlich benötigte Funktionalität mit der erwarteten Qualität bereitstellt und nicht nur die möglicherweise im Anforderungsdokument fehlerhaft beschriebene. In der Praxis werden die hier beschriebenen Abnahme- und Feldtests auch zusammengefasst, um dem Kunden das benötigte Produkt zu liefern und Folgeaufträge zu begünstigen. Rechtlich bindend ist jedoch der Inhalt der Anforderungsspezifikation.

Führt man den Software-Lebenszyklus fort, so schließen sich an die Erstinstallation des Produktes noch die „Phasen“ Wartung und Ablösung an. Insbesondere im Rahmen der Wartung wird man Modifikationen an der Software vornehmen, weshalb das geänderte Produkt erneut getestet wird. Dabei ist einerseits die Qualität der Modifikation zu untersuchen, also zum Beispiel, ob der Fehler auch tatsächlich behoben wurde, aber andererseits insbesondere auch sicherzustellen, dass durch die vorgenommenen Änderungen keine Funktionalität beeinträchtigt wurde, die gar keiner Modifikation bedurfte – also dass keine unerwünschten Seiteneffekte durch die Wartung entstanden sind. Dazu werden im Rahmen sogenannter *Regressionstests* die früher erfolgreich durchgeführten Testfälle, welche von den Modifikationen nicht betroffen sein sollten, erneut abgearbeitet und deren Ergebnisse mit den früher ermittelten verglichen. Für die geänderte oder neu hinzugekommene Funktionalität müssen entweder bestehende Testfälle angepasst oder zusätzliche neu erstellt werden, welche anschließend zusammen mit den unmodifiziert wiederverwendeten Testfällen ausgeführt werden müssen.

Da die Qualitätsprüfung im Allgemeinen und damit das Testen im Besonderen gerade für große, komplexe und sicherheitskritische softwarebasierte Systeme von eminenter Bedeutung ist, legt das in Kapitel 2.1 (Abbildung 2.1) vorgestellte V-Modell einen besonderen Schwerpunkt auf die Verifikation und Validierung der zu entwickelnden Systeme. Allerdings gehört zum Testen mehr als nur die Testdurchführung selbst.

Aus diesem Grund wurde das V-Modell erweitert, um den organisatorischen und vorbereitenden Schritten der einzelnen Testarten im Laufe der Softwareentwicklung Rechnung zu tragen. Dieses erweiterte Modell heißt aufgrund seines Aussehens W-Modell [Spi02]. Abbildung 2.2 nach [Spi02] zeigt exemplarisch, wie bereits in den frühesten Phasen des Softwarelebenszyklus die Grundsteine zur Durchführung der einzelnen Testphasen gelegt werden. So können zum Beispiel die Abnahmetests bereits bei der Erstellung oder nach Fertigstellung des Anforderungskataloges definiert werden oder gar in das Vertragswerk aufgenommen werden, was spätere Rechtsstreitigkeiten einzugrenzen vermag.

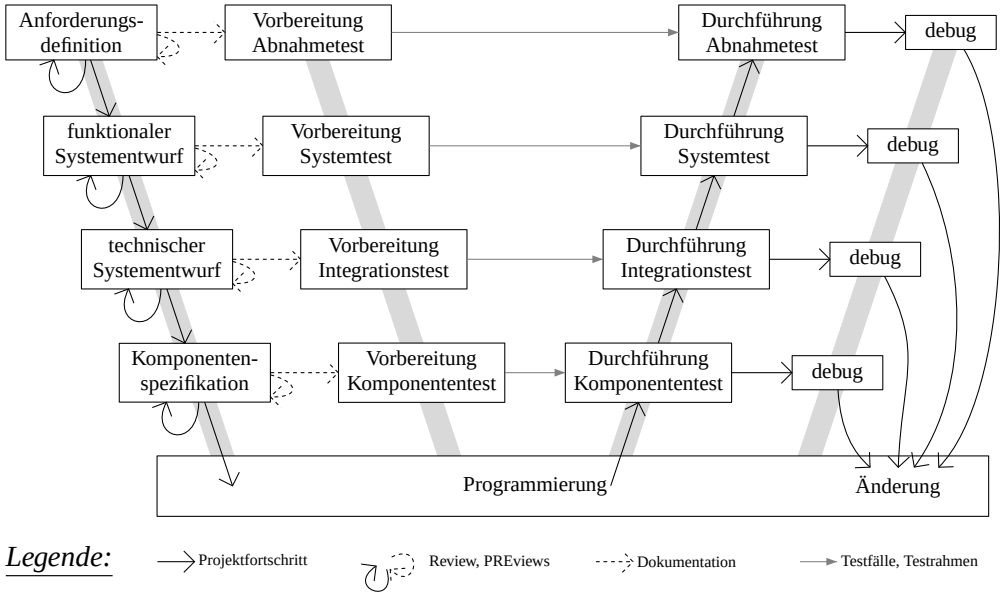


Abbildung 2.2: W-Modell

Bedeutsam ist hier auch die Darstellung des Prozesses der Fehlerkorrektur: Sollten während der jeweiligen Testphasen Versagen auftreten, müssen im Wesentlichen alle bereits abgeschlossenen Testphasen erneut durchlaufen werden. Nach der Fehlerlokalisierung erfolgt eine Rückkehr in die eigentliche Programmierphase, um den Produktfehler im Code zu entfernen (*debug*). Zumindest für den betroffenen Codeausschnitt ist anschließend im Rahmen des Regressionstests die korrekte Behebung des Produktfehlers sicherzustellen, was das Wiederholen der entsprechenden Verifikationsphasen erfordert.

2.2.2 Klassifikation nach Technik

So vielfältig die zu verifizierenden Aspekte eines Softwaresystems sind, so mannigfaltig ist die Menge der Prüfmethoden. Die verschiedenen Qualitätssicherungsverfahren sind meist prinzipiell unabhängig von der Phase des Lebenszyklus (wie im Kapitel 2.2.1 dargestellt) einsetzbar, der Aufwand für ihre Anwendung zur Überprüfung der Codequalität ist jedoch von Phase zu Phase verschieden. Dies liegt nicht zuletzt daran, dass die verschiedenen Testkriterien jeweils bestimmte Aspekte des zu testenden Systems detailliert betrachten. Um die Einordnung der hier angesprochenen Verfahren (siehe Kapitel 3) zu erleichtern, empfiehlt sich eine Klassifikation nach der jeweils verwendeten Technik beziehungsweise ihrer Definitionsgrundlage. Einen repräsentativen Auszug eines geeigneten Schemas zeigt Abbildung 2.3 in Anlehnung an [Lig02, Bal97].

Demnach unterscheidet man zunächst grob statische Qualitätssicherungsverfahren und dynamische Prüfmethoden. In der Fachliteratur findet man zuweilen die Bezeichnung „statische Testverfahren“ – da aber das Testen im allgemeinen Sprachgebrauch die Benutzung oder Ausführung des zu testenden Objektes erfordert und somit eher den dynamischen Aspekt suggeriert,

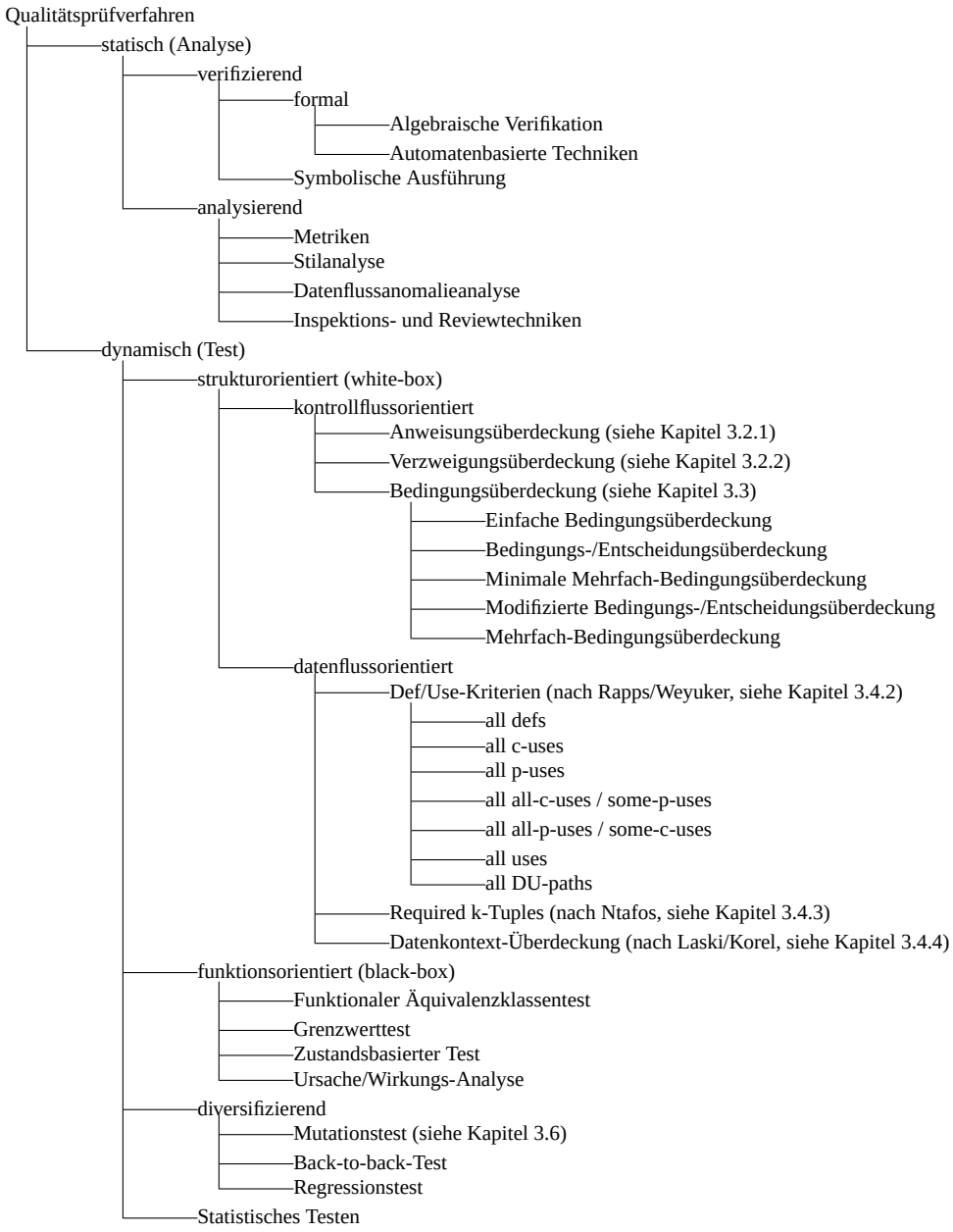


Abbildung 2.3: Klassifikationsschema für Qualitätssicherungsverfahren

werden die statischen Verfahren zur besseren Unterscheidung unter dem Sammelbegriff *(statische) Analyse* zusammengefasst. Zu dieser Klasse gehören unter anderem algebraische Verifikationsverfahren, wie beispielsweise formale Beweise mittels axiomatischen Hoare-Kalküls, und automatenbasierte Techniken mit *Model Checking* als wichtigstem Vertreter. Ebenso stellen die Vermessung der Software anhand von Komplexitätsmetriken sowie Inspektion und Review von Entwurfsdokumenten statische Qualitätsprüfverfahren dar.

Zur Familie der dynamischen Testverfahren zählen die beiden großen Klassen der struktur- und der funktionsorientierten Testmethoden. Letztere berücksichtigen den Quellcode der zu testenden Anwendung gar nicht, weshalb sie auch den Namen *black-box*-Verfahren tragen. Nach solchen Kriterien vorgehend, leitet der Tester die Testfälle und -daten aus der Spezifikation ab und untersucht auf diese Weise, ob alle spezifizierten Funktionen erwartungsgemäß funktionieren. Diese black-box-Testverfahren sind gleichsam auch zur Untersuchung nicht-funktionaler Eigenschaften geeignet, wie beispielsweise Performanztests (Last- und Stresstests, minimale/-maximale Ausführungszeiten [OVW98]) oder zur Überprüfung der Informationssicherheit (*security*) des Systems [ML04].

Ein typischer Vertreter dieser funktionalen Teststrategie ist der *funktionale Äquivalenzklassentest*. Dabei partitioniert man den gesamten Eingaberaum des zu testenden Programms anhand der Spezifikation in einzelne Klassen, so dass jede Eingabe aus der gleichen Klasse ein (erwartungsgemäß) äquivalentes Verhalten des Programms hervorruft. Diesem Testverfahren liegt die Annahme zugrunde, dass aus dem erfolgreichen Ausführen je eines Testfalls mit einer Eingabe (Repräsentanten) aus jeder der Äquivalenzklassen auf die korrekte Bearbeitung aller anderen Eingaben geschlossen werden kann. Soll beispielsweise die Verwaltungssoftware einer KFZ-Vermietung laut Spezifikation grundsätzlich nur eine Ausleihe von Fahrzeugen an Volljährige erlauben, so ergeben sich daraus zwei Äquivalenzklassen basierend auf dem eingegebenen Alter des Ausleihenden. Demnach sollte das Programm zumindest mit je einem minderjährigen und einem volljährigen „Kunden“ getestet werden.

Im Gegensatz dazu ist die Basis zur Definition der strukturellen Testkriterien, auch *white-box*-Kriterien genannt, der Quellcode eines Softwaresystems. Die Überdeckungsgrade, also die Maße zur Bestimmung der Testvollständigkeit hinsichtlich solcher Kriterien, beruhen auf dem Kontroll- oder Datenfluss – die Spezifikation wird lediglich zur Überprüfung der Testergebnisse herangezogen. Der Vorteil der white-box-Strategien gegenüber den black-box-Verfahren liegt darin begründet, dass strukturelles Testen auch unerwünschte, also unspezifizierte „Funktionalität“ zu entdecken vermag, wie sie durch Kombination erwarteter Funktionalitäten auftreten kann. Da die in der vorliegenden Arbeit entwickelte Methode besonders für die Testfall- und Testdatengenerierung hinsichtlich solcher Kriterien geeignet ist, werden diese Teststrategien ausführlicher in Kapitel 3 behandelt.

Während die klassischen black- und white-box-Testtechniken jeweils Testvollständigkeitskriterien definieren, erlauben die sogenannten *diversifizierenden* Testverfahren die Umgehung der meist sehr aufwändigen Überprüfung einzelner Testergebnisse anhand der Spezifikation [Lig02]. So kann zum Beispiel beim Regressionstesten derjenige Teil der bereits vorhandenen Testfälle wiederverwendet werden, der von den Änderungen nicht betroffen ist; für diese Testfälle muss sich die modifizierte Variante genauso verhalten (insbesondere die gleichen Ausgaben liefern) wie das ursprüngliche Programm, so dass ein direkter automatisierter Vergleich möglich ist.

2.3 Testautomatisierung

Aufgrund des enormen Fortschritts in der Forschung und in der industriellen Anwendung entsprechender Forschungsergebnisse auf dem Gebiet der Verifikation von Softwaresystemen hat der Begriff *Testautomatisierung* eine geradezu inflationäre Verbreitung gefunden. Dies macht die Einordnung der vorliegenden Arbeit umso schwieriger, da in der Literatur und insbesondere in Beschreibungen kommerzieller Softwarewerkzeuge mit dem Ausdruck *Testautomatisierung* zuweilen recht unterschiedliche Aspekte des Testens bezeichnet werden.

2.3.1 Klassifikation

Klassische Verfahren zur Automatisierung des Softwaretests, welche in den meisten gängigen Werkzeugen zur Unterstützung des Software Engineerings umgesetzt wurden, berücksichtigen unter dem Konzept der Testautomatisierung meist lediglich die automatische Ausführung der Testläufe. Dabei bleibt es weitgehend dem Tester überlassen, die relevanten Testfälle und insbesondere die dazu notwendigen Testdaten zu identifizieren und in einem geeigneten Format zu beschreiben, so dass diese Testspezifikation automatisch abgearbeitet werden kann.

Zur Beschreibung und Ausführung von Tests im Bereich des Modultestens haben sich die sogenannten *xUnit*-Frameworks etabliert, von denen es für die jeweilige Programmiersprache spezielle Ausprägungen gibt, unter anderem *JUnit*[2] für JAVA™ oder *NUnit* für die Microsoft.NET-Umgebung. Dabei „programmiert“ der Tester die gewünschten Testfälle, indem er sich der vom Framework vorgegebenen Funktionalitäten bedient. Dazu kann er beispielweise in JUnit eine Klasse erstellen, welche von der Testfallklasse des Frameworks (`junit.framework.TestCase`) abgeleitet ist und deren Methoden (welche mit dem Schlüsselwort `test` beginnen müssen) die einzelnen Testschritte des Testfalls darstellen. Da diese Testfälle somit eine definierte Syntax und Struktur haben, können sie sehr leicht von entsprechenden Testwerkzeugen vollautomatisch ausgeführt werden, welche meist schon Bestandteil des Frameworks sind oder in Softwareentwicklungsumgebungen integriert wurden.

Darüber hinaus kann der Testingenieur in xUnit-Tests auch Prüfroutinen erstellen, sogenannte *assertions*, die während der Testausführung oder am Ende der Testfallabarbeitung die Korrektheit bestimmter (Zwischen-)Ergebnisse untersuchen. Somit ist das Testwerkzeug auch in der Lage, auf unterschiedlichen Detaillierungsebenen ein Urteil (sog. *verdict*) über den Erfolg oder Misserfolg des Testlaufs und einen Testbericht automatisch zu generieren. Allerdings können solche Prüfbedingungen nicht vollautomatisch abgeleitet, höchstens in Sonderfällen aus bestimmten Dokumenten, wie zum Beispiel Kommentaren im Quellcode des Testobjekts, automatisch übertragen werden.

Ein anderer Weg, Testfälle zu ermitteln, besteht bei vollständig ablauffähigen Programmen und insbesondere bei Applikationen mit ausgeprägter Benutzerinteraktion durch eine graphische Benutzeroberfläche (GUI, graphical user interface) in der sogenannten *capture/replay*-Technik. Dabei wird die Software von einer Testperson bedient, während ein Testwerkzeug im Hintergrund alle Aktivitäten des Benutzers, typischerweise Mausbewegungen oder Tastatureingaben,

[2] `www.junit.org`

protokolliert (*capture*). Anschließend können die aufgezeichneten Aktionen beliebig oft wiederholt werden (*replay*), was inbesondere im Rahmen des Regressionstestens eine erhebliche Einsparung an Testaufwand bedeutet. Leistungsfähigere capture/replay-Werkzeuge generieren aus den mitgeschnittenen Benutzerinteraktionen editierbare Anweisungsabfolgen in sogenannten Skriptsprachen, welche sich gezielt anpassen lassen, um daraus neue Testfälle zu erstellen. Typischerweise würde man die eingegebenen Werte, die zunächst im Testablaufskript festgehalten sind, durch Zugriffe auf externe Datenspeicher ersetzen, um damit beliebige Testfälle mit gleicher Abfolge aber jeweils unterschiedlichen Daten automatisiert ausführen zu können.

Aufgrund des heutzutage etablierten Einsatzes von Modellierungswerkzeugen in den frühen Phasen der Softwareentwicklung, darunter insbesondere solcher, denen die semi-formale Modellierungssprache *UML (Unified Modeling Language)* zugrunde liegt, wurde das sogenannte *modellbasierte Testen* weitgehend als willkommene Unterstützung der Testphase angenommen. So bestehen vielfältige Ansätze darin, die in den Entwurfsmodellen enthaltenen Informationen zu extrahieren und in Testszenarien zu übertragen. Im einfachsten Fall können bereits die in den Sequenzdiagrammen modellierten Abläufe direkt in Testszenarien und somit in Testskripte übersetzt werden. Im Bereich des *zustandsbasierten Testens* nutzt man die in den UML-Statecharts modellierten Zustände und Zustandsübergänge, um daraus ebenfalls Ablaufszenarien zu generieren. Die Kombination unterschiedlicher Sichten [Sok04], wie sie typischerweise bei UML-Modellen auf verschiedenen Diagrammen verteilt sind, erlaubt zwar eine genauere Charakterisierung der Testszenarien, doch ein vollständiges und allgemeingültiges Verfahren zur Identifikation notwendiger Testdaten ist auch damit analytisch und somit statisch nicht möglich.

Um den Softwaretest effizient und effektiv zu gestalten, ist es insbesondere bei strukturellen Testverfahren wichtig, eine weitestgehend automatisierte Unterstützung bei der Bestimmung des jeweils erreichten Überdeckungsgrades zu erhalten. Hat der Tester zunächst eine Menge Testfälle identifiziert und ausgeführt, so müssen diejenigen Entitäten (zum Beispiel Anweisungen, Verzweigungen oder ganze Teilpfade) des Programmcodes noch überdeckt werden, die bislang noch nicht berücksichtigt wurden. Dazu müssen diese Bereiche aber zunächst identifiziert und dem Tester in geeigneter Weise kommuniziert werden. Die meisten Testwerkzeuge bieten hier nur bedingt Unterstützung. Meist beschränken sich diese Tools auf einfache Kontrollflusskriterien wie die Anweisungsüberdeckung oder auf die gröbere Granularität einzelner Codezeilen, wie im Falle der aktuellen Version des *Microsoft Visual Studio Team System*.

Hat ein Tester einen Grundstock an Testfällen ausgeführt und verbleiben noch nicht überdeckte Entitäten, so besteht der nächste Schritt darin, der bereits bestehenden Testfallmenge weitere Testfälle hinzuzufügen, um das Überdeckungsmaß gemäß dem vorgegebenen Testkriterium zu maximieren. Bedauerlicherweise sucht man auch in renommierten, kommerziellen Testwerkzeugen vergebens nach einer Unterstützung bei der Identifikation derjenigen Testfälle, die die noch nicht überdeckten Codestrukturen zur Ausführung bringen – selbst im Falle einfacher Kriterien wie der Anweisungsüberdeckung. Der zum Teil immense Aufwand zur manuellen Bestimmung noch fehlender Testfälle ist wohl auch der Grund, warum schon die Unterstützung der Überdeckungsgradmessung in kommerziellen Werkzeugen kaum über die der einfachsten kontrollflussbasierten Kriterien hinausgeht.

Die Automatisierbarkeit des Testens endet im Allgemeinen beim letzten Schritt, der Überprüfung der Testergebnisse. Unabhängig davon, ob die Testfälle manuell oder automatisiert ent-

wickelt wurden, müssen die Testergebnisse und das Verhalten des zu testenden Systems während der Testausführung auf Konformität mit der entsprechenden Testspezifikation untersucht werden. Zwar kann der reine Vergleich der Testergebnisse automatisiert werden, die tatsächlich erwarteten Werte müssen jedoch meist manuell identifiziert werden. Erste Ansätze der Automatisierung unterstützen zum Teil das sogenannte *conformance testing*, bei dem beispielsweise ein Testwerkzeug automatisiert anhand eines Modells, typischerweise eines Zustandsautomaten, die an dem zu testenden System auszuführende Folge von Aktionen ermittelt und das System dementsprechend schrittweise stimuliert, wobei nach jedem Schritt jeweils die Reaktion und der neue Zustand des Testobjekts ebenfalls gegen das Modell überprüft werden [TB03].

Die vorliegende Arbeit beschäftigt sich mit der möglichst vollständigen Automatisierung der Testfallgenerierung, -optimierung und -bewertung für komplexere strukturelle Kriterien (vor allem datenflussbasierte Teststrategien). Dabei werden insbesondere auch die zur Testausführung notwendigen Testdaten automatisch ermittelt. Die Optimierung der Testfälle besteht darin, die strukturelle Überdeckung der zu testenden Software zu maximieren und gleichzeitig die Menge der dazu erforderlichen Testfälle zu minimieren, um den Aufwand der meist manuellen Auswertung der Testläufe gering zu halten und somit eine hohe Testeffizienz zu erzielen.

2.3.2 Bestehende Ansätze und Abgrenzung

Der Grundgedanke, Evolutionäre Algorithmen zur Automatisierung des Softwaretestens einzusetzen, ist in dieser Arbeit nicht revolutionär. Die im Folgenden skizzierten Ansätze verfolgen zum Teil verwandte Ziele auf diesem Themengebiet. Dennoch unterscheidet sich das hier präsentierte Verfahren grundlegend von den bekannten Ansätzen. Um diesen Sachverhalt darzustellen und zu begründen, wird in diesem Kapitel ein kurzer Überblick über bestehende Ansätze erarbeitet und zugleich eine Abgrenzung gegenüber der hier präsentierten Methode vorgenommen. Die Darstellung bedient sich einer Begriffswelt, die in Kapitel 3 und Kapitel 4 ausführlich beschrieben und definiert wird.

Im Bereich des black-box-Testens seien hier insbesondere zwei Verfahren genannt, welche die Eignung Genetischer Algorithmen zur Testautomatisierung unterstreichen. Das erste setzt solche Algorithmen zur Überprüfung der temporalen Korrektheit von Echtzeitsystemen ein, während das zweite die funktionale Korrektheit im Hinblick auf die Vermeidung kritischer Situationen untersucht.

Bestimmung maximaler Ausführungszeiten von Echtzeitsystemen ([OVW98])

Nebst Bereitstellung der tatsächlichen Funktionalität muss bei der Entwicklung eingebetteter Echtzeitsysteme auf eine wichtige nicht-funktionale Eigenschaft besonders geachtet werden: Die Einhaltung der vorgegebenen zeitlichen Einschränkungen. Eine Verletzung dieser Bedingungen tritt auf, weil entweder Zwischenergebnisse zu früh ermittelt und an nachgelagerte Subsysteme weitergereicht werden oder die Berechnung solcher Ergebnisse zu lange dauert. Daher müssen in der Testphase diejenigen Ablaufszenarien ermittelt werden, welche die kürzeste beziehungsweise längste Ausführungszeit erfordern, um somit zu untersuchen, inwieweit obengenannte temporale Versagen auftreten. Da die manuelle Suche nach solchen Szenarien bei der Komplexität heutiger

Systeme in den meisten Fällen kaum noch sinnvoll durchzuführen ist, bietet sich hier die Darstellung dieser Aufgabe als Optimierungsproblem an [OVW98]. Wissenschaftler der Stanford University und Mitarbeiter der Daimler-Benz AG Berlin haben zur Lösung dieses Problems Genetische Algorithmen eingesetzt, wobei der Eingaberaum des zu testenden Systems dem Suchraum des Optimierungsverfahrens entspricht und die Ausführungszeiten des Testobjekts zugleich die Werte der sogenannten Bewertungsfunktion (*Fitness*, siehe Kapitel 4.5) darstellen. Ein Abbruchkriterium für die Optimierung wird in [OVW98] auf Basis der Cluster-Analyse definiert. Dabei werden die nach jeder Generation des Genetischen Algorithmus identifizierten Zwischenlösungen aufgrund ihres euklidischen Abstandes im Suchraum zu Clustern zusammengefasst und die durchschnittliche Fitness jedes Clusters berechnet. Sind diese Mittelwerte für alle Cluster annähernd gleich, so ist der Genetische Algorithmus konvergiert und ein weiterer Fortschritt ist nicht zu erwarten. Bricht man nun die Optimierung ab, so stellen Individuen aus verschiedenen Clustern jeweils unterschiedliche aber gleichwertige Lösungen des Problems dar, in diesem Fall also diejenigen Eingaben an das Testobjekt, die je nach Optimierungsziel die kürzeste oder längste Ausführungsdauer erfordern.

Untersuchung kritischer Ablaufszenarien ([BW03, BSW04])

Die voranschreitende Automatisierung macht natürlich auch vor dem Straßenverkehr nicht halt. Daher ist es nicht verwunderlich, dass Automobilhersteller sowie deren Zulieferer zunehmend kritische Bereiche der Kraftfahrzeugsteuerung eingebetteten Softwaresystemen anvertrauen. Eine Komponente der Fahrerassistenzsysteme, die mehr und mehr Gegenstand der praktischen Einführung in Serienfahrzeuge ist, soll dem Fahrer das rückwärts längs Einparken vollautomatisch abnehmen. Dazu erkennt das *Automated Parking System (APS)* im Vorbeifahren, ob eine Parklücke ausreichend Platz für das automatische Manöver bietet, so dass der Fahrer das Auto am Startpunkt des klassischen Einparkweges abstellen und das System auf Knopfdruck starten kann. Das APS berechnet daraufhin eine ideale Route und steuert das Fahrzeug entlang dieser Route in die Lücke. Über die Verletzung von Passanten hinaus besteht die Kritikalität eines solchen Systems in der Gefahr, bereits parkende Fahrzeuge zu beschädigen, wenn die Route nicht korrekt berechnet oder abgefahren wird. Weniger kritisch aber unerwünscht ist es, wenn das automatisch geparkte Fahrzeug eine unangepasste Lage einnimmt, zum Beispiel schief in der Parklücke steht, über die Lücke hinausragt oder in der Umgebung parkende Fahrzeuge zuparkt.

Um die Qualität der Software eines neu entwickelten APS zu untersuchen, wurden ebenfalls Evolutionäre Algorithmen eingesetzt [BW03, BSW04]. Der Suchraum des Optimierungsproblems enthält alle möglichen Startszenarien für das Manöver, jedes Szenario ist dabei durch fünf Parameter charakterisiert: Länge und Breite der Parklücke, Querabstand des Fahrzeugs von der Parkreihe und Längsabstand vom Beginn der Lücke sowie den Winkel zwischen Fahrzeug und Parkreihe. Um die „Güte“ eines Startszenarios zu bestimmen, wird das Einparken ausgehend vom vorgegebenen Szenario simuliert. Die Bewertungsfunktion ist dabei der minimale Abstand zwischen dem Fahrzeug und den sogenannten Kollisionsflächen (andere parkende Fahrzeuge oder Straßenrand) – wobei eine Abstandsmessung nach jedem Schritt der Simulation vorgenommen wird. Das eingesetzte Evolutionäre Verfahren war in der Lage, kritische Szenarien zu identifizieren, die zu einem Überschreiten der Parklücke geführt haben, nämlich wenn das Fahrzeug

beim Start des Parkmanövers zu nah an der Parkreihe aber zu weit vom Beginn der Parklücke abgestellt wurde.

Mutationstestbasierte Erweiterung von Komponentenselbsttests ([BHJT00, BFJT02])

Ein Ansatz, welcher im Übergang vom funktionalen Testen (einschließlich black-box-Testen nicht-funktionaler Eigenschaften) hin zu einer strukturorientierten Sicht anzusiedeln ist, wird in [BHJT00, BFJT02] vorgestellt. Kern der dort vorgestellten Methoden ist das Mutationstesten, welches im Kapitel 3.6 genauer vorgestellt wird, wobei der Schwerpunkt der Arbeit auf dem Testen objekt-orientierter Komponenten beziehungsweise Systeme liegt.

Das Verfahren aus [BHJT00] setzt voraus, dass die zu testenden Komponenten nach dem Ansatz des „design by contract“ entwickelt wurden, also dass die Spezifikation iterativ durch Verfeinerungsschritte in ausführbare Programmeinheiten überführt wird, welche die „Verträge“ oder Zusicherungen weiterhin in Form von Invarianten sowie Vor- und Nachbedingungen beinhalten. Auf diese Weise enthalten die Komponenten einen eingebauten Selbstprüfungsmechanismus, der wie folgt das Vertrauen in die Qualität der Komponente erhöht: Sind die eingebauten Bedingungen ausreichend, dann werden sie verletzt und melden daher einen fehlerhaften Zustand (ein Hinweis auf einen Produktfehler) falls die Komponente fehlerhaft umgesetzt wurde und ein Testfall die fehlerhafte Programmstelle zur Ausführung bringt.

Der vorgestellte Ansatz verfolgt das Ziel, die Überprüfung der Konsistenz zwischen Spezifikation, Implementierung und Selbsttest zu unterstützen, indem eine adäquate Menge von Testfällen für den Selbsttest generiert wird. Das Verfahren wurde in einem experimentellen Werkzeug namens *μSlayer* für die Programmiersprache EIFFEL entwickelt. *μSlayer* generiert Mutanten der zu testenden Komponente und initiiert den Selbsttest. Werden Mutanten vom bestehenden Test nicht getötet, so muss eine manuelle Diagnose entscheiden, ob der Mutant dem ursprünglichen Programm äquivalent oder der Selbsttest nicht angemessen ist. Ein Genetischer Algorithmus wird angewandt, um den bereits bestehenden Selbsttest zu verbessern, wobei die zugrundeliegende Fitnessfunktion den *mutation score* (siehe Seite 78) des betrachteten Testfalls wiedergibt. Die Variante des Algorithmus selbst ist dem Prinzip des „Jäger und Gejagten“ (predator and prey) entlehnt, wobei jeder Mutant eine Beute und jedes Individuum einen Jäger darstellt. Ziel des Genetischen Algorithmus ist es demnach, die Population der Jäger im Hinblick auf die Ausrottung der Beute zu evolvieren – dabei repräsentiert ein Individuum einen Testfall bestehend aus mehreren Methodenaufrufen.

Die Erweiterung des initialen Selbsttests wird bei [BHJT00] als „Optimierung des Tests“ bezeichnet, welche im Gegensatz zur vorliegenden Arbeit keine Minimierung der Testfallmenge zugunsten einer effizienteren Validierung der Testergebnisse bei gleichzeitig hoher Überdeckung verfolgt. Die Entwickler des Verfahrens selbst plädieren sogar für eine kontinuierliche Vergrößerung der Population im Laufe der „Optimierung“ und damit auch der Menge der einzubauenden Selbsttests.

Die Anwendung Genetischer Algorithmen zur automatischen Generierung von mutationsbasierten Testfällen wird in [BFJT02] verallgemeinert und verbessert. Die Betrachtung selbst-testender Komponenten wird zugunsten des Systemtests aufgegeben. Dieses neue Verfahren wurde exemplarisch für die Programmiersprache C# in der .NET-Umgebung umgesetzt und experi-

mentell untersucht. Nebst der in [BHJT00] angewandten Variante der Genetischen Algorithmen wird außerdem ein sogenannter *bakteriologischer* Ansatz verfolgt, welcher in den durchgeführten Experimenten bessere Ergebnisse hinsichtlich Performanz der Generierung und Qualität der Testfälle lieferte. Dabei wird auf die Kreuzung von Individuen verzichtet; stattdessen werden die Individuen lediglich reproduziert und verstärkt mutiert.

Generierung kontrollflussbasierter Testfälle

Da die Ermittlung von Testdaten zur strukturellen Überdeckung des Testobjektes besonders schwierig und zeitaufwendig ist, beschäftigt sich eine Vielzahl von Forschungsarbeiten mit der Automatisierung dieses Schrittes im Testprozess. Einen umfassenden Überblick zu diesen Methoden, von den frühen Ansätzen bis zu den aktuellen Weiterentwicklungen, bietet [McM04]. Da eine ausführliche Beschreibung den Rahmen dieser Arbeit sprengen würde, seien hier nur exemplarisch die grundlegenden Methoden auf einem aktuellen Stand der Technik skizziert. Die unterschiedlichen Verfahren werden nach [PHP99, McM04] wie folgt grob in mehrere Klassen eingeteilt.

Zufallsbasierte Datengeneratoren berücksichtigen keinerlei Überdeckungskriterien unmittelbar, stattdessen erstellen sie Testdaten nach vorgegebenen Verteilungen über den möglichen Eingaberaum. Das Verfahren hat im Allgemeinen den entscheidenden Nachteil, dass sehr viele Eingabedaten generiert werden, welche die gleichen Pfade wiederholt überdecken, während gleichzeitig vereinzelte Pfade nie ausgeführt werden, zum Beispiel weil der zugehörige Eingaberaum nur wenige Elemente besitzt und der Zufallsgenerator daher nur mit geringer Wahrscheinlichkeit eine solche Eingabe auswählt. Verschärft wird der Nachteil dadurch, dass die vielen, meist äquivalenten Testergebnisse üblicherweise manuell auf Korrektheit untersucht werden müssen. Diesbezügliche Untersuchungen haben gezeigt, dass im Durchschnitt lediglich ein Viertel der zufällig generierten Testfälle tatsächlich genauer untersucht werden musste [JSE96].

Die sogenannten *pfad-orientierten* Generatoren nutzen typischerweise den Kontrollflussgraphen des Testobjekts zur Identifikation eines (vollständigen) Pfades durch den Graphen, bei dessen Überdeckung auch das vorgegebene Testziel, zum Beispiel die Ausführung einer bestimmten Anweisung, erreicht ist. Anschließend wird meist mittels symbolischer Ausführung oder lokaler Suchheuristiken (z.B. Hillclimbing, siehe Kapitel 4.3) nach einem Eingabedatum gesucht, welches den identifizierten Pfad zur Ausführung bringt [MLK03]. Problematisch hierbei ist, dass prinzipiell Pfade auftreten können, die zwar graphentheoretisch aus dem Kontrollflussgraphen abgeleitet werden, für die es jedoch keinen Testfall gibt, der sie zur Ausführung bringt (siehe Definition 3.11 auf Seite 38). Einen Sonderfall stellt das Verfahren in [Grz04] dar, da hierbei kein klassisches strukturelles Testziel, wie bestimmte Anweisungen oder Verzweigungen, vorab festgelegt wird. Stattdessen werden Testdaten nach dem Kriterium des Äquivalenzklassentests generiert, wobei die Äquivalenzrelation aufgrund der Kontrollflusspfade definiert ist: Alle Testfälle, welche den gleichen Programmpfad durchlaufen, gehören der gleichen Klasse an, wobei für jede Schleife eine Obergrenze n festgelegt wird, so dass alle Pfade, die mehr als n Wiederholungen des Schleifenrumpfes enthalten, ansonsten aber einen äquivalenten Pfad durchlaufen, der gleichen Klasse zugeordnet werden. Die symbolische Ausführung erstellt für jeden äquivalenten Pfad ein Gleichungssystem, dessen Lösungen genau den äquivalenten Eingabedaten entsprechen.

Die am weitesten verbreiteten Verfahren gehören zur Klasse der *ziel-orientierten* Testdatengeneratoren. Diese identifizieren anhand eines vorgegebenen strukturellen Testkriteriums (zum Beispiel Anweisungsüberdeckung) und basierend auf einer geeigneten Representation des Programms (zum Beispiel mittels Kontrollflussgraphen) alle zu überdeckenden Testziele (im Beispiel sind das alle Anweisungen des Programms beziehungsweise alle Knoten des Graphen), um das Kriterium zu erfüllen. Anschließend versuchen sie zu jedem noch nicht erreichten Testziel jeweils einen Testfall zu ermitteln. Der letzte Schritt wird mittels symbolischer Ausführung, statischer Analyse (sofern möglich) oder Suchheuristiken, wie zum Beispiel Genetische Algorithmen, angegangen. Die verschiedenen Ansätze der letzten Klasse unterscheiden sich meist lediglich in der Art, wie die einzelnen Testziele ermittelt werden oder welche zusätzliche Information, typischerweise bezüglich bestimmter Knoten und Kanten entlang potentieller Pfade zum Testziel, hinzugezogen wird.

Eines der frühen Verfahren [MMS98], welches evolutionäre Suchheuristiken zur Ermittlung von Testdaten zur Verzweigungsüberdeckung beliebiger Programme in der Sprache `C/C++` einsetzt, basiert auf dem Prinzip der Funktionsminimierung und stützt sich unter anderem auf den Grundgedanken bekannter pfad-orientierter Generatoren [Kor90, CCCL96, GN97]. Für jede Bedingung im Programm und jede der beiden möglichen Ergebnisse (Zweige *wahr* oder *falsch*) der Bedingungsauswertung wird je eine Funktion so definiert, dass der gewünschte Zweig genau dann überdeckt wird, wenn diese Funktion ihren minimalen Wert einnimmt [Kor90]. Zum Beispiel sei „`if (c >= d)`“ ein Ausdruck, welcher laut Testziel zu *wahr* ausgewertet werden soll und `x` ein Testfall, dann ist die zu minimierende Funktion gegeben durch [MMS98]:

$$\mathcal{F}(x) = \begin{cases} d - c, & falls\ d \geq c, \\ 0, & sonst. \end{cases}$$

Wird der zu überdeckende Ausdruck bei Ausführung des Programms mit `x` gar nicht erst erreicht, so wird der Wert der Funktion auf den größtmöglichen Wert gesetzt. Falls der Ausdruck jedoch zu *falsch* ausgewertet wird, so kann diese Funktion als Fitnessfunktion eines Genetischen Algorithmus dienen, welche Eingaben an das zu testende Programm mit dem Ziel evolviert, die Funktion zu minimieren, was im Falle eines Erfolges die Auswertung des Ausdrucks zu *wahr* bedeutet. Die Reihenfolge, in der die einzelnen zu überdeckenden Verzweigungen verfolgt werden, ergibt sich dynamisch aufgrund sogenannter „Überdeckungstabellen“ [CCCL96]. In einer solchen Tabelle wird für jede Bedingung im Programm vermerkt, welche der beiden daraus entstehenden Alternativen von den bisher identifizierten Testfällen bereits überdeckt wurde. Somit ist der *Wahr*-Zweig einer Bedingung erst dann ein potentielles Testziel, wenn der *Falsch*-Zweig von mindestens einem bereits ermittelten Testfall überdeckt wurde und entsprechend umgekehrt; ansonsten wird diese Bedingung zunächst nicht weiter berücksichtigt, da bis dahin noch kein Testfall die Auswertung des Ausdrucks überhaupt notwendig gemacht hat.

Das in [JSE96] vorgestellte Verfahren wurde für die Programmiersprache ADA83 exemplarisch umgesetzt und sollte mittels Genetischer Algorithmen das Kriterium der Verzweigungsüberdeckung für Ada-Programme verfolgen. Die Vorgehensweise basiert auf einer baumartigen Repräsentation des Kontrollflussgraphen (*CFG*). Dabei werden alle schleifenfreien Pfade im CFG von entsprechenden Pfaden durch den zugehörigen Baum repräsentiert. Jede Schleife wird

im Baum durch vier verschiedene Knoten dargestellt, welche jeweils die Anzahl (keine, eine, zwei beziehungsweise mehr als zwei) der Schleifenwiederholungen widerspiegeln, wobei jeder Knoten ein gleichberechtigter Nachfolger des gleichen Vorgängerknotens ist. Somit ähnelt das Schleifenkonstrukt einer Vierfachverzweigung, weshalb die Überdeckung aller „Kanten“ nun der Ausführung von Pfaden mit keiner, einer, zwei und mehr als zwei Schleifenwiederholungen (für jede Schleife unabhängig betrachtet) entspricht. Der so entstandene Kontrollflussbaum wird nun in einer Breitensuche abgearbeitet, so dass als nächstes Testziel jeweils eine der noch nicht überdeckten Kanten ausgewählt wird, welche von einem zuletzt überdeckten Knoten ausgeht. Das Verfahren nutzt Genetische Algorithmen, das heißt jedes Individuum entspricht einer Programmeingabe bestehend aus mehreren Parametern, welche jeweils als binäre Zeichenkette repräsentiert und zum Individuum konkateniert werden. Sind e und $\bar{e}$ die beiden Kanten einer Verzweigung mit einer atomaren Bedingung (z.B. „`A = 0`“), so nimmt die Fitnessfunktion für jeden Testfall t (Individuum) und die zu überdeckende Kante e Werte wie folgt an:

- einen sehr großen Wert, wenn e bei der Ausführung von t überdeckt wurde;
- einen Wert, welcher proportional zum Hammingabstand zwischen den Binärdarstellungen (alternativ zum Betrag der reziproken Differenz) der tatsächlichen Werte links und rechts des relationalen Operators (im Beispiel der Wert von A sowie 0) in der Bedingung ist, falls die Kante $\bar{e}$ von t überdeckt wurde (auf diese Weise werden bevorzugt Testfälle an den Grenzen der durch die Bedingung induzierten Äquivalenzklassen generiert);
- einen sehr kleinen Wert, wenn weder e noch $\bar{e}$ unter der Ausführung von t überdeckt wurden.

Der Hauptkritikpunkt an diesem Verfahren ist die stark vereinfachte Darstellung des Kontrollflusses als Baum durch Abrollen der Zyklen. So kann es durchaus vorkommen, dass der Rumpf bestimmter Schleifen immer mindestens einmal (oder gar öfter) ausgeführt werden muss, was die Erfüllung des „Verzweigungskriteriums“ nach [JSE96] verhindert, da stets eine „Kante“ zu überdecken ist, welche dem Überspringen des Rumpfes entspricht.

Um den Schwachpunkt des Verfahrens von [JSE96] zu überwinden, bedient sich die Methode nach [PHP99] einer Transformation des Kontrollflussgraphen in einen sogenannten *Kontrollabhängigkeitsgraphen* (engl. *control dependence graph*, kurz *CDG*) [Nat88]. Wie der Kontrollflussgraph (siehe Definition 3.5) ist auch der CDG ein gerichteter Graph, dessen Knoten ebenfalls Anweisungen repräsentieren, während die Kanten im Gegenzug lediglich eine Kontrollabhängigkeit darstellen. Abbildung A.1 zeigt einen solchen Kontrollabhängigkeitsgraphen für das Beispielprogramm aus Listing 3.1 basierend auf dem Kontrollflussgraphen in Abbildung 3.1. Darin ist Knoten n_6 von n_4F kontrollabhängig, das heißt die Anweisungen in Knoten n_6 werden nur ausgeführt, wenn vorher die Bedingung in Knoten n_4 erreicht und zu *falsch* ausgewertet wird. Gleiches gilt auch für Knoten n_9, da zu allen Pfaden die die Kante (n_4, n_6) enthalten, stets auch Knoten n_9 gehört, unabhängig davon, welchen Wert die Bedingung in Knoten n_6 annimmt, weshalb n_9 nicht etwa von n_6, n_7 oder n_8 kontrollabhängig ist, sondern direkt von n_4F. Um Testfälle nach dem Kriterium der Anweisungs- und Verzweigungsüberdeckung zu generieren, werden azyklische Pfade im Kontrollabhängigskeitsgraphen von der Wurzel zum jeweils betrachteten Knoten

beziehungsweise zur relevanten Kante untersucht [PHP99]. Jeder dieser sogenannten *control-dependence predicate paths* (kurz CDPT) enthält eine Menge von Bedingungen, die allesamt erfüllt sein müssen, um den jeweiligen Knoten oder die Kante zu überdecken. In Abbildung A.1 ist $\{n_{Start}, n_2T, n_4F, n6T\}$ ein solcher CDPT für den Knoten n_7 beziehungsweise für die Kante (n_6, n_7). Der eingesetzte Genetische Algorithmus erstellt zunächst den CDG des Programms und leitet daraus alle zu überdeckenden Testziele einschließlich ihrer jeweiligen CDPTs ab. Daraufhin wird eine initiale Testfallmenge zufällig generiert. Um jedem Testfall einen Fitnesswert im Sinne der Fitnessfunktion des Genetischen Algorithmus zuzuordnen, werden alle Testfälle einzeln ausgeführt und dabei die jeweils überdeckten Verzweigungen (in Analogie zu den Bedingungen der CDPTs) protokolliert. Die Fitness eines Testfalls ist dann proportional zur Anzahl der Bedingungen, die das zugehörige Protokoll und der CDPT des aktuellen Testziels gemeinsam haben. Auf diese Weise ist das gesamte Verfahren unabhängig von der Anzahl der möglichen beziehungsweise vom Testfall tatsächlich ausgeführten Schleifeniterationen bei zyklischen Kontrollflüssen - was für die Betrachtung der Anweisungs- und Verzweigungsüberdeckung ohnehin irrelevant ist, die Anwendbarkeit dieser Methode aber zugleich auf ähnlich einfache Kriterien beschränkt. Die für Genetische Verfahren typische Evolutionsschleife (Abbildung 4.3) wird solange wiederholt, bis entweder das gesetzte Testziel oder eine vorgegebene maximale Anzahl Versuche pro Testziel erreicht wurde. Anschließend wird ein neues Testziel ausgewählt und der Prozess der Evolution beginnt von vorne. Der gesamte Algorithmus endet, sobald alle Testziele erreicht oder eine vorgegebene maximale Rechenzeit abgelaufen ist.

Aufbauend auf den Vorarbeiten aus [PHP99] wird in [Ton04] eine Übertragung des Verfahrens auf das strukturelle Modultesten objekt-orientierter Softwarekomponenten (genauer: jeweils einer einzelnen Klasse) vorgestellt. Auch bei dieser Methode werden zunächst die Testziele identifiziert und anschließend einzeln verfolgt. Eine Neuerung stellt jedoch die Codierung der Individuen dar, welche für den Test prozeduraler Programme lediglich eine Reihe von Eingabedaten darstellen, nun jedoch speziell auf die Anforderungen beim Testen einer Klasse zugeschnitten sind. Dabei wird insbesondere berücksichtigt, dass vor dem Aufruf der zu testenden Methode eventuell eine Instanz der entsprechenden Klasse erstellt und in einen geeigneten Zustand gebracht werden muss, was das Aufrufen weiterer Methoden dieser Instanz und damit womöglich die Instantiierung weiterer Klassen erfordert. Somit ist ein Testfall bei [Ton04] eine Abfolge von Objektinstantiierungen (das zu testende Objekt oder notwendige Parameter), Methodenaufrufen (um die Objekte in einen geeigneten Zustand zu bringen) und dem Aufruf der zu testenden Methode. Das ursprüngliche Grundverfahren ([PHP99]) neigt dazu, „redundante“ Testfälle zu erzeugen, indem eventuell später in die Ergebnismenge aufgenommene Testfälle ebenfalls Testziele überdecken, welche bereits in vorangehenden Schritten verfolgt wurden. Dieses Problem wird in [Ton04] dadurch abgemildert, dass überzählige Testfälle nach Abschluss der Testfallgenerierung verworfen werden. Dazu wird ein einfacher *Greedy*-Algorithmus eingesetzt, welcher die Testfälle in der Reihenfolge ihres größten Überdeckungsbeitrags in die Ergebnisliste aufnimmt und die übrigbleibenden Testfälle verwirft. Das Verfahren wurde in einem Werkzeug namens *ETOC* umgesetzt und zur Ermittlung von Testfällen zur Verzweigungsüberdeckung einzelner JAVA™-Klassen eingesetzt.

Eine Mischform aus *pfad-* und *ziel-orientierten* Generatoren wird in [WBP02] vorgestellt. Die Autoren erheben den Anspruch, das erste Verfahren entwickelt zu haben, welches Test-

daten mittels Evolutionärer Verfahren auf Basis beliebiger kontroll- und datenflussorientierter Teststrategien generiert. Ähnlich [JSE96] nutzt diese Methode den Kontrollflussgraphen eines Programms, um die einzelnen Testziele entsprechend dem vorgegebenen Überdeckungskriterium zu identifizieren, wobei ein Testziel eine beliebige Programmstruktur (zum Beispiel eine Anweisung) darstellt, deren Ausführung notwendig zur Erfüllung des Kriteriums ist. Für jedes noch offene Ziel wird mittels eines Genetischen Algorithmus nach einem Testdatum gesucht, welches die zu überdeckende Programmstruktur ausführt. Die dem Evolutionären Verfahren zugrunde liegende Fitnessfunktion besteht in diesem Ansatz jedoch aus zwei Teilbeiträgen: Der sogenannten *Annäherungsstufe (approximation level)* und dem *lokalen Abstand (local distance)* [Bar00]. Für jeden Testfall `t` gibt die Annäherungsstufe die minimale Anzahl der Verzweigungen wieder, die zwischen den von `t` tatsächlich überdeckten und der eigentlich zu überdeckenden Programmstruktur liegen. Dabei werden lediglich diejenigen Verzweigungen berücksichtigt, die eine ausgehende Kante besitzen, deren Ausführung zwangsweise zum Verfehlen des Ziels führt. Zur Veranschaulichung betrachte man den Kontrollflussgraphen aus Abbildung 3.1: Falls n_7 das gesetzte Testziel ist, dann erhält ein Testfall, welcher lediglich die Kante (n_4, n_5), jedoch *nicht* (n_4, n_6) überdeckt, eine niedrigere Annäherungsstufe (hier: Stufe 2) als ein Testfall, welcher die Kante (n_6, n_8) aber nicht (n_6, n_7) überdeckt (Stufe 1). Der zweite Beitrag zur Fitnessfunktion, der sogenannte *lokale Abstand*, ist im Wesentlichen vergleichbar mit der bei [MMS98] verwendeten Funktion, wie sie exemplarisch auf Seite 24 dieser Arbeit vorgestellt wurde.

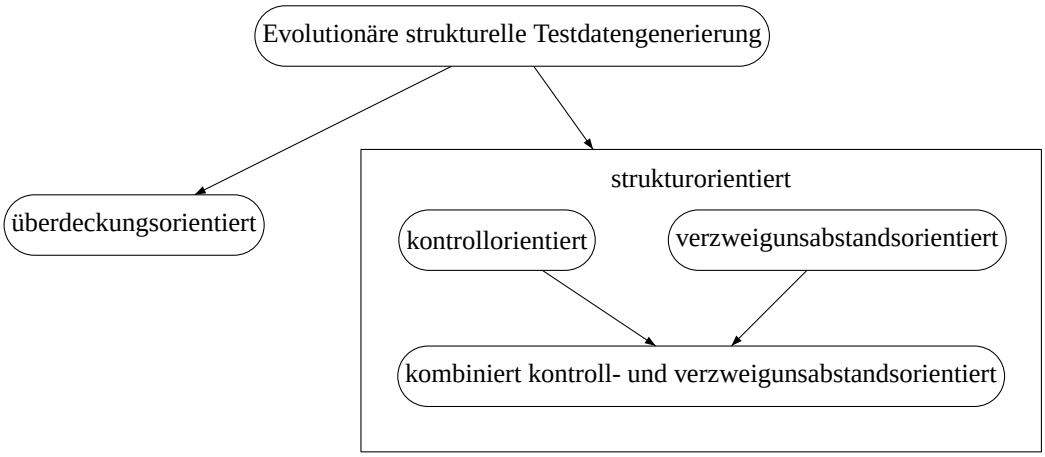


Abbildung 2.4: Klassifikation: Testdatengeneratoren mittels Evolutionärer Algorithmen

Betrachtet man nur die Testdatengeneratoren, welche auf Evolutionären Verfahren beruhen, so lassen sie sich nach [McM04] wie in Abbildung 2.4 dargestellt, klassifizieren. Die Bezeichnungen in den Knoten beziehen sich dabei auf die Informationsquellen zur Berechnung der Fitnessfunktion. So wird die Fitness eines Testfalls bei den „überdeckungsorientierten“ (*coverage-oriented*) Generatoren allein aufgrund der von diesem Testfall erreichten Überdeckung bestimmt, wie zum Beispiel aus der Anzahl der Verzweigungen bei der Verzweigungsüberdeckung. Die Fitnessfunktion der „kontrollorientierten“ (*control-oriented*) Verfahren beruht auf der Anzahl der vom Testfall überdeckten Strukturelemente, welche entlang des Pfades vom Startknoten zum

erwünschten Zielknoten zwingend überdeckt werden müssen. Das Verfahren nach [PHP99] ist ein typischer Vertreter dieser Klasse, da die Fitness eines Testfalls mit der Anzahl der Knoten/-Verzweigungen vom Startknoten zum Zielknoten/zur Zielkante im *CDG* identifiziert wird. Die beiden Varianten nach [MMS98] und [JSE96] gehören der Klasse der „verzweigunsabstands-orientierten“ (*branch-distance-oriented*) Vorgehen an. Dabei spiegelt die Fitness eines Testfalls den „Abstand“ des tatsächlich ausgeführten Pfades vom erwünschten Pfad (welcher zum Testziel geführt hätte) wider, indem hier lediglich die letzte Bedingung berücksichtigt wird, deren Auswertung zur Ausführung einer zwischen den beiden Pfaden abweichenden Kante geführt hat. Die Vorteile beider letztgenannten Methoden vereinigt die Klasse der „kombiniert kontroll- und verzweigunsabstandsorientierten“ Ansätze (*combined control and branch distance approaches*), die sowohl den bereits überdeckten Teilpfad auf dem Pfad zur Zielstruktur als auch die letzte abweichende Verzweigung in den Fitnesswert einfließen lassen, wie dies bei [WBP02] der Fall ist.

Zusammenfassende Abgrenzung

Vergleicht man das in dieser Arbeit präsentierte Verfahren mit den bestehenden Ansätzen, von denen vorangehend unterschiedliche, repräsentative Varianten vorgestellt wurden, so lässt es sich in folgenden Aspekten davon abgrenzen:

- Erstmals wird die klassische Testfallgenerierung um die Optimierung von Testfallmengen zu einem integrierten Vorgehen erweitert, welches sowohl die Maximierung der von den Testfällen erreichten Überdeckung als auch die gleichzeitige Minimierung der Größe dieser Menge anstrebt. Erreicht wird dies durch Einsatz multi-kriterieller Such- und Optimierungsheuristiken.

- Einen bedeutenden Fortschritt stellt die Erweiterung des Einsatzgebietes der automatischen Testdatengenerierung auf Programmeinheiten beliebiger Größe und Komplexität dar, selbst wenn dem Testprozess strukturelle Kriterien zugrunde gelegt werden. Dies wird unter anderem dadurch erreicht, dass das vorgestellte Verfahren nicht auf eine statische Analyse des zu testenden Objektes zur Bestimmung der einzelnen Testziele angewiesen ist (wie bei [JSE96, PHP99, Ton04]), denn mit heutigen Mitteln kann sie meist kaum noch vollständig und präzise durchgeführt werden.

- Durch den Verzicht auf eine statische Analyse als Grundlage der Testdatengenerierung wendet das hier entwickelte Verfahren keine Ressourcen für die Untersuchung von Testzielen auf, die gar nicht oder nicht innerhalb der vorgegeben Zeit erreicht werden können.

- Im Zuge der Anwendung multi-kriterieller Heuristiken ermöglicht das Verfahren die Verbesserung der Testeffizienz durch Erhöhung der Fehleraufdeckungsquote minimaler Testfallmengen mittels gleichzeitiger Verfolgung orthogonaler, einander ergänzender struktureller Testkriterien während der Testdatengenerierung.

- Das hier vorgestellte Verfahren eignet sich sowohl für klassische Programmierparadigmen als auch für moderne objekt-orientierte Ansätze. Dabei ist hervorzuheben, dass nicht nur

einfache klassische Strukturtestkriterien unterstützt werden, sondern insbesondere auch anspruchsvollere Teststrategien, welche auf den Datenfluss einzelner Softwareteile beliebiger Granularität oder gar der gesamten Applikation beruhen.

- Darüber hinaus wird ein Gesamtprozess definiert und vollständig automatisiert, welcher sich von der Testtreibergenerierung über die eigentliche Testdatengenerierung und -optimierung, über die Analyse der Fehleraufdeckungsquote mittels Mutationsanalyse und die bedarfsgerechte Erweiterung der Testfallmengen zur Verbesserung der Fehleraufdeckung bis zur Darstellung der Testfälle in einem benutzerfreundlichen Format erstreckt.

- Eine funktionsfähige Umsetzung, die den gesamten Sprachschatz von JAVA™ unterstützt, und die beeindruckende Resonanz und die Nachfrage seitens der Softwareindustrie zeigen die praktische Relevanz des Verfahrens.

Entsprechend der Klassifikation von [McM04] in Abbildung 2.4 gehört das hier vorgestellte Grundverfahren, basierend alleine auf der *„globalen Optimierung“* (siehe Kapitel 5.1), im Wesentlichen zur Klasse der „überdeckungsorientierten“ Methoden. In der voll ausgebauten Endstufe, also nach der Hybridisierung mit der *„lokalen Optimierung“* (Kapitel 5.2), handelt es sich insgesamt um einen hybriden Ansatz, welcher die beiden Hauptklassen „überdeckungsorientiert“ und „strukturorientiert“ in sich vereint. Dabei ist die Methodik so allgemein gestaltet, dass sie sich für jede Teststufe vom Modultest bis zum Systemtest eignet (siehe Abbildung 2.1 beziehungsweise Kapitel 2.2.1), solange diese dynamisch und strukturorientiert (zum Teil sogar funktionsorientiert, siehe Abbildung 2.3) erfolgt. Beschränkt wird sie lediglich durch die zur Verfügung stehenden Ressourcen wie Rechenzeit und Speicherkapazität. Darüber hinaus kann dieses Verfahren auch beim Regressionstesten während der Wartungsphase angewandt werden, wobei die bestehenden Testfälle automatisch geeignet eingeschränkt (bei entfernter Teilfunktionalität) oder nach Bedarf erweitert (bei neu hinzugekommenem Programmcode) werden, um die so geänderte Funktionalität angemessen zu testen.

Kapitel 3

Kontroll- und Datenflusstesten

„Variables won't; constants aren't."
Don Osborn

Nachdem im Kapitel 2.2 eine grobe Übersicht unterschiedlicher Teststrategien in ihrem organisatorischen Kontext mit ihren jeweiligen Testzielen präsentiert wurde, soll hier zunächst die grundlegende Terminologie definiert werden, wie sie im weiteren Verlauf der Arbeit zur formalen Darstellung der datenflussorientierten Testüberdeckungskriterien verwendet wird, mit denen sich die vorliegende Arbeit vorwiegend beschäftigt. Anschließend werden die wichtigsten Vertreter der Kontroll- und Datenflussteststrategien formal beschrieben und die hier betrachteten genauer untersucht. Das Kapitel wird um eine Beschreibung wichtiger Problemfelder ergänzt, auf welche man bei der Behandlung und Umsetzung von Datenflussstrategien im Rahmen objekt-orientierter Programmiersprachen stößt und skizziert mögliche Lösungsansätze. Es schließt mit einer zusammenfassenden Einordnung der wichtigsten Kriterien in einer hierarchischen Subsumptionsrelation. Als Ausblick wird eine orthogonale Teststrategie vorgestellt, welche im Rahmen des hier entwickelten Verfahrens zur nachträglichen objektiven Bewertung der Fehleraufdeckungsfähigkeit der generierten Testfallmengen eingesetzt wird.

3.1 Basisterminologie

Vergleicht man diverse Literaturquellen, stellt man unweigerlich fest, dass selbst grundlegende Begriffe aus dem Bereich des Softwaretestens (noch) nicht ausreichend standardisiert sind und daher oft mit unterschiedlicher Bedeutung verwendet werden. Um Abhilfe bemüht sich das *„International Software Testing Qualifications Board" (ISTQB)*, dem 2004 die Veröffentlichung einer ersten Version des *ISTQB Glossary of Testing Terms* gelang, welches jedoch bereits ein Jahr später überarbeitet wurde.

Zunächst sei das zu testende Programm(teil) etwas genauer beleuchtet [AO00]:

Definition 3.1 (Testobjekte) *Eine Einheit (oder* unit*) bezeichnet in dieser Arbeit je nach zugrundeliegendem Programmierparadigma eine einzelne Prozedur, Funktion oder Methode. Ein*

Modul *ist eine Sammlung verwandter Einheiten, wie zum Beispiel eine Datei in der Sprache* C *oder eine Klasse in* JAVA^{TM}*. Zuweilen wird der Begriff* Modul *auch für eine einzelne Einheit (Methode) oder eine ganze Sammlung mehrerer Klassen verwendet (z.B. ein Package in* JAVA^{TM}*).*

Ein zentraler Begriff der vorliegenden Arbeit ist der sogenannte *Testfall*. Das *ISTQB Glossary of Testing Terms* übernimmt die Definition des Standards IEEE 610 wie folgt:

Definition 3.2 (Testfall) *Ein Testfall umfasst die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts), die Menge der erwarteten Ergebnisse (z.B. Ausgabewerte), die Prüfanweisung (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind) sowie die erwarteten Nachbedingungen und hat zum Ziel, einen bestimmten Programmpfad zu überdecken oder die Erfüllung einer bestimmten Anforderung zu überprüfen.*

Definition 3.3 (Testlauf) *Ein Testlauf bezeichnet die Ausführung eines Testobjektes mit einem Testfall.*

Definition 3.4 (Testfallmenge) *Eine Testfallmenge stellt hier eine Menge mehrerer, in sich vollständiger und unabhängiger Testfälle dar. Die Testfälle einer Testfallmenge sind insofern unabhängig, als dass die Reihenfolge ihrer Ausführung keinen Einfluss auf die Ergebnisse jedes einzelnen Testlaufs hat.*

In der Literatur (zum Beispiel im Kontext des xUnit-Framework) wird der Begriff *Testsuite* für eine Menge einzelner „Testcases“ verwendet. Typischerweise müssen diese einzelnen Testfälle einer Testsuite nicht voneinander unabhängig sein, das heißt, jeder Testfall greift auf die Daten und den Zustand des Testobjekts zurück, die die vorhergehenden Testfälle nach ihrer Ausführung hinterlassen haben. Im Gegensatz dazu, werden die Testfälle einer Testfallmenge nach Definition 3.4 einzeln (daher müssen sie „in sich vollständig“ sein) jeweils mit dem gleichen Ausgangszustand des Testobjekts ausgeführt.

Da die erwarteten Ergebnisse eines Testlaufs nur in seltenen Fällen automatisiert ermittelt werden können, lassen sich für die hier entwickelte automatische Testgenerierung zwei Kernkomponenten eines Testfalls unterscheiden:

- *Testszenario:* Das Testszenario beschreibt den wesentlichen Ablauf eines Testfalls. Werden zum Beispiel Zustände und Zustandsübergänge eines Objektes mittels eines endlichen Automaten (FSM: *finite state machine*) beschrieben, deren Transitionen den Methodenaufrufen entsprechen, so können aus dieser FSM einzelne Folgen von Transitionen (also Methodenaufrufen) abgeleitet werden, welche jeweils unterschiedliche Testszenarien darstellen.

- *Testdaten:* Testdaten repräsentieren die tatsächlichen Eingabewerte an das zu testende System. Betrachtet man erneut die Beschreibung eines Moduls mittels einer FSM, so benötigt man im Allgemeinen für jeden Methodenaufruf eine Reihe von Parametern – die Testdaten.

Oftmals sind Testdaten und Testszenarien nicht explizit getrennt darstellbar. Dies ist zum Beispiel beim Testen von Komponenten oder Applikationen der Fall, bei denen das Verhalten des Programms nur von den Eingabedaten selbst abhängt. Somit repräsentieren die Testdaten zugleich ein implizites Testszenario. Während die Ableitung von Testszenarien, zum Beispiel aus endlichen Automaten, relativ einfach ist, besteht die Schwierigkeit der automatischen Generierung von Testfällen in der Bestimmung derjenigen Testdaten, die genau die Ausführung des geforderten Testszenarios ermöglichen. Als Beispiel stelle man sich den Zustand eines Objektes vor, welcher mit dem gleichen Methodenaufruf in unterschiedliche Nachfolgezustände überführt werden kann: In einem endlichen Automaten sind beide Transitionen mit dem gleichen „Trigger“ (Methodenaufruf) gekennzeichnet – welcher Übergang tatsächlich stattfindet, hängt von den tatsächlichen Parametern des Aufrufs ab, welche in Form eines „Guards“ den bedingten Übergang beeinflussen.

Alle Teststrategien, die der Klasse der strukturellen Testverfahren angehören (sogenannte White-Box-Tests), beziehen sich definitionsgemäß unmittelbar auf den Programmcode sowie dessen Ausführung. Sie repräsentieren demnach jeweils eine Vorschrift, die vorgibt, welche Anweisungen eines Programmes in welcher Reihenfolge von einer Testfallmenge auszuführen sind. Leider enthalten sie keinerlei Hinweise darauf, *welche* Testfälle zu ihrer Erfüllung auszuwählen sind. Die funktionalen Eigenschaften einer Implementierung sind dabei nur von nachgelagertem Interesse und werden erst bei der Bewertung der Testergebnisse herangezogen.

Zu diesen Teststrategien gehören sowohl die rein kontrollflussbasierten Verfahren, wie zum Beispiel die Verzweigungsüberdeckung, als auch Teststrategien, welche lediglich auf den Kontrollfluss aufbauen, diesen um weitere Aspekte anreichern und deren Vorschriften sich auf den annotierten Kontrollflussgraphen beziehen, zu denen die hier behandelte Klasse der Datenflussüberdeckungskriterien zählt.

Weil die Definition der Anforderungen dieser strukturellen Kriterien anhand des textuellen Programmcodes und dessen dynamischer Ausführung nur schwerlich gelingt, repräsentiert man Programme visuell mittels sogenannter Kontrollflussgraphen. Da sich die vorliegende Arbeit mit aktuellen Programmierparadigmen beschäftigt und die Konzepte anhand der Programmiersprache JAVA™ erarbeitet werden, beziehen sich die folgenden Definitionen auf objekt-orientierte prozedurale Programme. Diese bestehen hier in ihrer kleinsten atomaren Einheit aus einzelnen Ausdrücken[1] (engl. *expression*), beispielsweise Addition zweier Zahlen und Zuweisung des Ergebnisses an eine Variable, Verzweigungs- oder Schleifenkonstrukte, welche zu komplexeren Funktionen (sogenannten „Methoden“ im Falle objekt-orientierter Programme) zusammengesetzt werden. Eine Menge solcher Methoden ist auf der nächsthöheren Strukturierungsebene Bestandteil der sogenannten Klasse, einem Verbund aus Variablen (den Feldern) und Methoden, welcher zur Ausführungszeit mehrfach instantiiert werden kann. Die für die Objekt-Orientierung typischen Elemente Vererbung und Polymorphie werden in speziell erweiterten Kontrollflussgraphen ebenfalls berücksichtigt.

Definition 3.5 (Kontrollflussgraph) *Ein Kontrollflussgraph* $G := (N, E)$ *ist ein gerichteter Graph, bestehend aus Knoten* $n_i \in N$ *mit* $i = 1, \ldots, |N|$ *und Kanten* $e_j \in E$ *mit* $j = 1, \ldots, |E|$ *und* $E \subseteq N \times N$.

[1] Eine feinere Granularität wird nur in den Annotationen berücksichtigt.

Jede Kante $e_j = (n_a, n_z) \in E$ stellt eine gerichtete Verbindung zwischen den Knoten $n_a \in N$ und $n_z \in N$ dar. Dabei werden n_a als Vorgänger *von n_z und n_z als* Nachfolger *von n_a bezeichnet.*

Ein Knoten n_i repräsentiert hierbei eine maximale[2] Abfolge von Anweisungen $\langle a_1^{n_i}, \ldots, a_k^{n_i} \rangle$, so dass diese in jeder möglichen Programmausführung stets in der gleichen zusammenhängenden Reihenfolge vollständig auftreten. Falls also der Knoten n_i mehrere Anweisungen darstellt ($k > 1$), dann ist für alle $s = 2, \ldots, k$ die Anweisung $a_s^{n_i}$ die einzige die immer unmittelbar auf Anweisung $a_{s-1}^{n_i}$ folgt und umgekehrt ist $a_{s-1}^{n_i}$ die einzige Anweisung die stets vor $a_s^{n_i}$ ausgeführt wird.

Eine gerichtete Kante $e_j = (n_a, n_z)$ verbindet den Ausgangsknoten n_a der Kante mit ihrem Zielknoten n_z und stellt somit einen Transfer der Ausführungskontrolle von einer Anweisungssequenz zur nächsten dar, das heißt, es gibt mindestens eine mögliche Ausführung des Programms, bei der zunächst Anweisungsblock n_a und anschließend Block n_z ausgeführt werden.

```
1   int low = 0;
2   int high = list.length - 1;
3   int mid = -1;
4   boolean found = false;
5   while ((low <= high) && !found) {
6      mid = (low + high) / 2;
7      if (list[mid] == searchitem) {
8         found = true;
9      } else if (list[mid] < searchitem) {
10        low = mid+1;
11     } else {
12        high = mid-1;
13     }
14  }
15  if (found) {
16     System.out.println("found␣at␣" + mid);
17  } else {
18     System.out.println("not␣found");
19  }
```

Listing 3.1: Quellcodeausschnitt des Programms *BinarySearch*

Betrachtet man den Programmausschnitt aus Listing 3.1, so wird dieser durch den Kontrollflussgraphen in Abbildung 3.1 leicht vereinfacht dargestellt. Eine Besonderheit des gezeigten Graphen stellen die Knoten n_{Start} und n_{End} dar. Sie repräsentieren die Eintritts- bzw. Austrittspunkte des Kontrollflusses dieses Programmausschnittes, das heißt, jede Ausführung des Programmteils beginnt stets mit Knoten n_{Start} und endet mit n_{End}, weshalb n_{Start} als einziger Knoten keinen Vorgänger und n_{End} keinen Nachfolger besitzt. Bei der Einbettung dieses Ausschnittes in

[2] Zuweilen wird eine solche Abfolge auch durch mehrere sequentielle Knoten dargestellt, um einzelne Anweisungen hervorzuheben oder ihre gesamte Reihenfolge zu betonen.

die gesamte Applikation stellen diese beiden Knoten die Verbindungspunkte zum umliegenden Kontrollfluss dar. Im Rahmen der Datenflusskriterien werden ihnen zusätzliche Variablendefinitionen bzw. -verwendungen zugeordnet, die nicht textuell im Codeausschnitt zu finden sind – sie kennzeichnen damit den sogenannten Import bzw. Export der verwendeten Variablen.

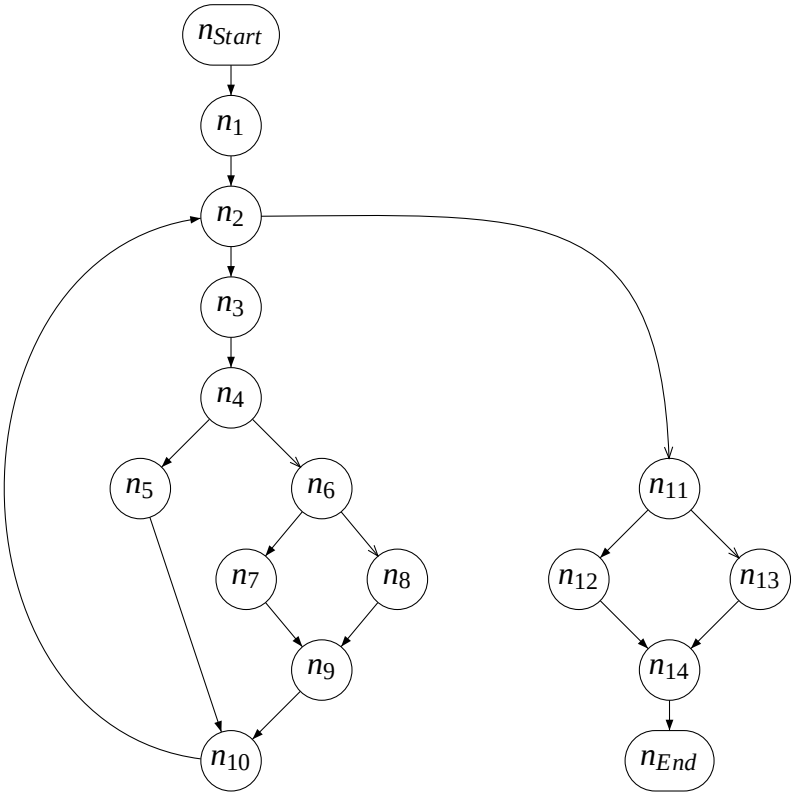


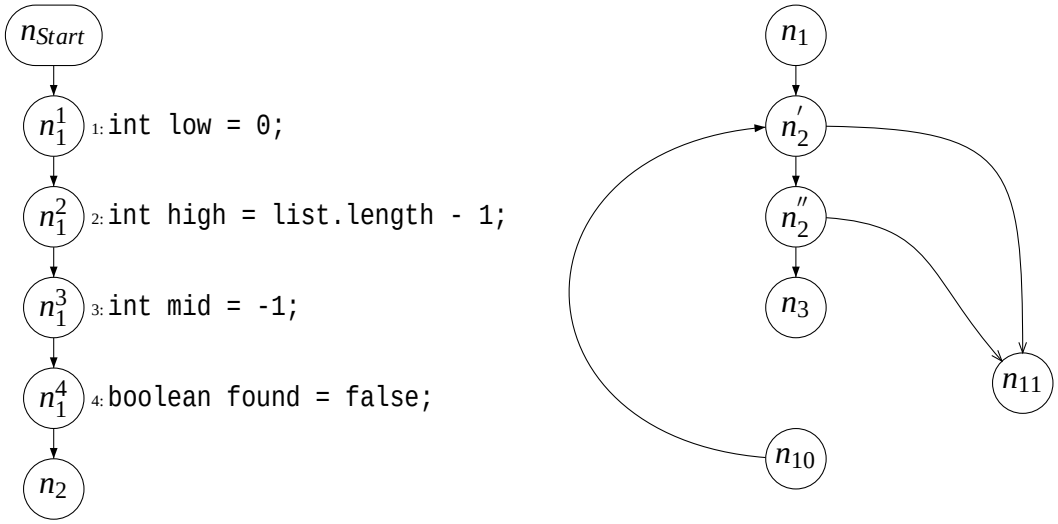
Abbildung 3.1: Kontrollflussgraph des Programms *BinarySearch*

Weiterhin wird die Abfolge der Initialisierungsanweisungen in den Zeilen 1 bis 4 durch den Knoten n_1 zusammengefasst, da diese Anweisungen stets in der gleichen Reihenfolge und stets zusammenhängend ausgeführt werden. Alternativ kann der Knoten n_1 auch aufgespaltet werden, wie in Abbildung 3.2(a) zu sehen. Knoten n_2 stellt symbolisch die Verzweigung des Kontrollflusses dar, welche aufgrund der `while`-Schleifenbedingung entsteht: Falls die Bedingung beim Erreichen der Anweisung während der Ausführung zu wahr ausgewertet wird, dann wird das Programm mit der dem Knoten n_3 zugeordneten Anweisung in Zeile 6 fortgesetzt, sonst springt der Kontrollfluss nach Zeile 15, welche durch Knoten n_{11} repräsentiert wird[3]. Analog entsprechen die Verzweigungen in den Knoten n_4, n_6 und n_{11} den einfachen Verzweigungsanweisungen (`if`) in den Zeilen 7, 9 beziehungsweise 15.

Eine Vereinfachung der Darstellung des vom Programm zur Ausführungszeit tatsächlich

[3]Im Kontrollflussgraphen werden *wahr*- und *falsch*-Kanten zur Verdeutlichung mit unterschiedlichen Pfeilspitzen dargestellt.

durchlaufenen Kontrollflusses in Abbildung 3.1 ist die Abstraktion der Auswertung der zusammengesetzten Bedingung „`(low <= high) && !found`“ in Zeile 5, welche im Graphen durch den Knoten n_2 repräsentiert wird. Diese Vereinfachung ist nur dann korrekt, wenn komplexe Bedingungen dieser Art stets vollständig ausgewertet werden, insbesondere auch im Falle einer Ausnahme während ihrer Auswertung. Moderne Compiler könnten an dieser Stelle eine Optimierung bezüglich der Ausführungsgeschwindigkeit vornehmen und eine so genannte *Kurzschlussauswertung* (*shortcut evaluation* oder *partial evaluation*) umsetzen, sofern die Sprachspezifikation nichts anderes vorschreibt. Demnach würde eine zusammengesetzte Bedingung nur soweit evaluiert, wie es zur Feststellung des Gesamtergebnisses vonnöten ist. Neuere Sprachen, dazu gehört auch JAVA™, schreiben eine solche Auswertung für bestimmte Operatoren (z.B. „&&“) sogar explizit vor und bieten für die vollständige Auswertung einen eigenen Operator („&“ mit Boole'schen Operanden) an. Im vorliegenden Beispiel kann also auf die Auswertung der Teilbedingung „`!found`“ verzichtet werden, wenn die linke Seite der Konjunktion bereits als falsch erkannt wurde, wodurch der Knoten n_2 zur korrekte Darstellung der Bedingungsauswertung wie in Abbildung 3.2(b) skizziert, zu ersetzen ist. Dabei repräsentiert Knoten n_2' die Auswertung der linken Teilbedingung „`(low <= high)`“ während n_2'' für die Evaluierung der rechten Teilbedingung „`!found`“ steht.



(a) Initialisierungssequenz (b) `While`-Bedingung

Abbildung 3.2: Ausführliche Darstellung zusammengesetzter Anweisungssequenzen

Kontrollflussbasierte Testüberdeckungskriterien fordern zu ihrer Erfüllung eine Testfallmenge, welche bestimmte Knoten oder Kanten des Kontrollflussgraphen zur Ausführung bringen müssen. Komplexere Kriterien dieser Art erfordern jedoch die Überdeckung einer bestimmten Abfolge von Knoten, sogenannter (Teil)Pfade.

Definition 3.6 (Teilpfad) *Ein Teilpfad* $p = (n_1^p, n_2^p, \ldots, n_m^p)$ *eines Kontrollflussgraphen* $G = (N,$

E) ist eine endliche Folge von Knoten $n_i^p \in N$ mit $i = 1,\ldots,m$, so dass es je eine Kante $e_j = (n_k^p, n_{k+1}^p) \in E$ mit $k = 1,\ldots,m-1$ gibt.

Definition 3.7 (Pfad) *Ein Pfad $q = (n_{Start})p(n_{End}) := (n_{Start}, n_1^p, n_2^p, \ldots, n_m^p, n_{End})$ ist ein spezieller Teilpfad, welcher mit dem ersten Knoten n_{Start} des Kontrollflussgraphen beginnt und mit dem letzten Knoten n_{End} endet, wobei $p = (n_1^p, n_2^p, \ldots, n_m^p)$ ein Teilpfad ist und $(n_{Start}, n_1^p) \in E$ sowie $(n_m^p, n_{End}) \in E$ gilt.*

Jeder vollständigen Programmausführung entspricht demnach ein Pfad durch den Kontrollflussgraphen. Für bestimmte Kontroll- und Datenflussüberdeckungskriterien sind Pfade beziehungsweise Teilpfade mit besonderen Eigenschaften von Bedeutung:

Definition 3.8 (Einfacher Teilpfad) *Ein Teilpfad $p = (n_1^p, n_2^p, \ldots, n_m^p)$ eines Kontrollflussgraphen $G = (N,E)$ heißt* einfach*, falls alle Knoten $n_i^p \in N$, außer eventuell der erste Knoten n_1^p und der letzte Knoten n_m^p, jeweils paarweise verschieden sind.*

Definition 3.9 (Schleifenfreier Teilpfad) *Ein Teilpfad $q = (n_1^q, n_2^q, \ldots, n_m^q)$ heißt* schleifenfrei*, falls alle seine Knoten paarweise verschieden sind, also $\forall i \neq j : n_i^q \neq n_j^q$ mit $i, j \in \{1,\ldots,m\}$.*

Im Graphen aus Abbildung 3.1 sind p_1 = (n_4, n_6, n_7, n_9, n_{10}) und p_2 = (n_5, n_{10}, n_2, n_3, n_4, n_5) zwei Teilpfade des Kontrollflussgraphen für den Programmausschnitt aus Listing 3.1. Dabei ist p_1 schleifenfrei während p_2 einen einfachen Teilpfad darstellt, da im Falle von p_2 der erste und der letzte Knoten identisch sind, alle anderen jedoch paarweise verschieden. Im gleichen Graphen ist p_3 = (n_{Start}, n_1, n_2, n_3, n_4, n_6, n_8, n_9, n_{10}, n_2, n_{11}, n_{13}, n_{14}, n_{End}) ein Pfad.

Definition 3.10 (Überdeckung) *Eine Anweisung heißt von einer Testfallmenge beziehungsweise einem einzelnen Testfall* überdeckt*, falls die Testfallmenge mindestens einen Testfall enthält, so dass bei der Ausführung des Programms mit diesem Testfall, die entsprechende Anweisung mindestens einmal ausgeführt wird.*

Für einen Kontrollflussgraphen $G = (N,E)$ gilt analog: Ein Knoten $n \in N$, eine Kante $e = (n_a, n_z) \in E$ beziehungsweise ein (Teil)Pfad $p = (n_1^p, n_2^p, \ldots, n_m^p)$ heißen von einem Testfall überdeckt, falls alle Anweisungen des Knotens n beziehungsweise alle Anweisungen der Knoten n_a und n_z sowie der Knoten $n_1^p, n_2^p, \ldots, n_m^p$ in genau dieser Abfolge ausgeführt werden.

Betrachtet man im gleichen Beispiel den Knoten n_5 näher, so kann man anhand des Graphen leicht unbegrenzt viele Teilpfade finden, welche diesen Knoten mit irgendeinem der Knoten n_2 bis n_{End} verbindet, da der Schleifenrumpf beliebig oft und mit unterschiedlichen inneren Teilpfaden wiederholt werden kann. Ein Beispiel für einen solchen Teilpfad stellt $p_{5/8}$ = (n_5, n_{10}, n_2, n_3, n_4, n_6, n_7, n_9, n_{10}, n_2, n_3, n_4, n_6, n_8) dar, welcher syntaktisch abgeleitet werden kann und graphentheoretisch die beiden Knoten n_5 und n_8 miteinander verbindet. Tatsächlich aber kann dieser Teilpfad nie zur Ausführung gebracht werden, da es aufgrund der Schleifenbedingung keine Eingabe gibt, für die der Schleifenrumpf erneut betreten wird, nachdem Knoten n_5 beziehungsweise Zeile 8 des Programms ausgeführt wurde. Demnach können (Teil)Pfade hinsichtlich ihrer tatsächlichen Ausführbarkeit wie folgt unterschieden werden:

Definition 3.11 (feasibility) *Ein Teilpfad heißt* ausführbar *(oder engl.* feasible*), falls es mindestens einen Testfall gibt, bei dessen Ausführung der Teilpfad überdeckt wird, ansonsten bezeichnet man den Teilpfad als* nicht ausführbar *(beziehungsweise* infeasible*).*

Ob ein syntaktisch aus dem Graphen abgeleiteter (Teil)Pfad auch tatsächlich ausführbar ist, kann in einfacheren Fällen durch eine statische „semantische“ Analyse [GG02] bestimmt werden, zum Beispiel durch Aufbauen und Lösen von (Un)Gleichungssystemen über Verzweigungsbedingungen für einzelne Pfade oder mittels symbolischer Auswertung. Solche Untersuchungen sind im Allgemeinen jedoch sehr aufwändig, so dass sie mit heutigen Rechenkapazitäten zum Teil nicht mehr bewältigt werden können – im Falle von Schleifen gelingt diese Analyse meist gar nicht, da die Anzahl der Schleifendurchläufe oft nicht a priori bekannt ist [Grz04].

Die bisher eingeführten Konzepte eignen sich sehr gut zur Darstellung des Kontrollflusses innerhalb in sich abgeschlossener und voneinander unabhängiger Funktionen oder Prozeduren, wie sie in klassischen prozeduralen Programmiersprachen (zum Beispiel C) ohne Ausnahmekonstrukte als Strukturierungsmittel eingesetzt werden. Der Fokus der vorliegenden Arbeit liegt jedoch auf objekt-orientierten Paradigmen, deren zentrales Element die Daten sind, während die Operationen auf diesen Daten in den einzelnen Funktionen, den sogenannten Methoden, untergebracht sind. Ebenso wie in prozeduralen Programmiersprachen kommt es hierbei häufig vor, dass einzelne Methoden lediglich einfachste Funktionalität besitzen. Um komplexere Dienste zu erbringen, müssen die Methoden einander aufrufen, womit sie voneinander abhängig sind und wodurch sich der Kontrollfluss erheblich komplexer gestaltet.

Darüber hinaus sind weitere wesentliche Merkmale der Objekt-Orientierung die sogenannte *Vererbung*, die *Polymorphie* beziehungsweise das *dynamische Binden* und die *Ausnahmebehandlung*. Die Vererbung dient sowohl der übersichtlicheren Gestaltung der Programmarchitektur als auch der einfachen Wiederverwendung und Ergänzung bestehender Funktionalität. Dabei kann eine Unterklasse sowohl Datenfelder als auch Methoden der übergeordneten Klasse erben, wodurch Funktionen, welche in der Oberklasse deklariert, also textuell vorhanden sind, so verwendet werden können, als gehörten sie zur Unterklasse. Insbesondere können nun neue Methoden der Unterklasse auf Datenfelder zugreifen, welche in den Oberklassen oder deren Methoden definiert wurden, was das Nachvollziehen des Datenflusses anhand einer statischen Analyse (siehe Kapitel 5.2.1) erheblich erschwert.

Außerdem können einzelne Methoden der Oberklassen in den Unterklassen überschrieben werden, was zusammen mit der Polymorphie eine deutliche Erhöhung der Komplexität des statisch identifizierbaren Kontrollflusses zur Folge hat. Mittels Polymorphie können Methodenaufrufe der Form `objektReferenz.methode()` transparent implementiert werden, das heißt, es wird aus statischer Sicht lediglich eine Instanz einer typkompatiblen Klasse für `objektReferenz` erwartet, deren Methode ausgeführt werden soll. Während also an der Stelle des Funktionsaufrufs im Programmcode der Aufruf einer scheinbar durch den Typ von `objektReferenz` vorgegebenen Methode festgestellt werden kann, können sich zur Laufzeit verschiedene Instanzen beziehungsweise sogar Objekte verschiedenen Typs (nämlich auch Instanzen aller Unterklassen) als Empfänger der Nachricht ausgeben, wodurch eine größere Menge möglicher Methoden als potentielle Aufrufkandidaten berücksichtigt werden muss.

Zwar sind die hier vorgestellten Konzepte weitgehend unabhängig von der Wahl einer be-

stimmten Programmiersprache, jedoch soll hier exemplarisch JAVA™ als Leitsprache dienen. Daher müssen die bereits vorgestellten Methoden zur Darstellung des Kontrollflusses um zusätzliche Aspekte erweitert werden, um insbesondere die Notation des sogenannten interprozeduralen Kontrollflusses und der Besonderheiten der objekt-orientierten Sprachfamilie zu ermöglichen. Dabei schließt der interprozedurale Kontrollflussgraph die verschiedenen Methodenaufrufe innerhalb eines Codeblocks mit ein, im Gegensatz zum intraprozeduralen Kontrollflussgraph, welcher keine Mittel zur Darstellung von Methodenaufrufen dediziert zur Verfügung stellt. Dies gelingt für die Sprache JAVA™ mittels sogenannter *Java Interclass Graphs* (*JIG*) [HJL+01], welche auch in der Lage sind, komplexe Ausnahmebehandlungskonstrukte darzustellen [Pol04].

```
1   int i, j, c;
2   Obst oi, oj;
3   i = 0;
4   while (i < korb.length) {
5       j = 0;
6       while (j < korb.length) {
7           try {
8               oi = (Obst)korb[i];
9               oj = (Obst)korb[j];
10              c = oi.vergleicheMit(oj);
11              if (c < 0) {
12                  tausche(i, j);
13              }
14          } catch (ClassCastException e) {
15              System.err.println("Kein Obst!");
16          } finally {
17              j++;
18          }
19      }
20      i++;
21  }
```

Listing 3.2: Quellcodeausschnitt der statischen Methode *BubbleSort.sort()*

Zur Veranschaulichung des Problems soll die Sortierfunktion aus Listing 3.2 dienen, welche ein Feld von Objekten aufsteigend sortieren soll. Die Funktion erwartet, dass die zu sortierenden Objekte in der Liste `korb` typkompatibel mit der Klasse `Obst` sind, das heißt, es müssen entweder Instanzen dieser oder einer ihrer Unterklassen sein.

Betrachtet man den Programmausschnitt, so findet man in Zeile 12 den Aufruf einer statischen Methode (sog. Klassenmethode) `tausche(int, int)`, also einer Prozedur, die eindeutig in der sie umgebenden Klasse vorhanden ist und zu ihrer Ausführung kein Zielobjekt benötigt. Um diesen Sachverhalt mittels eines Kontrollflussgraphen darzustellen, kann man den Graphen der Funktion entweder direkt anstelle des Aufrufs aufführen (sog. inlining) oder mit einem Aufrufknoten verbinden. Ersteres gelingt jedoch nicht mit dem Aufruf der nicht-statischen Methode

`vergleicheMit(Object)` in Zeile 10, sobald bezüglich Klasse `Obst` zum Beispiel die in Abbildung 3.3 dargestellte Vererbungshierarchie vorliegt.

In diesem Fall kann zur Ausführungszeit anstelle des Zielobjektes `oi` sowohl eine Instanz der Klasse `Apfel` als auch ein Objekt vom Typ `Birne` genutzt werden, da beide typkompatibel zur Oberklasse `Obst` sind. Instanzen der abstrakten Klasse `Obst` können ausgeschlossen werden, weil diese Klasse ohnehin nicht instantiiert werden kann. Da außerdem die Klasse `Apfel` die Methode `vergleicheMit(Object)` überschreibt (zum Beispiel weil Äpfel zusätzlich nach `farbe` sortiert werden müssen), während die Klasse `Birne` sie lediglich von der Oberklasse erbt, müssen aus statischer Sicht beide Möglichkeiten berücksichtigt werden.

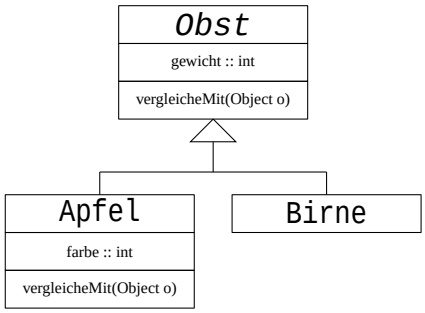


Abbildung 3.3: Vererbungshierarchie *Obst*

Diesen Sachverhalt stellt Abbildung 3.4 vollständig dar. Die Kanten (n_7, n_{17}), (n_7, n_{18}) sowie (n_9, n_{19}) sind sogenannte *Aufrufkanten* und stellen den Transfer der Kontrolle während der Ausführung zur jeweiligen Methode und *wieder zurück* dar. Welche Methoden hier überhaupt mögliche Aufrufkandidaten sind, zum Beispiel dass im Falle einer `Birne` als Zielobjekt die von `Obst` geerbte Methode `Obst.vergleicheMit(Object)` zu beachten ist, muss eine statische Analyse des Vererbungsbaums ergeben (siehe Kapitel 5.2.1). Die sich nach der Rückkehr aus den Methodenblöcken anschließenden Kanten (n_7, n_8) beziehungsweise (n_9, n_{10}) heißen *Pfadkanten* und dienen lediglich der Verdeutlichung des weiteren Programmablaufs. Sie sind daher keine einfachen Kontrollflusskanten im üblichen Sinne, denn die Kontrolle an sie ergeht streng genommen aus dem letzten Knoten des Graphen der aufgerufenen Methode. Zuweilen findet sich in der Literatur als Zielknoten der Pfadkante auch ein sogenannter *return*-Knoten, dem keine Anweisungen des Programms zugeordnet sind [HJL+01].

Darüber hinaus kann in diesem Programmausschnitt nicht sichergestellt werden, dass im Feld `korb` ausschließlich zur Klasse `Obst` typkompatible Objekte enthalten sind. Falls demnach zur Laufzeit „Fremdobjekte“ auftreten, wird der Versuch in Zeile 8 oder 9 und prinzipiell auch in Zeile 10, diese Objekte in den Typ `Obst` zu erzwingen (type-cast), mit einer *Exception* (Ausnahme) geahndet. Um die restlichen Objekte trotzdem sinnvoll sortieren zu können, wird der gefährdete Codeblock mittels des `try-catch-finally`-Konstruktes umschlossen und somit eine solche Ausnahme behandelt. Da im Allgemeinen sowohl gleiche als auch unterschiedliche Ausnahmen in verschiedenen Anweisungen innerhalb eines auf diese Art gesicherten Codeblocks auftreten können, ist es aus Gründen der Übersichtlichkeit wenig sinnvoll, von jeder solchen Anweisung

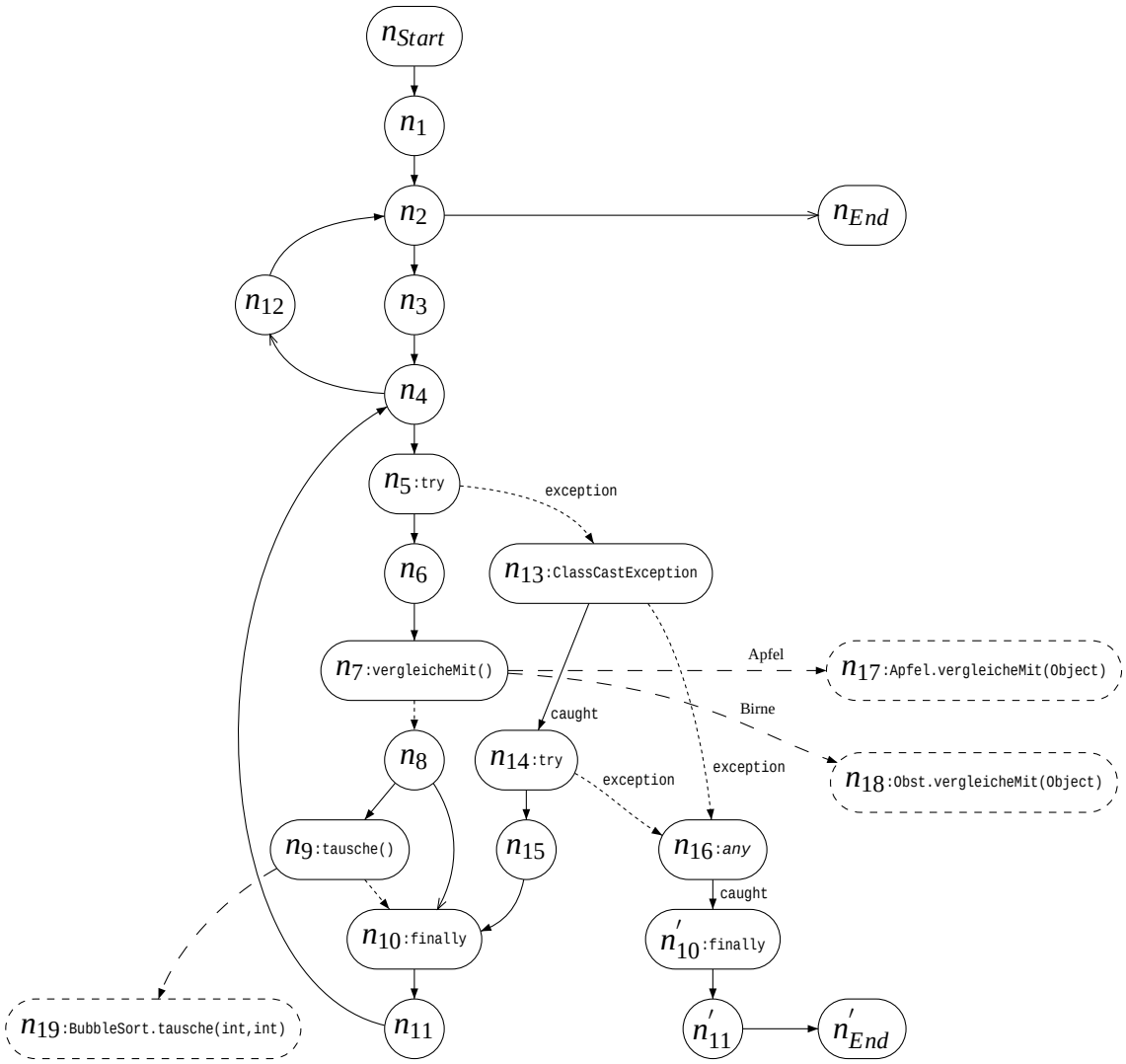


Abbildung 3.4: Kontrollflussgraph der Methode *BubbleSort.sort()*

eine Ausnahmekante zur entsprechenden Ausnahmebehandlungsroutine darzustellen. So könnte sich an den ersten Behandlungsblock vor der `finally`-Anweisung in Zeile 16 eine Reihe weiterer Ausnahmebehandlungen anschließen, welche zur Ausführung gelangen, sobald andere, mit der ersten Ausnahmeart nicht verwandte (also nicht typkompatible) Ausnahmen im gesicherten Block auftreten.

In *Java Interclass Graphs* wird das Problem auf eine Weise gelöst, wie sie exemplarisch in Abbildung 3.4 dargestellt ist. Beginn und Ende eines `try`-Blocks werden mit entsprechenden Knoten, hier n_5 und n_{10}, vermerkt, welche andeuten, dass im Falle einer Ausnahme im Kontrollfluss zwischen diesen Knoten, die dem Knoten n_5 assoziierte Ausnahmebehandlung ab Knoten n_{13} aktiviert wird. Falls mehrere Behandlungsblöcke für unterschiedliche Ausnahmearten vorgesehen wären, dann würde sich an Knoten n_{13} eine weitere `exception`-Kante ähnlich (n_{13}, n_{16}) anschließen, wobei die hier dargestellte Kante (n_{13}, n_{16}) den Fall berücksichtigt, dass irgendeine andere (*any*) als die bedachte `ClassCastException` aufgetreten ist. Ziel ist, damit zu beschreiben, dass der `finally`-Codeblock $n'_{11} = n_{11}$ unabhängig davon ausgeführt wird, ob und wenn ja, welche Ausnahme aufgetreten ist, beziehungsweise ob diese behandelt wird oder nicht. Dem gleichen Zweck dient auch der „virtuelle“ `try`-Knoten n_{14}, der in dieser Form nicht im Programmcode zu finden ist, jedoch eine eventuelle Ausnahme in der Ausnahmebehandlungsroutine n_{15} selbst berücksichtigt.

Hierbei ist jedoch n_{End} nicht mit n'_{End} gleichzusetzen, da n_{End} ausgeführt wird, falls die Methode im Falle einer ausnahmefreien Abarbeitung beendet und verlassen wird, während n'_{End} ein Beenden dieser Methode durch eine in dieser Methode nicht behandelte Ausnahme mit anschließender Weitergabe (Delegation) dieser Ausnahme an die aufrufende Methode beschreibt.

3.2 Kontrollflusskriterien

Während bei funktionalen Tests die Spezifikation als Quelle zur Bestimmung der auszuführenden Testfälle herangezogen wird, orientieren sich kontrollflussbasierte Testverfahren an der Programmstruktur. Um die Testziele zu definieren, benutzen diese Verfahren die Strukturelemente des Programmcode wie Anweisungen, Verzweigungen oder Bedingungen.

Ziel der funktionalen Tests ist es zu prüfen, ob alle laut Spezifikation erwarteten Funktionalitäten auch wirklich entsprechend der Vorgabe umgesetzt wurden. Im Gegenzug gehen strukturorientierte Tests der Frage nach, ob auch „nur“ die erwartete Funktionalität umgesetzt wurde – also ob nicht durch eine Kombination mehrerer Einzelfunktionalitäten, die jede für sich erwünscht ist, womöglich eine kritische und daher unerwünschte „Funktion“ ausgeführt werden kann. Dabei ist zu beachten, dass Testen im Allgemeinen *kein* Korrektheitsbeweis ist, sondern höchstens die Anwesenheit von Fehlern nachweisen kann.

Die ersten Ansätze, Testverfahren zu definieren, welche sich an der Programmstruktur orientieren, lassen sich bis ins Jahr 1963 zurückverfolgen [MM63]. Da die klassischen kontrollflussorientierten Testkriterien daher eine lange Tradition haben und in diversen Literaturquellen ausführlich behandelt werden, sei hier in Anlehnung an [Bal97] nur kurz das Wesentliche der verbreiteten Methoden zusammengefasst.

Man beachte folgende Unterscheidung: Sei K ein Testüberdeckungskriterium, dann bezeich-

net hier C_K je nach Kontext sowohl das Kriterium K selbst, als auch die zum Kriterium gehörende Metrik. Während das „Kriterium C_K“ konzeptionell ein Prädikat darstellt, welches entweder erfüllt ist oder nicht, gibt die „Metrik C_K“ den Anteil oder Grad der Überdeckung nach Kriterium K an. Ist also Φ die Menge der Entitäten eines zu testenden Programms, welche anhand des Kriteriums K zu überdecken sind (zum Beispiel die Menge aller Anweisungen) und $\phi \subseteq \Phi$ die Menge aller von einer Testfallmenge T tatsächlich überdeckten Entitäten, dann ist das Kriterium C_K genau dann erfüllt, wenn $\phi = \Phi$ ist, sonst ist C_K nicht erfüllt; die Metrik $C_K := \frac{|\phi|}{|\Phi|} \in [0;1]$ hingegen gibt den durch T erreichten Grad der Überdeckung nach Kriterium K an.

3.2.1 Anweisungsüberdeckung

Das einfachste Kontrollflusstestverfahren ist der *Anweisungsüberdeckungstest*, in der Literatur meist[4] als C_0-Kriterium oder *statement coverage* bezeichnet. Ziel dieses Verfahrens ist es, jede Anweisung des Programms mindestens einmal auszuführen. Dies entspricht der Überdeckung aller Knoten des Kontrollflussgraphen (siehe Kapitel 3.5, Seite 33) des betrachteten Testobjekts, welches sowohl ein vollständiges Programm als auch lediglich ein Programmausschnitt sein kann.

Sei $G = (N,E)$ der Kontrollflussgraph des Testobjekts und $N_{cov} \subset N$ die Menge aller von einer Testfallmenge überdeckten Knoten, dann beschreibt die folgende Metrik C_0 die von dieser Testfallmenge erreichte Anweisungsüberdeckung:

$$C_0 = \frac{|N_{cov}|}{|N|}$$

Der Vorteil dieses Testverfahrens ist, dass eine Abdeckung von $C_0 = 100\%$ sicherstellt, dass es nach dem Test keine Anweisung im Programm gibt, welche überhaupt nicht ausgeführt wurde. Darüber hinaus ist es mit dieser Methode möglich, nicht-ausführbare Programmteile (dead code) zu identifizieren. Als eines der einfachsten Testkriterien hat es jedoch den Nachteil, wichtige Zusammenhänge und Datenabhängigkeiten zwischen verschiedenen Programmteilen nicht zu berücksichtigen. So kann es trotz 100%iger C_0-Überdeckung noch Kanten geben, welche von der Testfallmenge überhaupt nicht ausgeführt werden, womit möglicherweise wichtige Teilpfade ungetestet bleiben.

3.2.2 Verzweigungsüberdeckung

Um den Nachteil der Anweisungsüberdeckung auszugleichen und sicherzustellen, dass alle Kanten des Kontrollflussgraphen mindestens einmal ausgeführt wurden, sollte der *Verzweigungsüberdeckungstest*, auch C_1-Kriterium beziehungsweise *branch coverage* genannt, in Betracht gezogen werden.

Für ein Testobjekt, dargestellt durch den Kontrollflussgraphen $G = (N,E)$, welches mit einer Testfallmenge T ausgeführt wurde, wobei $E_{cov} \subset E$ die Menge aller durch T überdeckten Kanten von G sei, ist die folgende Metrik C_1 ein Maß für die Vollständigkeit des Tests nach dem

[4] Manche Autoren nennen es irrtümlich C_1-Test.

Kriterium der Verzweigungsüberdeckung:

$$C_1 = \frac{|E_{cov}|}{|E|}$$

Wie die Subsumptionshierarchie in Abbildung 3.9 zeigt, ist die Anweisungsüberdeckung vollständig in der Verzweigungsüberdeckung enthalten (siehe Kapitel 3.5) – letztere gilt bei zuverlässiger Software weithin als *das* Minimalkriterium [Bal97] in der Testphase. Von Vorteil ist, dass bei diesem Verfahren auch Teilpfade getestet werden müssen, die keinen Code im herkömmlichen Sinne tragen – dies ist zum Beispiel bei Verzweigungen der Fall, deren Alternative keine Anweisungen enthalten, wie es häufig bei `IF-THEN`-Konstrukten ohne `ELSE`-Block vorkommt. Dementsprechend wird der Kontrollfluss an Verzweigungsstellen etwas intensiver untersucht als bei der Anweisungsüberdeckung, was auch das Aufdecken nicht ausführbarer Zweige im Kontrollflussgraphen ermöglicht. Der Nachteil dieses Verfahrens ist wie bei der Anweisungsüberdeckung die mangelhafte Überprüfung von Schleifenwiederholungen und zusammengesetzten Bedingungen.

3.2.3 Pfadüberdeckung

Das umfassendste Testkriterium, welches *alle* kontroll- und datenflussbasierten Verfahren subsumiert, ist die sogenannte *Pfadüberdeckung* oder *path coverage*, typischerweise mit C_∞ abgekürzt. Dieses fordert zu seiner Erfüllung die Ausführung einer Testfallmenge, welche *alle* Pfade durch den Kontrollflussgraphen überdeckt.

Sei $G = (N, E)$ der Kontrollflussgraph des Testobjekts, P die Menge aller Pfade nach Definition 3.7 von G und $P_{cov} \subset P$ die Menge aller überdeckten Pfade, dann berechnet sich die Pfadüberdeckung zu:

$$C_\infty = \frac{|P_{cov}|}{|P|}$$

Auch wenn dieses Kriterium das mächtigste kontrollstrukturorientierte Verfahren ist, garantiert auch dieses nicht die Fehlerfreiheit eines danach getesteten Programms. Da die zu überdeckende Pfadanzahl bei ineinander geschachtelten Verzweigungen exponentiell wächst und bei Schleifen potentiell gegen unendlich strebt (da jeder weitere Schleifendurchlauf mindestens einen weiteren Pfad erzeugt), besitzt dieses Kriterium für realistische Module oder gar vollständige Programme eine sehr eingeschränkte Durchführbarkeit und hat daher keine praktische Bedeutung.

Um das Kriterium dennoch praktisch nutzbar zu machen, besteht die Möglichkeit, die Anzahl der geforderten Schleifendurchläufe zu begrenzen, um somit eingeschränkte Varianten der Pfadüberdeckung zu erzielen, welche für schleifenfreie Programme mit dem C_∞-Kriterium identisch sind. Zu diesen Varianten zählen der *Strukturierte Pfadtest* ($C_{\infty,k}$) sowie die Kombination *Boundary/Interior-Pfadtest* ($C_{\infty,0}$/$C_{\infty,1}$) aus den beiden Sonderfällen des strukturierten Pfadtestes mit $k = 0$ beziehungsweise $k = 1$. Beim Strukturierten Pfadtest sind dabei alle Pfade durch den Kontrollflussgraphen auszuführen, welche im Falle von Schleifendurchläufen höchstens k

Wiederholungen der Schleifenrümpfe enthalten. Dabei müssen die Rümpfe jeder durchlaufenen Schleife selbst ebenfalls nach diesem Kriterium getestet werden – also falls eine Schleife eine weitere, geschachtelte Schleife enthält, muss diese ebenfalls bis zu k-mal wiederholt werden, ansonsten ist das Schleifeninnere nach dem Kriterium der Pfadüberdeckung zu testen. Für den Fall dass eine bestimmte, vom Kriterium geforderte Anzahl $l \leq k$ Schleifenwiederholungen aufgrund der Ausführbarkeit (Definition 3.11) nicht getestet werden kann, sieht eine weitere Auflockerung des Strukturierten Pfadtests vor, mehrere Durchgänge durch das Schleifeninnere zuzulassen, jedoch nur die ersten l Wiederholungen zur Bestimmung der Überdeckung zu betrachten.

3.3 Bedingungsüberdeckungskriterien

Die im Kapitel 3.2 vorgestellten Kriterien zur Kontrollflussüberdeckung stellen sicher, dass während des Tests zumindest jede Anweisung mindestens einmal ausgeführt wird. Höhere Kriterien fordern sogar die Überdeckung bestimmter (vollständiger) Pfade – jedoch werden selbst dabei die Bedingungen in den Verzweigungsknoten stets als atomar betrachtet, wie zum Beispiel Knoten n_2 in Abbildung 3.1 unter der Annahme vollständiger Bedingungsauswertung. Inbesondere im Falle zusammengesetzter Ausdrücke ist das eine gravierende Vereinfachung, welche erhebliches Testpotential ungenutzt lässt, zumal gerade Verzweigungsanweisungen häufige Fehlerquellen darstellen.

Um komplexe Bedingungen genauer zu testen, gibt es eine Reihe sogenannter Bedingungsüberdeckungskriterien, welche im Folgenden kurz skizziert werden. Zu Verdeutlichung diene dabei folgendes Beispiel in der Programmiersprache JAVA™: Seien `a` und `c` ganze Zahlen, `b` eine Zeichenkette sowie `d` eine boole'sche Variable (also jeweils vom Datentyp `int`, `String` beziehungsweise `boolean`) dann ist „`(a > 2 | b.length == 0) & (c != 0 | d)`“ eine zusammengesetzte Bedingung. Diese besteht aus zwei nicht-atomaren Teilbedingungen, nämlich „`a > 2 | b.length == 0`“ und „`c != 0 | d`“, welche ihrerseits jeweils in sogenannte atomare (weil nicht weiter in Boole'sche Ausdrück teilbare) Bedingungen „`a > 2`“ und „`b.length == 0`“ sowie „`c != 0`“ und „`d`“ zerlegt werden können. Zur Vereinfachung seien diese wie folgt abgekürzt:

- $\mathcal{A}$:= „`a > 2`“
- $\mathcal{B}$:= „`b.length == 0`“
- $\mathcal{C}$:= „`c != 0`“
- $\mathcal{D}$:= „`d`“
- $\mathcal{L}$:= „$\mathcal{A}$ | $\mathcal{B}$“ := „`a > 2 | b.length == 0`“
- $\mathcal{R}$:= „$\mathcal{C}$ | $\mathcal{D}$“ := „`c != 0 | d`“
- $\mathcal{G}$:= „$\mathcal{L}$ & $\mathcal{R}$“ := „`(a > 2 | b.length == 0) & (c != 0 | d)`“

Man beachte, dass die Operatoren | und & im Falle boole'scher Operanden ebenfalls ein boole'sches Ergebnis liefern, jedoch im Gegensatz zu || beziehungsweise && die Auswertung beider Operanden erzwingen - selbst wenn das Ergebnis des Gesamtausdrucks bereits durch die Auswertung eines Operanden sicher feststeht. Um den Fall partieller Auswertung ebenfalls behandeln zu können, sei die obige Übersicht um folgende Ausdrücke ergänzt:

- $\hat{\mathcal{L}}$:= „$\mathcal{A}$ || $\mathcal{B}$“ := „`a > 2 || b.length == 0`“
- $\hat{\mathcal{R}}$:= „$\mathcal{C}$ || $\mathcal{D}$“ := „`c != 0 || d`“
- $\hat{\mathcal{G}}$:= „$\mathcal{L}$ && $\mathcal{R}$“ := „`(a > 2 || b.length == 0) && (c != 0 || d)`“

Fall	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{L} = \mathcal{A} \mid \mathcal{B}$	$\mathcal{R} = \mathcal{C} \mid \mathcal{D}$	$\mathcal{G} = \mathcal{L}$ & $\mathcal{R}$
1	0	0	0	0	0	0	0
2	0	0	0	1	0	1	0
3	0	0	1	0	0	1	0
4	0	0	1	1	0	1	0
5	0	1	0	0	1	0	0
6	0	1	0	1	1	1	1
7	0	1	1	0	1	1	1
8	0	1	1	1	1	1	1
9	1	0	0	0	1	0	0
10	1	0	0	1	1	1	1
11	1	0	1	0	1	1	1
12	1	0	1	1	1	1	1
13	1	1	0	0	1	0	0
14	1	1	0	1	1	1	1
15	1	1	1	0	1	1	1
16	1	1	1	1	1	1	1

Tabelle 3.1: Wahrheitstabelle für den Ausdruck „$\mathcal{G} = (\mathcal{A} \mid \mathcal{B})\&(\mathcal{C} \mid \mathcal{D})$“

Tabelle 3.1 und Tabelle 3.2 zeigen alle möglichen Wahrheitswertkombinationen für die beiden betrachteten Ausdrücke $\mathcal{G}$ beziehungsweise $\hat{\mathcal{G}}$ die grundsätzlich auftreten können. Dabei bedeutet eine „0“, dass der entsprechende Ausdruck zu *falsch* und bei einer „1“ zu *wahr* ausgewertet wurde. Das Zeichen „–“ in Tabelle 3.2 deutet an, dass der entsprechende Teilausdruck gar nicht ausgewertet wird (*don't care*); wie zum Beispiel im Fall $\hat{1}$, bei dem das Ergebnis der Konjunktion bereits feststeht, nachdem der linke Operand ausgewertet wurde und die Auswertung des rechten Operanden damit überflüssig ist.

Zum Erreichen einer vollständigen Verzweigungsüberdeckung würde für eine Bedingung dieser Form die Ausführung dieser Verzweigungsanweisung mit zwei Wertekombinationen genügen, beispielweise aus den Fällen 3 und 11 aus Tabelle 3.1 beziehungsweise $\hat{1}$ und $\hat{7}$ aus

Fall	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	$\hat{\mathcal{L}} = \mathcal{A} \,\|\|\, \mathcal{B}$	$\hat{\mathcal{R}} = \mathcal{C} \,\|\|\, \mathcal{D}$	$\hat{\mathcal{G}} = \mathcal{L}$ && $\mathcal{R}$
$\hat{1}$	0	0	–	–	0	–	0
$\hat{2}$	0	1	0	0	1	0	0
$\hat{3}$	0	1	0	1	1	1	1
$\hat{4}$	0	1	1	–	1	1	1
$\hat{5}$	1	–	0	0	1	0	0
$\hat{6}$	1	–	0	1	1	1	1
$\hat{7}$	1	–	1	–	1	1	1

Tabelle 3.2: Wahrheitstabelle für den Ausdruck „$\hat{\mathcal{G}} = (\mathcal{A} \,||\, \mathcal{B})\&\&(\mathcal{C} \,||\, \mathcal{D})$“

Tabelle 3.2. Damit ist der Wert des Gesamtausdrucks jeweils einmal *falsch* und einmal *wahr*, dennoch würden für dieses Kriterium nur zwei von sieben sowie sogar nur zwei von 16 möglichen Kombinationen getestet. Noch gravierender ist, dass bei den Wertkombinationen aus den Fällen 3 und 11 lediglich der Wert des Teilausdrucks $\mathcal{A}$ variiert wurde, während die anderen Ergebnisse nicht geändert werden mussten – sie wurden im Falle von Ausdruck $\hat{\mathcal{G}}$ zum Teil gar nicht erst ausgewertet. Daher gibt es Reihe unterschiedlicher Testkriterien, welche sich der gezielten Überprüfung zusammengesetzter Bedingungen dieser Art widmet.

Das einfachste Verfahren aus dieser Klasse ist die sogenannte ***Einfache Bedingungsüberdeckung***, meist unter der englischen Bezeichnung *simple condition coverage* bekannt. Zur Erfüllung dieses Kriteriums müssen beim Test alle atomaren Bedingungen eines jeden Verzweigungsausdrucks im Programm mindestens einmal sowohl den Wahrheitswert *wahr* als auch den Wert *falsch* annehmen. Für das Beispiel $\mathcal{G}$ genügt laut Tabelle 3.1 die Ausführung des Programms mit denjenigen Testdaten, bei denen die (Teil)Bedingungen wie in den Fällen 6 und 11 ausgewertet werden: $\mathcal{A}$ zu *falsch*, $\mathcal{B}$ zu *wahr*, $\mathcal{C}$ zu *falsch* und $\mathcal{D}$ zu *wahr* sowie analog $\mathcal{A}$ = 1, $\mathcal{B}$ = 0, $\mathcal{C}$ = 1, $\mathcal{D}$ = 0.

Wie man an diesem Beispiel erkennen kann, beinhaltet dieses Kriterium im Falle vollständiger Auswertung aller Teilbedingungen nicht die Verzweigungsüberdeckung (siehe Kapitel 3.5, Abbildung 3.9). Diese Einschränkung gilt jedoch nicht für die unvollständige Auswertung. Anschaulich lässt sich das damit begründen, dass der Wahrheitswert einer zusammengesetzten Bedingung von mindestens einer atomaren Teilbedingung (o.B.d.A. sei $\mathcal{X}$ eine solche) abhängen muss, ansonsten läge ja keine Verzweigung vor. Da bei der einfachen Bedingungsüberdeckung jede atomare Teilbedingung sowohl einmal *wahr* als auch einmal *falsch* werden muss, gilt das insbesondere auch für $\mathcal{X}$ und damit für die gesamte Bedingung. Wählt man für das vorliegende Beispiel $\hat{\mathcal{G}}$ die zu den Fällen 6 und 11 für $\mathcal{G}$ analogen Fälle $\hat{3}$ und $\hat{7}$, dann müssen zusätzlich noch die atomaren Teilbedingungen $\mathcal{B}$ und $\mathcal{D}$ zu falsch ausgewertet werden, was durch Hinzunahme der Fälle $\hat{1}$ und wahlweise $\hat{2}$ oder $\hat{5}$ erreicht wird.

Um sicherzustellen, dass mittels Bedingungsüberdeckungstesten auch bei vollständiger Auswertung der Bedingungen die Verzweigungsüberdeckung subsumiert wird, kann die einfache Bedingungsüberdeckung um die explizite Forderung erweitert werden, dass nicht nur alle atomaren Teilbedingungen jeweils zu *wahr* und *falsch* ausgewertet werden, sondern auch die Gesamtbedin-

gung beide Werte anzunehmen hat. Das auf diese Weise geänderte Kriterium wird ***Bedingungs-/Entscheidungsüberdeckung*** oder *condition/decision coverage* genannt. Für den Ausdruck $\mathcal{G}$ erfüllen die „Testfälle“ 5 und 12 aus Tabelle 3.1 dieses Kriterium. Problematisch an diesem Kriterium ist, dass es möglicherweise auch erfüllt werden kann, ohne dass die nicht-atomaren Teilbedingungen jeweils für sich beide möglichen Wahrheitswerte annehmen. Dies kann man an den gewählten Testfällen für $\mathcal{G}$ erkennen, unter deren Ausführung die Teilbedingung $\mathcal{L}$ nie *falsch* wird.

Um den erwähnten Nachteil der Bedingungs-/Entscheidungsüberdeckung abzufangen, fordert das umfassendere Kriterium namens ***Minimale Mehrfach-Bedingungsüberdeckung*** beziehungsweise *minimal multiple condition coverage*, dass jede Teilbedingung im Test mindestens einmal sowohl den Wert *wahr* als auch den Wert *falsch* annehmen muss – dies gilt demnach sowohl für alle atomaren und alle nicht-atomaren Teilbedingungen als auch insbesondere für den gesamten Bedingungsausdruck. Für die betrachteten Ausdrücke $\mathcal{G}$ und $\hat{\mathcal{G}}$ kann dieses Kriterium zum Beispiel mit den Fällen 1 und 16 aus Tabelle 3.1 beziehungsweise $\hat{1}$, $\hat{2}$, $\hat{6}$ sowie $\hat{7}$ aus Tabelle 3.2 vollständig erfüllt werden.

Bedauerlicherweise stellt keines der bisher vorgestellten Kriterien sicher, dass sogenannte invariante Teilbedingungen eines zusammengesetzten Prädikats erkannt werden. Eine Teilbedingung heißt *invariant* wenn sie unter allen möglichen Ausführungen des Programms stets zum gleichen Wahrheitswert ausgewertet wird. Zum Beispiel ist der Teilausdruck „`a != 0`“ in der Bedingung „`a > 2 && b.length == 0 && a != 0`“ invariant, da er entweder bei Erfüllung der atomaren Teilbedingung „`a > 2`“ ebenfalls zu wahr oder wegen der partiellen Auswertung gar nicht ausgewertet wird. Weitaus schwieriger zu erkennen sind invariante Teilbedingungen wenn sie durch Schachtelung von Prädikaten entstehen, wobei die umschließende Bedingung bestimmte Wahrheitswertkombinationen des inneren Prädikats ausschließt. Das Vorhandensein solcher invarianten Bedingungen lässt im Allgemeinen entweder auf ungenutztes Optimierungspotential oder auf einen Programmfehler schließen.

Die Aufdeckung invarianter Prädikate in Bedingungsausdrücken unterstützt das Testkriterium ***Modifizierte Bedingungs-/Entscheidungsüberdeckung***, besser bekannt als *modified condition/decision coverage*. Dazu sind Testfälle so zu wählen, dass für jedes atomare Prädikat nachgewiesen wird, dass die Änderung des Wahrheitswertes dieses und nur dieses Prädikats auch eine Änderung des Wertes des gesamten Bedingungsausdrucks nach sich zieht. Für den Beispielausdruck $\mathcal{G}$ zeigen die Testfälle 2 und 10 dass der Wert von $\mathcal{A}$ für das Gesamtergebnis ausschlaggebend sein kann, denn obwohl in diesen beiden Testfällen nur $\mathcal{A}$ unterschiedlich bewertet wird, hat $\mathcal{G}$ insgesamt jeweils unterschiedliche Wahrheitswerte. Analog dazu muss dieser Nachweis noch für die verbleibenden atomaren Teilausdrücke erbracht werden. Beispielsweise für $\mathcal{B}$ mittels 2 und 6, für $\mathcal{C}$ durch 5 und 7 sowie für $\mathcal{D}$ mit 5 und 6. Demnach erfüllen die Testfälle 2, 5, 6, 7 und 10 aus Tabelle 3.1 dieses Kriterium für diesen Ausdruck vollständig.

Das umfassendste Kriterium zum Testen zusammengesetzter Bedingungen ist die sogenannte ***Mehrfach-Bedingungsüberdeckung*** beziehungsweise *multiple condition coverage*. Es fordert zu seiner vollständigen Erfüllung die Ausführung jeder Bedingung im Programm so, dass während des Tests alle möglichen Wahrheitswertkombinationen sämtlicher atomaren Teilbedingungen auftreten. Der Vorteil des vollständigen Tests erkauft man sich dabei jedoch mit einer in der Anzahl der atomaren Teilbedingungen exponentiell wachsenden Anzahl notwendiger Testdurch-

läufe. Demnach müssen die beiden Beispielausdrücke $\mathcal{G}$ und $\hat{\mathcal{G}}$ jeweils mit allen 16 respektive sieben Testfällen der Tabelle 3.1 beziehungsweise Tabelle 3.2 ausgeführt werden.

3.4 Datenflusskriterien

Betrachtet man die in Kapitel 3.2 und Kapitel 3.3 beschriebenen Testüberdeckungskriterien, so legen diese den Testschwerpunkt auf die Überdeckung bestimmter Anweisungen oder Teilpfade beziehungsweise auf die gründliche Untersuchung der Kontrollflusses, wie er mittels der Bedingungen in den Verzweigungsprädikaten gesteuert wird. Während diese Teststrategien für bestimmte Softwaresysteme ein akzeptables Kosten/Nutzen-Verhältnis im Bezug auf die Fehleraufdeckungsquote bieten, z.B. für eingebettete Systeme zur Gerätesteuerung, sind viele Programmpakete primär zur Verarbeitung von Daten ausgelegt. Um diesem Aspekt auch im Test die entsprechende Beachtung zukommen zu lassen, wurden diverse Überdeckungskriterien definiert, deren Ziel die Untersuchung des Datenflusses durch das Programm ist. Treffend motivieren die Erfinder dieser Strategien die Notwendigkeit des Datenflusstestens wie folgt: *„Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed“* [RW85].

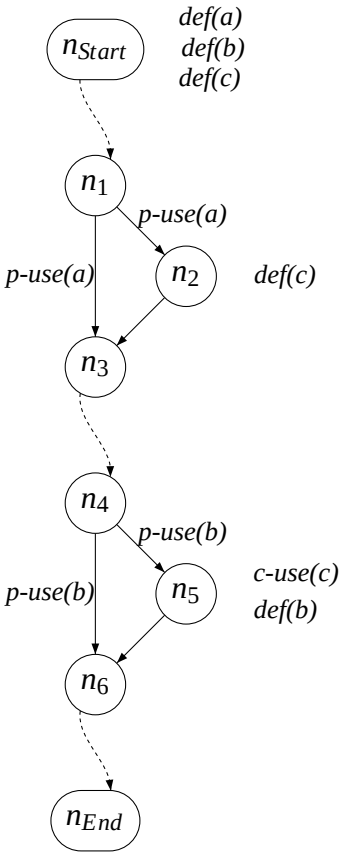
Die Bedeutung dieses Zitats sei exemplarisch an dem Beispiel aus Abbildung 3.5 demonstriert. Das durch den Kontrollflussgraphen dargestellte Programmfragment ist so aufgeführt, dass die Anweisungen jeweils auf Höhe der sie darstellenden Knoten stehen. Dabei können zwischen den Knoten n_3 und n_4 beliebige weitere Anweisungen vorkommen (dargestellt durch eine gestrichelte geschwungene Kante), jedoch sei angenommen, dass diese keine Zuweisungen an Variable `c` vornehmen.

Für dieses Programmbeispiel ist es möglich, eine vollständige Verzweigungsüberdeckung und damit auch Anweisungsüberdeckung zu erzielen: Zum Beispiel durch die Wahl zweier Eingabevektoren, mittels derer die beiden Pfade $p_1 = (n_{Start}, \ldots, n_1, n_2, n_3, \ldots, n_4, n_6, \ldots, n_{End})$ und $p_2 = (n_{Start}, \ldots, n_1, n_3, \ldots, n_4, n_5, n_6, \ldots, n_{End})$ ausgeführt werden. Dennoch wurde der (hier) offensichtliche Codefehler nicht gefunden, der aufgrund eines Methodenaufrufs an einer null-Referenz dann ein Versagen auslöst, wenn nach Ausführung von Knoten n_2 auch Knoten n_5 ausgeführt wird.

Wünschenswert wäre demnach ein Testkriterium, welches die Überdeckung eines Pfades der Form $p = (n_{Start}, \ldots, n_1, n_2, n_3, \ldots, n_4, n_5, n_6, \ldots, n_{End})$ fordert, also insbesondere den „Fluss der Information“ von Knoten n_2 zu Knoten n_5 testet. Dieser sogenannte Datenfluss entsteht hier durch Zuweisung eines Datums (in diesem Fall `null`) an Variable `c` in Knoten n_2 und darauf folgendes Auslesen dieses Datums im Knoten n_5, hier durch den Zugriff auf das Objekt, welches `c` referenziert, mittels des Methodenaufrufs `c.length()`.

3.4.1 Datenflussterminologie

Während die im Rahmen dieser Arbeit behandelten Kontrollflussteststrategien mit den im Kapitel 3.1 vorgestellten Mitteln formal definiert werden können, erfordert die Klasse der datenfluss-




```
public int f(int a, int b, String c) {
      ...
      if (a > 0) {
            c = null;
      }
      ...
      if (b < 0) {
            b = c.length();
      }
      ...
}
```

Abbildung 3.5: Beispiel zum Vergleich: *branch – all-uses*

orientierten Überdeckungskriterien weitergehende Konzepte, die im Folgenden in Anlehnung an [RW85] hier vorgestellt werden. Während die wichtigsten Kontrollflusstestmethoden auf die Ausführung bestimmter Knoten, Kanten und (Teil)Pfade des Kontrollflussgraphen $G = (N, E)$ eines Moduls abzielen, beschäftigen sich die Datenflusstesteststrategien mit der Überdeckung bestimmter Aspekte des Informationsflusses durch ein Programm während dessen Ausführung. Ein solcher Datenfluss entsteht während der Abarbeitung des Codes dadurch, dass Variablen zunächst Werte zugewiesen werden, die im anschließenden Ablauf zur Weiterverarbeitung wieder ausgelesen werden.

Im Folgenden bezeichne $\mathcal{P}$ ein Programm oder einen Programmausschnitt (zum Beispiel eine Funktion beziehungsweise Methode), $G = (N, E)$ den (datenflussannotierten) Kontrollflussgraphen und V die Menge aller in $\mathcal{P}$ verwendeten Variablen, genauer gesagt Speicherstellen[5]. Demnach kann der Zugriff auf Variablen zunächst grob in schreibende und lesende Zugriffe unterschieden und somit wie folgt klassifiziert werden.

Definition 3.12 (def) *Eine* Definition *(im Weiteren auch kurz als* def *bezeichnet) einer Variablen $v \in V$ im Knoten $n_d \in N$ liegt vor, wenn der Variablen v bei Ausführung des Knotens n_d ein Wert zugewiesen wird, der den bisherigen Wert von v überschreibt.*

Betrachtet man erneut den Kontrollflussgraphen in Abbildung 3.1, der den Programmcode aus Listing 3.1 repräsentiert, so ist die Zuweisung „`found = true`“ in Codezeile 8 beziehungsweise im entsprechenden Knoten n_5 ein typisches Beispiel einer solchen Definition, in diesem Falle der Variablen `found`. Diese Information trägt man üblicherweise als Annotation an den entsprechenden Stellen im Kontrollflussgraphen ein, wie in Abbildung 3.5 bereits angedeutet. Dabei besagt die Annotation $def(v_1, v_2, \ldots, v_n)$ an Knoten n_i, dass die Variablen $v_1, v_2, \ldots, v_n$, bei Ausführung der von n_i dargestellten Anweisungen, in genau dieser Reihenfolge definiert werden. Dadurch ergibt sich für das betrachtete Beispiel *BinarySearch* der sogenannte datenflussannotierte Kontrollflussgraph aus Abbildung 3.6. Stellt man jede Anweisung durch einen eigenen Knoten dar und werden Bedingungsausdrücke mittels Kurzschlussauswertung evaluiert, dann ergibt sich der im Anhang dargestellte Graph aus Abbildung A.2.

Definition 3.13 (use) *Eine* Verwendung *(kurz:* use*) einer Variablev $v \in V$ in Knoten $n_u \in N$ liegt vor, wenn der Wert der Variablen v bei Ausführung des Knotens n_u ausgelesen wird, ohne den bisherigen Wert von v zu verändern.*

Zuweilen wird in wissenschaftlicher Literatur [GG02, HFGO94, OW91, CPRZ89] lediglich zwischen *def* und *use* unterschieden, wobei man diese Ereignisse dem jeweiligen Knoten zuordnet, der die entsprechende Anweisung enthält. Da aber die Entscheidung über den weiteren Programmablauf an Stellen mit Verzweigungen aufgrund des aktuellen Zustands der Ausführung getroffen werden, damit also aufgrund der aktuellen Werte der Variablen, welche in der Bedingung der Verzweigungsanweisung ausgelesen werden, ist es sinnvoller, lesende Zugriffe dieser Art von rein datenverarbeitenden zu unterscheiden [RW85].

[5] Je nach Programmiersprache können evtl. mehrere Variablen mit verschiedenen Namen die gleiche Speicherstelle referenzieren (zum Beispiel Zeiger in C) oder mit dem gleichen Variablennamen unterschiedliche Speicherstellen gemeint sein (zum Beispiel Instanzvariablen in JAVA™), siehe *pointer-aliasing* in Kapitel 3.4.6

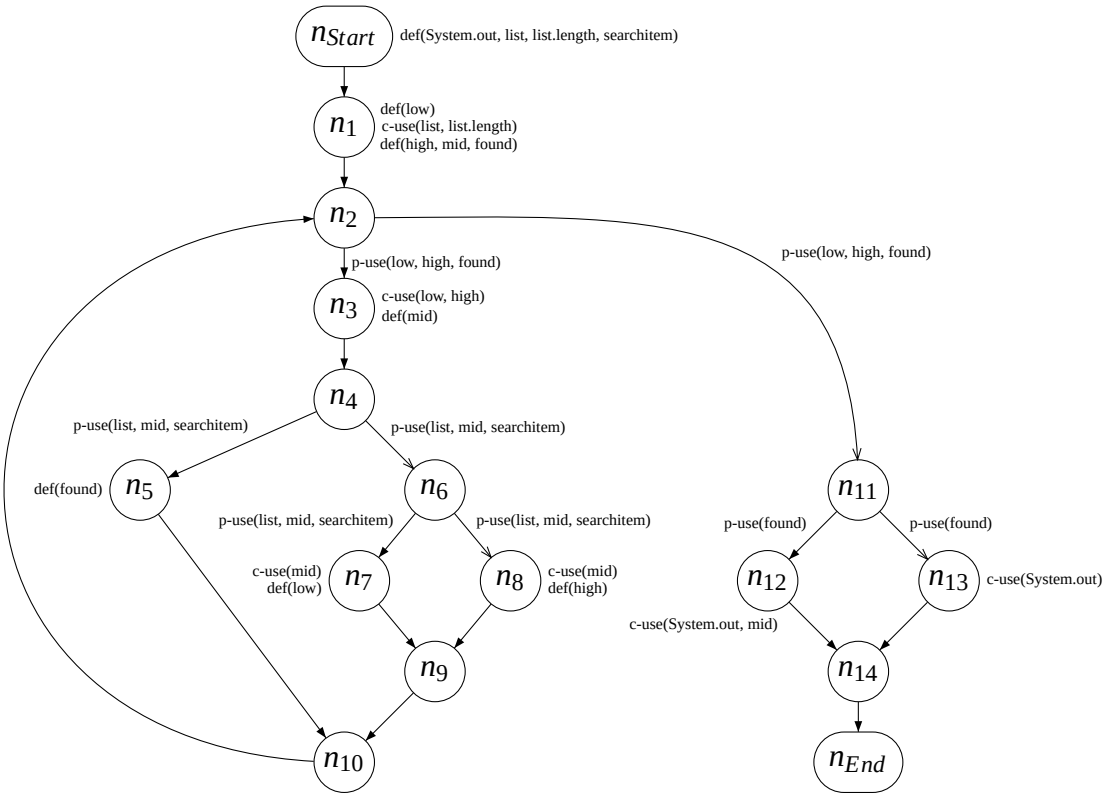


Abbildung 3.6: Datenflussannotierter Kontrollflussgraph des Programms *BinarySearch*

Definition 3.14 (c-use) *Eine* berechnende Verwendung *(computational use, kurz:* c-use*) einer Variablen $v \in V$ in Knoten $n_c \in N$ liegt vor, wenn bei Ausführung des Knotens n_c der Wert der Variablen v ausgelesen wird, um diesen in einer datenverarbeitenden Aktion, zum Beispiel einer arithmetischen Operation, zu verwenden.*

Im Quelltext aus Listing 3.1 findet man zum Beispiel berechnende Verwendungen der Variablen `low` und `high` (und eine anschließende Definition der Variablen `mid`) in Zeile 6. Diese c-uses (mit anschließendem def) werden gemeinsam und in dieser Reihenfolge dem Knoten n_3 des entsprechenden Kontrollflussgraphen wie in Abbildung 3.6 zugeordnet.

Definition 3.15 (p-use) *Eine* prädikative Verwendung *(predicative use, kurz:* p-use*) einer Variablen v entlang der Kante $(n_p, n_q) \in E$ liegt vor, wenn bei Ausführung des Verzweigungsknotens n_p der Wert der Variablen v ausgelesen wird, um den Wahrheitsgehalt der Bedingung zu bestimmen, welche darüber entscheidet, ob die Programmausführung nach dem Knoten n_p mit dem Nachfolger n_q oder einem anderen Nachfolger von n_p fortgesetzt wird.*

Definitionen und berechnende Verwendungen werden denjenigen Knoten des Kontrollflussgraphen $G = (N, E)$ eines Programms assoziiert, beziehungsweise annotiert, die die entsprechenden Anweisungen repräsentieren. Im Gegenzug ordnet man prädikative Verwendungen einer Variablen v *allen* Kanten $(n_p, n_q^i) \in E$ zu, deren Ausgangsknoten n_p die Auswertung der Bedingungen darstellen, deren Ergebnis vom Wert der verwendeten Variablen v abhängt. Ziel dabei ist es, die Datenflusstestkriterien geeignet in die Subsumptionshierarchie der kontrollflussbasierten Überdeckungskriterien einzubinden, wie in Kapitel 3.2 und Kapitel 3.4.2 dargestellt. Für die Zeile 15 des Moduls *BinarySearch* aus Listing 3.1 bedeutet dies, dass die prädikative Verwendung der Variablen `found` nicht dem Knoten n_{11} zugeordnet wird, wie dies aus Sicht des Kontrollflusses mit der eigentlichen Bedingung geschieht. Stattdessen assoziiert man zu jeder von Knoten n_{11} ausgehenden Kante (n_{11}, n_{12}) und (n_{11}, n_{13}) ein p-use der Variablen `found`.

Für die datenflussorientierten Überdeckungskriterien sind Paare von Definitionen und Verwendungen der gleichen Variablen von Bedeutung, die entlang eines betrachteten Pfades nicht von einer erneuten Definition der selben Variablen getrennt sein dürfen.

Definition 3.16 (definitionsfrei) *Ein Teilpfad $(n_d, n_1, n_2, \ldots, n_m, n_c)$ eines Kontrollflussgraphen $G = (N, E)$ heißt* definitionsfrei *(in der englischsprachigen Literatur:* def-clear*)* bezüglich Variable v von Knoten n_d zu Knoten n_c*, falls die Knoten n_1 bis n_m keine Definition der Variablen $v \in V$ enthalten, wobei $n_d, n_c, n_i \in N$ mit $i = 1 \ldots m$, $m \geq 0$.*

Analog heißt ein Teilpfad $(n_d, n_1, n_2, \ldots, n_m, n_p, n_q)$ definitionsfrei bezüglich Variable v von Knoten n_d zur Kante $(n_p, n_q) \in E$*, falls keine Definition der Variablen $v \in V$ in den Knoten n_1 bis n_p vorkommt, wobei $n_d, n_p, n_q, n_i \in N$ mit $i = 1 \ldots m$, $m \geq 0$.*

Da ein Knoten, laut ursprünglicher Definition (siehe Seite 33) eine maximale Sequenz von Anweisungen repräsentiert, können einem einzigen Knoten durchaus mehrere Zugriffe auf die gleiche Variable zugeordnet werden. Deshalb ist die genaue Abfolge der Variablenzugriffe innerhalb eines Knotens wesentlich, weshalb zur Verdeutlichung des Datenflusses bei seiner Annotation im Kontrollflussgraphen die einzelnen Anweisungen des Quelltextes besser von getrennten

Knoten dargestellt werden sollten. Um den allgemeinen Fall vollständiger Anweisungssequenzen in Knoten genauer behandeln zu können, unterscheidet man globale und lokale Variablenzugriffe wie folgt:

Definition 3.17 (globale/lokale Verwendung) *Eine berechnende Verwendung einer Variablen $v \in V$ heißt* global*, falls innerhalb der gleichen Anweisungssequenz (evtl. dargestellt durch den gleichen Knoten) keine Definition der Variablen v vor der betrachteten Verwendung vorkommt. Ansonsten heißt die Verwendung* lokal*.*

Bei p-uses macht diese Unterscheidung keinen Sinn, da diese den Kanten im Kontrollflussgraphen zugeordnet werden. Analog kann man jedoch unterschiedliche Arten von Definitionen identifizieren:

Definition 3.18 (globale/lokale Definition) *Eine Definition einer Variablen $v \in V$ im Knoten $n_d \in N$ heißt* global*, falls sie die letzte Definition von v in der von Knoten n_d repräsentierten Anweisungssequenz ist und es mindestens einen definitionsfreien Teilpfad bezüglich v vom Knoten n_d zu einem Knoten mit einer globalen berechnenden Verwendung oder einer Kante mit einer prädikativen Verwendung gibt.*

Eine Definition einer Variablen v im Knoten n_d heißt lokal*, falls sie nicht global ist, jedoch eine lokale Verwendung im gleichen Anweisungsblock auf die Definition folgt und keine weitere Definition von v zwischen der betrachteten Definition und der lokalen Verwendung vorkommt.*

Eine Definition, die weder global noch lokal ist, weist auf eine Programmanomalie hin, da der Wert, welcher der Variablen an der Stelle der Definition zugewiesen wurde, während der weiteren Ausführung des Codes nicht mehr verwendet wird – wodurch die Definition sinnlos ist.

Definition 3.19 (Definition *erreicht* Verwendung) *Falls es für eine Variable v in einem Kontrollflussgraphen G einen definitionsfreien Pfad von einem Knoten $n_d \in N$ mit einer Definition von v zu einem Knoten $n_c \in N$ mit einer (globalen) berechnenden Verwendung oder zu einer Kante $e_p \in E$ mit einer prädikativen Verwendung von v gibt, dann heißt es, die Definition von v in n_d* erreicht (reaches) *die Verwendung von v in n_d bzw. entlang e_p.*

Um die Beschreibung der datenflussorientierten Überdeckungskriterien zu vereinfachen, seien noch folgende Mengen definiert, welche sich jeweils auf einen Kontrollflussgraphen $G = (N, E)$ eines Moduls beziehen:

Definition 3.20 (def(n_d)) *Die Menge* def(n_d) $\subseteq V$ *enthält genau diejenigen Variablen des von G dargestellten Codeausschnitts, die in Knoten $n_d \in N$ global definiert werden.*

Definition 3.21 (c-use(n_c)) *Analog stellt* c-use(n_c) *die Menge derjenigen Variablen des von G dargestellten Codeausschnitts dar, die in Knoten $n_c \in N$ global berechnend verwendet werden.*

Definition 3.22 (p-use(n_p, n_q)) *Ebenso enthält die Menge* p-use(n_p, n_q) *genau diejenigen Variablen des von G dargestellten Codeausschnitts, die entlang der Kante $(n_p, n_q) \in E$ prädikativ verwendet werden.*

Bezogen auf das Beispiel aus Listing 3.1 beziehungsweise Abbildung 3.6 ergeben sich demnach exemplarisch folgende Mengen:

- def(n_3) = {`mid`}
- c-use(n_3) = {`low`, `high`}
- p-use(n_2, n_3) = {`low`, `high`, `found`}

Darüber hinaus sind zur Beschreibung der Datenflusskriterien noch die Mengen der von einer Definition erreichbaren Knoten beziehungsweise Kanten relevant. Eine Definition einer Variablen in Knoten n_d *erreicht* eine Verwendung dieser Variablen in Knoten n_c oder entlang Kante (n_p, n_q), wenn es einen definitionsfreien Teilpfad gibt, der n_d mit n_c beziehungsweise n_p verbindet.

Definition 3.23 (dcu) *Die Menge* dcu(v, n_d) *enthält genau diejenigen Knoten $n_c \in N$, für die die Variable $v \in V$ im Knoten n_d global definiert wird, also $v \in$ def(n_d), im Knoten n_c global berechnend verwendet wird, also $v \in$ c-use(n_c), und es mindestens einen definitionsfreien Pfad zwischen Knoten n_d und n_c im Kontrollflussgraphen G gibt.*

Definition 3.24 (dpu) *Analog stellt* dpu(v, n_d) *die Menge aller Kanten $(n_p, n_q) \in E$ dar, so dass die Variable $v \in V$ im Knoten n_d global definiert wird, also $v \in$ def(n_d), entlang der Kante (n_p, n_q) prädikativ verwendet wird, also $v \in$ p-use(n_p, n_q), und es mindestens einen definitionsfreien Pfad zwischen Knoten n_d und n_p im Kontrollflussgraphen G gibt.*

Betrachtet man erneut den datenflussannotierten Kontrollflussgraphen aus Abbildung 3.6 basierend auf dem Code-Beispiel aus Listing 3.1, kann man exemplarisch folgende Mengen identifizieren:

- dcu(`mid`, n_3) = $\{n_7, n_8, n_{12}\}$
- dpu(`mid`, n_3) = $\{(n_4, n_5), (n_4, n_6), (n_6, n_7), (n_6, n_8)\}$

Etwas überraschen mag vielleicht, dass $n_{12} \in$ dcu(`mid`, n_1), also die Definition der Variablen `mid` in Zeile 3 eine Verwendung dieser Variablen in Zeile 16 „erreicht“, obwohl dieser Fall aufgrund der Initialisierung der Variablen `found` nie auftreten kann. Dass jedoch kein Testfall existiert, der einen Pfad von Knoten n_1 zu Knoten n_{12} unter Umgehung des Knotens n_3 ($\hat{=}$ Zeile 6) zur Ausführung bringt, kann nur semantisch ermittelt werden – syntaktisch und damit graphentheoretisch ist ein solcher Pfad dennoch feststellbar. Dies ist ein weiteres Beispiel für das Problem der statischen Analyse, zwischen ausführbaren und nicht-ausführbaren Pfaden (siehe Definition 3.11) zu unterscheiden.

Aufgrund bisher eingeführter Konzepte seien spezielle Teilpfade zwischen der Definition einer Variablen und einer davon erreichbaren Verwendung, sogenannte DU-Teilpfade, in Anlehnung an [RW85] wie folgt festgelegt:

Definition 3.25 (DU-Teilpfad) *Ein Teilpfad $p = (n_d, n_1, \ldots, n_m, n_p, n_q)$ eines Kontrollflussgraphen $G = (N, E)$ mit einer globalen Definition einer Variablen $v \in def(n_d)$ im Knoten n_d heißt* DU-Teilpfad bezüglich *v falls eine der beiden folgenden Bedingungen zutrifft:*

1. *Knoten $n_q \in dcu(v, n_d)$ enthält eine berechnende Verwendung von v und (n_d, n_1, …, n_m, n_p, n_q) ist ein bezüglich v definitionsfreier, einfacher Teilpfad*[6].

2. *Kante $(n_p, n_q) \in dpu(v, n_d)$ enthält eine prädikative Verwendung von v und (n_d, n_1, …, n_m, n_p) ist ein bezüglich v definitionsfreier, schleifenfreier Teilpfad*[7].

3.4.2 Datenflusskriterien nach Rapps/Weyuker

Basierend auf den damals bereits im Rahmen des Compilerbaus eingesetzten Datenflussanalysen entwickelten Sandra Rapps und Elaine J. Weyuker Ende der 70er Jahre eine Hierarchie unterschiedlicher Datenflusskriterien zur Bestimmung entsprechender Testdaten [RW82, RW85]. Wie sie befanden, haben die sinnvoll einsetzbaren, rein kontrollflussorientierten Kriterien diverse Schwachstellen, weshalb die entsprechend erstellten Testfälle bei weitem nicht ausreichend sind. Natürlich schafft ein vollständiger Test aller Eingaben oder aller Pfade eine ausreichende Vertrauensbasis in die Zuverlässigkeit der Systeme, doch sind diese Teststrategien aufgrund ihres Umfangs nur in Ausnahmefällen überhaupt durchführbar. Also müssen Strukturtestkriterien herangezogen werden, für welche die notwendigen Testfallmengen so klein wie möglich sein sollen, um sie vollständig ausführen zu können, jedoch so umfangreich wie möglich, um viele Fehler aufdecken zu können [RW82]. Dementsprechend sind die von Rapps und Weyuker definierten Kriterien im Wesentlichen im Mittelfeld zwischen Verzweigungsüberdeckung und Pfadüberdeckung angesiedelt und legen zugleich den Schwerpunkt auf den Informationsfluss, welcher von den dazu orthogonalen, klassischen Kontrollflusskriterien nicht betrachtet wird (siehe Abbildung 3.9).

Die im Folgenden zusammengefassten Beschreibungen der klassischen Datenflusskriterien basieren auf den Definitionen aus Kapitel 3.4.1. Für weitergehende Betrachtungen sei hier auf die entsprechende Literatur verwiesen, z.B. [RW85, Bal97, CPRZ89, FB01, HL91]. Zur Verdeutlichung der Kriterien wird der in Abbildung 3.1 dargestellte Programmausschnitt *BinarySearch* herangezogen, welcher durch den datenflussannotierten Kontrollflussgraphen aus Abbildung 3.6 repräsentiert wird. Die annotierten Datenflussinformationen sind zur besseren Übersicht in Tabelle 3.3 zusammengefasst. Ein Testfall t für diesen Suchalgorithmus wird als $t :=$ (`searchitem`, $\langle le_1, \ldots, le_n \rangle$) angegeben, wobei $le_1, \ldots, le_n$ die Elemente der nach `searchitem` zu durchsuchenden Liste `list` := $\langle le_1, \ldots, le_n \rangle$ sind.

Das sogenannte ***all-defs***-Kriterium ist eines der schwächsten aus der Familie der Datenflussüberdeckungskriterien. Ziel dieser Strategie ist es zu zeigen, dass jeder einer Variablen zugewiesene Wert, auch mindestens einmal verwendet wird – also dass die entsprechende Zuweisung im Programm nicht vollkommen sinnlos ist. Sei also V die Menge der in einem Programm $\mathcal{P}$ verwendeten Variablen, und $G := (N, E)$ der datenflussannotierte Kontrollflussgraph dieses

[6] Siehe Definition 3.8
[7] Siehe Definition 3.9

(a) def, c-use

n_i	$def(n_i)$	$c-use(n_i)$
n_{Start}	{list, list.length, searchitem, System.out}	$\emptyset$
n_1	{low, high, mid, found}	{list, list.length}
n_3	{mid}	{low, high}
n_5	{found}	$\emptyset$
n_7	{low}	{mid}
n_8	{high}	{mid}
n_{12}	$\emptyset$	{System.out, mid}
n_{13}	$\emptyset$	{System.out}
$\forall n_i \in \{n_2,n_4,n_6,n_9,n_{10},n_{11},n_{14},n_{End}\} : def(n_i) = \emptyset$ $\forall n_i \in \{n_2,n_4,n_6,n_9,n_{10},n_{11},n_{14},n_{End}\} : c-use(n_i) = \emptyset$		

(b) p-use

e_j	$p-use(e_j)$
(n_2,n_3)	{low, high, found}
(n_2,n_{11})	{low, high, found}
(n_4,n_5)	{list, mid, searchitem}
(n_4,n_6)	{list, mid, searchitem}
(n_6,n_7)	{list, mid, searchitem}
(n_6,n_8)	{list, mid, searchitem}
(n_{11},n_{12})	{found}
(n_{11},n_{13})	{found}
für alle anderen $e_j \in E : p-use(e_j) = \emptyset$	

(c) dcu, dpu

v	n_i	$dcu(v,n_i)$	$dpu(v,n_i)$
list	n_{Start}	$\{n_1\}$	$\{(n_4,n_5),(n_4,n_6),(n_6,n_7),(n_6,n_8)\}$
list.length	n_{Start}	$\{n_1\}$	$\emptyset$
searchitem	n_{Start}	$\emptyset$	$\{(n_4,n_5),(n_4,n_6),(n_6,n_7),(n_6,n_8)\}$
System.out	n_{Start}	$\{n_{12},n_{13}\}$	$\emptyset$
low	n_1	$\{n_3\}$	$\{(n_2,n_3),(n_2,n_{11})\}$
high	n_1	$\{n_3\}$	$\{(n_2,n_3),(n_2,n_{11})\}$
mid	n_1	$\{n_{12}\}$	$\emptyset$
found	n_1	$\emptyset$	$\{(n_2,n_3),(n_2,n_{11}),(n_{11},n_{12}),(n_{11},n_{13})\}$
mid	n_3	$\{n_7,n_8,n_{12}\}$	$\{(n_4,n_5),(n_4,n_6),(n_6,n_7),(n_6,n_8)\}$
found	n_5	$\emptyset$	$\{(n_2,n_3),(n_2,n_{11}),(n_{11},n_{12}),(n_{11},n_{13})\}$
low	n_7	$\{n_3\}$	$\{(n_2,n_3),(n_2,n_{11})\}$
high	n_8	$\{n_3\}$	$\{(n_2,n_3),(n_2,n_{11})\}$
für alle anderen $v \in V, n_i \in N : dcu(v,n_i) = \emptyset = dpu(v,n_i)$			

Tabelle 3.3: Datenflussannotation zum Programm *BinarySearch*

Programms, dann fordert das Kriterium formal: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu mindestens einem Element aus $dcu(v, n_d)$ oder $dpu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten aus $dcu(v, n_d)$ beziehungsweise zur entsprechenden Kante aus $dpu(v, n_d)$ ist.

Um das Kriterium bezüglich der Definition von `mid` im Knoten n_3 zu erfüllen, genügt es zum Beispiel in einer Liste der Form `list` = $\langle 1,2,3 \rangle$ nach dem Element `searchitem` = 2 zu suchen. Von diesem Testfall t_d := (2, $\langle$ 1, 2, 3 $\rangle$) wird der Pfad p_{t_d} = (n_{Start}, n_1, n_2, $\underline{n_3}$, $\underline{n_4, n_5}$, n_{10}, n_2, n_{11}, $\underline{n_{12}}$, n_{14}, n_{End}) überdeckt, wobei bezüglich `mid` ein definitionsfreier Teilpfad von n_3 zur Kante $(n_4, n_5) \in dpu(\texttt{mid}, n_3)$ sowie zusätzlich zum Knoten $n_{12} \in dcu(\texttt{mid}, n_3)$ ausgeführt wird. Natürlich muss darüber hinaus eine entsprechende Betrachtung für jeden einzelnen Knoten und jeder darin definierten Variable erfolgen.

Der Nachteil dieses Kriteriums ist, dass viele Datenflusspaare nicht zwingend überdeckt werden müssen. Im obigen Beispiel ist es bezüglich $\texttt{mid} \in def(n_3)$ während des Testens nicht notwendig, weitere definitionsfreie Teilpfade von n_3 zu den restlichen Knoten $n7, n8 \in dcu(\texttt{mid}, n_3)$ sowie Kanten $(n_4, n_6), (n_6, n_7), (n_6, n_8) \in dpu(\texttt{mid}, n_3)$ auszuführen. Dadurch entgeht einem Tester unter Umständen, dass die Zuweisung „`low = mid+1;`“ möglicherweise fehlerhaft ist, zum Beispiel wenn sie tatsächlich „`low = low+1;`“ lauten müsste, da der Fluss der Daten mittels der Variablen `mid` von Knoten n_3 zu Knoten n_7 nie untersucht wurde.

Um diesen Nachteil speziell für berechnende Verwendungen auszugleichen, fordert das Kriterium ***all-c-uses*** die Überdeckung aller von einer Definition einer Variablen erreichbaren c-uses. Formal bedeutet dies: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu jedem Element aus $dcu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten aus $dcu(v, n_d)$ ist.

Im obigen Beispiel verfehlte t_d bezüglich `mid` die beiden von n_3 aus erreichbaren Verwendungen in den Knoten $n7$ und n_8. Erweitert man die Testfallmenge um t_c := (4, $\langle$ 1, 3, 5 $\rangle$), so wird davon der Pfad p_{t_c} = (n_{Start}, n_1, n_2, $\underline{n_3}$, n_4, n_6, $\underline{n_7}$, n_9, n_{10}, n_2, $\overline{n_3}$, n_4, n_6, $\overline{n_8}$, n_9, n_{10}, n_2, n_{11}, n_{13}, n_{14}, n_{End}) überdeckt, wodurch die mit t_d nicht erreichten Elemente aus $dcu(\texttt{mid}, n_3)$ nun ebenfalls getestet wurden.

Als Ausblick auf die Minimierung der Testdatenmenge sei hier noch der Testfall $\hat{t_c}$:= (5, $\langle$ 1, 2, 3, 4, 5, 6, 7 $\rangle$) genannt, welcher durch Ausführung des Pfades $p_{\hat{t_c}}$ = (n_{Start}, n_1, n_2, $\underline{n_3}$, n_4, n_6, $\underline{n_7}$, n_9, n_{10}, n_2, $\overline{n_3}$, n_4, n_6, $\overline{n_8}$, n_9, n_{10}, n_2, $\underline{n_3}$, n_4, n_5, n_{10}, n_2, n_{11}, $\underline{n_{12}}$, n_{14}, n_{End}) alle von der Definition der Variablen `mid` im Knoten n_3 erreichbaren Verwendungen aus $dcu(\texttt{mid}, n_3)$ gleichzeitig überdeckt.

Analog zu *all-c-uses* fordert das Kriterium ***all-p-uses*** die Überdeckung aller von einer Definition einer Variablen erreichbaren prädikativen Verwendungen. Formal bedeutet dies: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu jedem Element aus $dpu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zur entsprechenden Kante aus $dpu(v, n_d)$ ist.

Allein bezüglich der Definition der Variablen `mid` im Knoten n_3 ist das *all-p-uses*-Kriterium mit den beiden obigen Testfällen t_d und t_c oder alternativ mit dem einzelnen Testfall $\hat{t_c}$ zur *all-c-uses*-Überdeckung ebenfalls erfüllbar.

Betrachtet man die Tabelle 3.3 genauer, so fällt auf, dass von manchen Definitionen gar keine berechnende oder prädikative Verwendung erreicht wird. Zum Beispiel ist $dcu(\texttt{found}, n_5)$

ebenso leer wie $dpu(\texttt{mid}, n_1)$. In solchen Fällen müsste bei Verwendung des *all-c-uses*- beziehungsweise *all-p-uses*-Kriteriums der von solchen Definitionen ausgehende Datenfluss gar nicht getestet werden. Diese Schwäche der beiden Kriterien stellt auch den Grund dar, warum sie das *all-defs*-Kriterium nicht enthalten, wie in Abbildung 3.9 zu sehen ist.

Um selbst im Falle einer leeren $dpu(v, n_i)$-Menge trotzdem zumindest einen Datenfluss von der entsprechenden Definition der Variablen v im Knoten n_i zu irgendeiner Verwendung wie bei der *all-defs*-Strategie zu testen, fordert das sogenannte ***all-c-uses/some-p-uses***-Kriterium formal: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu jedem Element aus $dcu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten aus $dcu(v, n_d)$ ist; falls jedoch $dcu(v, n_d) = \emptyset$, dann ist mindestens ein Teilpfad von n_d zu mindestens einem Element aus $dpu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zur entsprechenden Kante aus $dpu(v, n_d)$ ist.

Analog dazu ist das sogenannte ***all-p-uses/some-c-uses***-Kriterium erfüllt, wenn für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ mindestens ein Teilpfad von n_d zu jedem Element aus $dpu(v, n_d)$ ausgeführt wurde, welcher definitionsfrei bezüglich v von n_d zur entsprechenden Kante aus $dpu(v, n_d)$ ist; falls jedoch $dpu(v, n_d) = \emptyset$, dann ist mindestens ein Teilpfad von n_d zu mindestens einem Element aus $dcu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten aus $dcu(v, n_d)$ ist.

Das ***all-uses***-Kriterium führt schließlich alle bisher vorgestellten Datenflusskriterien zusammen und fordert formal: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu jedem Element aus $dcu(v, n_d)$ und zu jedem Element aus $dpu(v, n_d)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten aus $dcu(v, n_d)$ beziehungsweise der entsprechenden Kante aus $dpu(v, n_d)$ ist. Demnach ist dieses Kriterium erfüllt, wenn für jede Definition jeder Variablen in einem Programm der Datenfluss zu allen davon definitionsfrei erreichbaren Verwendungen dieser Variablen getestet wird.

Kriterium	*defs*	*c-uses*	*p-uses*	*Teilpfade*
all-defs	alle	mind. eines		mind. einer
all-c-uses	alle	alle	keines	mind. einer
all-p-uses	alle	keines	alle	mind. einer
all-c-uses/ some-p-uses	alle	alle	mind. eines, falls kein c-use erreichbar	mind. einer
all-p-uses/ some-c-uses	alle	mind. eines, falls kein p-use erreichbar	alle	mind. einer
all-uses	alle	alle	alle	mind. einer
all-DU-paths	alle	alle	alle	alle schleifenfreien

Tabelle 3.4: Übersicht aller klassischen Datenflusskriterien nach Rapps/Weyuker

Die in Tabelle 3.4 dargestellte Übersicht der klassischen Datenflusskriterien nach Rapps und Weyuker [RW85] offenbart, dass alle bisher vorgestellten Kriterien lediglich die Überdeckung mindestens eines Teilpfades zwischen jeder Definition und bestimmten davon erreichbaren Verwendungen einer Variablen fordern. So gibt es zum Beispiel in Abbildung 3.6 zwischen der

Definition von `found` im Knoten n_1 und der davon erreichbaren prädikativen Verwendung entlang der Kante (n_{11}, n_{12}) prinzipiell unendlich viele Teilpfade der Form $p = (n_1, n_2, [q_1 \mid q_2]^*, n_{11}, n_{12})$, wobei die Teilpfade $q_1 := (n_3, n_4, n_6, n_7, n_9, n_{10}, n_2)$ und $q_2 := (n_3, n_4, n_6, n_8, n_9, n_{10}, n_2)$ beliebig oft und in beliebiger Reihenfolge wiederholt werden können.

Würde man nun ein Kriterium definieren, welches *alle* Teilpfade zwischen jeder Definition und davon erreichbaren Verwendungen fordert, so wäre dieses Kriterium im Allgemeinen ebenso unerfüllbar beziehungsweise nur mit unverhältnismäßig hohem Aufwand erreichbar wie die im Kapitel 3.2.3 vorgestellte Pfadüberdeckung. Daher wird in [RW85] als Kompromiss das ***all-DU-paths***-Kriterium eingeführt. Dieses ist erfüllt, wenn für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ alle DU-Teilpfade bezüglich v (siehe Definition 3.25) von n_d zu jedem Element aus $dcu(v, n_d)$ und zu jedem Element aus $dpu(v, n_d)$ getestet wurde.

Man beachte, dass dieses Kriterium laut ursprünglicher Definition die Überdeckung aller anhand eines datenflussannotierten Kontrollflussgraphen statisch identifizierbaren DU-Teilpfade fordert. Dies kann in besonderen Fällen dazu führen, dass das Kriterium gar nicht erfüllt werden kann, zum Beispiel wenn zur Ausführungszeit eine Schleife zwischen einer Definition und einer erreichbaren Verwendung mindestens einmal durchlaufen werden muss. Da die zuvor eingeführten Kriterien beliebige Teilpfade fordern, also insbesondere auch solche, welche Schleifen enthalten können, subsumiert das *all-DU-paths* Kriterium nicht zwingend das *all-uses*-Kriterium. In Abbildung 3.9 ist daher die leicht modifizierte Variante „*all-DU-paths+*“ dargestellt, welche im Kapitel 3.5 beschrieben wird.

Für die Definition der Variablen `mid` im Knoten n_3 und der berechnenden Verwendung in Knoten n_{12} des Kontrollflussgraphen aus Abbildung 3.6 genügen nun nicht mehr die für *all-c-uses* gewählten Testfälle t_d und t_c. Zwar überdeckt t_c den DU-Teilpfad $p_1^{DU} := (\underline{n_3}, n_4, n_5, n_{10}, n_2, n_{11}, \underline{n_{12}})$, jedoch müssten noch zusätzlich die Teilpfade $p_2^{DU} := (\underline{n_3}, n_4, n_6, n_7, n_9, n_{10}, n_2, n_{11}, \underline{n_{12}})$ und $p_3^{DU} := (\underline{n_3}, n_4, n_6, n_8, n_9, n_{10}, n_2, n_{11}, \underline{n_{12}})$ getestet werden. Allerdings gibt es weder einen Testfall der p_2^{DU} zur Ausführung bringt, noch einen zur Überdeckung von p_3^{DU}. Aufgrund der Belegungen der Variablen `found` muss jeder Pfad, welcher Knoten n_{12} enthält, vorher zwingend auch Knoten n_5 enthalten. Dies wiederum führt dazu, dass die Schleife nach Ausführung von Knoten n_5 verlassen wird, ohne mit einer gültigen Definition aus Knoten n_3 jemals entlang p_2^{DU} beziehungsweise p_3^{DU} zur Verwendung in Knoten n_{12} zu gelangen.

3.4.3 Datenflusskriterien nach Ntafos

Während die Anweisungs- und Verzweigungsüberdeckungskriterien sowie die einfacheren klassischen Datenflusskriterien nach Rapps und Weyuker nur das Testen einzelner Entitäten eines Kontrollflussgraphen fordern, wie zum Beispiel die Ausführung jeder Kante für sich bei Anwendung der Verzweigungsüberdeckung, gibt es eine Reihe diverser Überdeckungskriterien, welche das Ausführen ganzer Ketten solcher Entitäten fordern. Zu dieser Kategorie gehört die Familie der ***LCSAJ***-Tests (*Linear Code Sequence And Jump*) aus der Klasse der kontrollflussorientierten Kriterien. Diese Teststrategie wurde ursprünglich für Programme mit vielen Sprüngen entwickelt [Bal97] und spielt daher in der heutigen Zeit kaum noch eine Rolle, zumal moderne Programmiersprachen keine Sprunganweisungen mehr enthalten. Betrachtet man eine Verzweigung als

„Sprung“, kann man das Kriterium auch auf aktuelle Softwareprodukte anwenden. Ziel des Kriteriums ist es, Ketten bestimmter Länge zu überdecken, deren Glieder lineare Codesequenzen sind, welche mit einer Sprunganweisung enden. Dabei stellen Ketten der Länge $n = 1$ definitionsgemäß einzelne Anweisungen dar, Ketten der Länge $n = 2$ entsprechend Verzweigungen. Die zur Messung der erzielten Überdeckung verwendeten Metriken werden mit TER_n (für *Testing Effectiveness Ratio*) abgekürzt. In der Subsumptionshierarchie ordnen sie sich wie in Abbildung 3.9 dargestellt, ein.

Eine ähnliche Kriterienfamilie, jedoch mit Bezug zu Datenflussentitäten wurde von Simeon C. Ntafos [Nta81, Nta84, CPRZ89] definiert. Der Grundgedanke der sogenannten ***required k-tuples*** ist es, Ketten alternierender Definitionen und Verwendungen von Variablen zu überdecken, wobei diese Ketten als *k-dr-Interaktionen* bezeichnet werden. Jede Definition d_i einer *k-dr*-Interaktion erreicht die nächste Verwendung in der Kette, welche im gleichen Knoten stattfindet wie die nächste Definition d_{i+1} in dieser Kette, so dass der Wert der Variablen aus d_i in den Wert einfließt, welcher in d_{i+1} zugewiesen wird. Eine *k-dr*-Interaktion propagiert demnach Informationen entlang eines Teilpfades, dem sogenannten *Interaktionsteilpfad*. Somit besteht das Ziel dieser Strategie darin, den Informationsfluss entlang solcher Interaktionspfade zu testen.

Um die formale Definition des Kriteriums leichter nachzuvollziehen, sei angenommen, dass jeder Knoten eines Kontrollflussgraphen $G = (N, E)$ nur noch eine einzelne Anweisung repräsentiert, also zum Beispiel eine Zuweisung oder die Auswertung eines Prädikates. Außerdem seien prädikative Verwendungen dem entsprechenden Knoten zugeordnet, welcher die Bedingungsanweisung mit dem lesenden Zugriff darstellt, wobei Knoten mit einer Prädikatsauswertung mindestens zwei Nachfolgeknoten haben (siehe Definition 3.15). Die im Folgenden vorgestellten Kriterien lassen sich jedoch leicht auf den üblichen Fall übertragen, bei dem die prädikativen Verwendungen den jeweiligen Kanten zugeordnet werden.

Für $k \geq 2$ ist eine *k-dr*-Interaktion eine Abfolge $\kappa = [d_1(x_1), u_2(x_1), d_2(x_2), u_3(x_2), \ldots, d_{k-1}(x_{k-1}), u_k(x_{k-1})]$ von $k-1$ Definitionen und $k-1$ Verwendungen, welche jeweils k unterschiedlichen Knoten $n_1, n_2, \ldots, n_k$ zugeordnet sind, wobei für alle i mit $1 \leq i < k$ gilt: Die i-te Definition $d_i(x)$ in Knoten n_i erreicht die i-te Verwendung $u_{i+1}(x)$ im Knoten n_{i+1} entlang eines bezüglich x definitionsfreien Teilpfades von Knoten n_i zu Knoten n_{i+1}. Während die Knoten jeweils paarweise verschieden sein müssen, gilt diese Einschränkung nicht für die Variablen x_1, $x_2, \ldots, x_{k-1}$.

Ein Interaktionsteilpfad für κ ist dann ein Teilpfad der Form $p_\kappa = (n_1) \circ p_1 \circ (n_2) \circ \ldots \circ (n_{k-1}) \circ p_{k-1} \circ (n_k)$[8], wobei für alle i ($1 \leq i < k$) gilt: Teilpfad p_i ist definitionsfrei bezüglich Variable x_i.

Eine Testfallmenge T erfüllt das *required k-tuples*-Kriterium, falls von T mindestens ein Interaktionspfad für jede *l-dr*-Interaktion λ des Programms mit $2 \leq l \leq k$ überdeckt wird. Falls eine *l-dr*-Interaktion λ mit einer prädikativen Verwendung $u_l(x_{l-1})$ im Knoten n_l endet, dann müssen darüber hinaus alle Kanten, welche den Knoten n_l verlassen, ebenfalls überdeckt werden. Damit ist sichergestellt, dass alle Verzweigungen getestet wurden, also die Verzweigungsüberdeckung in diesem Kriterium enthalten ist. Falls die erste Definition in Knoten n_1 innerhalb einer Schleife liegt, muss die Testfallmenge solche Pfade überdecken, welche eine minimale Anzahl w_1 und

[8] „$\circ$“ bezeichnet die Konkatenation von Teilpfaden: ist $p = (n_2, n_3, n_4)$, dann gilt $(n_1) \circ p = (n_1, n_2, n_3, n_4)$

eine größere Anzahl $w_2 > w_1$ Wiederholungen der innersten Schleife mit Knoten n_1 enthalten. Analoges gilt auch für die letzte Verwendung in Knoten n_k, falls diese innerhalb einer Schleife vorkommt.

Die ursprüngliche Definition dieses Kriteriums von Ntafos forderte nur die Überdeckung aller *k-dr*-Interaktionen. Da es für jedes Programm eine obere Schranke K gibt, ab der keine *k-dr*-Interaktionen für $k > K$ existieren, wäre die Subsumptionshierarchie innerhalb der Familie nicht mehr gegeben, das heißt, das *required k-tuples*-Kriterium würde das *required (k-1)-tuples*-Kriterium nicht mehr enthalten. Aus den Beispielen von Ntafos geht jedoch hervor, dass er dies beabsichtigte, daher die abgewandelte Fassung nach [CPRZ89].

Um die eingeführten Begriffe und das Kriterium zu verdeutlichen, betrachte man exemplarisch erneut das Programm *BinarySearch* aus Listing 3.1 mit dem datenflussannotierten Kontrollflussgraphen aus Abbildung A.3. Daran kann man zum Beispiel folgende 4-*dr*-Interaktion erkennen: $\kappa^{ex} = [d_{n_{Start}}(list), u_{n_1^2}(list), d_{n_1^2}(high), u_{n_3}(high), d_{n_3}(mid), u_{n_7}(mid)]$. Die drei Definitionen und drei Verwendungen sind entsprechend den vier Knoten n_{Start}, n_1^2, n_3 und n_7 zugeordnet. Ein Interaktionsteilpfad für κ^{ex} ist hier eindeutig gegeben durch $p_{\kappa^{ex}} = (n_{Start}) \circ p_1^{ex} \circ (n_1^2) \circ p_2^{ex} \circ (n_3) \circ p_3^{ex} \circ (n_7) = (n_{Start}, n_1^1, n_1^2, n_1^3, n_1^4, n_2^{'}, n_2^{''}, n_3, n_4, n_6, n_7)$ mit den jeweiligen Teilpfaden $p_1^{ex} = (n_1^1)$, $p_2^{ex} = (n_1^3, n_1^4, n_2^{'}, n_2^{''})$ sowie $p_3^{ex} = (n_4, n_6)$. Für die Variable `found` gibt es hingegen nur 2-*dr*-Interaktionen, darunter zum Beispiel $\kappa_1^{found} = [d_{n_1^4}(found), u_{n_{11}}(found)]$ und $\kappa_2^{found} = [d_{n_5}(found), u_{n_2^{''}}(found)]$.

3.4.4 Datenflusskriterien nach Laski/Korel

Einen ähnlichen Ansatz wie Ntafos verfolgten auch Janusz W. Laski und Bogdan Korel bei der Definition der Familie der sogenannten *Datenkontextüberdeckung* [LK83]. Während die *Required-k-Tuples*-Kriterien jedoch Ketten von abwechselnden Definitionen und Verwendungen betrachten und damit den Fluss der Information entlang dieser Kette testen, stützen sich die Datenkontextüberdeckungskriterien auf die Tatsache, dass ein Knoten eines Kontrollflussgraphen die Verwendung mehrerer Variablen repräsentiert, deren Definitionen in unterschiedlichen vorangehenden Knoten stehen.

Sei auch hier wie im Kapitel 3.4.3 zum besseren Verständnis die Annahme getroffen, dass jeder Knoten genau eine Anweisung repräsentiert und die prädikativen Verwendungen von Variablen jeweils denjenigen Knoten zugeordnet werden, die die enstprechende Bedingungsauswertung darstellen [CPRZ89]. Darüber hinaus sei für einen datenflussannotierten Kontrollflussgraphen $G = (N, E)$ eines Programm(ausschnitt)s $\mathcal{P}$ mit Variablenmenge V, in Anlehnung an Definition 3.21 und Definition 3.22, noch die Menge use(n_u) wie folgt eingeführt:

Definition 3.26 (use(n_u)) *Die Menge* use(n_u) $\subseteq V$ *enthält genau diejenigen Variablen des von G dargestellten Codeausschnitts, die in Knoten* $n_u \in N$ *verwendet werden.*

Zur Beschreibung der verschiedenen Kriterien von Laski und Korel sind weitere Definitionen notwendig:

Definition 3.27 (ODC(n_u)) *Sei $n_u \in N$ ein Knoten des Kontrollflussgraphen $G = (N,E)$ und $\{v_1, v_2, \ldots, v_k\} = use(n_u)$ die Menge der im Knoten $n_u \in N$ verwendeten Variablen.*

Dann ist die Abfolge von Definitionen ODC(n_u) = [$d_1(v_1)$, $d_2(v_2)$, …, $d_k(v_k)$], mit $v_i \in def(n_i)$ für $i = 1 \ldots k$, ein sogenannter geordneter Definitionskontext *(engl.* ordered definition context*), falls es einen Teilpfad $p^{n_u}_{ODC}$ mit folgender Eigenschaft gibt: Für alle i mit $1 \leq i \leq k$ kann $p^{n_u}_{ODC}$ dargestellt werden als $p^{n_u}_{ODC} = p_i \circ (n_i) \circ p_i' \circ (n_u)$, so dass die Definition $d_i(v_i)$ im Knoten n_i vorkommt und der Teilpfad $(n_i) \circ p_i' \circ (n_u)$ definitionsfrei bezüglich v_i von Knoten n_i zu Knoten n_u ist, sowie für alle j mit $i < j \leq k$ ist entweder $n_i = n_j$ oder n_j kommt entlang des Teilpfades p_i' vor.*

Ein solcher Teilpfad $p^{n_u}_{ODC} = p \circ (n_u)$ heißt geordneter Kontextteilpfad *(engl.* ordered context subpath*) für ODC(n_u).*

Demnach ist ein geordneter Definitionskontext eines Knotens eine Sequenz von Definitionen, welche alle entlang des gleichen Teilpfades auftreten und jeweils eine Verwendung im betrachteten Knoten entlang dieses Teilpfades erreichen. Die Reihenfolge der Definitionen in der Sequenz ist die gleiche wie die ihres Auftretens entlang des entsprechenden geordneten Kontextteilpfads.

Definition 3.28 (DC(n_u)) *Sei erneut $\{v_1, v_2, \ldots, v_k\} = use(n_u)$ die Menge der in einem Knoten $n_u \in N$ verwendeten Variablen.*

Ein Definitionskontext *(engl.* definition context*) des Knotens n_u ist eine Menge von Definitionen DC(n_u) = $\{ d_1(v_1), d_2(v_2), \ldots, d_k(v_k) \}$.*

Analog zu Definition 3.27 ist $p^{n_u}_{DC}$ ein Kontextteilpfad *(engl.* context subpath*) für DC(n_u), falls die Definitionen des Definitionskontextes entsprechend entlang $p^{n_u}_{DC}$ vorkommen.*

An Definition 3.27 und Definition 3.28 kann man erkennen, dass es zu jedem Definitionskontext DC(n_u) mindestens einen geordneten Definitionskontext ODC(n_u) gibt, welcher die gleichen Definitionen enthält wie DC(n_u). Demnach stellen einzelne Permutationen der Menge DC(n_u) einen geordneten Definitionskontext des Knotens n_u dar. Ebenso ist jeder geordnete Kontextteilpfad für einen Kontext ODC(n_u) zugleich auch ein Kontextteilpfad für den zugehörigen Kontext DC(n_u).

Betrachtet man erneut das Beispiel aus Listing 3.1 mit der detaillierten Darstellung des Knoten n_1 wie in Abbildung A.4 unter der Annahme der vollständigen Bedingungsauswertung, so werden hier die prädikativen Verwendungen der Variablen `low`, `high` und `found` nicht wie in Abbildung 3.6 beziehungsweise Abbildung A.5 den Kanten (n_2, n_3) und (n_2, n_{11}), sondern stattdessen dem Knoten n_2 zugeordnet, also gilt: $use(n_2)$ = { `low`, `high`, `found` }. Bezüglich `low` gibt es zwei Definitionen, von denen aus die Verwendung in Knoten n_2 erreicht wird, nämlich $d_1^1(low)$ im Knoten n_1^1 und $d_7(low)$ im Knoten n_7. Analog dazu gibt es für die Variable `high` die beiden Definitionen $d_1^2(high)$ und $d_8(high)$ in den Knoten n_1^2 beziehungsweise n_8 sowie bezüglich `found` jeweils $d_1^4(found)$ im Knoten n_1^4 und $d_5(found)$ im Knoten n_5.

Die daraus resultierenden Definitionskontexte für Knoten n_2 sind in Tabelle 3.5 so zusammengestellt, dass die aus einem Definitionskontext hervorgehenden geordneten Kontexte erkennbar sind. Zum Beispiel gibt es zum Kontext $DC_7(n_2) = \{d_1^1(low), d_8(high), d_5(found)\}$ nur die beiden geordneten Definitionskontexte $ODC_9(n_2) = [d_1^1(low), d_8(high), d_5(found)]$ und

i	$DC_i(n_2)$	j	$ODC_j(n_2)$
1	$\{d_1^1(low), d_1^2(high), d_1^4(found)\}$	1	$[d_1^1(low), d_1^2(high), d_1^4(found)]$
2	$\{d_7(low), d_1^2(high), d_1^4(found)\}$	2	$[d_1^2(high), d_1^4(found), d_7(low)]$
3	$\{d_1^1(low), d_8(high), d_1^4(found)\}$	3	$[d_1^1(low), d_1^4(found), d_8(high)]$
4	$\{d_1^1(low), d_1^2(high), d_5(found)\}$	4	$[d_1^1(low), d_1^2(high), d_5(found)]$
5	$\{d_7(low), d_8(high), d_1^4(found)\}$	5	$[d_1^4(found), d_7(low), d_8(high)]$
		6	$[d_1^4(found), d_8(high), d_7(low)]$
6	$\{d_7(low), d_1^2(high), d_5(found)\}$	7	$[d_1^2(high), d_7(low), d_5(found)]$
		8	$[d_1^2(high), d_5(found), d_7(low)]$
7	$\{d_1^1(low), d_8(high), d_5(found)\}$	9	$[d_1^1(low), d_8(high), d_5(found)]$
		10	$[d_1^1(low), d_5(found), d_8(high)]$
8	$\{d_7(low), d_8(high), d_5(found)\}$	11	$[d_7(low), d_8(high), d_5(found)]$
		12	$[d_7(low), d_5(found), d_8(high)]$
		13	$[d_8(high), d_7(low), d_5(found)]$
		14	$[d_8(high), d_5(found), d_7(low)]$
		15	$[d_5(found), d_7(low), d_8(high)]$
		16	$[d_5(found), d_8(high), d_7(low)]$

Tabelle 3.5: Definitionskontexte des Knotens n_2 von *BinarySearch*

$ODC_{10}(n_2) = [d_1^1(low), d_5(found), d_8(high)]$. Einen geordneten Kontextteilpfad für den Kontext $ODC_{13}(n_2) = [d_8(high), d_7(low), d_5(found)]$ erzielt man zum Beispiel mit dem Testfall $t_{13}^{ODC} := (2, \langle\, 1,2,3,4,5 \,\rangle)$ durch Überdeckung des Pfades $p_{t_{13}^{ODC}} = (n_{Start}, n_1^1, n_1^2, n_1^3, n_1^4, n_2, n_3, n_4, n_6, \underline{n_8}, n_9, n_{10}, n_2, n_3, n_4, n_6, \underline{n_7}, n_9, n_{10}, n_2, n_3, n_4, \underline{n_5}, n_{10}, \underline{n_2}, n_{11}, n_{12}, n_{14}, n_{End})$.

Man beachte jedoch, dass die in Tabelle 3.5 dargestellten $ODC_j(n_2)$ lediglich die mittels einer statischen Analyse ermittelbaren Kontexte sind. Insbesondere gibt es darunter geordnete Definitionskontexte für die kein geordneter Kontextteilpfad überdeckt werden kann. Dies trifft zum Beispiel auf alle geordneten Definitionskontexte zu, die eine Definition von `found` im Knoten n_5 enthalten, welches nicht die letzte Definition in der Abfolge ist. Da nach Durchlaufen des Knotens n_5 die Schleife nicht erneut iteriert wird, können die darauf folgenden Definitionen nicht mehr erreicht werden, wie es beispielsweise bei $ODC_{15}(n_2) = [d_5(found), d_7(low), d_8(high)]$ und den nicht mehr erreichbaren Definitionen $d_7(low)$ und $d_8(high)$ der Fall ist.

Darauf aufbauend kann nun die Familie der Datenkontextüberdeckungsstrategien nach Laski und Korel definiert werden, welche im Wesentlichen aus den drei Kriterien *reach coverage*, *context coverage* und *ordered context coverage* besteht.

Eine Testfallmenge T erfüllt das ***reach coverage***-Kriterium, falls die von T überdeckte Teilpfadmenge mindestens einen Teilpfad von jeder Definition jeder Variablen zu jeder davon erreichbaren Verwendung enthält. Formal heißt das: Für jeden Knoten $n_d \in N$ und jede Variable $v \in def(n_d)$ ist mindestens ein Teilpfad von n_d zu jedem Knoten $n_u \in N$ mit $v \in use(n_u)$ auszuführen, welcher definitionsfrei bezüglich v von n_d zum entsprechenden Knoten n_u ist. Dieses Kriterium ähnelt dem *all-uses*-Kriterium von Rapps und Weyuker (siehe Kapitel 3.4.2), mit dem

entscheidenden Unterschied, dass bei Letzterem, im Falle einer prädikativen Verwendung in n_u, alle von n_u ausgehenden Kanten ebenfalls überdeckt werden müssen, während bei *reach coverage* nur der Knoten n_u mit der zugeordneten Verwendung (gleich welcher Art) erreicht werden muss. Durch eine in Kapitel 3.5 beschriebene Erweiterung zum sogenannten *reach coverage+* erhält man jedoch ein zu *all-uses* äquivalentes Kriterium.

Das ***context coverage***-Kriterium fordert die Überdeckung aller Definitionskontexte eines Programms. Zur Erfüllung dieses Kriteriums muss demnach die von einer Testfallmenge überdeckte Teilpfadmenge für jeden Knoten $n_u \in N$ und alle Definitionskontexte $DC(n_u)$ mindestens einen Kontextteilpfad für $DC(n_u)$ enthalten.

Analog dazu ist das ***ordered context coverage***-Kriterium erfüllt, wenn die während der Testausführung überdeckte Teilpfadmenge für jeden Knoten $n_u \in N$ und alle geordneten Definitionskontexte $ODC(n_u)$ mindestens einen geordnete Kontextteilpfad für $ODC(n_u)$ enthält.

In [CPRZ89] findet man für die letzten beiden Kriterien jeweils eine gegenüber der ursprünglichen Fassung von Laski und Korel leicht angepasste Variante. Dabei enthalten die dort definierten (geordneten) Definitionskontexte nicht mehr zwangsweise die Definitionen *aller* in einem Knoten verwendeten Variablen wie bei Laski und Korel, sondern lediglich die einer beliebigen nicht-leeren Teilmenge davon.

3.4.5 Interprozeduraler Datenfluss nach Alexander/Offut

Die in Kapitel 3.4.2, Kapitel 3.4.3 und Kapitel 3.4.4 vorgestellten Datenflussüberdeckungskriterien wurden ursprünglich für einzelne Funktionen prozeduraler Sprachen definiert und sind somit eher für die Betrachtung des intraprozeduralen Datenflusses geeignet. Andererseits bietet gerade das objekt-orientierte Programmierparadigma vielfältige Mechanismen zur Modularisierung der Software: Von einzelnen Methoden, über Klassen (als Zusammenfassung mehrerer Methoden und Attribute), bis hin zu größeren Klassenverbünden (wie Packages oder Komponenten). Dadurch entsteht, zusätzlich zum Unit-Test nach den vorangehend vorgestellten Kriterien, auch der Bedarf nach einer Berücksichtigung des interprozeduralen oder modulübergreifenden Kontroll- und Datenflusses, wie am Beispiel aus Abbildung 3.4 (Seite 41) angedeutet.

Während es bei den klassischen, intraprozeduralen Kriterien um die Verifikation einzelner Einheiten (Units) geht, beschäftigen sich die interprozeduralen Überdeckungsstrategien vorwiegend mit der Überprüfung der Interaktion und dem Zusammenspiel dieser Einheiten – womit sie eher der Integrationstestphase zuzuordnen sind (siehe Kapitel 2.2.1 bzw. Abbildung 2.1).

Eine Hierarchie solcher Kriterien mit besonderem Fokus auf den interprozeduralen Datenfluss hat eine Forschergruppe um Jefferson A. Offutt entwickelt [AO00]. Um die einzelnen Strategien zu beschreiben, seien zusätzlich zu Definition 3.12, Definition 3.13 sowie Definition 3.16, weitere Konzepte wie folgt festgelegt. Dabei seien $G_1 := (N_1, E_1)$ und $G_2 := (N_2, E_2)$ die Kontrollflussgraphen zweier Einheiten $\mathcal{P}_1$ und $\mathcal{P}_2$, welche zum Beispiel Methoden oder auch Module sein können.

Definition 3.29 (callsite) *Eine* Aufrufstelle (callsite) *ist ein Knoten $n_i^1 \in N_1$ des Kontrollflussgraphen von $\mathcal{P}_1$, welcher einen Aufruf von $\mathcal{P}_2$ darstellt.*

In Abbildung 3.4 ist Knoten n_7 eine solche Aufrufstelle, wobei der Einheit $\mathcal{P}_1$ hier die Methode `BubbleSort.sort()` beziehungsweise $\mathcal{P}_2$ analog `Apfel.vergleicheMit(Object)` entspricht.

Definition 3.30 (coupling-def, coupling-use) *Falls ein Knoten $n_i^1 \in N_1$ eine Definition einer Variablen x enthält und es einen bezüglich x definitionsfreien Teilpfad von n_i^1 zu einem Knoten $n_j^2 \in N_2$ mit einer Verwendung von x in $\mathcal{P}_2$ gibt, so heißt diese Definition* Kopplungsdefinition (coupling-def) *und die entsprechende Verwendung* Kopplungsverwendung (coupling-use).

Definition 3.31 (coupling-path, coupling path set) *Ein Teilpfad zwischen zwei Einheiten eines Programms ist ein* Kopplungsteilpfad (coupling-path)*, falls er mit einer Definition einer Variablen x im aufrufenden Modul beginnt, mit einer Verwendung von x im aufgerufenen Modul endet und bezüglich x definitionsfrei ist.*

Eine Kopplungsteilpfadmenge (coupling path set) *ist eine Menge von Knoten, welche entlang eines Teilpfades von einer Kopplungsdefinition zu einer zugehörigen, davon erreichbaren Kopplungsverwendung auftreten können.*

Darauf aufbauend werden in Anlehnung an die klassischen datenflussorientierten Kriterien nach Rapps/Weyuker nun folgende interprozedurale Kriterien definiert:

- *All-coupling-defs*: Von jeder Kopplungsdefinition jeder Variablen in $\mathcal{P}_1$ muss mindestens ein Kopplungsteilpfad zu mindestens einer zugehörigen, davon erreichbaren Kopplungsverwendung überdeckt werden.
- *All-coupling-uses*: Von jeder Kopplungsdefinition jeder Variablen in $\mathcal{P}_1$ muss mindestens ein Kopplungsteilpfad zu jeder davon erreichbaren Kopplungsverwendung überdeckt werden.
- *All-coupling-paths*: Von jeder Kopplungsdefinition jeder Variablen in $\mathcal{P}_1$ muss jede Kopplungsteilpfadmenge zu jeder davon erreichbaren Kopplungsverwendung überdeckt werden.

Die Forderung nach allen Kopplungsteilpfadmengen anstelle aller Kopplungsteilpfade im Kriterium *All-coupling-paths* trägt dem Problem Rechnung, dass eventuelle Schleifen entlang solcher Kopplungsteilpfade zu einer potentiell unendlichen Menge von zu überdeckenden Teilpfaden führen. Dieses Kriterium entspricht somit in etwa dem von Rapps/Weyuker vorgeschlagenen *all-DU-paths* (siehe Kapitel 3.4.2).

Darüber hinaus wird in [AO00] eine Reihe weiterer datenflussorientierter Kopplungskriterien definiert, welche im Wesentlichen die Besonderheiten der Aufrufstrukturen von Methoden in objekt-orientierten Systemen genauer beleuchten und im Folgenden kurz skizziert werden. Dazu wird das klassische Konzept des *def/use*-Paares auf die Definition und Verwendung von Instanzvariablen, über die Grenzen einzelner Methoden hinweg, ausgedehnt.

Sind `O` eine Klasse sowie `o` eine Variable mit einer Referenz auf eine Instanz `ô` dieser Klasse und wird die Methode `m()` des Objekts `ô` aufgerufen („`o.m()`“), so nennt man `o` die *Kontextvariable (context variable)* und es heißt, die Methode `m` wird im *Instanzkontext (instance context)*

von ô ausgeführt. Ruft eine Methode `f()` ihrerseits eine weitere Methode `r()` auf, so wird `r()` im Kontext der Methode `f()` ausgeführt.

Ein Paar zweier Methodenaufrufe, welche jeweils im Kontext einer gemeinsamen Methode, genannt *Kopplungsmethode (coupling method)*, und im gleichen Instanzkontext aufgerufen werden, stellt eine sogenannte *intra-method coupling sequence* (kurz *intraMCS*) dar, falls es zwischen den beiden Aufrufstellen mindestens einen Teilpfad gibt, welcher bezüglich der Kontextvariablen und mindestens einer weiteren Zustandsvariablen, die in der ersten Methode definiert und in der zweiten verwendet wird, definitionsfrei ist.

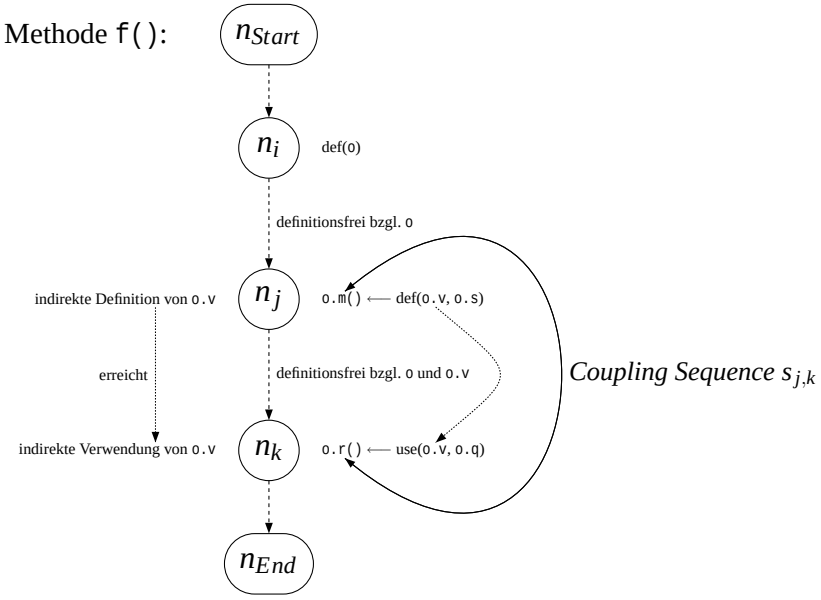


Abbildung 3.7: Beispiel einer *intra-method coupling sequence*

Ein einfaches Beispiel für eine solche *intraMCS* zeigt Abbildung 3.7 nach [AO00]. Dabei enthalten der Antezedenzknoten *(antecedent node)* n_j einen Aufruf der Antezedenzmethode *(antecedent method)* `o.m()` sowie der Konsequenzknoten *(consequent node)* n_k einen Aufruf der Konsequenzmethode *(consequent method)* `o.r()`.

Analog dazu stellt ein Paar zweier Methodenaufrufe (zum Beispiel `o.m()` und `o.r()`) eine sogenannte *inter-method coupling sequence* (kurz *interMCS*) dar, falls die Antezedenz- und die Konsequenzmethode (hier `m()` beziehungsweise `r()`) jeweils im Kontext unterschiedlicher Methoden (beispielsweise `g()` und `h()`) jedoch weiterhin im gleichen Instanzkontext (hier das Objekt ô, das von `o` referenziert wird) aufgerufen werden, und es zwischen den beiden Aufrufstellen mindestens einen Teilpfad gibt, welcher bezüglich der Kontextvariablen (hier `o`) und mindestens einer weiteren Zustandsvariablen, die in der Antezedenzmethode definiert und in der Konsequenzmethode verwendet wird, definitionsfrei ist. In diesem Falle ist diejenige Methode (zum Beispiel `f()`) die *Kopplungsmethode*, welche die beiden Kontextmethoden (hier `g()` und `h()`) nacheinander aufruft.

Eine Kopplungssequenz $s_{j,k}$ entsteht aufgrund einer Menge von Zustandsvariablen, welche in der Antezedenzmethode definiert und in der Konsequenzmethode benutzt werden, und zwischen deren Definition und davon erreichbarer Verwendung ein bezüglich der jeweiligen Variable definitionsfreier Teilpfad existiert. Diese Menge heißt *Kopplungsmenge (coupling set)* und jedes ihrer Elemente ist eine *Kopplungsvariable (coupling variable)*. Der bezüglich einer Kopplungsvariable definitionsfreie Teilpfad zwischen der letzten Definition der Kopplungsvariable in der Antezedenzmethode und ihrer ersten Verwendung in der Konsequenzmethode wird *Kopplungsteilpfad (coupling path)* genannt.

In Abbildung 3.7 ist $\Theta_{s_{j,k}} = \{class(o) :: v\}$ die Kopplungsmenge der *intraMCS* $s_{j,k}$ mit der einzigen Kopplungsvariablen `v`, wobei `class(o)` der deklarierte Typ ihrer Kontextvariablen `o` ist.

Aufgrund obiger Definitionen lassen sich nun weitere datenflussbasierte und kopplungsorientierte Überdeckungskriterien definieren, wobei sich [AO00] auf die Betrachtung der *intraMCS* beschränkt:

- *All-Coupling-Sequences*: Für jede Kopplungsmethode `f()` eines objekt-orientierten Programms und jede Kopplungssequenz $s_{j,k}$ in `f()`, wird mindestens ein Kopplungsteilpfad dieser Kopplungssequenz $s_{j,k}$ überdeckt.

- *All-Poly-Classes*: Für jede Kopplungsmethode `f()`, jede Kopplungssequenz $s_{j,k}$ in `f()` und jede Klasse aus der Typfamilie des Instanzkontextes der Kopplungssequenz $s_{j,k}$ wird mindestens ein Kopplungsteilpfad dieser Kopplungssequenz $s_{j,k}$ überdeckt. Dazu werden nur Klassen dieser Typfamilie berücksichtigt, die mit dem deklarierten Typ übereinstimmen oder eine Unterklasse dieses Typs sind und dabei mindestens eine der beiden Methoden (Antezedenz- oder Konsequenzmethode) überschreiben.

- *All-Coupling-Defs-and-Uses*: Für jede Kopplungsmethode `f()`, jede Kopplungssequenz $s_{j,k}$ in `f()`, jede Kopplungsvariable v der Kopplungssequenz $s_{j,k}$ und jeden Knoten n_d der Antezedenzmethode von $s_{j,k}$, welcher eine letzte Definition von v enthält, wird mindestens ein Kopplungsteilpfad überdeckt, welcher mit der letzten Definition von v in n_d beginnt und mit einem Knoten der Konsequenzmethode von $s_{j,k}$ endet, welcher eine erste Verwendung von v darstellt.

- *All-Poly-Coupling-Defs-and-Uses*: Für jede Kopplungsmethode `f()`, jede Kopplungssequenz $s_{j,k}$ in `f()`, jede Klasse aus der Typfamilie des Instanzkontextes der Kopplungssequenz $s_{j,k}$, jede Kopplungsvariable v der Kopplungssequenz $s_{j,k}$ und jeden Knoten n_d der Antezedenzmethode von $s_{j,k}$, welcher eine letzte Definition von v enthält, wird mindestens ein Kopplungsteilpfad überdeckt, welcher mit der letzten Definition von v in n_d beginnt und mit einem Knoten der Konsequenzmethode von $s_{j,k}$ endet, welcher eine erste Verwendung von v darstellt.

3.4.6 Probleme beim Datenflusstesten und mögliche Lösungsansätze

Die von Rapps und Weyuker erstmals eingeführten datenflussorientierten Testüberdeckungskriterien (siehe Kapitel 3.4.2) wurden zunächst nur anhand einer eigens definierten „Programmier-

sprache“ vorgestellt [RW85], welcher allerdings viele Konzepte moderner Programmiersprachen fehlten. Daher war die Theorie dieser Kriterien elegant und einfach nachzuvollziehen, jedoch praktisch kaum anwendbar. So gingen Rapps und Weyuker weder auf die damals in der Sprache C weit verbreitete Nutzung von Zeigern und strukturierten Daten ein, noch behandelten sie den sogenannten interprozeduralen Daten- und Kontrollfluss, sondern beschränkten sich auf die Datenflussanalyse einer einzigen Funktion in einer sehr primitiven Sprache. Mit der Verbreitung objekt-orientierter Programmiersprachen, wie zum Beispiel der hier betrachteten Sprache JAVA™, entstand auch der Bedarf nach Erweiterungen der ursprünglichen Theorie, da neue Konzepte wie Mehrfachinstantiierung oder Polymorphie sowohl die Komplexität des Kontrollflusses als auch insbesondere die des Datenflusses in den Programmen erheblich erhöhten.

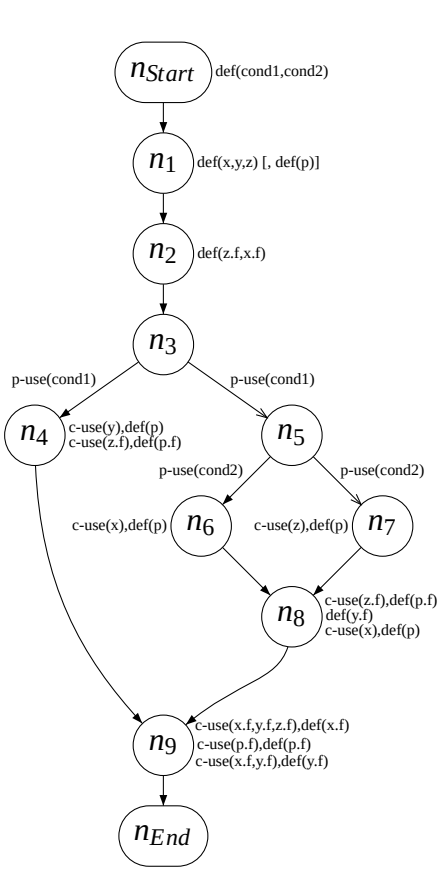
Im Zuge des aktuellen Trends hin zur komponentenbasierten Software-Entwicklung werden immer mehr Software-Systeme aus zum Teil vorgefertigten Bausteinen zusammengesetzt, deren Quellcode im neuen Kontext meist nicht verfügbar ist. Ähnlich verhält es sich auch in der Sprache JAVA™ mit plattformspezifischen Komponenten, welche nativ, das heißt außerhalb der Programmiersprache selbst (z.B. in Laufzeitbibliotheken zum Zugriff auf Betriebssystemfunktionen), implementiert sind. Da sich solche Codeportionen einer statischen Analyse des Datenflusses entziehen, erweist sich eine effiziente und präzise Datenflussanalyse als äußerst schwierig oder gar unmöglich. Die im Rahmen dieser Arbeit entwickelten Verfahren [OD04, Pol05, CRL99] sind speziell für diesen Einsatzzweck konzipiert worden (siehe Kapitel 5.1 und Kapitel 5.2).

Eines der schwierigsten Probleme der Datenflussanalyse stellt das in der Literatur unter dem Namen *pointer aliasing* bekannte Konzept dar. Dahinter verbirgt sich der Fall, dass es für einen Speicherort mehr als nur einen Namen gibt, also mindestens zwei Variablennamen den gleichen gemeinsamen Speicherbereich bezeichnen. Dieses Phänomen erhielt seinen Namen aufgrund von Programmiersprachen mit Zeigern, wie zum Beispiel C. Dennoch treten ähnliche Effekte auch bei objekt-orientierten Sprachen wie JAVA™ auf, denen zwar das Konzept des Zeigers fehlt, welche jedoch Instanzen über sogenannte Objekt-Referenzen adressieren.

Das sich daraus ergebende Problem für die (statische) Datenflussanalyse sei am Beispiel aus Abbildung 3.8 illustriert. Es basiert auf einem Programmfragment aus [OW91], welches in seiner ursprünglichen Form für die Sprache C in Abbildung A.6 zu finden ist und hier in eine entsprechende Darstellung für JAVA™ übersetzt wurde.

Demnach „referenzieren“ die Variablen `x`, `y` und `z` nach Ausführung der Zeile 7 jeweils paarweise verschiedene Instanzen der Klasse *PointerAliasingEx1*, jede mit einer eigenen Instanz des zugehörigen Feldes `f`. Nachdem die Zeile 11 ausgeführt wurde, bezeichnet die Variable `p` das gleiche Objekt wie `y`, weshalb nun die Definition von `p.f` in Zeile 12 einer Definition des Feldes `y.f` gleichzusetzen ist. Somit besteht in Knoten n_4 eine pointer-aliasing-Beziehung zwischen den Variablen `p` und `y`. Während dieser Zusammenhang hier noch statisch eindeutig identifiziert werden kann, ist es in Zeile 19 ungleich schwieriger. Dies liegt daran, dass `p.f` entweder mit `x.f` oder mit `z.f` gleichzusetzen ist, je nachdem ob unmittelbar vor Erreichen des Knotens n_8 der Knoten n_6 (`p = x;`) oder der Knoten n_7 (`p = z;`) ausgeführt wurde. Ähnlich verhält es sich auch mit den Verwendungen in Zeile 23: entweder der c-use von `y.f` oder derjenige von `x.f` stellt zugleich einen c-use von `p.f` dar, abhängig davon ob `cond1` wahr ist oder nicht.

Da die Kenntnis über Querbeziehungen dieser Art nicht nur für das datenflussorientierte




```
1    public class PointerAliasingEx1 {
2        public int f;
3        public void m(boolean cond1, boolean cond2) {
4            PointerAliasingEx1 x = new PointerAliasingEx1();
5            PointerAliasingEx1 y = new PointerAliasingEx1();
6            PointerAliasingEx1 z = new PointerAliasingEx1();
7            PointerAliasingEx1 p;
8            z.f = 17;
9            x.f = 13;
10           if (cond1) {
11               p = y;
12               p.f = z.f;
13           } else {
14               if (cond2) {
15                   p = x;
16               } else {
17                   p = z;
18               }
19               p.f = 7 + z.f;
20               y.f = 53;
21               p = x;
22           }
23           x.f = x.f + y.f + z.f;
24           p.f = p.f + 5;
25           y.f = x.f + y.f;
26       }
27   }
```

Abbildung 3.8: Kontrollflussgraph der Methode *PointerAliasingEx1.m(boolean, boolean)*

Testen von Bedeutung ist, sondern insbesondere zum Beispiel für Codeoptimierungen beim Compilerbau, gibt es eine Reihe unterschiedlicher Ansätze, das Problem des pointer-aliasing im Rahmen der statischen Analyse eines Programmcodes anzugehen. Man unterscheidet diese Verfahren im Wesentlichen anhand zweier Eigenschaften: Kontext-Sensitivität und Fluss-Sensitivität [Yur99].

Verfahren, welche weder kontext-sensitiv noch fluss-sensitiv sind, ignorieren die Reihenfolge, in der die Anweisungen eines Programms zur Laufzeit tatsächlich ausgeführt werden. Dadurch entsteht für jede Variable *v* eine Referenzliste (sogenannte *points-to*-Menge), welche alle Variablen *w* enthält, auf deren Speicherstelle *v* ebenfalls zeigen könnte. Diese ist daher an jeder Stelle im Programm gleich, unabhängig davon, ob bestimmte Beziehungen an der entsprechenden Stelle überhaupt auftreten können oder nicht. Im Beispiel aus Abbildung 3.8 würde ein solches Verfahren für die Variable `p` aufgrund der Programmzeilen 11, 15 und 17 die Menge `{x, y, z}` liefern, welche sowohl nach Zeile 22 als auch (besonders fatal) schon bezüglich des Datenflusses in Zeile 12 berücksichtigt werden müsste. Entsprechendes gilt dann auch für die jeweiligen Felder `p.f`, `x.f`, `y.f` und `z.f`.

Ein fluss-sensitives Verfahren erzeugt für jede Variable an jeder Stelle im Programm jeweils eine eigene *points-to*-Menge. Damit wird der Kontrollfluss berücksichtigt, welcher der gerade betrachteten Programmstelle vorangeht. Solch ein Algorithmus würde für `p` ab Zeile 22 nur noch die Menge `{x, y}` annehmen, da die Beziehung zwischen `p` und `z` aus Zeile 17 in der Zeile 21 mit Sicherheit aufgehoben wird.

Während sich die Fluss-Sensitivität auf den Kontrollfluss innerhalb eines Moduls (Funktion oder Methode) bezieht, betrifft die Kontext-Sensitivität den sogenannten Aufrufkontext verschiedener Methoden. Enthält zum Beispiel eine Methode `m1` zwei getrennte Aufrufe derselben Methode `m2`, welche sich auf die Alias-Beziehungen der Variablen aus `m1` auswirken, dann übernimmt ein kontext-insensitives Verfahren alle durch den Aufruf von `m2` neu entstehenden Beziehungen sowohl in die *points-to*-Mengen nach dem ersten Aufruf als auch in jene nach dem zweiten Aufruf, ohne Rücksicht darauf, dass die neuen Beziehungen nach dem zweiten Aufruf sich gar nicht auf die *points-to*-Mengen nach dem ersten auswirken können.

Das Beispiel aus Listing 3.3 in Anlehnung an [Yur99] soll dies verdeutlichen. Aufgrund einer fluss-sensitiven Analyse enthält die *points-to*-Menge von `p` nach Ausführung von Zeile 10 lediglich die Variable `y`. Demnach erfolgt der Aufruf der Methode `m2` mit einer Referenz auf `y` in Zeile 11 und auf `z` in Zeile 14 – wobei diese beiden Objekte jeweils der zu `m2` lokalen Variablen `w` zugewiesen werden. Im Falle einer kontext-insensitiven Analyse ergibt sich nach Ausführung der Methode `m2` in Zeile 19 für die Variable `q` die *points-to*-Menge `{y, z}`, da für die Analyse der Methode `m2` beide Aufrufe (sowohl aus Zeile 11 als auch aus Zeile 14) zusammengefasst werden. Wird diese aliasing-Beziehung nun an beide Aufrufstellen in den Zeilen 11 und 14 weitergereicht, muss in Zeile 12 sowohl eine potentielle Definition des Feldes `y.f` als auch eine des Feldes `z.f` angenommen werden, obwohl letztere tatsächlich gar nicht vorkommen kann. Ist die Analyse jedoch kontext-sensitiv, dann würde der jeweilige Aufrufkontext in den Zeilen 11 und 14 beachtet. Dadurch würde nach Rückkehr aus der Methode `m2` in Zeile 11 lediglich eine Beziehung zwischen `q` und `y` angenommen werden, da für den Aufruf von `m2` im Kontext der Zeile 11 nur die Beziehung zwischen `p` und `y` berücksichtigt wird.

Während demnach kontrollfluss- und aufrufkontext-insensitive Verfahren zur statischen Ana-

```
1  public class PointerAliasingEx2 {    // ContextSensitivity
2     public int f, t;
3     PointerAliasingEx2 x = new PointerAliasingEx2();
4     PointerAliasingEx2 y = new PointerAliasingEx2();
5     PointerAliasingEx2 z = new PointerAliasingEx2();
6     PointerAliasingEx2 p, q;
7
8     public void m1() {
9        p = x;
10       p = y;
11       m2(p);
12       q.f = 4711;
13       t = x.f + z.f;
14       m2(z);
15    }
16
17    private void m2(PointerAliasingEx2 w) {
18       q = w;
19    }
20 }
```

Listing 3.3: Quellcode-Beispiel zur fluss- und kontext-sensitiven Analyse

lyse des Datenflusses relativ schnell, dafür jedoch oft sehr ungenau sind, erfordert die durch sensitive Methoden erzielte Präzision einen ungleich höheren Analyseaufwand. In vielen Fällen ist eine exakte statische Analyse des Datenflusses mit heutigen Mitteln gar nicht mehr möglich. Dies tritt zum Beispiel auf, sobald der Kontrollfluss aufgrund von Schleifen zu komplex wird, was oft dann der Fall ist, wenn die Anzahl der Wiederholungen des Schleifenrumpfs nicht im Vorhinein ermittelt werden kann. Je nach Anforderung an die Präzision der Analyse gibt es eine Vielzahl unterschiedlicher Ansätze zur statischen Bestimmung des Datenflusses, von denen ein interessanter hier kurz skizziert sei.

Datenflussanalyse nach Ostrand/Weyuker [OW91]

Um das Problem des *pointer-aliasing* zu behandeln, wurden in [OW91] die ursprünglichen Datenflusskriterien nach Rapps und Weyuker erweitert. Mittels einer statischen Analyse, welche grundsätzlich beliebig sensitiv sein kann, wird für jede Variable und jede Programmstelle die zugehörige *points-to*-Menge („*alias set*“ in [OW91]) ermittelt. Darauf aufbauend werden Definitionen und Verwendungen genauer untersucht und in *definite def/use* beziehungsweise *possible def/use* klassifiziert.

Eine Definition (Verwendung) einer Speicherstelle ist ein *definite def (use)* einer Variablen *v*, falls es sich bei der betrachteten Speicherstelle eindeutig um die von Variable *v* referenzierte handelt[9]. Analog ist eine Definition (Verwendung) einer Speicherstelle ein *possible def (use)*

[9] Dies ist sowohl bei primitiven Variablen der Fall als auch bei Zeigern, deren *points-to*-Menge nur das Element *v* enthält.

einer Variablen *v*, falls der Zugriff eine Referenz betrifft, deren *points-to*-Menge mehr als ein Element enthält, darunter insbesondere die Variable *v*.

Eine entsprechende Unterscheidung wird auch bei den definitionsfreien (Teil-)Pfaden (siehe Definition 3.16 auf Seite 53) notwendig: Ein (Teil-)Pfad ist bezüglich Variable *v* nur dann *definite def-clear*, falls weder *definite* noch *possible defs* der Variable *v* entlang dieses (Teil-)Pfades auftreten. Gibt es hingegen keine *definite defs* jedoch mindestens ein *possible def*, so heißt der (Teil-)Pfad *possible def-clear*.

Eine def-use-Assoziation (auch DU-Paar genannt) bezüglich Variable *v* ist ein Tripel (n, m, v), falls Knoten *n* eine (definite oder possible) Definition von *v* enthält, Knoten *m* eine (definite oder possible) Verwendung von *v* darstellt und es mindestens einen bezüglich *v* (definite oder possible) definitionsfreien Teilpfad von *n* nach *m* gibt. Aufgrund obiger Definitionen wurden nun vier Klassen solcher def-use-Assoziationen eingeführt. Demnach heißt eine def-use-Assozation:

- ***strong:*** falls die Definition, die Verwendung und alle sie verbindenden definitionsfreien Teilpfade *definite def-clear* sind;
- ***firm:*** falls die Definition, die Verwendung und mindestens ein sie verbindender Teilpfad *definite def-clear* sind, sowie mindestens einer der sie verbindenden Teilpfade *possible def-clear* ist;
- ***weak:*** falls die Definition und die Verwendung *definite* sind, jedoch kein einziger der sie verbindenden Teilpfade *definite def-clear* ist;
- ***very weak:*** falls entweder die Definition oder die Verwendung oder beide *possible* sind.

Somit entstehen aus dem ursprünglichen *all-uses*-Kriterium nach Rapps und Weyuker vier neue Kriterien: *strong all-uses*, *firm all-uses*, *weak all-uses* beziehungsweise *very weak all-uses* analog zur Definition in Kapitel 3.4.2, jedoch nicht mehr mit dem Ziel der Überdeckung *aller* DU-Paare sondern nur noch derjenigen aus der entsprechenden Assoziationsklasse.

3.5 Subsumptionsrelation

Die in Kapitel 3.2, Kapitel 3.3 und Kapitel 3.4 dargestellten Testüberdeckungskriterien sind sogenannte strukturelle oder *white-box*-Kriterien und basieren allesamt auf einer graphischen Darstellung des Kontrollflusses der zu testenden Programmeinheit. Dieser sogenannte Kontrollflussgraph wird teilweise um zusätzliche Informationen angereichert, wie im Falle der Datenflussteststrategien. Dabei fordern die Kriterien die Überdeckung bestimmter Teilpfade im Graphen, weshalb man sie auch *Pfadauswahlkriterien* (engl. *path selection criteria*) nennt.

Sind $\mathcal{P}$ ein Programm oder Modul, C_K ein Testüberdeckungskriterium und T eine Testfallmenge zum Testen von $\mathcal{P}$, dann gilt: T ***erfüllt*** das Kriterium C_K bezüglich $\mathcal{P}$, in Prädikatschreibweise ausgedrückt durch $C_K(\mathcal{P}, T) = \mathit{wahr}$, wenn durch Ausführung der Testfälle aus T alle vom Kriterium geforderten Entitäten (zum Beispiel alle Anweisungen) von $\mathcal{P}$ überdeckt wurden und somit für die Metrik des Kriteriums $C_K = 100\%$ gilt.

Sind C_K und C_L zwei Testkriterien, dann sagt man C_K ***subsumiert*** C_L, wenn jedes beliebige Paar $(\mathcal{P}, T)$ welches Kriterium C_K erfüllt, auch das Kriterium C_L erfüllt. Stellt man diese Subsumptionsrelation als $C_K \rightarrow C_L$ dar, dann gilt: $C_K \rightarrow C_L \Leftrightarrow (\forall \mathcal{P}, T : C_K(\mathcal{P}, T) \Rightarrow C_L(\mathcal{P}, T))$. Entsprechend gelten zwei Kriterien C_K und C_L als ***äquivalent*** ($C_K \leftrightarrow C_L$), wenn sich die beiden Kriterien gegenseitig subsumieren: $C_K \leftrightarrow C_L \Leftrightarrow (C_K \rightarrow C_L \wedge C_L \rightarrow C_K)$ beziehungsweise $C_K \leftrightarrow C_L \Leftrightarrow (\forall \mathcal{P}, T : C_K(\mathcal{P}, T) \Leftrightarrow C_L(\mathcal{P}, T))$. Ein Kriterium C_K ***subsumiert streng*** C_L, wenn zwar C_K das Kriterium C_L subsumiert, jedoch nicht umgekehrt. Falls weder C_K das Kriterium C_L subsumiert noch umgekehrt, dann sind die beiden Kriterien C_K und C_L ***unvergleichbar***.

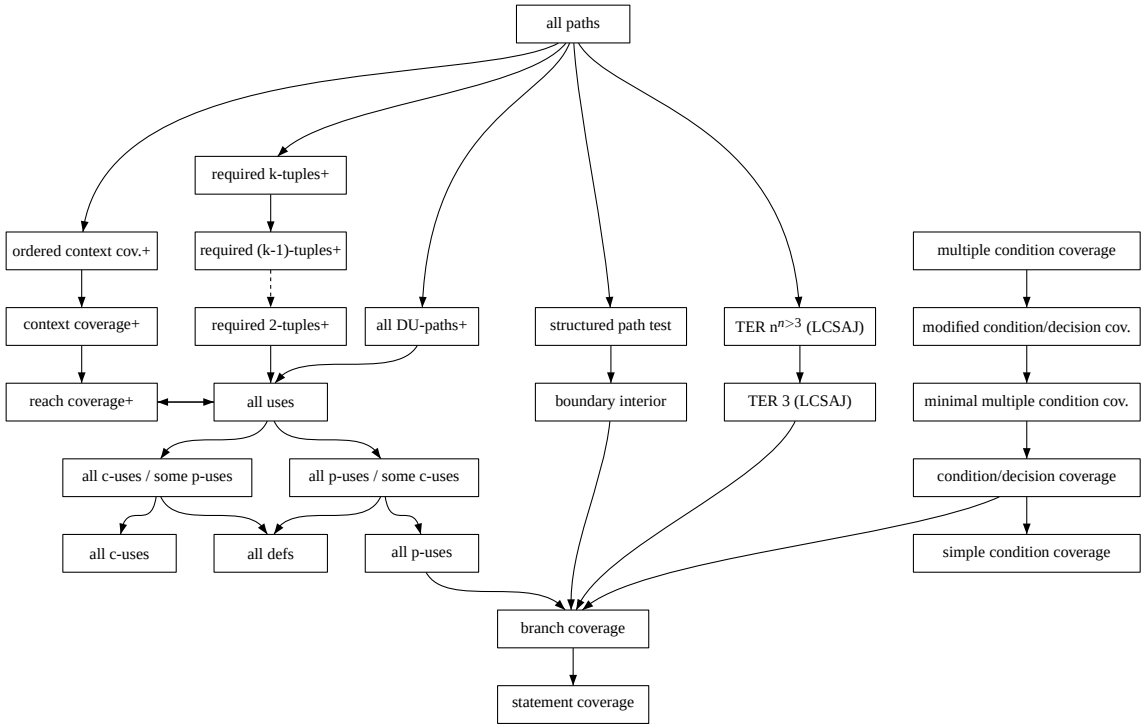


Abbildung 3.9: Subsumptionsrelation verbreiteter white-box-Strategien

Obige Subsumptionsrelation induziert eine partielle Ordnung über jede beliebige Menge solcher strukturellen Testüberdeckungskriterien, welche sich in einer sogenannten Subsumptionshierarchie wie in Abbildung 3.9 nach [ZHM94, ZHM97, CPRZ89] zusammenfassen lässt. Da der Beweis aller dargestellten Zusammenhänge den Rahmen dieser Arbeit sprengen würde, wird er hier nur exemplarisch für ausgewählte Paare anschaulich skizziert. Für weitergehende Betrachtungen sei auf die entsprechende Literatur verwiesen [RW85, HL91, CPRZ89].

Intuitiv ist nachvollziehbar, dass die Überdeckung aller Kanten eines Kontrollflussgraphen und damit jeder Verzweigung eines Programms auch automatisch die Überdeckung aller Knoten dieses Graphen, also jeder Anweisung des Programms, zur Folge hat. Dennoch kann es bei vollständiger Überdeckung aller Knoten noch Kanten geben, welche nicht überdeckt wurden, zum Beispiel bei einer `IF`-Verzweigung ohne Code im alternativen Zweig `ELSE`. Deshalb sub-

sumiert die Verzweigungsüberdeckung die Anweisungsüberdeckung streng: $C_{branch\ coverage} \rightarrow C_{statement\ coverage}$.

Als nicht allgemein gültig hat sich die Behauptung in [RW85] erwiesen, wonach das dort definierte Kriterium *all-DU-paths* stets das *all-uses*-Kriterium subsumiert. Wie in Kapitel 3.4.2 angedeutet, kann es bei Schleifen vorkommen, dass diese für jede beliebige Testeingabe stets mindestens einmal ausgeführt werden müssen. Liegt nun eine Definition einer Variablen in einem Knoten n_d vor einer solchen Schleife, eine davon erreichbare Verwendung im Knoten n_u dahinter und führt jeder Teilpfad von n_d zu n_u zwangsweise über die Schleife, dann kann das *all-uses*-Kriterium zwar erfüllt werden, problematisch wird dies jedoch für *all-DU-paths*. Während das erste einen beliebigen Teilpfad von n_d zu n_u fordert, müssen die zu überdeckenden DU-Teilpfade bei *all-DU-paths* schleifenfrei sein. Da die statische Analyse des Graphen solche DU-Teilpfade identifizieren kann, jedoch kein einziger Testfall existiert, welcher einen dieser DU-Teilpfade zur Ausführung bringt, gibt es nur zwei Konsequenzen:

- entweder kann das *all-DU-paths*-Kriterium nie erfüllt werden, weil keine ihr genügende Testfallmenge gefunden werden kann; dies tritt auf, falls man dem Kriterium nur die statisch identifizierbaren DU-Teilpfade zugrunde legt
- oder *all-DU-paths* subsumiert nicht *all-uses*, wenn man anstelle der statisch identifizierbaren DU-Teilpfade lediglich alle ausführbaren DU-Teilpfade fordert; in diesem Fall muss für das eben betrachtete DU-Paar kein Teilpfad nach *all-DU-paths* überdeckt werden, wohl aber nach dem *all-uses*-Kriterium

Lösungsmöglichkeiten zur Behebung dieser Schwäche in der Subsumptionshierarchie der datenflussbasierten Kriterien nach Rapps und Weyuker werden in [HL91] vorgeschlagen. Das somit entstehende Kriterium wurde in Abbildung 3.9 als „*all-DU-paths+*“ bezeichnet. Dabei wird jeder Teilpfad zwischen einer Definition im Knoten n_d und einer davon erreichbaren Verwendung in n_u, welcher eine dazwischenliegende Schleife beliebig oft iteriert, als ein Repräsentant des statisch identifizierten DU-Teilpfades ohne Wiederholung der Schleife angenommen. Durch das Kollabieren der Schleifenwiederholungen zu einem einzigen Schleifenkopfknoten entsteht aus jedem Repräsentanten wieder ein DU-Teilpfad für das betrachtete DU-Paar. Das Kriterium *all-DU-paths+* fordert nun die Überdeckung mindestens eines Repräsentanten dieser Art für jeden DU-Teilpfad.

Betrachtet man das Beispiel aus Abbildung 3.6, so muss nun für die Definition von `mid` in Knoten n_3 und die Verwendung in Knoten n_{12} nicht mehr der statisch identifizierbare, jedoch nicht ausführbare DU-Teilpfad $p_{DU} = (n_1, n_2, n_{11}, n_{12})$ überdeckt werden. Stattdessen genügt auch die Überdeckung eines beliebigen Repräsentanten, wie beispielsweise $p^{+}_{DU} = (n_1, n_2, n_3, n_4, n_5, n_{10}, n_2, n_{11}, n_{12})$, dessen Teilpfad $(n_2, n_3, n_4, n_5, n_{10}, n_2)$ zum Schleifenkopfknoten (n_2) kollabiert, womit der DU-Teilpfad p_{DU} als überdeckt angenommen wird.

Ein ähnliches Problem zeigt sich auch in der Subsumptionsrelation zwischen der in Kapitel 3.4.3 definierten Familie der *required k-tuples*-Kriterien und dem *all-uses*-Kriterium. Zwar wurde bei der Definition von *required k-tuples* angenommen, dass jedem Knoten genau eine Anweisung zugeordnet wird. Dennoch kann eine solche Anweisung eine Verwendung mit anschließender Definition der gleichen Variablen enthalten, wie zum Beispiel beim Post-Inkrement-

Operator in der Anweisung x++. Ist dies die einzige von dieser Definition erreichbare Verwendung, so erfordert keines der *required k-tuples*-Kriterien die Überdeckung dieses DU-Paares, da die von Ntafos definierten *k-dr*-Interaktionen in paarweise verschiedenen Knoten vorkommen müssen. Im Gegensatz dazu muss zur Erfüllung des *all-uses*-Kriteriums mindestens ein definitionsfreier Teilpfad für dieses DU-Paar überdeckt werden. Demnach würden die *required k-tuples*-Kriterien lediglich noch das *all-p-uses*-Kriterium subsumieren. Modifiziert man die *required k-tuples*-Familie so, dass die erste Definition einer *k-dr*-Interaktion im gleichen Knoten vorkommen darf wie die letzte Verwendung dieser Interaktion, erhält man die in Abbildung 3.9 dargestellte Familie der „*required k-tuples+*“-Kriterien [CPRZ89], welche sich wie gezeigt in die Subsumptionshierarchie einfügt.

Wie bei der Definition des *reach coverage*-Kriteriums in Kapitel 3.4.4 angemerkt, erfordert die Familie der Datenkontextüberdeckungskriterien nicht die Ausführung aller Nachfolgeknoten des jeweils betrachteten Verwendungsknotens n_u. Demnach subsumiert keines der Kriterien dieser Klasse die Verzweigungsüberdeckung. Da es Knoten gibt, welche keine Verwendungen von Variablen enthalten, fordert nicht einmal das *ordered context coverage*-Kriterium die Ausführung jedes Knotens und damit jeder Anweisung. Modifiziert man die Kriterienfamilie um die zusätzliche Forderung nach der Überdeckung aller Nachfolgeknoten von n_u, so erhält man die in Abbildung 3.9 eingeordneten Kriterien *reach coverage+*, *context coverage+* sowie *ordered context coverage+* [CPRZ89], wobei das neue Kriterium *reach coverage+* nun sogar äquivalent zu *all-uses* ist.

3.6 Mutationstesten/Mutationsanalyse

Die in dieser Arbeit vorgestellten kontroll- und datenflussorientierten Testverfahren sind typische Repräsentanten der strukturellen Teststrategien (*white-box*) und orientieren sich bei der Definition ihrer Kriterien sehr stark am Fluss der Kontrolle und der Information durch das Programm – basieren daher im Wesentlichen auf einer graphischen Darstellung des (datenflussannotierten) Kontrollflusses. Das im Folgenden skizzierte Mutationstesten ist dazu orthogonal und nimmt insofern eine Zwitterstellung zwischen strukturellen und funktionalen Testverfahren ein, als dass das praktische Vorgehen zwar stark codezentriert ist, die theoretischen Grundgedanken aber eher mit denen des funktionalen Testens verwandt sind.

Ursprünglich wurde das Mutationstesten von Richard A. DeMillo et al. [DLS78] als eigenständiges, fehlerbasierendes Testverfahren definiert und 1978 publiziert. Die zugehörigen Kriterien sollten, ähnlich wie diejenigen aus Kapitel 3.2, Kapitel 3.3 und Kapitel 3.4, zur Auswahl der Testfälle und zur Bestimmung einer entsprechenden Testüberdeckung dienen. Es hat sich jedoch bald gezeigt, dass die dabei eingesetzte Methodik auch zur Definition eines objektiven Maßes der Fehleraufdeckungsfähigkeit und damit der Güte einer beliebig ermittelten Testfallmenge herangezogen werden kann. Im Rahmen dieser Arbeit wird das Mutationstestkonzept hauptsächlich im letzteren Sinne verwendet und zur besseren Unterscheidung als *Mutationsanalyse* bezeichnet. Dabei werden die (z.B. datenflussorientiert) automatisch generierten Testfälle ebenfalls automatisch einer qualitativen Betrachtung unterzogen. Darüber hinaus kann, bei Bedarf nach Verbesserung der Fehleraufdeckung, das Generieren zusätzlicher Testfälle nach der

Strategie des Mutationstestens nachgeschaltet werden.

Das Mutationstesten basiert auf der Annahme, dass ein Programm zunächst als „ausreichend getestet“ angenommen werden kann, wenn alle einfachen (primitiven) Fehler entdeckt und behoben wurden. Primitive Fehler sind dabei solche, die ein durchschnittlicher Programmierer zum Beispiel aus Unachtsamkeit typischerweise macht. Je nach Programmiersprache und -paradigma gibt es eine Vielzahl unterschiedlicher Kataloge solcher Fehler, zu denen beispielsweise das Vertauschen logischer Operatoren ebenso gehört wie das fälschliche Hinzufügen beziehungsweise Weglassen des Schlüsselwortes `super` bei einem Methodenaufruf oder Feldzugriff. Tabelle 3.6 zeigt eine Auswahl repräsentativer Fehlerklassen, von denen diejenigen aus der unteren Hälfte typisch für objekt-orientierte Programme (zum Teil speziell für die Sprache JAVA™) sind [MKO02].

Kürzel	Beschreibung	Beispiel*	
		original	mutiert
ABS	Einfügen der Betragsfunktion	`z = x + y;`	`z = abs(x + y);`
AOR	Ersetzen arithmetischer Operatoren	`z = x / y;`	`z = x - y;`
LCR	Ersetzen logischer Verknüpfungen	`if (x && y)`	`if (x || y)`
ROR	Ersetzen relationaler Operatoren	`if (x < y)`	`if (x <= y)`
UOI	Einfügen unärer Operatoren	`if (done)`	`if (!(done))`
ISK	Entfernen des Schlüsselworts `super`	`a = super.m();`	`a = m();`
IPC	Entfernen des expliziten Superklassenkonstruktoraufrufs	`super();`	`// super();`
PNC	Neuinstantiierung mit Unterklassentyp	`A a = new A();`	`A a = new B();`
PMD	Deklaration von Instanzvariablen mit Superklassentyp	`protected B b = new B();`	`protected A b = new B();`
JTD	Entfernen des Schlüsselworts `this`	`a = this.f;`	`a = f;`
JID	Entfernen der Instanzvariableninitialisierung	`private int a = 4711;`	`private int a;`

* Es sei angenommen, dass Klasse B von Klasse A abgeleitet ist und in beiden Klassen jeweils eine Methode `m()` und eine Instanzvariable `v` deklariert sind.

Tabelle 3.6: Beispiele einiger Mutationsoperatoren für objekt-orientierte Programme

Das Testverfahren selbst wird durch die These des *coupling effect* gestützt, welche durch eine Vielzahl theoretischer und empirischer Studien weitgehend untermauert wird. Ihre Aussage ist, dass komplexere Programmfehler durch Verkettung einfacher entstehen; dabei hängen die komplexen Fehler mit den dazu beitragenden primitiven Fehlern insofern zusammen, als dass eine Testfallmenge, welche alle einfachen Fehler aufzudecken vermag, auch die meisten komplexen Fehler bloßstellen kann [DLS78, OPTZ96].

Um ein Programm oder Modul $\mathcal{P}$ einem Mutationstest zu unterziehen, müssen zunächst unterschiedliche Varianten von $\mathcal{P}$ erstellt werden – die sogenannten *Mutanten*. Einen Mutanten $\overline{\mathcal{P}}$ erhält man typischerweise durch Injektion eines primitiven Fehlers in die ursprüngliche Variante $\mathcal{P}$, wobei grundsätzlich auch mehrere Fehler injiziert werden können. Diese Injektion kann heutzutage automatisiert mittels entsprechender Mutationsoperatoren[10] (beispielhaft in Tabelle 3.6) erfolgen [MKO02, OMK04, MOK05].

Führt man nun $\mathcal{P}$ und einen Mutanten $\overline{\mathcal{P}}$ mit dem gleichen Testfall *t* aus und stellt dabei Unterschiede in der Abarbeitung von *t* zwischen den beiden Varianten fest, dann heißt es, *t* hat den Mutanten $\overline{\mathcal{P}}$ *getötet* (killed). Existiert kein einziger Testfall, bei dessen Ausführung mit $\mathcal{P}$ beziehungsweise $\overline{\mathcal{P}}$ eine solche Abweichung provoziert werden kann, so sind die beiden Vari-

[10] Nicht zu verwechseln mit den Mutationsoperatoren der evolutionären Verfahren, siehe Kapitel 4.5.1.

anten *äquivalent*, das heißt sie erbringen stets die gleiche Funktionalität trotz syntaktischer Unterschiede im Programmcode. Da die äquivalenten Mutanten mit dem ursprünglichen Programm gleichwertig sind, ist es sinnvoll, sie von der weiteren Betrachtung explizit auszuschließen, um das objektive Maß der Fehleraufdeckung nicht zu verfälschen. Problematisch dabei ist jedoch gerade die Erkennung äquivalenter Mutanten – der hier angesetzte Lösungsweg ist in Kapitel 3.6.1 beschrieben.

Betrachtet man bei der Ausführung er beiden Varianten lediglich das nach außen hin beobachtbare Verhalten (zum Beispiel die Ausgaben der Programme) so spricht man von *starkem Mutationstest*. Berücksichtigt man jedoch zusätzlich auch die Zustände (zum Beispiel Variablenbelegungen), die die beiden Varianten bei der Ausführung eines Testfalls einnehmen, so handelt es sich um einen *schwachen Mutationstest* [OL91].

Sei $\mathcal{P}$ ein Programm, $\mathcal{T}$ eine Testfallmenge, $M_t(\mathcal{P})$ die Menge aller aus $\mathcal{P}$ entstandenen Mutanten, $M_e(\mathcal{P}) \subset M_t(\mathcal{P})$ die Menge aller zu $\mathcal{P}$ äquivalenten Mutanten und $M_k(\mathcal{P},\mathcal{T}) \subseteq M_t(\mathcal{P})$ die Menge aller von $\mathcal{T}$ getöteten Mutanten, dann ist der sogenannte *mutation score*

$$MS(\mathcal{P},\mathcal{T}) = \frac{|M_k(\mathcal{P},\mathcal{T})|}{|M_t(\mathcal{P}) \backslash M_e(\mathcal{P})|}$$

ein Maß für die Fehleraufdeckungsquote und damit für die Güte der Testfallmenge $\mathcal{T}$ in Bezug auf Programm $\mathcal{P}$.

Ziel des Mutationstestens ist es, ungeachtet anderer Überdeckungskriterien, jedoch eventuell inkrementell, eine Testfallmenge zu identifizieren, welche einen möglichst hohen mutation score (idealerweise natürlich 100%) erzielt. Im Gegensatz dazu wird die Testfallmenge bei der Mutationsanalyse zunächst nach einem beliebigen Testkriterium erstellt und ihr mutation score anschließend lediglich zur objektiven Bewertung der Fehleraufdeckungsgüte bestimmt. Selbstverständlich können die beiden Ansätze auch kombiniert werden, wie in dieser Arbeit beschrieben.

Sind nach dem Ende der Testphase nicht alle Mutanten, welche zur ursprünglichen Variante $\mathcal{P}$ nicht äquivalent sind, getötet worden, so besteht kein Grund zur Annahme, das Programm $\mathcal{P}$ sei korrekt. Ist $\overline{\mathcal{P}}$ ein solcher nicht-äquivalenter und nicht-getöteter Mutant, so wurde $\mathcal{P}$ mit keinem Testfall getestet, der den potentiellen Fehler in $\overline{\mathcal{P}}$ als solchen entlarvt hätte. Somit ist auch die Annahme berechtigt, $\overline{\mathcal{P}}$ stelle die korrekte Implementierung dar, während $\mathcal{P}$ einen noch unentdeckten Fehler enthält.

3.6.1 Behandlung der Äquivalenz beim Mutationstesten

Wie bereits angedeutet, ist es für die Bestimmung des Fehleraufdeckungspotentials in Form des *mutation score* wichtig, nach der automatischen Generierung von Mutanten alle zum ursprünglichen Testobjekt äquivalenten Varianten auszusortieren. Da bei Anwendung möglichst unterschiedlicher Mutationsoperatoren schon bei kleinen Modulen schnell eine überwältigende Vielzahl von Mutanten entsteht, ist eine manuelle Untersuchung der Äquivalenzeigenschaft unter Berücksichtigung des dazu notwendigen Aufwands kaum zu rechtfertigen. Bedingt durch die Größe des Eingaberaums, kann man in den seltensten Fällen jeden Mutanten und das Original mit allen

möglichen Eingaben ausführen, um auf diesem Wege abzuleiten, welches die funktional äquivalenten Mutanten sind. Versuche, die Äquivalenz mittels statischer Analysatoren nachzuweisen oder zu widerlegen, scheitern meist an der Komplexität des Kontrollflusses heutiger Systeme, insbesondere im Falle von Schleifen mit statisch nicht ermittelbaren, maximalen Wiederholungen.

Wenn demnach kein allgemeingültiges Verfahren existiert, um äquivalente Mutanten mit absoluter Sicherheit zu identifizieren, so besteht dennoch die Möglichkeit, die Eigenschaft der Äquivalenz probabilistisch zu bewerten. Dazu wird im Rahmen dieser Arbeit der Grundgedanke des *statistischen Testens* [Ehr02] übertragen. Ziel dieses Verfahrens ist es, die unbekannte tatsächliche Zuverlässigkeit eines Softwaresystems nach unten abzuschätzen. Dabei führt man in der Testphase zunächst eine Reihe unabhängiger Testfälle aus. Versagte das System bei keinem dieser Tests (beziehungsweise nur bei einem geringen Anteil), so kann daraus eine obere Schranke für die Wahrscheinlichkeit bestimmt werden, mit der das getestete Programm im späteren Betrieb auf eine zufällig gewählte Eingabe fehlerhaft reagiert. Dabei ist zu beachten, dass der Zusammenhang selbstverständlich nur dann gilt, wenn die Eingaben während des Testens mit der gleichen Wahrscheinlichkeit (anhand des sogenannten *Operationsprofils*) unabhängig ausgewählt wurden wie die im späteren Betrieb auftretenden Anforderungen.

Die Überlegungen für das statistische Testen lassen sich auf das Problem der Identifikation äquivalenter Mutanten wie folgt übertragen. Sei erneut $\mathcal{P}$ ein Programm, $\mathcal{T}$ eine Testfallmenge der Größe $n = |\mathcal{T}|$, $M_t(\mathcal{P})$ die Menge aller aus $\mathcal{P}$ entstandenen Mutanten und $M_e(\mathcal{P}) \subseteq M_t(\mathcal{P})$ die (unbekannte) Menge aller zu $\mathcal{P}$ tatsächlich äquivalenten Mutanten. Außerdem sei $M_e^+(\mathcal{P}) \subseteq M_t(\mathcal{P})$ die Menge derjenigen Mutanten, welche während der Testphase nicht getötet wurden; es gilt demnach $M_e^+(\mathcal{P}) \supseteq M_e(\mathcal{P})$. Darüber hinaus sei p die Wahrscheinlichkeit, mit der ein beliebiger Mutant aus $M_e^+(\mathcal{P})$, bei einer zufällig und nach dem gleichen Profil wie in der Testphase ausgewählten Eingabe, ein gegenüber dem Originalprogramm abweichendes Verhalten aufweisen würde. Zwar kann p nicht deterministisch bestimmt werden, jedoch ist es möglich, mittels eines Hypothesentests bei vorgegebener oberen Schranke β für den zugehörigen Fehler 2. Art, eine obere Schranke p^* für p zu ermitteln, so dass $p \leq p^*$ gilt.

Dazu führt man zunächst das ursprüngliche Programm $\mathcal{P}$ und jeden Mutanten aus der zugehörigen Menge $M_t(\mathcal{P})$ mit jedem Testfall aus $\mathcal{T}$ aus. Dabei müssen die Testfälle entsprechend des zu erwartenden zukünftigen Einsatzes zufällig und unabhängig ausgewählt werden[11]. Auf diese Weise identifiziert man die mit Sicherheit nicht äquivalenten Mutanten, womit noch die restliche Menge $M_e^+(\mathcal{P})$ möglicherweise äquivalenter Mutanten näher zu betrachten ist.

Für die Wahrscheinlichkeit $P[\text{äquivalentes Verhalten}|p > p^*]$, dass ein beliebiger Mutant $\overline{\mathcal{P}} \in M_e^+(\mathcal{P})$ unter den gewählten n Testfällen stets ein zum Original äquivalentes Verhalten zeigt, obwohl $p > p^*$ und der Mutant somit nicht äquivalent ist, gilt:

$$P[\text{äquivalentes Verhalten}|p > p^*] = (1-p)^n \leq (1-p^*)^n$$

Da aus einer solchen Beobachtung fälschlich auf die Annahme $p \leq p^*$ geschlossen werden würde (Fehler 2. Art) und dieses Ereignis nur mit einer geringen Wahrscheinlichkeit von

[11] Mangels eines expliziten Operationsprofils, beruhen die experimentellen Daten in Kapitel 6 auf einem uniformen Profil, d.h. jeder Testfall wurde mit der gleichen Wahrscheinlichkeit ausgewählt.

höchstens β auftreten soll, muss $P[\text{äquivalentes Verhalten}|p > p^*] \leq \beta$ gelten. Aufgrund obiger Ungleichungen und der Tatsache, dass hier die kleinste Schranke p^* unter Berücksichtigung von β zu ermitteln ist, folgt somit unmittelbar:

$$(1-p^*)^n = \beta$$

Formt man diese Gleichung entsprechend um, so errechnet sich für die Wahrscheinlichkeit p, mit der bei einem beliebigen, im Test nicht getöteten Mutanten $\overline{\mathcal{P}} \in M_e^+(\mathcal{P})$ im späteren Betrieb ein unterschiedliches Verhalten festgestellt werden würde, eine obere Schranke p^* zu:

$$p^* = 1 - \sqrt[n]{\beta}$$

Kapitel 4

Metaheuristische Such- und Optimierungsverfahren

„The first 90% of the code accounts for the first 90% of the development time.
The remaining 10% of the code accounts for the other 90% of the development time.“
frequently quoted variant of the **ninety-ninety rule** by Tom Cargill, Bell Labs

Einer Studie der University of California in Berkeley zufolge wird das gesamte Wissen der heutigen Menschheit in Schrift, Ton und Video auf circa 12 ExaByte ($\approx 13,8 \cdot 10^{18}$ Byte) geschätzt; „der Anteil der Geschäftsdaten am monatlichen Internet-Verkehr von 9 ExaByte liegt doppelt so hoch wie der der privaten Nutzung“[1]. Betrachtet man im Vergleich dazu eine einfache Methode in der Sprache JAVA™ mit einem einzigen, ganzzahligen Parameter des Datentyps `long`, welche zum Beispiel die Quadratwurzel des übergebenen Wertes berechnet, so existieren für den Eingabeparameter insgesamt $2^{64} \approx 18,4 \cdot 10^{18}$ unterschiedliche Belegungen. Gibt es im Kontrollfluss eine Sonderbehandlung eines bestimmten Wertes (typischerweise der Zahl „0“), so gleicht die automatische Generierung geeigneter Testdaten nach einem kontroll- oder datenflussbasierten Kriterium der mehrfachen Durchsuchung des gesamten Menschheitswissens nach einem speziellen Byte.

Angesichts der stetig zunehmenden Menge an Information die gesammelt, verarbeitet, gespeichert und über Netzwerke übertragen wird, besteht ein Bedarf an effizienten Suchalgorithmen als zentraler Bestandteil nahezu jeder Applikation - von einfachen Internet-Suchmaschinen bis hin zum komplexen Data Mining in nahezu jeder unternehmensweiten Planungs- und Controllingumgebung. Da deshalb der naive Ansatz, die sogenannte *brute force*-Suche, bei der jede mögliche Lösung untersucht und hinsichtlich ihrer Eignung bewertet wird, in den meisten Fällen nicht mehr praktikabel ist, entstand eine Reihe unterschiedlicher Such- und Optimierungsverfahren. Manche dieser Algorithmen, die sogenannten *Heuristiken*, verzichten in Anbetracht der Komplexität und Größe des Eingaberaums zugunsten einer schnellen und effizienten Problemlösung auf den Anspruch, garantiert das beste Endergebnis zu identifizieren. Meist bezeichnet der Begriff Heuristik ein dediziertes Verfahren für eine bestimmte Problemklasse oder eine spezielle Lösungsstrategie. Der Natur haben Forscher allgemeinere Methoden abgeschaut, die durch

[1] Quelle: *Computerwoche* Nr. 23 vom 9. Juni 2006, S. 6

Kombination verschiedener, abstrahierter Vorgehensweisen (meist selbst Heuristiken) eine Lösung für eine breitere Problemklasse bieten. Da sie somit eher als allgemeines Rahmenwerk zu interpretieren und bei Bedarf jeweils anzupassen sind, bezeichnet man diese Verfahren als *Metaheuristiken*. Der Name leitet sich ab vom griechischen Präfix *meta* ($\mu\varepsilon\tau\alpha$, „darüber hinaus“) und dem Wort *Heuristik* ($\varepsilon\upsilon\rho\iota\sigma\kappa\varepsilon\iota\nu$, heuriskein, „finden“).

4.1 Suche und Optimierung

Den Themen Suche und Optimierung wurde (und wird) eine Vielzahl Forschungsarbeiten gewidmet, die mitunter stark unterschiedliche Herangehensweisen hervorgebracht haben. Einig ist man sich jedoch darin, dass Such- und Optimierungsprobleme äquivalent und somit ineinander überführbar sind. Anschaulich erklärt, stellt eine *Optimierung* lediglich die „Suche“ nach einer Lösung dar, die den besten Nutzen entsprechend einer vorgegebenen Bewertungsfunktion einbringt. Umgekehrt kann für jedes *Suchproblem* eine Zielfunktion definiert werden, so dass die ursprünglich gesuchte Lösung den höchsten Wert im Sinne dieser Zielfunktion hat, somit also das Ergebnis einer „Optimierung“ darstellt. Aus diesen Betrachtungen kann man schließen, dass beide Fragestellungen auch mit den gleichen Ansätzen gelöst werden können.

Ein umfassender Vergleich entsprechender Verfahren findet sich bei [Sha97]. Der Beitrag definiert eine „Optimierungsaufgabe“ durch ihre drei Basiskomponenten:

- eine Menge Variablen, die alle möglichen Lösungen eines Problems, welche den sogenannten *Suchraum* bilden, eindeutig und vollständig charakterisieren und deren Belegungen potentielle Lösungen des Problems darstellen;
- (mindestens) eine objektive *Bewertungsfunktion* (auch *Zielfunktion* oder *Fitnessfunktion* genannt), die jeder möglichen Lösung der Optimierungsaufgabe einen Wert zuordnet, der die Eignung (den Nutzen) dieser Lösung im Sinne der Aufgabe widerspiegelt;
- ein Satz Bedingungen (sogenannte *constraints*), die den Wertebereich der Variablen auf sinnvolle Lösungen der Aufgabe einschränken.

Demnach besteht die Optimierung darin, *diejenige(n) Belegung(en) für die Variablen zu finden, so dass die Auswertung der Funktion(en) mit diesen Eingangsvariablen jeweils einen Extremwert (Minimum oder Maximum) aufweist und die gleichzeitig alle Bedingungen erfüllen*. Man beachte, dass diese Extrema sowohl lokal als auch global sein können. Je nach Aufgabenstellung genügt manchmal die Lokalisierung eines globalen Extremwerts oder aber es sind alle möglichen Extrema von Interesse. Grundsätzlich kann die Aufgabe der Lokalisierung eines Minimums einer Bewertungsfunktion durch ein Verfahren zur Lokalisierung eines Maximums gelöst werden: $min_{f(x)} \equiv -max_{-f(x)}$. Daher werden die Algorithmen in diesem Kapitel o.B.d.A. mit dem Ziel der Maximierung der Bewertungsfunktion vorgestellt.

Da es sehr unterschiedliche Arten von Optimierungaufgaben gibt, müssen Metaheuristiken bis zu einem gewissen Grad an das jeweils vorliegende Problem angepasst werden. Eine detaillierte Klassifikation dieser Fragestellungen findet sich bei [Sha97]. Demnach können die Wertebereiche der Eingangsvariablen kontinuierlich (zum Beispiel $\mathbb{R}$) oder diskret (beispielsweise

$\mathbb{Z}$ oder Aufzählungstypen in entsprechenden Programmiersprachen) sein. Die zu optimierende Bewertungsfunktion kann selbst diskret oder kontinuierlich definiert sein und einen stetigen oder nicht-stetigen Verlauf aufweisen. Die Bedingungen können ebenfalls in Abhängigkeit von den Variablen stetig oder nicht-stetig sowie implizit oder explizit definiert sein. Grundsätzlich unterscheidet man dabei zwei Arten von Constraints:

- sogenannte *harte Bedingungen* dürfen nicht verletzt werden: Belegungen der Variablen, die harte Constraints nicht erfüllen, stellen keine gültigen Lösungen dar;

- *weiche Bedingungen* drücken lediglich einen Wunsch aus, und daher sollten sie nicht, aber können von den entsprechenden Variablenbelegungen verletzt werden.

Eine Zielfunktion heißt *unimodal*, falls sie genau ein lokales Extremum aufweist, das zugleich auch global ist. Hat die Funktion jedoch mehrere lokale oder globale Extrema, so bezeichnet man sie als *multimodal*. Manche Optimierungsprobleme werden nicht nur durch eine einzige Bewertungsfunktion beschrieben, sondern mittels einer ganzen Funktionsschar definiert, wobei die einzelnen Funktionen teilweise maximiert und teilweise minimiert werden sollen. Man spricht in diesem Fall von *multi-objektiver* Optimierung.

Die Generierung eines optimalen Testdatensatzes für ein Programm kann als Suche nach einer Menge einzelner Testfälle (entspricht der Belegung der Variablen) definiert werden, deren Anzahl möglichst klein sein soll, die von ihnen erzielte Überdeckung jedoch möglichst groß. Der Suchraum dieses Optimierungsproblems ist somit die Menge aller gültigen Testdatensätze (eingeschränkt durch entsprechende Constraints), während die Bewertung durch zwei getrennte Funktionen erfolgt, von denen eine minimiert und die andere gleichzeitig maximiert werden soll.

Eine detaillierte Beschreibung der verschiedenen Optimierungsarten ist nicht Gegenstand dieser Arbeit, jedoch sei hier ein Hinweis auf mögliche Probleme gegeben, die bei den meisten nichtstetigen Optimierungsaufgaben auftreten [Sha97] und diese deshalb selbst für eine automatische Verarbeitung zu einem aufwändigen Unterfangen werden lassen:

- Die gültigen Definitionsbereiche, die von nichtstetigen beziehungsweise nichtlinearen Nebenbedingungen begrenzt sind, können oft nur mit großem Aufwand im Vorhinein identifiziert werden. Das Sicherstellen, dass die Variablen nur diejenigen Belegungen annehmen, die alle Bedingungen erfüllen, wird meist als Teil der Optimierungsaufgabe selbst interpretiert oder erfordert komplexe Generatoren und Konsistenzprüfungen.

- Die von der Zielfunktion gebildete Suchraumbewertung kann sehr schroff sein und viele lokale Extrema aufweisen, wodurch benachbarte Punkte des Suchraums eine signifikant unterschiedliche Bewertung erhalten können. Sie kann jedoch auch weite und sehr flache Plateaus aufweisen, womit mehr oder weniger lose benachbarte Punkte im Suchraum (nahezu) gleiche Bewertungen erhalten würden.

- Die Dimension der Optimierungsaufgaben, also die Anzahl und Komplexität der Eingangsvariablen, ist in den meisten Anwendungen sehr groß, wodurch die Auswertung der Zielfunktion und der Bedingungen sehr viel Rechenzeit erfordern kann.

Das größte Problem besteht darin, lokale und globale Extrema zu unterscheiden. Viele der einfacheren Suchheuristiken werden von lokalen Extrema angezogen, das heißt die Verfahren tendieren dazu, schnell zu einem lokalen Extremum zu konvergieren. Sobald sie dieses lokalisiert haben, sind sie nicht mehr in der Lage sich davon zu lösen und nach einem eventuell vorhandenen globalen Extremwert zu suchen. Man nennt dieses Vorgehen (das man auch bewusst innerhalb einer Metaheuristik einsetzen kann) im allgemeinen Sprachgebrauch *lokale Optimierung*. Als Teil einer Metaheuristik werden diese Verfahren typischerweise dazu eingesetzt, ausgehend von einer bereits für sehr gut befundene Lösung innerhalb ihrer unmittelbaren Umgebung nach der besten zu suchen. Solche Algorithmen sind meist gradientenbasiert, sprich sie suchen ausgehend von einem beliebigen Startpunkt in Richtung des nächstgelegenen Extremwertes. Wenn die Funktion unimodal (quasi-konvex) ist, dann finden solche Methoden stets auch das globale Maximum oder Minimum. Da die genaue Bewertung einer Extremwertlösung meist nicht im Vorhinein bekannt ist, man denke zum Beispiel an die Anzahl notwendiger Testfälle für eine vorgegebene Überdeckung, und außerdem die mögliche Zahl unterschiedlicher Auswertungsversuche (sprich Optimierungsläufe) stets endlich und aufgrund der verfügbaren Ressourcen sehr klein gegenüber der Suchraumgröße ist, müssen gewinnversprechende Algorithmen sehr geschickt implementiert werden und vielfältige Mechanismen aufweisen, um auftretende Probleme dieser Art schnell und sicher überwinden zu können. Man spricht bei Techniken dieser Art, zu denen auch evolutionäre Verfahren gehören, von *globaler Optimierung*, da sie konzeptionell auf die Suche nach globalen Extrema ausgerichtet sind.

Der naive Ansatz, alle möglichen Lösungen aufzuzählen und einzeln hinsichtlich ihrer Eignung als beste Lösung für ein Optimierungsproblem zu untersuchen, typischerweise als *brute-force*-Ansatz bezeichnet, verbietet sich in den meisten Fällen aufgrund der Größe des Suchraumes – selbst in Zeiten exponentiell wachsender Rechenressourcen. Der unschlagbare Vorteil dieses Verfahrens ist jedoch, dass die Identifikation der absolut perfekten Lösung stets garantiert wird. Schränkt man die zur Verfügung stehenden Rechenressourcen (problematisch ist meist der Faktor Zeit) ein, bedeutet das fast unweigerlich die Aufgabe einer solchen Garantie zugunsten einer schnellen Lösung. Ist der Suchraum jedoch von unterschiedlich gut geeigneten Lösungen durchsetzt, so kann bereits die Ermittlung einer möglicherweise suboptimalen Lösung ein Fortschritt bedeuten. Metaheuristiken bieten einen Kompromiss indem sie Arbeitsressourcen gegen Ergebnisqualität eintauschen, wobei die beiden Eigenschaften bei fast allen Verfahren im Allgemeinen nahezu korrelieren.

Da die Vielzahl der verschiedenen existierenden Such- und Optimierungsverfahren nicht nur den Umfang dieser Arbeit sprengen, sondern auch ihren Fokus verlassen würde, werden an dieser Stelle lediglich die wichtigsten Grundalgorithmen skizziert, die im Rahmen dieses Forschungsprojektes in ein Werkzeug zur Testdatenoptimierung umgesetzt und verglichen wurden. Zum Studium weiterer Verfahren sei hier auf die umfangreiche Literatur[2] zu diesem Thema verwiesen. Zumindest dem Namen nach erwähnenswert sind hier insbesondere: *Branch and Bound*, *Greedy Randomized Adaptive Search Procedure (GRASP)*, *Tabu Search* und *Ant Colony Optimization*.

[2]Eine gute und dynamisch aktualisierte Übersicht zum Einstieg in dieses Themengebiet bietet die verbreitete Online-Enzyklopädie *Wikipedia* unter `http://en.wikipedia.org/wiki/Metaheuristic`.

Abkürzungen und Begriffe: Um im Rest dieser Arbeit eine einheitliche und somit leichter nachvollziehbare Terminologie zu verwenden, seien noch folgende Bezeichnungen und Abkürzungen eingeführt:

- $\mathcal{S}$: der Suchraum des Optimierungsproblems, also die Menge aller Elemente, die eine potentielle Lösung des Optimierungsproblems darstellen
- f beziehungsweise $f_1, f_2, \ldots, f_n$: Bewertungsfunktionen (auch Fitnessfunktionen), die jeder potentiellen Lösung eine Güte zuordnet und damit eine quantitative Aussage über die Qualität einer Lösung im Hinblick auf das zu optimierende Problem darstellt (typisch: $f, f_i : \mathcal{S} \mapsto \mathbb{R}$)
- $\mathcal{N}(s)$ (für $s \in \mathcal{S}$): die Menge aller Elemente aus der Umgebung (Nachbarschaft) von s, also potentielle Lösungen einer Optimierungsaufgabe, welche nur geringfügig von s abweichen

Da die Verfahren in Kapitel 4.2, Kapitel 4.3 und Kapitel 4.4 vorwiegend für nicht-multi-objektive Problemstellungen geeignet sind, wird zunächst angenommen, dass die Güte einer potentiellen Lösung durch eine einzige Bewertungsfunkion „gemessen“ wird. Ansätze zur Aufhebung dieser Beschränkung, beziehungsweise zur Reduktion von multi-objektiven Aufgaben auf nicht-multi-objektive werden im Kapitel 4.5.4 vorgestellt.

4.2 Random Search

Der einfachste „metaheuristische“ Ansatz ist die zufällige Suche oder *Random Search*. Abbildung 4.1 zeigt das Struktogramm dieses Verfahrens. Zunächst wird eine beliebige potentielle Lösung aus dem Suchraum zufällig ausgewählt und ihre Qualität als Lösung des Optimierungsproblems mittels der Bewertungsfunktion berechnet. Die sich anschließenden Schritte werden solange wiederholt, bis ein vorgegebenes Abbruchkriterium erfüllt wird; typischerweise also bis entweder die bestmögliche Lösung gefunden wurde, sofern dies überhaupt ermittelt werden kann, oder eine vorgegebene maximale Anzahl Iterationen erreicht beziehungsweise die Rechenzeit erschöpft ist. Innerhalb dieser eigentlichen Optimierungsschleife wird jeweils eine neue Lösung ebenfalls zufällig aus dem Suchraum gewählt und deren Güte mit der der besten bisher gefundenen Lösung verglichen. Falls die neue Lösung besser ist als die bisherige, so wird diese zunächst als bestes bisheriges Ergebnis angenommen, ansonsten wird sie verworfen und die alte Lösung zunächst beibehalten.

Der besondere Vorteil dieses Suchverfahrens liegt in seiner Einfachheit. Da zu jedem Zeitpunkt jeweils nur maximal zwei Lösungen aus dem Suchraum betrachtet werden, ist der Overhead in Bezug auf Rechen- und Speicherressourcen am geringsten im Vergleich zu allen anderen Metaheuristiken. Darüber hinaus ist das Verfahren prinzipiell in der Lage, globale Extrema zu identifizieren und kann nicht direkt von lokalen Maxima oder Minima angezogen und somit irregeführt werden. Die Einfachheit dieses Verfahrens führt jedoch unweigerlich auch zur größten Schwäche. Dadurch dass jeweils nur ein Punkt im Suchraum berücksichtigt wird, welcher außerdem rein zufällig gewählt wird, ist in jedem Durchgang die Wahrscheinlichkeit relativ gering, die

Random Search — Zufällige Suche

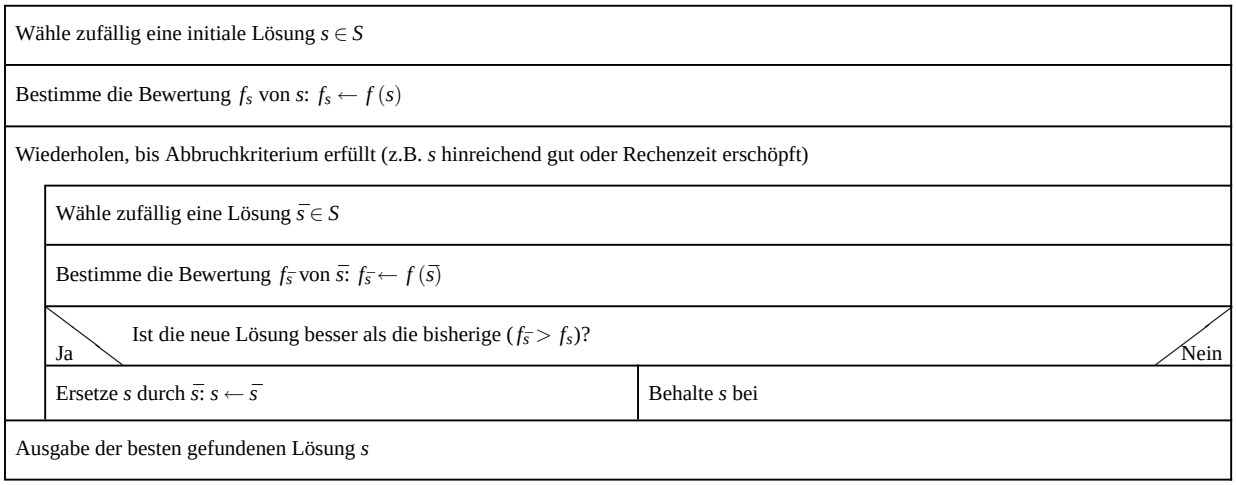


Abbildung 4.1: Struktogramm des Random Search Algorithmus

tatsächlich optimale Lösung zu identifizieren. Im Vergleich zu „intelligenteren“ Suchheuristiken erfordert dieses Verfahren daher erheblich mehr Zeit, um mit der gleichen Wahrscheinlichkeit ein ähnlich gutes Endergebnis zu erzielen. Trotz oder gerade wegen seiner Vor- und Nachteile wird dieser Algorithmus gerne als Vergleichsverfahren herangezogen, um die Effizienz anderer Ansätze zu verdeutlichen.

4.3 Hillclimbing

Der sogenannte *Hillclimbing*-Algorithmus geht eine der größten Schwächen des Random Search an: Die Wahrscheinlichkeit, während einer begrenzten Ausführungszeit die optimale Lösung zufällig auszuwählen, ist sehr gering. Selbst wenn Random Search womöglich eine relativ gute Lösung nahe der optimalen identifiziert hat, gibt es in diesem Verfahren keinen Mechanismus, der die Suche von einem solch suboptimalen Punkt zum nahegelegenen Optimum führt.

Das Ziel des Hillclimbing-Algorithmus ist die iterative Verbesserung einer zunächst zufällig gewählten, initialen Lösung aus dem Suchraum. Das Grundverfahren ist dem des Random Search aus Abbildung 4.1 ähnlich, jedoch mit der Ausnahme, dass die neue Lösung $\bar{s}$ nicht zufällig aus dem gesamten Suchraum ausgewählt wird, sondern ein Element aus der Nachbarschaft $\mathcal{N}(s)$ der bisherigen Lösung s ist. Die zwei Hauptvarianten des Hillclimbings sind in Abbildung A.7 und Abbildung A.8 dargestellt.

Bei dem sogenannten *„first ascent“*-Ansatz aus Abbildung A.7 wird in jeder Iteration eine einzige neue Lösung $\bar{s}$ zufällig aus der Nachbarschaft $\mathcal{N}(s)$ der bisherigen Lösung s ausgewählt und deren Bewertungen verglichen. Im Gegensatz dazu vergleicht die *„steepest ascent“*-Variante aus Abbildung A.8 zunächst die Bewertungen aller Nachbarn von s und wählt anschließend die-

jenige Lösung mit der höchsten Qualität.

Der Vorteil des Hillclimbing-Verfahrens ist seine Einfachheit. Darüber hinaus liefert es zügiger gute Ergebnisse als Random Search, da die Suche nach dem (lokalen) Optimum insbesondere bei „steepest ascent“ erheblich zielgerichteter ist. Im Falle unimodaler Suchräume kann dieser Ansatz garantiert die optimale Lösung schnell auffinden. Diese zielstrebige Orientierung hin zum nächstgelegenen Optimum ist zugleich auch der größte Nachteil dieses Ansatzes, da bei multimodalen Suchräume die Gefahr groß ist, dass das Verfahren lediglich zu einem lokalen Optimum konvergiert, welches sich in der Nähe des zufällig gewählten Startpunktes befindet. Auch problematisch für einen Hillclimbing-Algorithmus sind Suchräume, welche sogenannte Plateaus enthalten, also Bereiche, in denen alle benachbarten Elemente gleiche Bewertungen erhalten – in diesem Falle kann der Algorithmus keinen besten Nachbarn identifizieren und auf diesen zustreben.

Eine verbreitete Erweiterung der vorgestellten Hillclimbing-Varianten besteht darin, den gesamten Algorithmus mehrfach mit jeweils unterschiedlichen initialen Startlösungen auszuführen. Auf diese Weise versucht man dem Problem zu begegnen, dass das Verfahren vorwiegend lokale Optima identifiziert.

4.4 Simulated Annealing

Alternativ zum mehrfachen Neustart des Hillclimbing-Verfahrens kann man die unbedingte Konvergenz zum nächstgelegenen, eventuell lokalen Optimum dadurch etwas abmildern, dass man nicht in jedem Fall die beste Lösung aus der näheren Umgebung der aktuell betrachteten übernimmt, sondern stattdessen auch einen Sprung zu einer weniger guten Lösung zulässt. Diese Strategie findet man in der Literatur unter der Bezeichnung *Simulated Annealing* als einfache aber dennoch sehr leistungsfähige Metaheuristik.

Pate für Simulated Annealing stand ein Verfahren aus der Festkörperphysik, speziell der Kristallographie beziehungsweise Kristallzüchtung, wie es zum Beispiel zur Gewinnung von Silizium für Wafer in hochreiner Form eingesetzt wird. Das Problem bei der Waferherstellung ist nicht nur die Reinheit des Materials, sondern insbesondere das Silizium in eine geordnete Kristallstruktur zu bringen. Jede Fehlstellung im Kristall (zum Beispiel wenn Siliziumatome nicht an den dafür vorgesehenen Plätzen liegen oder Versetzungen in den Kristallschichten) verändert die elektrischen und magnetischen Eigenschaften des Halbleiters, was bei den heutigen Nanostrukturen in der Prozessorherstellung drastische Auswirkungen hat.

Um einen optimalen Kristall zu „züchten“, wie es in der Fachsprache heißt, bearbeitet man den Ausgangsstoff in zwei Stufen. In der ersten Stufe wird ein sogenannter Keim (ein kleiner, bereits erkalteter Siliziumkristall) in eine Siliziumschmelze getaucht und langsam aber kontinuierlich wieder herausgezogen. Dadurch lagert sich Schicht für Schicht neues Silizium an den nach und nach erkaltenden Kristall, der dann in etwa die Form eines Zylinders hat. Durch das relativ schnelle Erkalten wird den Atomen jedoch die kinetische Energie so schnell entzogen, dass sie oft nicht mehr in der Lage sind, eventuelle Fehlstellen aus eigener Kraft zu besetzen beziehungsweise zu korrigieren. Deshalb besteht die zweite Stufe im sogenannten Tempern – daher der Begriff *annealing*. Dabei wird der Siliziumzylinder schichtweise wieder bis knapp unterhalb

der Schmelztemperatur erhitzt und dann weitaus langsamer wieder abgekühlt.

Bereits 1953 hat man versucht, diesen Prozess virtuell im Computer zu simulieren. Die zugrundeliegende Modellannahme eines atomaren Systems im thermischen Gleichgewicht ist, dass Veränderungen in der Anordnung von Atomen immer dann akzeptiert werden, wenn diese die Gesamtenergie des Systems verringern. Mit einer Wahrscheinlichkeit von

$$P(\Delta E) = e^{-\frac{\Delta E}{k_B \cdot T}}$$

kommt es jedoch auch dann zu einer Neuordnung, wenn die innere Energie dabei zunimmt, wobei ΔE der Energieunterschied zwischen der Gesamtenergie vor und nach einer Umordnung im Kristallgitter, k_B die Boltzmannkonstante und T schließlich die Temperatur ist.

Simulated Annealing — Simulierte Abkühlung

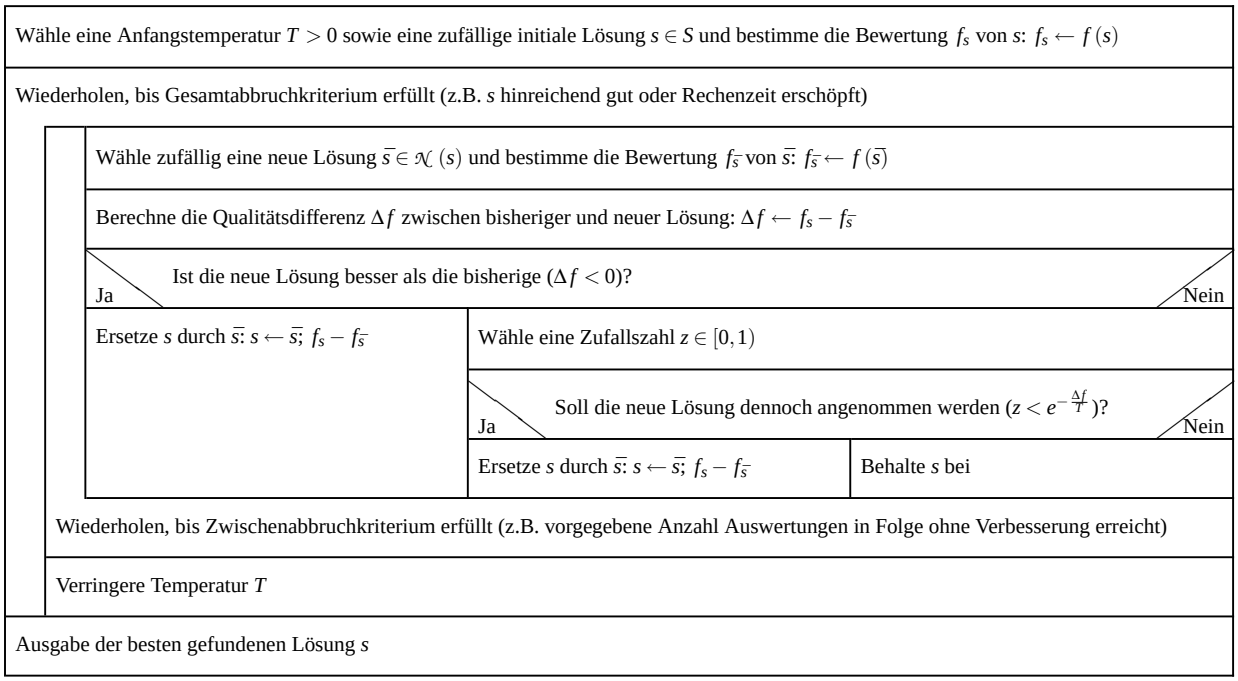


Abbildung 4.2: Struktogramm des Simulated Annealing Algorithmus

Im Jahre 1983 wurde dieses Vorgehen unter dem Namen *Simulated Annealing* als Verfahren zur kombinatorischen Optimierung beschrieben [CDM+96]. Eine allgemeine Darstellung des Grundprinzips zeigt Abbildung 4.2. Im Wesentlichen ist es vergleichbar mit der „first ascent"-Variante des Hillclimbing-Verfahrens aus Abbildung A.7, mit dem Unterschied, dass eine neue Lösung aus der Nachbarschaft der bisher besten auch dann akzeptiert wird, wenn sie im Sinne der Bewertungsfunktion schlechter ist. Eine Verschlechterung tritt jedoch nur mit der Wahrscheinlichkeit $P(\Delta f) = e^{-\frac{\Delta f}{T}}$ auf, wobei Δf die Differenz der Bewertungen der bisherigen und der

neuen Lösung ist. Der Kontrollparameter T (in Anlehnung an den Ursprung des Verfahrens auch „Temperatur“ genannt) wird nach und nach verringert, um somit auch die Wahrscheinlichkeit zu verkleinern, dass eine schlechtere Lösung eine gute verdrängt. Die gegenüber Hillclimbing hinzugekommene innere Schleife sorgt dafür, dass das System sich bei konstanter Temperatur auf ein vorläufiges „Gleichgewicht einpendelt“. Wann dies genau erreicht ist, lässt sich im Allgemeinen jedoch nur schwer festlegen, daher wählt man als Abbruchkriterium für diese innere Schleife eine maximale Anzahl an Versuchen, ein besseres Element aus der Nachbarschaft der aktuellen Lösung zu finden. Für die Reduktion der Temperatur, der sogenannten *Abkühlstrategie (cooling strategy)*, gibt es keine allgemeingültige Vorgabe, jedoch hat sich für T die Wahl einer geometrischen Reihe $T_1, T_2, \ldots, T_n$ bewährt, so dass $T_{i+1} = \alpha \cdot T_i$ mit $\alpha \in (0,1)$ gilt.

4.5 Evolutionäre Verfahren

Ein Großteil der Metaheuristiken zur Lösung von Such- und Optimierungsaufgaben haben ihren Ursprung in der Natur(wissenschaft). Während „Simulated Annealing“ (Kapitel 4.4) auf einem physikalischen Phänomen beruht, sind die meisten Metaheuristiken eher von der Biologie inspiriert, allen voran die hier näher betrachtete Klasse der *Evolutionären Verfahren* oder der „Ameisenalgorithmus“ (Ant Colony Optimization). Der zunächst augenscheinlichste Unterschied zwischen den in Kapitel 4.2, Kapitel 4.3 und Kapitel 4.4 vorgestellten Algorithmen und den Evolutionären Verfahren besteht darin, dass erstere zu jedem Zeitpunkt höchstens zwei potentielle Lösungen betrachten und gegeneinander vergleichen, während letztere einen größeren Teil des Suchraums gleichzeitig untersuchen.

Charles Darwin (1809-1882) widmete sein Lebenswerk der Erforschung der terrestrischen Evolution und veröffentlichte 1859 in *„The Origin of Species by Means of Natural Selection“* die Theorie, dass die gesamte Evolution durch die natürliche Auslese angetrieben wurde und noch immer wird. Demnach gibt es in jeder Population (ob Mensch, Tier oder Pflanze) Unterschiede zwischen den Individuen. Diejenigen, die mit einem herausragenden Merkmal geboren werden, zum Beispiel besonderes Sehvermögen oder gute Tarnung, haben einen Vorteil gegenüber den anderen ihrer Art. Insbesondere wenn sich die Umgebung ändert, kann es vorkommen, dass neue Eigenschaften vorteilhafter sind als alte, zum Beispiel eine neue Farbe, die eine bessere Tarnung verspricht. In ihrer Umwelt können sich Individuen mit diesen neuen Zügen besser behaupten (*„survival of the fittest“*) und zeugen daher mit hoher Wahrscheinlichkeit auch mehr Nachkommen. Wenn diese Individuen ihre besonderen Züge an ihre Nachkommen weitergeben, werden diese auch wiederum davon profitieren, insbesondere wenn die Nachkommen mittels Kreuzung mehrere herausragende Merkmale zugleich von den Eltern erben. Weniger gut angepasste Individuen bleiben auf der Strecke sobald sie mit den anderen um die beschränkten Ressourcen ihres Lebensraums konkurrieren müssen.

Offensichtlich handelt es sich bei diesem natürlichen Vorgang um eine „Optimierung“ einer bestimmten Spezies. Beim Versuch, dieses Verhalten der Natur virtuell nachzubilden, konnte man aus der Simulation nicht nur Erkenntnisse für die Evolutionsbiologie, sondern auch neue Metaheuristiken für die Informatik gewinnen. Als Väter dieser Klasse der sogenannten *Evolutionären Verfahren* sind Ingo Rechenberg und John Holland zu nennen. Rechenberg präsentierte

im Jahre 1965 den Grundstein seiner *Evolutionären Strategien* [Rec73]. Holland entwickelte ab 1962 mit seinen Kollegen und Studenten die *Genetischen Algorithmen* und veröffentlichte sie 1975 in seinem bahnbrechenden Buch [Hol75]. Im Jahre 1992 und damit rund zwanzig Jahre später, benutzte John Koza die Genetischen Algorithmen zur Züchtung von Computerprogrammen in der Sprache LISP, was seinem Verfahren den Namen *Genetische Programmierung* einbrachte.

Bevor in den folgenden Kapiteln auf relevante Aspekte Evolutionärer Verfahren im Allgemeinen eingegangen wird, seien hier noch wichtige Begriffe eingeführt, die hauptsächlich aus der Biologie in die Sprache der Metaheuristiken übernommen wurden. Die Gesamtheit aller Lebewesen unterteilt sich in einzelne, meist voneinander isoliert lebende Populationen. Eine *Population* P besteht aus einer bestimmten Anzahl *Individuen*: $P = \{a_1, a_2, \ldots, a_n\}$.

Jedes Individuum ist aus mehreren, einzelnen Zellen aufgebaut. In jeder Zelle befindet sich ein Satz *Chromosomen*[3] welche jeweils als Blaupause für den gesamten Organismus dienen und daher in jeder Zelle eines Organismus gleich[4] sind – daher wird bei den Evolutionären Verfahren das Individuum mit seinem Chromosomensatz identifiziert und nicht mehr zwischen einzelnen Chromosomen unterschieden.

Ein Chromosom besteht aus *Genen*, ist also eine Aneinanderreihung von DNA-Blöcken. Grundsätzlich verschlüsselt jedes Gen ein Protein und beeinflusst damit indirekt eine Eigenschaft des Individuums, zum Beispiel die Haarfarbe.

Verschiedene Ausprägungen (im Beispiel braun oder blond) dieses Gens bezeichnet man als *Allele*. Jedes Gen hat eine bestimmte Position im Chromosomensatz und man nennt diese *Lokus*. Das gesamte genetische Material einer Population, also alle existierenden Allele, werden als *Genom* bezeichnet. Eine bestimmte Kombination von Allelen, sprich eine „Chromosominstanz“, wird wie in der Begriffswelt der Genetik *Genotyp* genannt. Dieser Genotyp trägt die gesamte nötige Information, die zusammen mit anderen Umweltfaktoren für den tatsächlichen Aufbau eines Organismus, den sogenannten *Phänotyp*, benötigt wird.

4.5.1 Genetische Algorithmen

Der von Holland entwickelte Genetische Algorithmus basiert auf den genetischen Prozessen biologischer Systeme. Die Grundidee besteht darin, möglichst gute Lösungen für eine breite Klasse unterschiedlicher Optimierungsaufgaben der Realität nach den Prinzipien der natürlichen Auslese „heranzuzüchten“. Genetische Algorithmen (im Weiteren auch kurz *GA* genannt) haben in den letzten Jahren eine rasante Verbreitung in den unterschiedlichsten Domänen gefunden. Sie wurden beispielsweise erfolgreich zur Erstellung optimaler Einsatz- oder Stundenpläne für Schulen [WGO02] sowie zur Optimierung der Lastverteilung in Multiprozessorsystemen eingesetzt. Um solche Verfahren zur Lösung eines beliebigen Optimierungsproblems zu übertragen, muss für den Suchraum eine geeignete Codierung definiert werden, damit jede potentielle Lösung des Problems wie ein Chromosom in der Natur zusammengestellt und behandelt werden kann.

Genetische Algorithmen arbeiten stets mit einer ganzen Population von Individuen, wobei jedes Individuum eine mögliche Lösung des Problems darstellt [Bäc91]. Mittels der Bewertungs-

[3] Bei einem gesunden Menschen sind es im diploiden Zustand 2×23.

[4] Eine Ausnahme bilden vereinzelte Mutationen einer Zelle oder eines begrenzten Zellbereichs (Krebs).

funktionen aus Kapitel 4.1 wird jedem Individuum eine *Fitness* zugeordnet. In der Natur entspricht dies der Bewertung eines Individuums hinsichtlich seiner Anpassungsfähigkeit und damit der Fähigkeiten zu überleben und sich fortzupflanzen. Analog zur Natur gesteht man den Individuen die Möglichkeit zur Reproduktion mittels Kreuzung mit anderen Individuen zu, wodurch neue Individuen entstehen, die jeweils anteilig Eigenschaften beider Eltern in sich bergen. Da die Wahrscheinlichkeit der Reproduktion proportional zur Fitness eines Individuums ist, erzeugt der Genetische Algorithmus bevorzugt immer höherwertigere Nachkommen, während die schlechteren Lösungen seltener zur Zeugung zugelassen werden und somit nach und nach aussterben. Damit die Suche nach einer optimalen Lösung nicht auf das Kreuzprodukt der Wertebereiche aus dem Schritt „Initialisierung“ beschränkt bleibt, werden *Auswahl (Selection)* und *Kreuzung (Crossover)* um einen dritten, sogenannten *genetischen Operator* ergänzt: *Mutation*. Das Flussdiagramm in Abbildung 4.3 zeigt den schematischen Ablauf eines Genetischen Algorithmus in seiner Grundform.

Evolutionäre Verfahren — Grundalgorithmus

Initialisierung: Eine Anfangspopulation $P_{aktuell}$ mit zufälliger Initialisierung erzeugen.

Bewertung: Die Fitness jedes Individuums dieser Population $P_{aktuell}$ berechnen.

Wiederholen, bis das Abbruchkriterium erfüllt ist.
(z.B. bis bestes Individuum der aktuellen Population hinreichend gut oder vorgegebene maximale Rechenzeit abgelaufen ist)

- Eine neue, zunächst leere Population P_{neu} anlegen.
  - *Auswahl:* Zwei Individuen zufällig aus der alten Population $P_{aktuell}$ auswählen. Die Wahrscheinlichkeit, dass ein bestimmtes Individuum ausgewählt wird, ist dabei stets proportional zu seiner Fitness.
    Zu den unterschiedlichen Selektionsstrategien siehe Seite 94.
  - *Kreuzung:* Die ausgewählten Individuen so kreuzen, dass ein (beziehungsweise zwei) Nachkommen entstehen.
    Zu den verschiedenen *Crossover*-Strategien siehe Seite 100.
  - *Mutation:* Die vorangehend erzeugten Nachkommen nach einem zufälligen Muster jeweils leicht modifizieren (siehe Seite 102).
  - Diese mutierten Nachkommen in die neue Population P_{neu} einfügen.
- Wiederholen, bis neue Population vollständig.
- *Bewertung:* Die Fitness jedes Individuums der neuen Population P_{neu} berechnen.
- Die alte Population durch die neue ersetzen: $P_{aktuell} \leftarrow P_{neu}$

Das beste Individuum der aktuellen Population ausgeben.

Abbildung 4.3: Struktogramm des Genetischen Algorithmus (Grundform)

Die Stärke der Genetischen Algorithmen liegt in ihrer Robustheit [BBM93a]. Damit ist ihre

Fähigkeit gemeint, unterschiedlichste Optimierungsprobleme zu lösen, und das selbst wenn der Suchraum für klassische Algorithmen zu komplex erscheint, zum Beispiel weil er multimodal ist oder keine dedizierten Verfahren existieren. Leider erhält der Anwender eines Genetischen Algorithmus keine Garantie, dass ein globales Optimum seines Problems nach der Konvergenz des GAs auch tatsächlich identifiziert wurde – im Allgemeinen ist er jedoch in der Lage, *brauchbare Lösungen* innerhalb *akzeptabler Zeit* zu finden. Selbstverständlich können die Genetischen Algorithmen nicht immer mit spezialisierten Such- und Optimierungsalgorithmen Schritt halten – aber dort, wo es solche Möglichkeiten nicht gibt, sind sie nach heutigem Stand der Technik kaum mehr wegzudenken. Auch haben Forschungsprojekte gezeigt [WGO02], dass die Hybridisierung Genetischer Algorithmen mit dedizierten Spezialverfahren oder ihre Erweiterung um problemspezifisches Wissen zu einem deutlichen Performance-Gewinn führen kann, sowohl was die Qualität der Lösung angeht, als auch die Zeit bis zum Konvergieren zu einer idealen Lösung.

Die Schwierigkeit bei der Umsetzung und Nutzung Evolutionärer Verfahren zur Lösung von Such- und Optimierungsaufgaben liegt darin, einen Mittelweg zwischen *Erkundung (exploration)* und *Ausbeutung (exploitation)* zu finden. Erkundung bezeichnet dabei die Untersuchung möglichst vieler Regionen des potentiellen Suchraumes. Da dieser in den meisten Anwendungsfällen eine sehr hohe Kardinalität aufweist, kann eine Erkundung ohnehin nur stichprobenartig erfolgen. Je größer die Anzahl der Individuen und je gleichmäßiger diese im Suchraum verstreut sind, desto geringer ist die Wahrscheinlichkeit, ein eventuelles Extremum zu übersehen. Im Gegensatz dazu bezeichnet die Ausbeutung eine lokale Suche in der Nähe eines herausragenden Individuums. Mit einer gut umgesetzten Ausbeutung kann sichergestellt werden, dass (zumindest lokale) Optima identifiziert und lokalisiert werden. Das Prinzip der Ausbeutung ist im Hillclimbing-Algorithmus aus Kapitel 4.3 perfektioniert, weshalb es auch durchwegs erfolgreiche Versuche gab, evolutionäre Verfahren mit der Hillclimbing-Strategie zu hybridisieren.

Der Mittelweg zwischen Erkundung und Ausbeutung ähnelt einer Gratwanderung: Trimmt man einen Evolutionären Algorithmus zu sehr auf Erkundung, verschwendet das Verfahren viel Zeit auf die Untersuchung unbrauchbarer Lösungen; verstärkt man stattdessen die Neigung zur Ausbeutung, konvergieren Genetische Algorithmen vorzeitig zu einem suboptimalen, da lokalen Extremum. Daher ist die Wahl geeigneter Selektions-, Rekombinations- und Mutationsoperatoren sowie zugehöriger Wahrscheinlichkeiten entscheidend. Alternativ kann auch eine automatische Anpassung der genetischen Parameter im Laufe der Ausführung eines Evolutionären Verfahrens in Betracht gezogen werden (siehe Kapitel 4.5.3).

Codierung

Evolutionäre Verfahren sind Metaheuristiken und müssen daher vor ihrer Anwendung erst an das zu lösende Optimierungsproblem adaptiert werden. Der erste Schritt dabei besteht in der Wahl einer geeigneten Codierung des Suchraums, also einer passenden, maschinell verarbeitbaren Repräsentation aller potentiellen Lösungen der Optimierungsaufgabe. Dies ist zugleich der entscheidende Schritt bei der Umsetzung einer evolutionären Optimierung, da hiermit die Wahl der möglichen genetischen Operatoren (Selektion, Kreuzung und Mutation) eingeschränkt und die Performanz der Implementierung gravierend beeinflusst werden können. Grundsätzlich gibt es zwei Klassen unterschiedlicher Ansätze.

Bei der *direkten Codierung* spiegelt sich der zu erwartende Phänotyp unmittelbar im entsprechend codierten Genotyp wieder. Werden Evolutionäre Verfahren zum Beispiel wie in [WGO02] zur Generierung optimaler Stundenpläne für Schulen eingesetzt, so repräsentiert ein Individuum den Stundenplansatz aller Schulklassen. Somit besteht dessen Chromosom aus einer $s \times t \times k$-Matrix U für die s Unterrichtsstunden eines jeden der t Wochentage jeder der k Schulklassen, wobei jedes Feld $u_{a,b,c}$ das Unterrichtsfach zur entsprechenden Stunde a am Wochentag b der Klasse c repräsentiert. Der Vorteil einer direkten Codierung liegt in der effizienten Umsetzung der Fitnessbewertung, da der Phänotyp nicht erst aus dem Genotyp entschlüsselt werden muss, ehe die Bewertung ermittelt werden kann. Nachteilig ist jedoch die größere Komplexität der genetischen Operatoren Kreuzung und Mutation.

Die *indirekte Codierung* ist die von Holland ursprünglich vorgestellte und bevorzugte Art der Repräsentation. Das Chromosom ist dabei eine Abfolge von Elementen aus einem binären Alphabet, typischerweise ein Bit-String mit den Einzelwerten 0 oder 1. In diesem Sinne stellen bestimmte Teilfolgen dieses Genotyps eine „Bauanleitung“ für den Phänotyp dar. Im Beispiel der Stundenplanoptimierung für Schulen ist eine mögliche Interpretation eines binär codierten Individuums wie folgt denkbar: Die ersten drei Bits stellen einen Wochentag, die nächsten vier eine Anfangszeit, weitere sechs die Schulklasse und die folgenden dreizehn das Unterrichtsfach dar, womit alles zusammen eine bestimmte Unterrichtsstunde repräsentiert - die weiteren Blöcke werden dann entsprechend festgelegt. Der offensichtliche Vorteil dieses Ansatzes ist, dass Kreuzungs- und Mutationsoperatoren nicht immer wieder problemspezifisch angepasst werden müssen. Nachteilig ist insbesondere die aufwändige Rekonstruktion des Phänotyps zur Bestimmung der Fitness eines Individuums.

Bei Benutzung der einfachen Binärcodierung ergibt sich noch eine Reihe weiterer Probleme. Zwei benachbarte ganze Zahlen im Dezimalsystem können sehr unterschiedliche Binärdarstellungen aufweisen, sich also in mehr als nur einem Bit unterscheiden. Dies trifft zum Beispiel auf die Zahlen $7_{(10)} = 0111_{(2)}$ und $8_{(10)} = 1000_{(2)}$ zu, bei denen kein einziges Bit übereinstimmt. Im Falle Genetischer Algorithmen führt das zu einer weiteren Multimodalität, die die Suche nach einem Optimum deutlich erschwert. Wie im Kapitel 4.5.1 (ab Seite 102) beschrieben, besteht die Mutation bei einer binären Codierung in der Negation eines Bits. Problematisch dabei ist, dass die einzelnen Bit-Positionen unterschiedlich signifikant sind. So bewirkt eine Mutation an der i-ten Stelle (von rechts zählend) eine Änderung des entsprechenden Wertes um 2^{i-1}, wie zum Beispiel bei $0101_{(2)} \overset{mut}{\mapsto} 1101_{(2)} \Rightarrow 5_{(10)} \overset{mut}{\mapsto} 13_{(10)}$ aber $0101_{(2)} \overset{mut}{\mapsto} 0100_{(2)} \Rightarrow 5_{(10)} \overset{mut}{\mapsto} 4_{(10)}$. Um beiden Problemen zu begegnen, kann man den sogenannten *Gray-Code* verwenden. Dieser ist derart beschaffen, dass sich zwei benachbarte Dezimalzahlen in ihrer Gray-Codierung in *genau* einem Bit unterscheiden. Darüber hinaus muss bei der Codierung pseudo-reeller Zahlen je nach Maschinengenauigkeit eine entsprechende Mindestlänge des Bitstrings vorgesehen werden, da ansonsten die Gefahr besteht, dass relevante „reelle“ Werte nicht in der Binärcodierung vorkommen. Bei diskreten Parametern entstehen Schwierigkeiten mit der binären Codierung, wenn die Kardinalität ihrer Wertemenge keine Potenz von zwei ist, womit eine bijektive Abbildung zwischen Genotyp und Phänotyp nur umständlich realisiert werden kann. Bei einer trivialen Codierung könnten so Genotypen entstehen, deren Phänotypen keine sinnvollen Lösungen des Optimierungsproblems darstellen oder es können gewisse Phänotypen häufiger vorkommen, da

sie mehrere Genotyp-Entsprechungen aufweisen.

Die Codierungsstrategie spaltet die Verfechter Evolutionärer Verfahren in zwei Lager. Beide glauben, ihre Codierungsvariante biete die größere Anzahl Schemata (siehe Kapitel 4.5.1 ab Seite 104) und sei damit im Hinblick auf die implizite Parallelisierung ideal [BBM93b]. Welche Codierung nun am besten ist, hängt stark vom gestellten Optimierungsproblem ab und muss im Einzelfall abgewogen werden. Wichtig dabei ist, dass *alle* möglichen Lösungen eines Problems erfasst werden können, dass möglichst keine Werte mehrfach auftreten und dass die Codierung der Problemstellung und der verwendeten Algorithmus-Variante adäquat ist. Letzteres ist deshalb so entscheidend, da für die Berechnung der Fitness der Phänotyp benötigt wird, also der Genotyp meist erst entschlüsselt werden muss, und weil andere genetische oder hybride Operatoren häufig auf den Chromosomen arbeiten. Eine schlechte Codierung hat dann einen ineffizienten Programmablauf zur Folge.

Auswahlstrategien

Der Genetische Algorithmus erzeugt während der Initialisierung im ersten Schritt eine Population von Individuen, deren Gene typischerweise uniform verteilt aus dem Definitionsbereich der Parameter gewählt werden. Da die Population im Allgemeinen weitaus kleiner als der potentielle Suchraum ist (zum Beispiel 40 Individuen gegenüber 10^{1840} mögliche Stundenplansätze [WGO02]), wird mit hoher Wahrscheinlichkeit zunächst *keine* optimale Lösung dabei sein.

Da der Zweck des Genetischen Algorithmus die Suche nach einem (o.B.d.A.) Maximum ist, erscheint es erfolgsversprechend, in der Nähe derjenigen bisherigen Lösungen weiterzusuchen, für die die objektive Bewertungsfunktion den größten Wert annimmt. In Anlehnung an die Natur werden deshalb für die nächste Generation bevorzugt die Individuen mit höherer Fitness zur Reproduktion durch Kreuzung zugelassen. Das bedeutet, je besser ein Individuum in der aktuellen Generation ist, desto mehr Nachkommen kann es in die nächste Generation einbringen, also desto häufiger sind seine Eigenschaften (sprich Parameterwerte) in der kommenden Generation vertreten. Die Folge ist, dass sich die Suche mehr und mehr um die (bisher) beste Lösung konzentriert. Wählt man die Populationsgröße und die Art der anderen genetischen Operatoren geschickt, dann konzentriert sich die Suche nicht nur einseitig auf das beste Individuum. Der Genetische Algorithmus verfolgt stattdessen parallel mehrere potentielle Lösungen. Seit der Erfindung und dem breiten Einsatz Genetischer Algorithmen wurden vielfältige Techniken entwickelt und evaluiert, um „Eltern“ für die Individuen der nächsten Generation aus der jeweils aktuellen auszuwählen. Alle sogenannten Selektionsmechanismen ([BT95, Han94, BH91]) weisen unterschiedliche Vor- und Nachteile auf, so dass sie für bestimmte Optimierungsaufgaben auch unterschiedlich gut geeignet sind.

Elitismus und Steady-State Selection: Bevor im Folgenden diverse klassische Selektionsverfahren kurz skizziert werden, seien hier noch zwei Hilfsmechanismen aus der Natur vorgestellt, deren Ziel es ist, einen monotonen Fitnessverlauf über die gesamte Optimierung hinweg zu erreichen. In der einfachen Variante des Genetischen Algorithmus aus Abbildung 4.3 gibt es stets zwei Generationen: Eine alte, aus denen die Eltern gewählt werden und eine neue, in die die Nachfahren eingesetzt werden. Sobald die neue Generation vollständig gefüllt ist, wird die alte

verworfen und von der soeben neu erzeugten Generation vollständig ersetzt. Da die genetischen Operatoren Kreuzung und Mutation neue Individuen generieren, die sich von ihren Eltern meist signifikant unterscheiden, kann es bei diesem Grundverfahren vorkommen, dass die besten Lösungen der alten Generation nicht mehr in der neuen vertreten sind. Folglich gehen die guten Eigenschaften dieser Individuen verloren, sobald die alte Elterngeneration verworfen wird. Um diesem Informationsverlust vorzubeugen, erweitert man die Grundvariante der GAs um einen weiteren Operator, dem sogenannten *Elitismus*. Dabei wird ein bestimmter Anteil der besten Individuen völlig unverändert in die nächste Generation kopiert.

Alternativ kann der Genetische Algorithmus auch ohne sichtbaren Generationswechsel implementiert werden. Bei diesem sogenannten *Steady-State*-Verfahren werden die Nachkommen nicht zu einer neuen Generation hinzugefügt. Stattdessen ersetzen diese Nachkommen gleich nach ihrer Entstehung bestimmte Individuen der aktuellen Generation. Kandidaten für eine Ersetzung durch neue Nachkommen sind beispielsweise Individuen mit der kleinsten Fitness oder diejenigen, die am längsten in der Population existieren. Ersteres garantiert ebenfalls, dass die besten Individuen stets überleben.

Ein Vorteil der Steady-State-Variante ist, dass die jeweils direkt in die aktuelle Population eingeführten Individuen auch sofort als Elternteil zur Verfügung stehen, wodurch der Genetische Algorithmus nicht erst nach der Erstellung einer ganzen Generation von einer eventuell besseren Lösung profitieren kann. Der Nachteil dieser inkrementellen Variante ist, dass die Einführung eines neuen und häufig stärkeren Individuums die durchschnittliche Fitness anhebt, wodurch schlechtere Individuen eine geringere Chance zur Reproduktion bekommen und der Algorithmus möglicherweise vorzeitig zu einem lokalen Optimum konvergiert. Darüber hinaus erfordert das Steady-State-Vorgehen bei manchen Selektionsstrategien eine vollständige Neuberechnung der Fitnesswerte für jeden neu eingefügten Nachkommen, sofern das Auswahlverfahren auf einer normierten Fitness beruht.

Best Fitness: Von einer rein zufälligen Selektion abgesehen besteht die intuitivste Auswahlstrategie darin, stets die besten zwei Individuen zu Eltern zu küren. Da hierzu keine Umrechnung der Fitnesswerte, keine Sortierung der Individuen und auch keine explizite Suche innerhalb der potentiellen Lösungen in der aktuellen Generation notwendig ist, weist dieser Ansatz den geringsten Zusatzaufwand im Bezug auf Rechen- oder Speicherressourcen im Laufe der Optimierung auf.

Bedauerlicherweise ist dieser Ansatz nur sehr bedingt produktiv. Da insbesondere bei der Variante Genetischer Algorithmen mit explizitem Generationswechsel immer wieder die gleichen Eltern zur Reproduktion herangezogen werden, gehen die Informationen aller anderen Individuen der alten Generation plötzlich oder nach und nach vollständig verloren. Man spricht dann von einem starken Verlust der Vielfalt („loss of diversity“) im Genom, bedingt durch eine Ausrichtung des Verfahrens weg von der Erkundung hin zur Ausbeutung. Somit ähnelt das Verhalten eines solchen Genetischen Algorithmus eher einem Hillclimbing (siehe Kapitel 4.3), das heißt es wird bevorzugt in der Umgebung des besten Individuums gesucht und der Algorithmus konvergiert zum nächstgelegenen, meist lokalen Extremum.

Roulette Wheel: Zwar gibt es kein ideales Selektionsverfahren, jedoch haben empirische Untersuchungen gezeigt, dass diejenigen Varianten Genetischer Algorithmen in den meisten Einsatzszenarien am effizientesten waren, bei denen die Wahrscheinlichkeit für die Auswahl eines Individuums zur Reproduktion direkt proportional zu seiner Fitness innerhalb der jeweiligen Generation war. In Anlehnung an Kapitel 4.1 sei $P_{aktuell} = (a_1, \ldots, a_n) \in \mathcal{S}^n$ die Population der aktuellen Generation, wobei a ein Individuum aus der Menge $\mathcal{S}$ aller möglichen Lösungen ist, und $f : \mathcal{S} \to \mathbb{R}$ die objektive Bewertungsfunktion, die jedem Individuum seine Fitness zuordnet. Dann wird das Individuum $a_i \in P_{aktuell}$ unter der sogenannten *Roulette Wheel Selection* mit folgender Wahrscheinlichkeit $p_{RW}(a_i)$ zum Elternteil gewählt:

$$p_{RW}(a_i) = \frac{f(a_i)}{\sum_{j=1}^{n} f(a_j)}$$

Anschaulich liegt diesem Verfahren die Vorstellung eines Roulette-Rades zugrunde, bei dem jeder Roulette-Zahl ein Individuum zugeordnet ist. Abweichend von einem klassischen Roulette-Spiel ist die Größe der Fläche jeder Zahl, auf die die Roulette-Kugel landen kann, proportional zur Fitness des entsprechenden Individuums – daher der Name „Roulette Wheel Selection“ oder auch *Proportional Selection*. Abbildung 4.4 zeigt exemplarisch ein solches Roulette-Rad basierend auf der Beispielgeneration aus Tabelle 4.1.

Individuum	*Fitness*	*Selektionswahrscheinlichkeiten*					
a_i	$f(a_i)$	$p_{BF}(a_i)$	$p_{RW}(a_i)$	$p^w_{RW}(a_i)$	$p^\sigma_{RW}(a_i)$	$p^B_{RW}(a_i)$	$p_R(a_i)$
a_1	143	0,0%	11,0%	0,0%	0,0%	7,2%	6,7%
a_2	299	100,0%	23,0%	26,7%	26,8%	20,4%	26,7%
a_3	247	0,0%	19,0%	17,8%	13,7%	14,4%	20,0%
a_4	429	100,0%	33,0%	48,9%	59,6%	48,6%	33,3%
a_5	182	0,0%	14,0%	6,7%	0,0%	9,4%	13,3%

$p_{BF}(a_i)$: nach *Best Fitness* für die Auswahl beider Elternteile
$p_{RW}(a_i)$: nach *Roulette Wheel* pro Auswahl eines Elternteils
$p^w_{RW}(a_i)$: nach *Roulette Wheel* mit *Windowing* ($w = 1$) pro Auswahl eines Elternteils
$p^\sigma_{RW}(a_i)$: nach *Roulette Wheel* mit *Sigma Scaling* ($s = 0,6$) pro Auswahl eines Elternteils
$p^B_{RW}(a_i)$: nach *Roulette Wheel* mit *Boltzmann Selection* ($T = 150$) pro Auswahl eines Elternteils
$p_R(a_i)$: nach *Rank Selection* pro Auswahl eines Elternteils

Tabelle 4.1: Beispielhafte Fitnessverteilung und Selektionswahrscheinlichkeiten

Wie bereits angedeutet, hängt die Qualität des Ergebnisses und die Performance eines Genetischen Algorithmus stark vom sogenannten *Selektionsdruck* ab, also dem Verhältnis der Wahrscheinlichkeiten mit dem unterschiedlich gute Lösungen zur Reproduktion ausgewählt werden. Ist dieser zu hoch, wie zum Beispiel bei der „Best Fitness“-Selektion, so tritt eine frühzeitige Konvergenz zu einem lokalen Optimum auf. Dieser unerwünschte Fall kann in abgeschwächter Form auch bei der proportionalen Auswahlstrategie dann eintreten, wenn die Fitnessverteilung

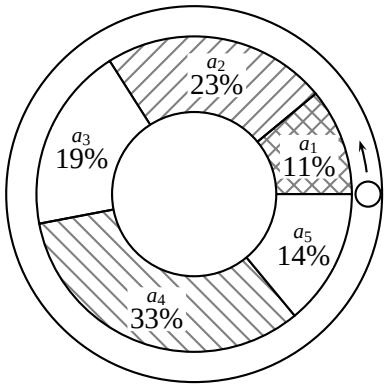


Abbildung 4.4: Beispiel für *Roulette Wheel Selection*

in der Population stark ungleichmäßig ist, also wenn einzelne Lösungen von relativ hoher Qualität sind, während andere hingegen eine nur geringe Fitness aufweisen. In diesem Fall bevorzugt die Selektion überdurchschnittlich oft die starken Individuen, wodurch der Genetische Algorithmus zur Ausbeutung ihrer Umgebung neigt und dabei die Erkundung vernachlässigt. Umgekehrt ist der Selektionsdruck zu niedrig, wenn nahezu alle Individuen eine ähnlich hohe Fitness aufweisen. Mögliche Gegenmaßnahmen sind zum Beispiel das sogenannte *Windowing*, das *Sigma scaling* und die *Boltzman selection* [Han94, BBM93a].

Bei **Windowing** handelt es sich um eine Fitness-Transformation mittels einer dynamischen Basis, welche vor Anwendung einer klassischen Selektionsstrategie (zum Beispiel „Roulette Wheel“) ausgeführt werden kann. Dazu wird der schlechteste Fitnesswert f^w_{min} der letzten w Generationen vermerkt und als Basis für die aktuelle Generation verwendet, weshalb der Parameter w auch *Fenstergröße (window size)* genannt wird. Übliche Werte für w liegen zwischen 2 und 10, im einfachsten Fall kann jedoch auch $w = 1$ gewählt werden, so dass nur die Fitness der aktuellen Generation berücksichtigt werden muss. Die Selektionswahrscheinlichkeit $p^w_{RW}(a_i)$ bei „Roulette Wheel“ nach dem „Windowing“ berechnet sich dann zu:

$$p^w_{RW}(a_i) = \frac{f(a_i) - f^w_{min}}{\sum_{j=1}^{n} \left(f\left(a_j\right) - f^w_{min}\right)}$$

Das sogenannte **Sigma Scaling** verstärkt den Selektionsdruck indem die ebenfalls dynamische Basis $f^\sigma_{min} := \mu - s \cdot \sigma$ auf die s-fache Standardabweichung $\sigma := \sigma_{f(a_1),\ldots,f(a_n)}$ unterhalb des Mittelwerts $\mu := \mu_{f(a_1),\ldots,f(a_n)}$ aller Fitnesswerte der aktuellen Generation angesetzt wird, wobei s ein Skalierungsfaktor ist. Alle Individuen der Elterngeneration, deren Fitness kleiner als f^σ_{min} ist, werden bei der Selektion gar nicht berücksichtigt. Diese Methode erlaubt es daher, besonders schlechte Individuen einer Generation, die sogenannten *lethals*[5], gesondert zu behandeln. Der Selektionsdruck kann unmittelbar über den Parameter s konfiguriert werden und verhält sich

[5] *Engl., hier*: dem Tode Geweihte, Unfruchtbare

umgekehrt proportional zu s. Mit $f^{\sigma}_{min} = \mu_{f(a_1),\ldots,f(a_n)} - s \cdot \sigma_{f(a_1),\ldots,f(a_n)}$ berechnet sich die transformierte Fitness $f^{\sigma}(a_i)$ zu:

$$f^{\sigma}(a_i) := \begin{cases} 0 & : \quad f(a_i) \leq f^{\sigma}_{min}, \\ f(a_i) - f^{\sigma}_{min} & : \quad sonst. \end{cases}$$

Für die Auswahlwahrscheinlichkeit $p^{\sigma}_{RW}(a_i)$ anhand „Roulette Wheel“ nach dem „Sigma Scaling“ gilt damit:

$$p^{w}_{RW}(a_i) = \frac{\max(0, f(a_i) - f^{\sigma}_{min})}{\sum_{j=1}^{n} \max\left(0, f(a_j) - f^{\sigma}_{min}\right)}, \; mit \; f^{\sigma}_{min} := \mu_{f(a_1),\ldots,f(a_n)} - s \cdot \sigma_{f(a_1),\ldots,f(a_n)}, \; s \in \mathbb{R}$$

Gegenüber „Windowing“ hat „Sigma Scaling“ den Vorteil, dass die Grundlinie nicht nur vom schlechtesten Individuum der letzten w Generationen abhängt. Ist die Fitness dieses Individuums ungewöhnlich niedrig, so beeinflusst die Transformation des „Windowing“ kaum die ursprünglichen Werte, was sich besonders unangenehm äußert (siehe „Rank Selection“), wenn gleichzeitig ein sehr starkes Individuum in der aktuellen Population vorhanden ist. Bei Sigma Scaling hingegen wird die Grundlinie dynamisch am Mittelwert der aktuellen Population ausgerichtet und berücksichtigt daher sowohl sehr gute als auch sehr schlechte Lösungen.

Die sogenannte **Boltzmann Selection** ist mit „Sigma Scaling“ stark verwandt, allerdings versucht man dabei die stärkeren Individuen überdurchschnittlich zu gewichten. Die Transformation der Fitness erfolgt dazu nach folgendem Schema:

$$f^{B}(a_i) := \frac{e^{\frac{f(a_i)}{T}}}{\frac{1}{n} \cdot \sum_{j=1}^{n} e^{\frac{f(a_j)}{T}}}$$

Der Parameter T hat eine ähnliche Wirkung wie die Temperatur bei Simulated Annealing (siehe Kapitel 4.4). Dadurch, dass der Wert dieses Parameters im Verlauf der Ausführung kontinuierlich gesenkt wird, vergrößert man entsprechend das Verhältnis zwischen kleinen und großen Fitnesswerten nach der Transformation, womit der Selektionsdruck laufend wächst.

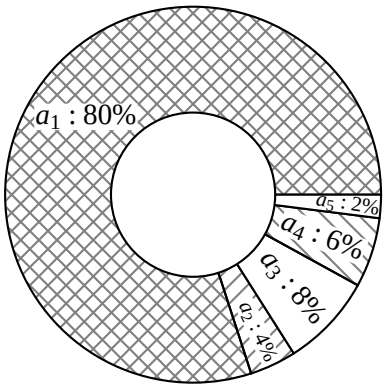
Sind die Fitnessbewertungen innerhalb einer Generation extrem unterschiedlich, wie zum Beispiel in Abbildung 4.5(a), so kann der hohe Selektionsdruck bei der sogenannten **Rank Based Selection** mit weniger Rechenaufwand und trotzdem höherer Wirkung als bei „Windowing“, „Scaling“ oder „Boltzman“, mittels eines *Rankings* reduziert werden. Die eigentliche Selektion kann anschließend mit den üblichen Auswahlstrategien, wie zum Beispiel bei „Roulette Wheel“, durchgeführt werden.

Wie in Abbildung 4.5 exemplarisch demonstriert, werden die Individuen der jeweils aktuellen Generation durch Sortierung nach ihrer Fitness klassifiziert, wodurch sich in Analogie zur Natur eine Rangordnung ergibt. Anschließend wird ihnen eine „neue“ Fitness zugeordnet, welche nun ihrem Rang in der Population entspricht: Das schlechteste Individuum beginnt mit Rang 1, das zweitbeste mit Rang 2, usw. Individuen mit gleicher Fitness werden natürlich auch dem gleichen Rang zugeordnet.

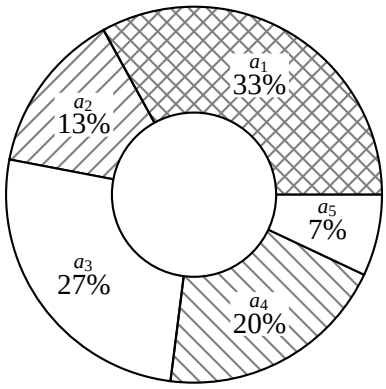
Vergleicht man die Verteilung der Auswahlwahrscheinlichkeiten vor dem Ranking in Abbildung 4.5(b) mit der nach dem Ranking in Abbildung 4.5(c), so stellt man fest, dass nach dem Ranking alle Individuen eine realistische Chance zur Kreuzung und Reproduktion haben. Der Nachteil dieses Vorgehens ist eine möglicherweise langsamere Konvergenz zum Optimum, denn die ursprünglich besten Lösungen einer Generation unterscheiden sich nach dem Ranking in ihrer Fitness nicht mehr wesentlich von den ursprünglich deutlich weniger fitten Individuen, womit auch die sehr schlechten Lösungen eventuell weiterhin verfolgt werden.

a_i	$f(a_i)$	$f_R(a_i)$
1	80	5
2	4	2
3	8	4
4	6	3
5	2	1

(a) Fitnesstransformation



(b) *vor* Ranking



(c) *nach* Ranking

Abbildung 4.5: Anpassung der Fitnesslandschaft beim *Ranking*

Sampling und Tournament Selection: Viele der eben erwähnten Selektionsmechanismen, darunter insbesondere die beliebte „Roulette Wheel“-Variante, haben den Nachteil, dass sie einen sehr großen Rechenaufwand für jedes auszuwählende Individuum verursachen. Da die Auswahl einen zentralen und häufig wiederkehrenden Teil des Genetischen Algorithmus ausmacht, wirkt sich dies insbesondere bei der Betrachtung großer Populationen deutlich auf die Effizienz der Implementierung aus. Andererseits bedeutet eine große Anzahl an Individuen in einer Generation auch eine große Artenvielfalt und damit eine sehr gute Erkundung des Suchraums. Eine Lösung des Effizienzproblems besteht darin, die Selektion nicht auf die gesamte Elterngeneration zu erstrecken, sondern vorher eine Vergleichsstichprobe zu entnehmen, also ein sogenanntes Sampling durchzuführen.

Eine verbreitete Variante dieses Ansatzes ist die sogenannte *Tournament Selection*. Dabei werden aus der Elterngeneration zufällig und idealerweise ohne Wiederholung eine bestimmte Anzahl Lösungen entnommen. Der Sieger dieses „Wettkampfes“ und damit das zur Reproduktion zugelassene Individuum ist die potentielle Lösung mit der besten Fitness innerhalb dieser entnommenen Stichprobe. Zur Bestimmung des zweiten Elternteils verfährt man analog, wobei man üblicherweise eine neue Vergleichsstichprobe wählt.

Crossovermechanismen

Im Umfeld Evolutionärer Verfahren haben sich zwei unterschiedliche Ansätze bezüglich des Übergangs von der „Auswahl“ zur „Kreuzung“ von Individuen etabliert. In der einen Variante werden die beiden Phasen vollständig getrennt und aufeinanderfolgend ausgeführt. Dabei werden zuerst alle potentiellen Eltern durch einen beliebigen Selektionsmechanismus nacheinander ausgewählt und in einem sogenannten *mating-pool* aufgenommen. Solange noch nicht alle Eltern aufgebraucht sind und die neue Generation vollständig ist, werden daraus in einem zweiten Durchlauf je zwei Individuen nach einer uniformen Verteilung entnommen und nach einer der nachfolgend beschriebenen Strategien gekreuzt. Alternativ können Selektion und Rekombination auch abwechselnd durchgeführt werden, bis die neue Generation vollständig ist.

Jedes Individuum einer Population stellt eine potentielle Lösung eines Optimierungsproblem dar, welche sich je nach Codierung aus unterschiedlichen Teilbeiträgen zusammensetzt. Schon während der Initialisierung und immer wieder im weiteren Lauf der Evolution entstehen Individuen, die einzigartige vorteilhafte Eigenschaften enthalten und deshalb eine höhere Fitness erzielen. Diese Eigenschaften können dabei aus unterschiedlichen Parametern hervorgehen, sich also jeweils an einem anderen Lokus im Chromosom befinden. Genetische Algorithmen verfolgen das Ziel, ein „Superindividuum“ zu züchten, welches möglichst viele vorteilhafte Allele in sich vereinigt. Aufgabe der Selektion war es, diejenigen Individuen (Eltern) auszuwählen und zusammenzuführen, welche mit großer Wahrscheinlichkeit mindestens eine in diesem Sinne geeignete Eigenschaft aufweisen. Damit ein neues Individuum (Kind) die vorteilhaften Allele in sich vereinigen kann, muss ein genetischer Operator den Austausch von Allelen ermöglichen. In Anlehnung an die Inspiration aus der Natur wird dieser Mechanismus *Crossoveroperator* genannt.

1-point-crossover: Die einfachste Variante einer Rekombination besteht darin, die Chromosomen der beiden Eltern zunächst zu kopieren, anschließend die entstandenen Kopien jeweils an einer beliebigen Stelle aufzubrechen und die letzten beiden Bruchteile auszutauschen. Somit entstehen zwei Nachkommen, welche jeweils einen Teil des Chromosoms beider Elternteile erben.

Abbildung 4.6 zeigt das Vorgehen des *1-point-crossover*-Operators. Alternativ kann auch nur eines der beiden Kinder direkt generiert und übernommen werden. Dazu wird ein Teil des Chromosoms von der Mutter und der restliche Teil vom Vater kopiert, wie dies in Abbildung 4.7 für die Zwei-Punkt-Rekombination angedeutet ist. Die erste Variante erscheint jedoch vielversprechender, da hierbei die Wahrscheinlichkeit höher ist, eine Kombination zu erhalten, die beide attraktiven Eigenschaften enthält.

Bei der Umsetzung der Kreuzungsoperatoren muss die Konsistenz der Chromosomen aller Nachkommen sichergestellt werden. Eine typische Anforderung mancher binär codierter Lösungen ist, dass die Länge der Chromosomen bei allen Individuen stets gleich bleibt. In diesem Falle müssen Mutter- und Vatererbgut am gleichen Lokus aufgetrennt werden. Bei direkter Codierung kann das Chromosom im Allgemeinen nicht an einer beliebigen Stelle zerlegt und mit einem anderen Bruchstück zusammengesetzt werden, da ansonsten die Dekodierung zu einem gültigen Phänotyp nicht mehr möglich ist. Die Konsistenz kann entweder direkt bei der Kreuzung durch

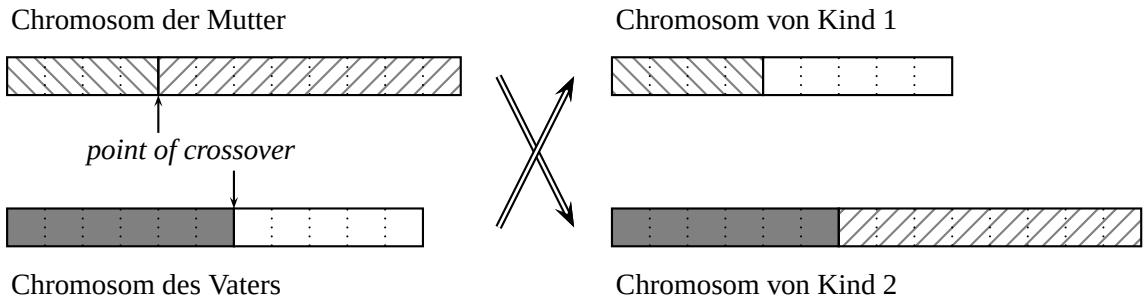


Abbildung 4.6: Darstellung des *1-point-crossover* mit *zwei* Nachkommen

komplexere Crossoveroperatoren sichergestellt oder nachträglich mittels sogenannter *Reparatur-operatoren* gezielt wiederhergestellt werden.

2-point-crossover: Eine verwandte Art der Rekombination stellt der sogenannte *2-point-crossover*-Operator dar. Abbildung 4.7 zeigt das Funktionsprinzip dieser Strategie. Es entspricht im Wesentlichen dem „1-point-crossover“-Verfahren, jedoch mit dem Unterschied, dass in diesem Falle die Kopien der Elternchromosomen an zwei unterschiedlichen Stellen aufgebrochen und die beiden Mittelstücke getauscht werden.

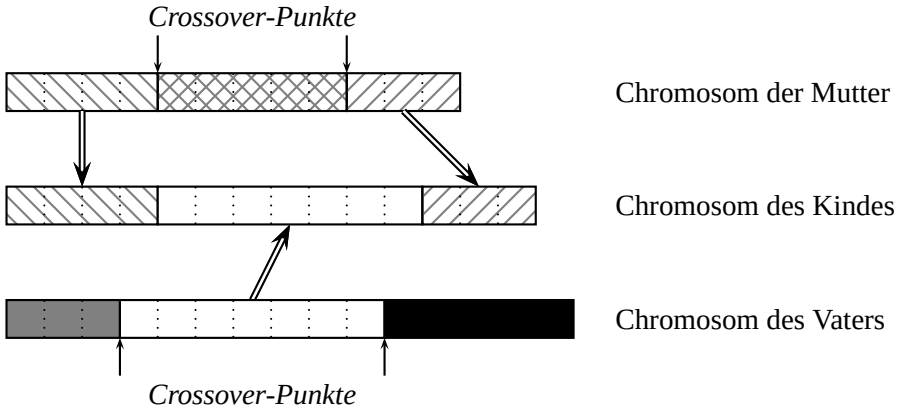


Abbildung 4.7: Darstellung des *2-point-crossover* mit *einem* Nachkommen

Ein empirischer Vergleich verschiedener Kreuzungsverfahren hat ergeben [BBM93b], dass mit diesem „2-point-crossover“ im Allgemeinen eine deutliche Verbesserung der Performanz gegenüber der ursprünglichen „1-point“-Variante erreicht werden kann, jedoch würde die Einführung weiterer Rekombinationspunkte keinen weiteren Gewinn erbringen. Dennoch ist die Eignung der unterschiedlichen Crossover-Strategien stark problemabhängig, insbesondere variiert sie mit der Art der Codierung und damit auch der Chromosomlänge. Eine einfache Lösung des Dilemmas besteht darin, die Auswahl der idealen Kreuzungsmethode einem selbstadaptiven

Genetischen Algorithmus zu überlassen, wie das im Kapitel 4.5.3 beschrieben wird.

Uniform-crossover: Selbstverständlich kann man die Anzahl *n* der Rekombinationspunkte nahezu beliebig erhöhen und somit einen allgemeinen *n-point-crossover*-Operator definieren. Im Grenzfall, also wenn die Anzahl der Crossoverstellen *n* gleich der Anzahl der Gene im Chromosom ist, spricht man vom sogenannten *uniform-crossover*. Bei Codierungen mit konstanter Länge werden keine Kreuzungspunkte mehr explizit bestimmt, statt dessen wird eine sogenannte *Bit-Maske* zufällig generiert, anhand derer die einzelnen Gene vertauscht werden. Das Verfahren ist exemplarisch in Tabelle 4.2 dargestellt.

Bit-Maske	1	0	0	1	0	1	1	1	0	0
$\Rightarrow$ *Gen ist zu übernehmen von*	V	M	M	V	M	V	V	V	M	M
Chromosom der Mutter (M)	1	0	1	0	0	0	1	1	1	0
Chromosom des Vaters (V)	0	1	0	1	0	1	0	0	1	1
Chromosom des Kindes	0	0	1	1	0	1	0	0	1	0

Tabelle 4.2: Anschauliche Darstellung des *uniform-crossover* mit *einem* Nachkommen

Im Falle direkter Codierung kann die Anzahl und Reihenfolge der Gene im Chromosom unter Umständen irrelevant sein. Dies trifft zum Beispiel auf den Teilschritt „globale Optimierung“ aus Kapitel 5.1 der hier vorgestellten automatischen Testdatengenerierung zu, bei dem ein Chromosom eine Menge von Testfällen darstellt (Abbildung 5.4). Daher kann der Operator „uniform-crossover“ dadurch abgewandelt werden, dass zunächst alle Gene der beiden Elternteile in einer Menge zusammengeführt und dabei zugleich alle Duplikate eliminiert werden. Anschließend wird eine zufällig gewählte Anzahl Gene aus dieser Menge entnommen und daraus das Chromosom des Nachkommens zusammengestellt.

Bei der Kreuzung kann ein nichtdeterministisches Verhalten einerseits durch die zufällige Wahl entsprechende Kreuzungspunkte erzielt werden, andererseits aber auch durch die Vorgabe einer Rekombinationswahrscheinlichkeit. Mittels dieses Operatorparameters kann gesteuert werden, ob und wie häufig die beiden zur Kreuzung erkorenen Individuen tatsächlich gekreuzt werden oder stattdessen lediglich eine identische Kopie eines der Eltern zum Nachkommen erklärt wird.

Mutation

In seiner ursprünglichen Variante der Genetischen Algorithmen hat ihr Erfinder John Holland die binäre Codierung gewählt, deren Verfechter er stets geblieben ist. Falls darüber hinaus die Population hinreichend groß gegenüber der Mächtigkeit des Suchraums ist, so genügen meist die beiden Operatoren Selektion und Rekombination, um schnell brauchbare Lösungen zu identifizieren, wie vielfältige Versuche belegen konnten. Mit zunehmender Größe und Komplexität des Suchraums sowie der Codierung wirkt sich die Beschränkung auf diese Operatoren jedoch nachteilig aus. Der Grund dafür ist, dass mit einer vergleichsweise kleinen Anfangspopulation

im Allgemeinen auch nur ein beschränkter Wertebereich für jedes Gen abgedeckt wird. Selektion und Rekombination erreichen damit höchstens alle Kombinationen derjenigen Parameterwerte, welche zufällig bei der Initialisierung ausgewählt wurden.

Die Lösung dieses Problems kommt ebenfalls aus der Natur in Form eines weiteren Operators, der sogenannten *Mutation*. Um unabhängig von der ursprünglichen Population auch Ausprägungen von Genen zu erreichen, die in der ersten Generation nicht vorkommen, werden nach der Kreuzung lokal beschränkte, zufällige Veränderungen am Chromosom der Nachkommen vorgenommen. Bei Wahl einer binären Codierung für die Chromosomen eines Individuums besteht die Mutation in einem zufälligen *bitflip*, also einer Veränderung von 1 nach 0 oder umgekehrt, an einer beliebigen Stelle im Chromosom. Verschlüsselt man die Parameter einer Problemlösung durch reelle Zahlen, so mutiert man ein Gen meist dadurch, dass zum bisherigen Wert eine normalverteilte Zufallszahl mit Erwartungswert $\mu = 0$ hinzuaddiert wird.

Empfehlungen

Evolutionäre Verfahren haben eine ganze Reihe unterschiedlicher Freiheitsgrade, darunter die Wahl verschiedener genetischer Operatoren und zugehöriger Parameter. Zwar gibt es keine allgemeingültige und zugleich optimale Parametrisierung dieser Operatoren für beliebige Arten von Optimierungsaufgaben, jedoch kann man aus dem Durchschnitt vielfältiger empirischer Studien zumindest bestimmte Empfehlungen aussprechen, welche sich als Ausgangspunkt für eine individuelle Anpassung eignen. Ist eine solche Anpassung zu aufwendig oder nur schwer realisierbar, so kann man die Konfiguration auch dem Evolutionären Verfahren selbst überlassen (siehe Kapitel 4.5.3).

Einer dieser Freiheitsgrade ist die zu wählende Populationsgröße. Sehr große Populationen können den Suchraum scheinbar gut erkunden, jedoch weisen sie meist kaum eine Verbesserung der Performance (im Sinne der Zeit bis zur Konvergenz zu einem Optimum) eines Genetischen Algorithmus gegenüber kleinerer Populationen auf. Eine Population sollte etwa 20 bis 30 Individuen umfassen, jedoch waren manche Versuche auch mit 50 bis 100 Chromosomen pro Generation erfolgreich. Manche Studien konnten einen Zusammenhang zwischen der optimalen Populationsgröße und der Länge des binär codierten Chromosoms aufzeigen. Demnach sollte eine Population, deren Individuen eine Chromosomlänge von 32 Bit aufweisen, idealerweise auch 32 Individuen groß sein.

Bezüglich der Selektion hat sich die „Roulette Wheel“-Auswahl als sehr effektiv erwiesen, wobei in manchen Fällen eine vorangehende Fitnesstransformation (zum Beispiel „Ranking“) von großem Vorteil war. Wenn an Stelle einer „Steady-State“-Strategie eine generationengebundene Evolution bevorzugt wird, dann sollte auf einen Elitismus-Operator nicht verzichtet werden.

Welche Art der Rekombination ideal ist, hängt sehr stark von der Problemstellung und der Codierung der Gene ab. Manche Autoren empfehlen „2-point-crossover“, andere haben mit „uniform-crossover“ die besten Erfahrungen machen können. Allen empirischen Ergebnissen ist jedoch zu entnehmen, dass die Kreuzungsrate relativ hoch sein sollte, das heißt, die Wahrscheinlichkeit, dass zwei Individuen überhaupt gekreuzt werden, sollte etwa 80% bis 95% betragen, wenngleich manchmal auch mit 60% von guten Ergebnissen berichtet wurde. Bei Nutzung des Elitismus-Operators kann die Crossover-Wahrscheinlichkeit durchaus auch 100% betragen.

Im Gegensatz zu einer hohen Rekombinationswahrscheinlichkeit hat sich eine niedrige Mutationswahrscheinlichkeit als besonders vorteilhaft herausgestellt. Bei Wahl einer binären Codierung sollte ein Bit eines Gens demnach lediglich mit einer Wahrscheinlichkeit von $0,5\%$ bis 1% invertiert werden. Ebenso ist bei reellwertigen Genen eine Mutation durch Addition einer normalverteilten Zufallsvariablen mit kleiner Varianz einer zufälligen und uniformverteilen Neuwahl des Gens vorzuziehen.

Das Schema-Theorem

Ein Großteil der Forschung auf dem Gebiet der Evolutionären Verfahren hat sich vorwiegend mit der Verbesserung der Algorithmen selbst, dabei insbesondere der genetischen Operatoren, beschäftigt. Dennoch gibt es unterschiedliche Hypothesen zur Erklärung des Optimierungspotentials dieser Heuristiken, wenngleich sich bisher keine wirklich allgemeingültige durchsetzen konnte [BBM93a]. Zwei dieser Erklärungsansätze stammen von den „Erfindern“ der Evolutionären Verfahren selbst, nämlich das sogenannte *Schema-Theorem* [Hol75] von John Holland und die *Building-Block-Hypothese* [Gol89] von Goldberg.

Die erste und rigorose Hypothese, das Schema-Theorem von Holland, bezieht sich zunächst auf eine binäre Codierung der Chromosomen, lässt sich aber sinngemäß auf beliebige Codierungen erweitern. Das Alphabet $\mathbb{B} := \{0,1\}$ der Binär-Codierung wird hierbei um ein weiteres Symbol „$\star$“ (dem sogenannten *„don't-care“*) zu $\Sigma := \{0,1,\star\}$ erweitert.

Definition 4.1 (Schema) *Ein* Schema *ist eine Abfolge* $\sigma = (s_1\ s_2\ \ldots\ s_m) \in \Sigma^m$ *der Länge m und stellt ein Muster von Genen dar, welches alle binären Sequenzen* $\overline{\sigma} = (\overline{s_1}\ \overline{s_2}\ \ldots\ \overline{s_m}) \in \mathbb{B}^m$ *repräsentiert, deren Stellen* $\overline{s_i}$ *mit den entsprechenden Stellen* s_i *von* σ *übereinstimmen, wobei* $\star$ *sowohl zu* 0 *als auch zu* 1 *äquivalent ist (*$\forall 1 \leq i \leq m : \overline{s_i} = s_i \vee s_i = \star$*).*

Ein Chromosom *enthält* eine bestimmtes Schema, wenn es nach Definition 4.1 mit diesem Schema übereinstimmt. Das Chromosom (0101) enthält unter anderen zum Beispiel die Schemata $(0\star\star1)$ sowie $(\star1\star1)$.

Definition 4.2 (Ordnung) *Sei* $\sigma = (s_1, s_2, \ldots, s_m)$ *mit* $\forall 1 \leq i \leq m : s_i \in \Sigma$ *ein Schema der Länge m, dann ist* $o(\sigma) := |\{s_i | s_i \neq \star\}|$ *die sogenannte* Ordnung *des Schemas* σ *und stellt die Anzahl aller feststehenden (also nicht-“don't-care“-) Stellen dar.*

Die Ordnung eines Schemas ist ein Maß für seine Spezialisierung, also für die Anzahl der mit ihm übereinstimmenden Genen. Das Schema $\sigma_1 = (0\star01\star1)$ hat die Ordnung $o(\sigma_1) = 4$ und ist spezifischer als das Schema $\sigma_2 = (\star\star0\star\star1)$ mit $o(\sigma_2) = 2$.

Definition 4.3 (definierende Länge) *Sei* $\sigma = (s_1, s_2, \ldots, s_m)$ *mit* $\forall 1 \leq i \leq m : s_i \in \Sigma$ *ein Schema. Die sogenannte* definierende Länge $\delta(\sigma)$ *ist der Abstand zwischen der ersten und der letzten feststehenden Stelle.*

Die definierende Länge ist ein Maß für die Kompaktheit der in einem Schema codierten Information. Ein Schema mit nur einer feststehenden Stelle hat per definitionem die definierende

Länge 0. Die beiden Schemata $\sigma_1 = (0\star01\star1)$ und $\sigma_2 = (\star\star0\star\star1)$ weisen die definierenden Längen $\delta(\sigma_1) = 6-1 = 5$ beziehungsweise $\delta(\sigma_2) = 6-3 = 3$ auf.

Aufgrund obiger Definitionen kann eine wahrscheinlichkeitstheoretische Betrachtung der Auswirkungen genetischer Operatoren auf die innerhalb einer Population vorhandenen Schemata erfolgen [Mic94]. Sei $P = \left(a_1^t, a_2^t, \ldots, a_n^t\right)$ eine Population mit n binär codierten Individuen $a_i^t \in \mathbb{B}^m$ in der Generation t, deren Chromosomen die Länge m haben. Darüber hinaus bezeichne

- $\xi(\sigma,t)$ die Anzahl der Individuen, deren Chromosom in der Generation t das Schema σ enthält,
- $f(\sigma,t)$ den Mittelwert der Fitnessbewertungen dieser Individuen,
- $F(t) := \sum_{i=1}^{n} f(a_i^t)$ die Summe der Fitnesswerte aller Individuen a_i^t und
- $\overline{F(t)} := \dfrac{F(t)}{n}$ den Mittelwert der Fitnessbewertungen aller Individuen a_i^t.

Werden n Individuen mittels einer Roulette Wheel Selection ausgewählt und ohne Rekombination und Mutation in die neue Generation übernommen, so befinden sich erwartungsgemäß $\xi(\sigma,t+1)$ Individuen in der nächsten Generation, welche das Schema σ weiterhin enthalten:

$$\xi(\sigma,t+1) = \xi(\sigma,t) \cdot n \cdot \frac{f(\sigma,t)}{F(t)} = \xi(\sigma,t) \cdot \frac{f(\sigma,t)}{\overline{F(t)}}$$

Demnach nimmt der Erwartungswert der Anzahl der Individuen zu, deren Chromosomen überdurchschnittlich bewertete Schemata enthalten, während die „schlechteren“ Individuen zunehmend vom „Aussterben“ bedroht werden.

Nach der Selektion der Eltern werden je zwei Individuen aus diesem mating-pool mit der Wahrscheinlichkeit p_c gekreuzt. Werden die Eltern einem *1-point-crossover* unterzogen, so wird der Kreuzungspunkt mit einer uniform verteilten Wahrscheinlichkeit aus den $m-1$ möglichen Stellen gewählt. Dabei bleibt das Schema σ eines Elternteils mit der Wahrscheinlichkeit $p_s^c(\sigma)$ erhalten oder wird mit der Wahrscheinlichkeit $p_d^c(\sigma) = 1 - p_s(\sigma)$ zerstört:

$$p_s^c(\sigma) \geq 1 - p_c \cdot \frac{\delta(\sigma)}{m-1}$$

Ein Schema wird zerstört, wenn der Kreuzungspunkt zwischen der ersten und der letzten feststehenden Stelle im Chromosom gewählt wird. Mit einer (sehr geringen) Wahrscheinlichkeit bleibt jedoch ein Schema auch dann erhalten, falls die relevanten Abschnitte in beiden Elternchromosomen vorkamen und damit zwar zunächst getrennt, aber durch den Austausch wieder zum ursprünglichen Schema zusammengestellt werden (daher „$\geq$“). Wird das Chromosom des Kindes mit der Wahrscheinlichkeit p_m an genau einer beliebigen Stelle mutiert, so kann die Wahrscheinlichkeit $p_s^m(\sigma)$ dass das Schema σ die Mutation unbeschadet übersteht, wie folgt berechnet beziehungsweise wegen $p_m \ll 1$ abgeschätzt werden:

$$p_s^m(\sigma) = (1-p_m)^{o(\sigma)} \approx 1 - o(\sigma) \cdot p_m$$

Unter Berücksichtigung der Kreuzung und der Mutation der Individuen ergibt sich insgesamt für die erwartete Anzahl der Chromosomen in der nachfolgenden Generation, die ein bestimmtes Schema enthalten:

$$\xi(\sigma,t+1) \geq \xi(\sigma,t) \cdot \frac{f(\sigma,t)}{\overline{F(t)}} \cdot \left[1 - p_c \cdot \frac{\delta(\sigma)}{m-1} - o(\sigma) \cdot p_m\right]$$

Daraus kann das *Schema-Theorem* entnommen werden, wonach *kurze, überdurchschnittlich gute Schemata kleiner Ordnung eine exponentiell ansteigende Anzahl Repräsentanten in den jeweils nachfolgenden Generationen eines Genetischen Algorithmus erhalten*. Eine unmittelbare Folgerung aus diesem Theorem ist die *Building-Block-Hypothese* [Mic94]: *Ein Genetischer Algorithmus strebt durch Aneinanderreihung kurzer, qualitativ hochwertiger Schemata niedriger Ordnung nach einem Optimum*.

4.5.2 Evolutionäre Strategien

Nahezu zeitgleich mit den Genetischen Algorithmen entwickelte Ingo Rechenberg ein ähnliches Verfahren namens *Evolutionäre Strategie* (kurz *„ES“*) [Rec73], das sich nur im Detail davon unterscheidet. Beide Algorithmen basieren auf der Idee der Evolution, arbeiten mit einer größeren Population von Individuen und setzen die üblichen genetischen Operatoren Selektion, Rekombination und Mutation ein. Zwar haben sich die unterschiedlichen Varianten der Genetischen Algorithmen rasant verbreitet, während die Evolutionären Strategien eher ein Schattendasein führen, jedoch sind letztere für bestimmte Aufgaben dennoch besser geeignet [Bäc95].

Die ursprünglich 1973 vorgestellte Variante der Evolutionären Strategien ähnelte zunächst eher dem sogenannten Hillclimbing-Algorithmus (siehe Kapitel 4.3), jedoch waren Ablauf und Codierung bereits den typischen Evolutionären Verfahren gleich. Demnach bestand damals die gesamte Population aus lediglich einem Individuum. Diese Lösung wurde mehrfach einer Mutation unterzogen, wodurch zunächst eine Menge unterschiedlicher Nachfolger generiert wurde. Das beste Individuum dieser neuen Population wurde als Ausgangspunkt für die nächste Generation gewählt und ersetzte damit das bisherige Mutter-Individuum. Erst im Jahre 1981 führte Hans-Paul Schwefel auch die genetischen Operatoren Rekombination und größere Populationen ein.

Der Ablauf eines ES-Programms sieht dem des klassischen Genetischen Algorithmus aus Abbildung 4.3 sehr ähnlich. Da jedoch Evolutionäre Strategien ursprünglich zur Lösung hydrodynamischer Optimierungsaufgaben eingesetzt wurden [Sea93], codierte man dabei die Gene der Individuen üblicherweise als reell-wertige Vektoren anstatt binär. Im Gegensatz zu den Genetischen Algorithmen wird in der ursprünglichen Variante der Evolutionären Strategien eine rein zufällige, uniformverteilte Auswahl der Eltern durchgeführt. Die eigentliche Selektion anhand der Fitness eines Individuums findet erst unter den Nachkommen statt. Da es sich bei den Chromosomen um reell-wertige Vektoren handelt, besteht die Mutation darin, zu einzelnen Komponenten des Vektors jeweils eine normalverteilte Zufallszahl mit Erwartungswert $\mu = 0$ zu addieren. Dadurch, dass die Varianzen dieser Zufallsvariablen (eine für jede Komponente des Vektors) zusätzlich in den Chromosomen codiert werden, erhält jedes Gen eines Individuum auf natürliche Art eine selbst-adaptive Mutationsrate.

Ein weiterer wesentlicher Unterschied zu den Genetischen Algorithmen besteht darin, dass bei den Evolutionären Strategien zunächst mehr Nachkommen erzeugt werden, als die ursprüngliche Population Individuen hatte. Damit aber die neuen Generationen nicht zunehmend größer werden, findet bei der engültigen Zusammenstellung der neuen Population eine Selektion entsprechend der Fitness statt. Dabei gibt es unterschiedliche Varianten, darunter die folgenden zwei:

- Bei der sogenannten (μ, λ)-ES werden aus μ Eltern-Individuen jeweils λ Nachkommen ($\lambda \geq \mu$) mittels einer der klassischen Rekombinationsoperatoren erzeugt. Anschließend werden die μ besten Nachkommen ausgewählt, welche dann die Grundlage für die nächste Generation bilden. Man beachte dabei, dass bei dieser Variante ausschließlich innerhalb der Nachkommen-Population ausgewählt wird.

- Im Gegensatz dazu wird bei der sogenannten $(\mu + \lambda)$-ES die nachgeschaltete Selektion auf die μ Eltern ausgeweitet. Die zweite Variante ist dem „Steady-State“-Verfahren der Genetischen Algorithmen ähnlich und macht einen Elitismus-Operator überflüssig.

In den letzten Jahren haben sich Evolutionäre Strategien und Genetische Algorithmen gegenseitig befruchtet und somit nachhaltig angenähert, weshalb heute zwar oft nur noch von Genetischen Algorithmen berichtet wird, dabei meist aber ein Hybrid gemeint ist, welcher die Vorteile beider Verfahren in sich vereint. So sind die Genetischen Algorithmen schon lange nicht mehr auf eine Binärcodierung angewiesen. Auch die Selbstanpassung der Operatoren und ihrer Parameter, ursprünglich ein Mechanismus der Evolutionären Strategien, wurde erfolgreich auf Genetische Algorithmen übertragen (siehe Kapitel 4.5.3). Die Performance und Zuverlässigkeit Genetischer Algorithmen wurde zum Beispiel durch die Hybridisierung mit anderen Suchalgorithmen und die Nutzung domänenspezifischen Wissens ([WGO02]) ebenfalls nachhaltig verbessert.

4.5.3 Adaptive und selbstadaptive Verfahren

Die Robustheit[6] Evolutionärer Verfahren rührt hauptsächlich von der Einfachheit und Problemunabhängigkeit der genetischen Operatoren (Selektion, Rekombination und Mutation) her. Diese Algorithmen können auch bei Optimierungsaufgaben eingesetzt werden, zu denen nur wenig oder gar kein fachspezifisches Wissen zur Verfügung steht und für die daher kaum spezialisierte Suchverfahren existieren. Falls aber domänenspezifisches Wissen verfügbar ist, so kann diese Information in die Wahl einer geeigneten Codierung oder in die Implementierung sowie Parametrisierung der Operatoren einfließen [WGO02], um einen idealen Kompromiss zwischen Erkundung und Ausbeutung zu etablieren.

Für den Fall, dass vor der Ausführung eines Genetischen Algorithmus nur wenig oder gar kein problemabhängiges Wissen bekannt ist, gibt es diverse Methoden um die evolutionären Verfahren dynamisch an die jeweils aktuellen Anforderungen anzupassen, unter anderem zum Beispiel durch Auswertung des zwischenzeitlich über den Suchraum angesammelten Wissens.

Diese sogenannten *Adaptiven Evolutionären Verfahren (adaptive evolutionary computations)* können auf drei unterschiedlichen Ebenen beeinflusst werden [Ang95], je nachdem ob sich die

[6] Ausführliche Definition dieses Begriffs im Kontext Evolutionärer Verfahren: [Gol89]

anzupassende Parameterkonfiguration auf die gesamte Population, ein einzelnes Individuum oder lediglich einzelne Gene auswirkt:

Anpassung auf der Ebene der Population: Bei klassischen Evolutionären Verfahren gelten die Parameter Rekombinationswahrscheinlichkeit und Mutationsrate global für die gesamte Population. Eine Veränderung dieser Parameter wirkt sich stets auf jedes Individuum gleich aus. Die von Ingo Rechenberg theoretisch abgeleitete $1/5$-*Erfolgsregel (*$1/5$ *success rule)* besagt, dass der Quotient aus der Anzahl erfolgreicher Mutationen, die eine Verbesserung der Fitness bewirken, und der Anzahl aller durchgeführten Mutationen etwa ein Fünftel betragen sollte. Nach dieser Empfehlung kann das tatsächliche Verhältnis laufend bestimmt und entsprechend gesteuert werden: Ist der Quotient kleiner als $1/5$, so ist die Varianz der Zufallsvariablen bei der Mutation zu verringern; ist er hingegen größer als $1/5$, so sollte die Varianz auch entsprechend angehoben werden. Diese empirische Heuristik wird durch die Annahme begründet, dass sich die Verteilung der Nachkommen auf einen kleinen Bereich um die Eltern beschränkt hat, sobald das Verhältnis der erfolgreichen Mutationen größer als $1/5$ ist. Wird die Mutation aufgrund einer solchen Feststellung verstärkt, so wird diese Verteilung „aufgelockert“ und der Algorithmus neigt wieder mehr zur Exploration. Ist das Verhältnis hingegen kleiner als $1/5$, so sind die Nachkommen zu sehr im Suchraum verteilt und die Suche muss durch eine geringere Mutation stärker fokussiert werden. Ein anderer Ansatz besteht darin, die Chromosomen der Individuen im Falle einer binären Codierung jeweils um ein einziges Bit zu erweitern. Dieses Bit wird zwar nicht zur Ermittlung des Phänotyps und damit für die Fitness-Berechnung genutzt, ansonsten aber als regulärer Bestandteil des Genoms behandelt, also insbesondere auch der Mutation unterzogen. Relevant wird dieses Bit dann zum Beispiel in der Phase der Rekombination, wenn aufgrund des Verhältnisses der beiden Bitwerte dynamisch entschieden wird, welche Kreuzungsmethode jeweils anzuwenden ist. Beispielsweise könnte der Anteil der 1-Bits in der aktuellen Population als relative Wahrscheinlichkeit für die Anwendung des „uniform-crossover“- anstelle des „single-point-crossover“-Operators interpretiert werden.

Anpassung auf der Ebene der Individuen: Eine ähnliche Situation ergibt sich bei der Anpassung auf der Ebene einzelner Individuen. Im Gegensatz zu anderen Suchalgorithmen verarbeiten Evolutionäre Verfahren eine ganze Population verschiedener Individuen. Da die Lösungen daher im gesamten Suchraum verteilt sein können, liegt die Vermutung nahe, dass die Umgebung auch entsprechend von Individuum zu Individuum unterschiedlich ist. Deshalb kann es sinnvoll sein, die Parameter der verschiedenen genetischen Operatoren auf das jeweilige Individuum anzupassen. So kann beispielsweise das Chromosom eines Individuums stärker verändert werden (zum Beispiel durch eine höhere Mutationsrate oder -varianz), wenn sich dieses Individuum in einer relativ flachen Region des Suchraums befindet, also die Qualität der Lösungen in der Umgebung des betrachteten Individuum kaum variiert. In der Literatur findet sich eine ganze Reiher solcher Ansätze. Die Gemeinsamkeit aller Verfahren ist die individuelle Codierung der Parameter anpassungsfähiger Operatoren in das Chromosom jedes einzelnen Individuums. Im Falle Genetischer Algorithmen kann beispielsweise das binär codierte Chromosom des Phänotyps um drei zusätzliche Bit pro Gen erweitert werden, womit sich jeweils ein Parameterwert zwischen 0 und 7 darstellen lässt. Auch hier werden diese Bits nicht zur Berechnung der Fitness herangezogen, wohl aber der Kreuzung und Mutation unterzogen. Die Werte x_i dieser Parameter werden als relative Häufigkeit interpretiert, mit der bei „uniform-crossover“ eine Kreuzung am entsprechen-

den Lokus stattfindet. Demnach berechnet man die Wahrscheinlichkeit p_i, dass ein Crossover nach dem i-ten Gen stattfindet, zu:

$$p_i = \frac{x_i}{\sum_j x_j}$$

Anpassung auf der Ebene der Gene: Das Vorgehen bei der Modifikation der Operatorparameter auf Ebene der einzelnen Chromosom-Komponenten (Gene) entspricht dem auf der Ebene eines Individuums, wie vorhin beschrieben. Der Unterschied liegt lediglich darin, dass nun jedes Gen einen eigenen Parameter-Wert mit sich trägt und somit für jede Komponente eine individuelle Behandlung möglich ist. Ein anschaulicher Ansatzpunkt für diese Art der Anpassung ist die Mutation bei den Evolutionären Strategien. Dabei wird zu jedem reell-wertigen Gen jeweils ein zusätzlicher Wert mitgeführt (und natürlich auch der Evolution preisgegeben), welcher als Varianz der Zufallszahl interpretiert wird, die bei der Mutation zum aktuellen Allel hinzuaddiert wird.

Unabhängig davon, ob von einer Anpassung der Operatoren die gesamte Population gleichartig betroffen ist oder jedes einzelne Gen unterschiedlich behandelt wird, unterscheidet man die Methoden zur Adaptation Evolutionärer Verfahren hauptsächlich nach dem Zeitpunkt, zu dem die Art und Stärke einer Änderung festgelegt wird:

Random: Ein typischer Vertreter dieser Klasse ist der sogenannte *Meta-Operator*, welcher bei der Generierung optimaler Stundenpläne erfolgreich eingesetzt wurde [WGO02]. Dieser verfolgt die Entwicklung der Fitness des jeweils besten Individuums über mehrere Generationen hinweg. Wegen des eingesetzten Elitismus kann die beobachtete Qualität nur zunehmen oder konstant bleiben. Hat sich die Fitness des besten Individuums über eine bestimmte Anzahl von Generationen (die sogenannte *Startschwelle* des Meta-Operators) hinweg nicht geändert, so wird eine Rekonfiguration des Genetischen Algorithmus eingeleitet. Dabei wird jeder Freiheitsgrad, wozu insbesondere die Art der Selektion, der Crossovermechanismus und die zugehörige Kreuzungswahrscheinlichkeit sowie die Mutationswahrscheinlichkeit zählen, mit einer bestimmten Wahrscheinlichkeit völlig neu aus dem jeweils erlaubten Bereich gewählt. Allerdings wurde der Wertebereich der Wahrscheinlichkeiten auf eine Umgebung der empfohlenen Parameter eingeschränkt (siehe Seite 103). Anschließend wird der Genetische Algorithmus wie vorhin, jedoch mit den neuen Parametern fortgesetzt, insbesondere setzt die Überwachung erneut ein, so dass bei Bedarf eine erneute Reparametrisierung durchgeführt wird. Somit nutzt die zufällige Rekonfiguration keinerlei direkte Information über die Qualität der jeweils neu gewählten Parameter.

Statisch (a priori): Da der Suchraum für die Parameter eines evolutionären Verfahrens sehr groß sein kann, erscheint es angebracht, die Rekonfiguration nicht wahllos irgendeine Parametrisierung zusammenstellen zu lassen. Alternativ kann man die Erfahrung anwenden, die man in verschiedenen Versuchen mit Evolutionären Verfahren gesammelt hat. So wird zum Beispiel von erfolgreichen Experimenten berichtet [BBM93b], bei denen die Rekombinationswahrscheinlichkeit während des Programmablaufs linear oder exponentiell reduziert und die Mutationswahrscheinlichkeit gleichzeitig schrittweise erhöht wurden. Damit erzielt man eine ähnliche Wirkung

wie mit dem Temperaturverlauf bei Simulated Annealing (Kapitel 4.4). Da die Population während ihrer Evolution zunehmend konvergiert, besteht der Sinn dieser Rekonfiguration darin, der Mutation durch Reduktion der Crossover-Rate eine höhere Wahrscheinlichkeit zu geben, neue Variationen in die Population einzubringen. Allerdings ist diese Art der Reparametrisierung insofern „statisch“, als dass sie im Voraus festgelegt ist und deshalb auch nicht auf die tatsächliche Entwicklung der Generationen Rücksicht nimmt.

Dynamisch (absolut): Im Gegensatz zu den statischen Strategien basieren die dynamischen auf sogenannten *absolute Update-Regeln* [Ang95]. Dabei wird über eine bestimmte Anzahl Generationen hinweg eine Statistik über die aktuelle Entwicklung der Individuen geführt. Darauf aufbauend kann die Reduktion der Rekombinationswahrscheinlichkeit und Erhöhung der Mutationsrate anhand der statistischen Konvergenz der Lösungen ermittelt werden [BBM93b]. Ebenso können die bisher eingesetzten Operatoren hinsichtlich der jeweils durch sie erreichten Verbesserung der Fitness bewertet werden, so dass damit stets der beste Operator ausgewählt und eingesetzt werden kann. Die Bezeichnung „absolut“ spiegelt die Tatsache wider, dass diese Regeln deterministisch vorbestimmt sind - ihre „Dynamik“ beziehen sie aus Zeitpunkt und Ausmaß ihrer Anwendung. Ein Beispiel für ein solches Vorgehen wurde bereits unter der Bezeichnung $1/5$ *success rule* eingeführt. Diese Strategie läuft jedoch Gefahr, dass das evolutionäre Verfahren von einem lokalen Optimum angezogen wird, wodurch womöglich die falschen Operatoren belohnt werden.

Evolutionär: Die evolutionären Ansätze zur optimalen Parametrisierung genetischer Operatoren setzen den Grundgedanken Evolutionärer Verfahren konsequent fort und wenden diesen auf die Optimierung der Parameter selbst an. Dazu kann parallel zum Genetischen Algorithmus GA_1 der sich mit der Lösung des Hauptproblems befasst, noch ein zweiter Algorithmus GA_2 eingesetzt werden, dessen Aufgabe es ist, die Parameter des ersten kontinuierlich zu optimieren. Die Individuen der Population von GA_2 repräsentieren dabei die Parameter des Algorithmus GA_1. Um die Fitness jedes dieser Individuen zu bestimmen, parametrisiert man GA_1 mit den Daten des entsprechenden Individuums und evaluiert eine vorgegebene Anzahl davon generierter Problemlösungen. Diejenigen Parameter, die sich am besten bewährt haben, die also zur Erschaffung der besten Nachkommen beigetragen haben, erhalten somit die größte Fitness, weshalb sie in den nächsten Generationen vermehrt eingesetzt werden.

Leider hat die obige Strategie eine immense Belastung der Performanz zur Folge, denn es müssen gleichzeitig zwei evolutionäre Verfahren ausgeführt werden. Erweitert man jedoch die Chromosomen der Individuen des Hauptproblems um die Codierung der aktuellen Parameter, so genügt lediglich die Ausführung eines einzigen Genetischen Algorithmus. Vor der Generierung einer neuen Population können diese Parameter aus dem besten Individuum der Elterngeneration extrahiert werden, so dass die genetischen Operatoren diese neue Parametrisierung während der Erzeugung der gesamten Nachkommen-Population behalten. Der Vorteil dieses Verfahrens ist seine Effektivität gepaart mit seiner Einfachheit: Da die Parameter direkt in die Individuen hineincodiert sind, werden sie automatisch durch die Fitness-Berechnung des eigentlichen Individuums mitbewertet. Der Strategie liegt dabei die Annahme zugrunde, dass je besser ein

Nachkommen in der neuen Generation ist, um so besser die Operatoren gearbeitet haben müssen, aufgrund derer das Individuum entstanden ist. Aufgrund dieser eigenständigen Kontrolle und Rekonfiguration nennt man diese evolutionären Verfahren auch *selbst-adaptiv*.

4.5.4 Multi-objektive Optimierung

Betrachtet man erneut die in dieser Arbeit behandelte Generierung optimaler Testfallmengen, so gibt es für die einzelnen „Lösungen“ des Suchproblems zwei gegenläufige Bewertungsfunktionen, sogenannte *Objektive*: Die Größe der Testfallmenge einerseits und die von den Testfällen erreichte Überdeckung andererseits. Aus diesem Grund hat man für diese Klasse der Such- und Optimierungsaufgaben den Begriff *multi-objektiv* geprägt. Für nicht-triviale Testobjekte sind diese beiden Fitnessbeiträge insofern gegensätzlich, als dass man mit wenigen Testfällen grundsätzlich auch nur eine geringere Überdeckung erreichen kann als mit einer größeren Menge an Testfällen – dennoch ist eine kleine Testfallmenge nicht „besser“ oder „schlechter“, solange zum Beispiel das Verhältnis von Validierungsaufwand und Fehleraufdeckungsquote dem einer größeren Testfallmenge entspricht. Das Ziel einer multi-objektiven Optimierung ist es, alle Lösungen zu finden, welche hinsichtlich aller Objektiven optimal sind.

Bevor im Folgenden unterschiedliche Ansätze skizziert werden, sei die *multi-objektive Optimierungsaufgabe* in Anlehnung an [ZT98, BB00] allgemein und formal definiert. Dazu wird ein Optimierungsproblem mit n Bewertungskriterien betrachtet. Die Aufgabe lautet, alle globalen Optima von

$$f(a) := (f_1(a),\ f_2(a),\ \ldots,\ f_n(a))$$

zu finden, wobei $a \in \mathcal{S}$ eine potentielle Lösung aus dem Suchraum ist und $f : \mathcal{S} \mapsto \mathbb{R}^n$ gilt. Ohne Beschränkung der Allgemeinheit sei im Folgenden die Maximierung aller Teilbeiträge $f_i(a)$ angestrebt, wobei die Minimierung einzelner oder aller Teilbewertungen aufgrund des bereits genannten Zusammenhangs $min_{f_i(a)} \equiv -max_{-f_i(a)}$ trivial ist.

Definition 4.4 (Dominanz) *Eine potentielle Lösung $a_p \in \mathcal{S}$* dominiert *eine andere Lösung $a_q \in \mathcal{S}$ (dargestellt als $a_p \succ a_q$) dann und nur dann, wenn gilt:*

$$\forall i \in \{1,\ 2,\ \ldots,\ n\} : f_i(a_p) \geq f_i(a_q) \ \wedge\ \exists j \in \{1,\ 2,\ \ldots,\ n\} : f_j(a_p) > f_j(a_q).$$

Nach dieser Definition stellen diejenigen potentiellen Lösungen aus dem Suchraum $\mathcal{S}$, die von keiner anderen Lösung dominiert werden, selbstredend auch die *optimalen* Lösungen der Optimierungsaufgabe dar, denn schließlich sind sie hinsichtlich aller Bewertungskriterien mindestens so gut wie alle anderen und bezüglich mindestens einer am besten.

Definition 4.5 (Pareto-Front) *Sei $\mathcal{N} := \{a \in \mathcal{S} \mid \nexists \bar{a} \in \mathcal{S} : \bar{a} \succ a\}$ die Menge aller nicht-dominierten Elemente des Suchraums $\mathcal{S}$, dann sind die potentiellen Lösungen $a \in \mathcal{N}$* Pareto[7]-optimal *und bilden zusammen die sogenannte* Pareto-Front.

[7]Nach *Vilfredo Federico Damaso Pareto* (1848-1923). Er legte in seinem Buch *Manuale di economia politica* (1906) die Grundlagen der modernen sozialen Wirtschaft mit seinem Konzept vom sogenannten Pareto-Optimum. Darin behauptet er, dass die optimale Ressourcen-Verteilung in einer Gesellschaft nicht erreicht ist, solange man die Situation eines Individuums aus seiner Sicht verbessern kann, aber gleichzeitig die der anderen unverändert lässt.

Evolutionäre Verfahren sind für die Identifikation solcher Pareto-Fronten besonders gut geeignet, da sie für ihre Suche eine ganze Menge potentieller Lösungen (Population) gleichzeitig berücksichtigen. Eine übersichtliche Gegenüberstellung und vergleichende Bewertung der verschiedenen Strategien findet sich bei [ZT98]. Ehe komplexere Varianten Evolutionärer Verfahren entwickelt wurden, die die gesamte Pareto-Front mit nur einer einzigen Ausführung identifizieren, haben sich jedoch zunächst Ansätze zur sequentiellen Bestimmung einzelner Paretooptimaler Lösungen etabliert.

Multi-objektive Aggregation

Eine dieser intuitiven Strategien besteht darin, die verschiedenen Bewertungskriterien zu einer Fitness-Funktion zusammenzufassen [ZT98], wodurch wieder klassische Evolutionäre Verfahren angewandt werden können. Dabei muss vom Benutzer des Verfahrens jeder der *n* objektiven Bewertungsfunktionen $f_i(a)$ eine Gewichtung $w_i \in]0,\ 1[$ mit $\sum_i w_i = 1$ zugewiesen werden. Die gesamte Fitnessfunktion berechnet sich dann zu $f(a) = \sum w_i \cdot f_i(a)$, wobei die einzelnen Funktionen eventuell noch auf ein einheitliches Intervall normiert werden sollten.

Während bei der obigen Variante die gesamte Population zunehmend zu einer einzigen, gemäß der Gewichtung optimalen Lösung konvergiert, versuchen andere Ansätze die Verteilung der Population entlang der gesamten Pareto-Front zu stabilisieren. Vorausgesetzt wird dabei eine *Abstandsmetrik* $d : \mathcal{S}^2 \mapsto \mathbb{R}$, welche ein Maß für die „Ähnlichkeit“ beziehungsweise „Nähe“ zweier Elemente im Suchraum $\mathcal{S}$ darstellt. Diese Metrik sollte idealerweise die Phänotypen berücksichtigen, kann aber auch auf Ebene des Genotyps definiert werden. Im Falle binärcodierter Individuen, deren Phänotyp aufgrund des Gray-Codes entschlüsselt wird, kann zum Beispiel die euklidische Metrik auf den Phänotyp angewandt oder die Anzahl unterschiedlicher Bits im Genotyp herangezogen werden. Demnach weisen die drei Individuen $a_1 : (0,1,1,0)_{(Gray)} = 4_{(10)}$, $a_2 : (0,1,0,0)_{(Gray)} = 7_{(10)}$ und $a_3 : (1,0,1,1)_{(Gray)} = 13_{(10)}$ die phänotypischen (euklidischen) Abstände $d_p(a_1,a_2) = 3$, $d_p(a_1,a_3) = 9$ sowie $d_p(a_2,a_3) = 6$ beziehungsweise die genotypischen Abstände $d_g(a_1,a_2) = 1$, $d_g(a_1,a_3) = 3$ sowie $d_g(a_2,a_3) = 4$ auf.

Aufgrund einer solchen Abstandsmetrik kann ein weiterer Effekt aus der Natur in die Welt der Evolutionären Verfahren übertragen werden: Je dichter eine begrenzte Region (die sogenannte *Nische*) besiedelt ist, desto geringer sind die Überlebenschancen jedes einzelnen Individuums dieser Region, was typischerweise durch eine Verknappung der Ressourcen (Nahrung) bedingt wird. Die Folge ist, dass sich die Population weniger auf ein enges Gebiet beschränkt, sondern über ergiebigere Räume verteilt. Bei geeigneter Übertragung dieser Beobachtung auf die multiobjektive Optimierung verhindert man eine Konvergenz zu einem einzelnen Optimum.

Definition 4.6 (Sharing Function) *Sei $d_{ij} := d(a_i,a_j)$ ein Abstandsmaß für je zwei beliebige Elemente $a_i,a_j \in \mathcal{S}$ des Suchraums. Die sogenannte* Sharing Function *stellt eine normierende Abbildung dieser Abstandsmetrik wie folgt dar:*

$$sh(d_{ij}) = \begin{cases} 1 - \left(\frac{d_{ij}}{\sigma_s}\right)^{\alpha} & , \quad 0 \leq d_{ij} \leq \sigma_s, \\ 0 & , \quad sonst. \end{cases}$$

Der Parameter α beeinflusst den Grad, mit dem sich die Entfernung zweier Individuen auf ihre Zugehörigkeit zur gleichen Nische auswirkt. Wichtiger ist jedoch der sogenannte *Sharing Radius* σ_s als Maß für die Größe der zu berücksichtigenden Nische. Die beiden Parameter müssen vom Anwender eines entsprechenden Genetischen Algorithmus festgelegt werden und erfordern eine gewisse Kenntnis der Fitness-Landschaft [Hor97]. So sollte σ_s idealerweise den Mindestabstand zwischen zwei verschiedenen (lokalen) Extrema wiedergeben.

Definition 4.7 (Niche Count) *Sei $P = (a_1, a_2, \ldots, a_n) \in \mathcal{S}^n$ eine Population eines Evolutionären Algorithmus. Der sogenannte* „Niche Count"*-Faktor m_i des Individuums a_i ist ein Maß für die Dichte der Besiedelung des Suchraums in der Nähe von a_i durch die aktuelle Population und wird wie folgt errechnet [Mah95]:*

$$m_i = \sum_{j=1}^{n} sh(d_{ij})$$

Verwendet man als Basis für die Selektion anstelle der bereits vorgestellten, einfach aggregierten Fitness $f(a_i)$ die mit obigen Mitteln angepasste Fitness $f^*(a_i) = f(a_i)/m_i$ und berücksichtigt damit das jeweilige Ausmaß der Ausbeutung in der Umgebung des Individuums a_i, so werden Individuen „bestraft", die zu einem gemeinsamen Optimum konvergieren, wodurch sich die Population in den nächsten Generationen wieder entlang der Pareto-Front verteilt. Diese Fitnessumrechnung ist in der Literatur unter dem Begriff *Sharing* bekannt [DY96, Hor97, Mah95] und diente ursprünglich der Umsetzung einer multimodalen Optimierung.

Ein vergleichbares Verfahren berücksichtigt die reale Fitness bei der Auswahl der Elternindividuen nicht mehr direkt. Stattdessen wird eine „Ersatzfitness" als Kombination aus dem sogenannten *Dominanzmaß* $\mathcal{D}(a_i)$ des Individuums a_i und seines „Niche Count"-Faktors im Umfeld der aktuellen Population $P = (a_1, a_2, \ldots, a_n) \in \mathcal{S}^n$ bestimmt [BB00]. Ersteres berechnet sich zu:

$$\mathcal{D}(a_i) := \sum_{j=1}^{n} nd(a_i, a_j) \quad wobei \quad \forall i, j : nd(a_i, a_j) := \begin{cases} 1 & falls\ a_j \succ a_i; \\ 0 & sonst. \end{cases}$$

Somit ist $\mathcal{D}(a_i)$ die Anzahl aller Individuen der aktuellen Population, die das betrachtete Individuum a_i dominieren. Die zu minimierende Ersatzfitness $\mathcal{F}(a_i)$ eines Individuums a_i berechnet sich dann zu:

$$\mathcal{F}(a_i) := (1 + \mathcal{D}(a_i))^{\beta} \cdot (1 + \mathcal{N}(a_i)) \quad wobei \quad \mathcal{N}(a_i) := m_i = \sum_{j=1}^{n} sh(d_{ij}),\ d_{ij} = d(a_i, a_j)$$

wobei der Parameter β zur relativen Gewichtung des Dominanzmaßes im Verhältnis zum „Niche Count"-Faktor dient und typischerweise den Wert 1 annimmt. Man beachte, dass $\mathcal{F}(a_i)$ grundsätzlich zu minimieren ist – ob eine Objektive tatsächlich maximiert oder minimiert werden soll, wird hierbei indirekt über die Festlegung der Dominanzrelation gesteuert.

Vector Evaluated Genetic Algorithm (VEGA)

Einen Ansatz zur multi-objektiven Optimierung ohne Aggregation aller Bewertungsfunktionen zu einer einzigen Fitness präsentierte David Schaffer bereits im Jahre 1985 unter dem Namen

Vector Evaluated Genetic Algorithm (VEGA) [Sch85, ZT98]. Der Genetische Algorithmus in VEGA arbeitet generationenbasiert und mit einem „Mating Pool“ (siehe Seite 100), welcher zunächst alle ausgewählten Eltern aufnimmt, ehe diese zur Rekombination gelangen. Der Mating Pool wird bei VEGA jedoch zusätzlich entsprechend der Anzahl der Bewertungskriterien in n gleich große Partitionen unterteilt, wobei die i-te Untermenge nur mit Individuen aufgefüllt wird, welche lediglich anhand der Bewertungsfunktion f_i mittels „Roulette Wheel Selection“ ausgewählt wurden. Anschließend werden die Individuen aus diesem Mating Pool nach dem Zufallsprinzip (uniform verteilt) selektiert und in gewohnter Weise den genetischen Operatoren Kreuzung und Mutation zugeführt.

Obwohl Schaffer in seiner Veröffentlichung von großen Erfolgen mit diesem Verfahren berichtet, deckten darauf aufbauende empirische Studien auf, dass VEGA lediglich zu extremen Grenzpunkten an der Pareto-Front konvergiert, anstatt die gesamte Front zu besetzen. Eine mögliche Erklärung für dieses Verhalten ist, dass VEGA nicht nach einem Kompromiss zwischen den einzelnen Bewertungsfunktionen auswählt, sondern nur nach dem Maximum jeweils eines Kriteriums.

Niched Pareto Genetic Algorithm (NPGA)

Aufbauend auf VEGA und diverser Erweiterungen sowie entsprechender Studien haben Horn und Nafpliotis den sogenannten *Niched Pareto Genetic Algorithm (NPGA)* entwickelt [Hea94]. Da bei klassischer „Tournament Selection“ (siehe Seite 99) eine Untermenge der Individuen aus der aktuellen Population gewählt wird und das der Fitness nach stärkste Individuum dieser Menge zur Reproduktion gelangt, konvergiert die gesamte Population zügig zu einem Extremum. Bei der Auswahl im NPGA handelt es sich um eine Anpassung der Tournament Selection, bei der zunächst zwei beliebige Individuen $a_i \in P$ und $a_j \in P$ zufällig nach einer uniformen Verteilung aus der aktuellen Population gewählt werden. Darüber hinaus wird zusätzlich eine Vergleichsmenge $C \subseteq P$ der Kardinalität $t_{dom} = |C|$ ebenfalls zufällig bestimmt. Anschließend werden a_i und a_j mit jedem Individuum $a_c \in C$ bezüglich der Dominanzrelation verglichen. Wird eine der beiden Lösungen von der Vergleichsmenge dominiert und die andere nicht, so wird das nichtdominierte Individuum zur Reproduktion zugelassen. Wenn beide oder keines der Individuen dominiert wird, so wird eine Variante des „Sharing“ namens *Equivalence Class Sharing* [Hea94] angewandt, um den Gewinner im direkten Vergleich festzulegen.

Die Kardinalität t_{dom} der Vergleichsmenge erlaubt einen gewissen Einfluss auf den Selektionsdruck beziehungsweise den Dominanzdruck. Die Performanz des Algorithmus reagiert jedoch relativ empfindlich auf ein unangemessenes Verhältnis zwischen Dominanzdruck und Sharing-Druck. Aufgrund ihrer Versuchsreihen empfehlen die Autoren $t_{dom} \approx 10\% \cdot |P|$, also die Größe der Vergleichsmenge auf etwa ein Zehntel der Populationsgröße zu beschränken.

Nondominated Sorting Genetic Algorithm (NSGA)

Eine andere Variante eines multi-objektiven Optimierungsverfahrens namens *Nondominated Sorting Genetic Algorithm (NSGA)* wurde 1994 von Srinivas und Deb [SD94] vorgestellt. Während VEGA und NPGA die Selektionsverfahren variieren, beruht NSGA auf einer Fitnesstransforma-

tion namens *Pareto-ranking* und den sich anschließenden klassischen Verfahren zur Selektion, Kreuzung und Mutation.

Abbildung 4.8 zeigt ein Struktogramm des „Pareto-ranking“. Anschaulich entspricht das Vorgehen dem „Abschälen“ einzelner Pareto-Fronten aus der ursprünglich vollständigen Population. Dabei wird allen Individuen einer „Schale“ zunächst die gleiche Fitness zugewiesen, welche anschließend jedoch noch einem Sharing unterzogen wird, um eine Konvergenz des gesamten Verfahrens zu einem einzigen Optimum zu verhindern. Bei der Fitnessvergabe ist lediglich zu beachten, dass am Ende des Pareto-rankings jedes Individuum weiterhin eine größere Fitness haben muss als alle von ihm dominierten Individuen. Die Fitness der „äquivalenten“ Individuen innerhalb einer Schale variiert proportional zu ihrem „Niche Count“-Faktor (siehe Definition 4.7) und berücksichtigt somit die Dichte der Besiedelung in der Umgebung des jeweiligen Individuums.

Pareto-Ranking — Grundverfahren

Sei $P_{aktuell} \leftarrow P$ die zunächst vollständige Population und $f_{aktuell} \leftarrow FITNESS_{max}$ die zunächst maximale künstliche Fitness
Wiederholen bis $P_{aktuell} = \emptyset$
Identifiziere die Menge aller nicht-dominierten Individuen: $P_{nd} = \{a \mid \nexists \bar{a} \in P_{aktuell} : \bar{a} \succ a\} \subseteq P_{aktuell}$
Extrahiere alle nicht-dominierten Individuen aus der aktuellen Population: $P_{aktuell} \leftarrow P_{aktuell} \setminus P_{nd}$
Weise jedem Individuum aus P_{nd} die gleiche Fitness $f_{aktuell}$ zu: $\forall a \in P_{nd} : f(a) \leftarrow f_{aktuell}$
Transformiere die Fitness der Individuen aus P_{nd} gemäß einer Sharing-Strategie bezüglich der gesamten Population P.
Identifiziere das Individuum $a_{min} \in P_{nd}$ mit der kleinsten Fitness: $\forall \bar{a} \in P_{nd} : f(a_{min}) \leq f(\bar{a})$
Wähle die Fitness der nächsten Schale etwas kleiner als die kleinste Fitness aller Individuen aus P_{nd}: $f_{aktuell} \leftarrow f(a_{min}) - \varepsilon$

Abbildung 4.8: Struktogramm des *Pareto-Ranking*

Die in [ZT98] durchgeführten Vergleiche der verschiedenen Verfahren haben ergeben, dass NSGA in nahezu allen Versuchsreihen weitaus besser als die anderen vorgestellten Algorithmen war - mit Ausnahme des von [BB00] vorgestellten Verfahrens der Aggregation, welches nicht Gegenstand der Experimente war.

4.5.5 Weitere Varianten Evolutionärer Verfahren

Ausgehend von den klassischen Genetischen Algorithmen wurde eine Vielzahl unterschiedlicher Varianten entwickelt. Die Triebfeder dieser Entwicklung besteht einerseits aus der Erschließung neuer Einsatzgebiete für die Verwendung Evolutionärer Verfahren und andererseits aus der Perfektionierung dieser Methoden für dedizierte Aufgabenstellungen. Zwei dieser Ansätze wurden

in Kapitel 4.5.3 und Kapitel 4.5.4 ausführlich vorgestellt, da sie im Kontext dieser Arbeit Verwendung finden. Die beiden im Folgenden skizzierten Varianten haben eine große Verbreitung erfahren, tragen aber zur automatischen Testdatengenerierung im Rahmen dieser Arbeit nicht bei.

Multimodale Optimierung

Einfache Genetische Algorithmen berücksichtigen nur eine Objektive und streben im Laufe ihrer Ausführung die Konvergenz aller Individuen zum globalen Optimum an. Multi-objektive Strategien betrachten mehrere Bewertungsfunktionen und haben die Identifikation und gleichförmige Besetzung der Pareto-Front durch alle Individuen der Population zum Ziel. Oftmals enthalten Suchräume mit nur einer Bewertungsfunktion jedoch sogenannte *Multimodalitäten*. Solche Fitness-Funktion weisen mehrere lokale oder gar globale Maxima auf, weshalb man sie *multimodal* nennt. Für viele Optimierungsaufgaben ist die Bestimmung aller oder zumindest einer vorgegebenen Anzahl dieser Extrema von Interesse. Auch in diesem Fall stand die Natur Pate für die Entwicklung multimodaler Optimierungsstrategien [BBM93b, MS96], davon insbesondere der Prozess der *Artenbildung (speciation)* zur Besetzung unterschiedlicher ökologischer Nischen – man spricht dabei von der Aufrechterhaltung einer *Vielfältigkeit (diversity)* in der Population.

Booker [Boo85] verwendet die sogenannte *eingeschränkte Paarung (Restricted Mating)*, bei der nur Eltern gekreuzt werden, die einander ähnlich sind. Dabei hatte seine Beispielanwendung jedoch den Vorteil, dass die Zugehörigkeit der Individuen zu den Nischen diskret und damit eindeutig war, was im Allgemeinen nicht der Fall ist.

Cavicchio [GR87] führte den Mechanismus der *Vorauswahl (Preselection)* ein. Dabei wird in der Art des Steady-State-Verfahrens derjenige Elternteil[8] mit der geringsten Fitness durch sein Kind ersetzt, vorausgesetzt das Kind weist selbst eine höhere Fitness auf. Die Aufrechterhaltung der Vielfältigkeit in der Population liegt darin begründet, dass die Kinder den Eltern meist ähnlich sind, weshalb nur innerhalb einer Nische konkurriert wird.

Die von De Jong schon im Jahre 1975 vorgeschlagene Strategie aus der Klasse der sogenannten *Crowding*-Verfahren [MS96] ähnelt der „Preselection“ von Cavicchio, empfiehlt jedoch in einem Genetischen Algorithmus mit steady-state-Strategie dasjenige Individuum der alten Generation durch das Kind zu ersetzen, welches dem letzteren am ähnlichsten ist. Um das neue Individuum nicht mit jeder potentiellen Lösung der alten Generation vergleichen zu müssen, kann ein „Sampling“ angewandt werden. Dabei wird ein Anteil der Größe *CF* (*crowding factor*) aus der aktuellen Population zufällig gewählt und das ähnlichste Individuum daraus wird ersetzt.

Allerdings weist das „Crowding“ zwei Nachteile auf. Die gesamte Optimierung läuft verlangsamt ab, da eine zügige Konvergenz gezielt verhindert wird. Dennoch hat sich herausgestellt, dass das Verfahren unabhängig von der Komplexität der Bewertungsfunktion und damit des Suchraums nicht in der Lage ist, mehr als zwei Extrema auf Dauer zu behalten. Wird der Programmlauf nicht rechtzeitig abgebrochen, können weder Crowding noch Preselection mehrere Nischen aufrechterhalten. Einen Ausweg stellt eine Alternative namens *Deterministic Crowding* dar [Mah95], bei der alle Individuen der aktuellen Population genau einmal zur Reproduktion

[8] Beim klassischen Steady-State wird das schwächste Individuum der gesamten Population ersetzt (Seite 94).

zugelassen werden. Um dennoch einen Selektionsdruck einzuführen, ersetzt ein Nachkommen ein Elternteil nur dann, wenn das Kind eine bessere Fitness aufweisen kann. Eine weitere Variante dieses Verfahrens findet sich in der Literatur unter dem Namen *Probabilistic Crowding* [MG99].

Eine der leistungsfähigsten Erweiterungen Genetischer Algorithmen für die multi-objektive Optimierung ist das sogenannte *Sharing*, welches bereits 1987 von Goldberg und Richardson umgesetzt wurde [DY96, Hor97, Mah95] und dessen Grundlagen im Kapitel 4.5.4 mit Definition 4.6 (Sharing Function *sh*) und Definition 4.7 (Niche Count m_i) vorgestellt werden. Bei diesem Verfahren wird die Fitness jedes Individuums a_i nach der Vorschrift $f^*(a_i) = f(a_i)/m_i$ umgerechnet, ehe eines der klassischen Selektionsverfahren angewandt wird. Ziel dabei ist es, die Anzahl der Individuen, die sich in einer Nische einfinden, auf die Tragfähigkeit dieser Nische zu begrenzen. Dazu ordnet man jeder Nische eine feste *Belohnung (payoff)* in Abhängigkeit von der Qualität der sie repräsentierenden Individuen zu [Mah95], welche äquivalent zur Fitness im Sinne der Evolutionären Verfahren ist. Sobald sich Individuen in einer Nische niederlassen, müssen sie diese Belohnung mit allen anderen Anwesenden in der Nische teilen. Somit entsteht nach einer gewissen Zeit ein stabiles Gleichgewicht, in dem jede Nische proportional zu ihrer Belohnung besetzt ist. Sobald eine Nische übervölkert wird, ist es für die ansässigen Individuen von Vorteil (reduzierte Fitness in der Nische), wenn sie weniger belebte Nischen aufsuchen.

Diese einfache Art des Sharing ist zwar durchaus in der Lage, mehrere oder alle Extrema zu identifizieren, bedauerlicherweise kommt es jedoch nach der ersten Konvergenz zu einer Fluktuation in der Population. Ursache dafür ist die Tendenz aller Individuen, sich voreilig auf einem globalen Optimum niederzulassen. In den folgenden Generationen wird die Fitness durch das Sharing jedoch wieder verringert, da die Nischen übervölkert sind, wodurch der Selektionsdruck sinkt und die Individuen wieder andere Positionen im Lösungsraum aufsuchen. Um dem jedoch entgegenzuwirken und die Stabilität des Sharing zu erhöhen, wurde 1991 eine neue Variante namens *Continuously Updated Sharing* entwickelt [Oea91]. Der Unterschied zur klassischen Methode besteht darin, dass nicht erst die gesamte neue Population erstellt und anschließend die Fitness reduziert wird. Stattdessen wird diese „Shared Fitness“ schon während des Aufbaus der nächsten Generation als Maß gewählt, wobei sie in Relation zu der noch unvollständigen neuen Population berechnet wird.

Das sogenannte *Dynamic Shared Niching* greift die Idee des Sharing auf, um die einzelnen Nischen dynamisch zu bestimmen, kommt jedoch mit einer weitaus geringeren Rechenzeitkomplexität aus [MS96]. Bei der sogenannten *Greedy Dynamic Peak Identification* wird eine zunächst leere Optima-Menge *DPS* (*dynamic peak set*) nach und nach bis zu einer vorgegebene Größe q befüllt. Dazu wird jedes Individuum der aktuellen Population mit den Individuen in *DPS* verglichen und darin aufgenommen, falls es nicht innerhalb eines Sharing Radius σ_s um irgend ein anderes Individuum aus *DPS* liegt, womit es ein eigenes potentielles Extremum darstellt.

Alle Sharing-Varianten weisen drei gravierende Nachteile auf [DY96]. Zum einen ist der Sharing Radius σ_s konstant, weshalb für eine erfolgreiche Optimierung alle Extrema im Suchraum nahezu äquidistant sein müssen. Die Wahl eines geeigneten Wertes für σ_s erfordert Kenntnisse über die Abstände der Extrema im Lösungsraum. Bedingt durch den ausführlichen Vergleichsschritt jedes Individuums mit allen anderen einer Population weisen diese Algorithmen eine hohe Laufzeitkomplexität ($> O(n^2)$, n ist die Populationsgröße) auf.

Mit dem *Sequential Niching* [Bea93] gibt es eine ebenso einfache wie intuitive Lösung zur sequentiellen Bestimmung aller Extrema durch wiederholte Ausführung eines Evolutionären Verfahrens. Dazu werden alle bereits gefundenen Optima von der Fitnesslandschaft der nachfolgenden Ausführungen entfernt. Bei Sequential Niching wird dies entsprechend dem klassischen Sharing dadurch erreicht, dass die Fitness jedes aktuellen Individuums indirekt proportional zu seiner Entfernung von jeder bisher bekannten Lösung verringert wird. Bedauerlicherweise erfordert auch dieses Vorgehen eine gute Abschätzung der Breite eines Optimums. Unterschätzt man diese, so erhalten Individuen am Fuße des Maximums eine relativ hohe Fitness, da sie weit genug von der eigentlichen Spitze entfernt sind, wodurch „virtuelle“ Extrema entstehen können. Überschätzt man die Breite, so wird ein eventuell benachbartes Optimum in der Fitnesslandschaft ebenfalls unterdrückt. Künstliche Maxima können unter Umständen aufgrund der Beobachtung identifiziert und eliminiert werden, dass sie meist am Rande der Nische (entsprechend dem Sharing Radius) eines bekannten Hauptmaximums auftreten [Bea93]. Darüber hinaus kann eine lokale Suche (zum Beispiel mittels Hillclimbing) klären, ob eine identifizierte Lösung ein tatsächliches Maximum ist oder auf das benachbarte Extremum zurückgeführt werden kann.

Eine jüngere Variante der multimodalen Optimierung, das sogenannte *CoEvolutionary Shared Niching (CSN)*, basiert auf den Prinzipien der freien Marktwirtschaft getreu dem Motto „Angebot und Nachfrage regeln den Preis“ und simuliert einen monopolistischen Wettbewerb [GW98]. Dazu wird die Population in die Menge K der „Käufer“ und die Menge V der „Verkäufer“ unterteilt, wobei die Positionen der „Verkäufer“ im Lösungsraum auf die zu suchenden Extrema schließen lassen. Den Käufern und Verkäufern wird erlaubt sich in der Suchlandschaft so zu verteilen, dass sie ihren jeweiligen Profit maximieren. Käufer $k \in K$ wird *bedient von* oder *gehört zu* Verkäufer $v \in V$ wenn v der zu k nächstgelegene Verkäufer ist. Sei K_v die Menge derjenigen Käufer, die von Verkäufer v in der aktuellen Population bedient werden und deren Anzahl $m_v = |K_v|$ die Kardinalität der Menge ist.

Die an der individuellen Fitness der Käufer vorgenommene Transformation ähnelt dem klassischen Sharing, jedoch tritt hier die Anzahl der anderen Käufer, die jeweils vom gleichen Verkäufer bedient werden, anstelle des *Niche Count*: $\forall v \in V, k \in K_v : f^*(k) := f(k)/m_v$. Die Fitness der Verkäufer kann dann beispielsweise durch die Summe der ursprünglichen Fitnesswerte der eigenen Käufer ausgedrückt werden:

$$\phi(v) = \sum_{k \in K_v} f(k).$$

In der einfachen Variante des CSN wird die Population der Käufer nach den gewohnten Regeln der evolutionären Verfahren verarbeitet. Bei der Evolution der Verkäufer wurde hingegen auf die Rekombination verzichtet und die Mutation modifiziert. Dabei wird zunächst das Chromosom eines Verkäufers v_a an einer beliebigen Stelle mutiert. Der so entstandene neue Verkäufer v_n ersetzt nur dann v_a, wenn die Fitness von v_n besser als die von v_a ist sowie der Abstand zwischen v_n und jedem anderen Verkäufer mindestens eine vorgegebene untere Schranke d_{min} übersteigt. Falls innerhalb einer vorgegebene Anzahl Iterationen kein geeigneter Verkäufer durch Mutation von v_a erzielt wurde, so wird v_a unverändert beibehalten. Auf diese Art verfolgt man das Ziel, dass sich die Verkäufer auf die $|V|$ besten Optima verteilen, welche mindestens d_{min} voneinander entfernt sind.

Experimentelle Untersuchungen mit relativ einfachen multimodalen Funktionen haben vielversprechende Ergebnisse gezeigt, jedoch versagte der Algorithmus auf schwierigen, massiv multimodalen Suchräumen. Eine verbesserte Variante nutzt das sogenannte *Imprint* [GW98]. Damit ist eine Übertragung der besten Käufer in die Menge der Verkäufer gemeint, vorausgesetzt die beiden genannten Bedingungen (bessere Fitness und ausreichender Abstand) bleiben erfüllt. Um den enormen Overhead des Imprints zu verhindern, werden nicht alle Käufer als potentielle Verkäufer betrachtet. Stattdessen wird jeweils genau ein Käufer aus der Kundenmenge eines Verkäufers zufällig gewählt. Wenn dieser Käufer ein besserer Verkäufer als der bisherige wäre und der Mindestabstand zu den anderen Verkäufern bleibt gewahrt, dann wird der aktuelle Verkäufer durch den betrachteten Käufer ersetzt.

Fuzzy Genetic Algorithms

Als Teilgebiet der Künstlichen Intelligenz hat die *Fuzzy Logic* in den letzten Jahren eine rasante Verbreitung erfahren. Um für dieses breite Einsatzgebiet ebenfalls Evolutionäre Verfahren zur Optimierung einsetzen zu können, wurden die Heuristiken mit der Fuzzy Logik hybridisiert. Da *Fuzzy Genetic Algorithms* im Rahmen dieser Arbeit nicht von Belang sind und die Theorie der Fuzzy Logik selbst schon gar nicht, sei an dieser Stelle für weitergehende Betrachtungen auf die entsprechende, mittlerweile sehr umfangreiche Literatur verwiesen [KS93, Jan98]. Für eine Kombination Evolutionärer Verfahren und Fuzzy-Logik gibt es unterschiedliche Ansätze:

Fuzzy-basierte Operator-Parameter: Die Parameter der genetischen Operatoren können mit Hilfe der Fuzzy-Logik implementiert werden, was dem Benutzer Evolutionärer Verfahren eine genaue Spezifikation abnimmt und eine einfachere Anpassung ermöglicht, ohne dass detaillierte Kenntnisse erforderlich sind. Auch die Selbstadaptation der Operatoren kann auf diese Art und Weise mit „unscharfen“ Regeln beschrieben werden [HL96]. Im Falle einer reellen Codierung der Chromosomen mit mehreren Genen $(g_1,\ldots,g_m) \in \mathbb{R}^m$ sei $[a_i,\ b_i]$ der gültige Wertebereich des Gens g_i. Nach der Selektion zweier Individuen gilt es, ihre Gene $X = (x_1,\ldots,x_n)$ und $Y = (y_1,\ldots,y_n)$ zu kreuzen. Bei der klassischen Rekombination würde das Chromosom des Kindes abwechselnd Gene aus X beziehungsweise Y erhalten, zum Beispiel $K = (y_1,x_2,x_3,\ldots,y_n)$. Um einen leistungsfähigeren Kreuzungsoperator zu erhalten, kann man den möglichen Wertebereich des $i$ten Gens im Kindchromosom in drei Intervalle $[a_i,\ g_i^{min}]$, $[g_i^{min},\ g_i^{max}]$ und $[g_i^{max},\ b_i]$ unterteilen, dabei sei $g_i^{min} := min(x_i,y_i)$ und $g_i^{max} := max(x_i,y_i)$. Die äußeren Intervalle $[a_i,\ g_i^{min}]$ und $[g_i^{max},\ b_i]$ sind sogenannte „Erkundungs-Bereiche“, während das innere Intervall $[g_i^{min},\ g_i^{max}]$ einen „Ausbeutungs-Bereich“ darstellt. Aufbauend auf diese Betrachtungen präsentiert [HL96] verschiedene Fuzzy-Abbildungen, die eine Rekombination durch Neuwahl der Gene aus den entsprechenden Intervallen in Abhängigkeit vom gewünschten Grad der Erkundung beziehungsweise Ausbeutung darstellen.

Optimierung der Fuzzy-Domäne: Evolutionäre Algorithmen können als leistungsfähige Heuristiken zur automatisierten Erstellung und Optimierung von Fuzzy-Logik-Systemen eingesetzt werden. Damit können zum Beispiel die Zugehörigkeitsfunktionen aufgestellt und optimiert oder das Inferenzsystem eines Fuzzy-Logik-Controllers maschinell „erlernt“ werden. Dazu ermittelt man zunächst eine Menge von Musterentscheidungsfällen und überlässt einem Genetischen Algorithmus die Suche nach derjenigen Regelbasis, die diese Beispiele am besten erfüllt

[BK95, HL96, HM96, HLV93].

Fuzzy Optimierungsaufgaben: Von besonderem Interesse ist der Einsatz Evolutionärer Verfahren zur Lösung von Optimierungsaufgaben, welche ihrerseits auf der Fuzzy-Logik basieren, also die Anwendung Genetischer Algorithmen auf „unscharfe“ Suchräume [Ost01a]. Die Repräsentation der Chromosomen stellt weiter keine Hürde dar, denn man kann jeder Fuzzy-Menge entsprechende Kennzahlen zuordnen und sie somit im Gen verschlüsseln. Alternativ ist ebenso auch eine Codierung der Zugehörigkeit eines bestimmten Wertes möglich. Schwieriger gestaltet sich aufgrund der Fuzzifizierung hingegen die Modellierung der genetischen Operatoren und insbesondere die Bestimmung einer aussagekräftigen Fitness. Doch dazu gibt es mittlerweile vielversprechende Ansätze [HLV94].

Einer dieser Lösungsvorschläge besteht im Einsatz der sogenannten *fuzzy min-max-recombination* [Voi92, HL96], bei der für die Erzeugung der Nachkommen die Durchschnitts- und Vereinigungsoperatoren auf Basis der T-Norm und T-Conorm der Fuzzy-Mengen-Theorie eingesetzt werden. Erwähnenswert ist die bei diesem Verfahren eingesetzte Nachbildung der sogenannten *Epistasis* – ein Effekt der Genetik, bei dem eine phänotypische Eigenschaft nicht nur von einem einzigen Gen abhängt, sondern auch von weiteren Genen an anderer Stelle im Chromosom mitbestimmt wird.

Das Problem der Fuzzy-Fitness kann durch eine Abbildung der zugehörigen Fuzzy-Menge F auf exakte reelle Werte mittels einer „Defuzzifizierungs“-Funktion $d : F \mapsto \mathbb{R}$ gelöst werden [HLV94]. Die Auswahlwahrscheinlichkeit bei Roulette Wheel Selection berechnet sich dann zu:

$$p_s(a_i) = \frac{d(f(a_i))}{d(\sum_{j=1}^{n} f(a_j))},$$

wobei durch die Wahl von d sichergestellt werden muss, dass folgende Zusammenhänge gelten:

1. $\sum_{i=1}^{n} d(f(a_i)) = d(\sum_{j=1}^{n} f(a_j))$ und somit $\sum_{i=1}^{n} p_s(a_i) = 1$, damit wieder eine Wahrscheinlichkeitsverteilung erreicht wird,

2. $f(a_k) \leq f(a_l) \Leftrightarrow d(f(a_k)) \leq d(f(a_l))$, damit dem stärkeren Individuum auch die größere Fitness und somit die größere Selektionswahrscheinlichkeit zugeordnet wird, also weiterhin $f(a_k) \leq f(a_l) \Rightarrow p_s(a_k) \leq p_s(a_l)$ gilt.

Für die Wahl einer solchen Funktion d bieten sich sogenannte *fuzzy numbers ranking procedures (FNRP)* an [HLV94], die auf den Eigenschaften der α-*cuts* [Bez93] beruhen. Eine weitere Herangehensweise findet sich bei [PR96], deren Formalismus relativ aufwendig ist, aber eine interessante und verständliche Alternative darstellt.

Kapitel 5

Anwendung der Suchheuristiken zur automatischen Testdatengenerierung

„Testing can show the presence of bugs,
not their absence.“
Edsger Wybe Dijkstra (1930-2002), University of Texas

Kern der vorliegenden Arbeit ist es, zwei zunächst unabhängige Fachgebiete der Informatik zusammenzubringen: Software Engineering und Artificial Intelligence. Deshalb wurden zunächst im Kapitel 3 die Grundlagen des Softwaretestens vorgestellt und dabei insbesondere das strukturelle Testen beleuchtet, das gerade hinsichtlich der Identifikation geeigneter Testdaten Nachholbedarf aufweist. Kapitel 4 beschäftigt sich mit den unterschiedlichen Such- und Optimierungsheuristiken der künstlichen Intelligenz, allen voran mit den *multimodalen* Evolutionären Verfahren. In diesem Kapitel wird aufgezeigt, wie diese Heuristiken zur automatischen Generierung und Optimierung struktureller Testdaten eingesetzt werden können.

Bisherige Ansätze betrachten diese beiden Teilaufgaben meist als voneinander unabhängig [BHJT00]. Dazu generieren sie zunächst eine Vielzahl unterschiedlicher Testfälle, welche bezüglich des zugrunde liegenden Testkriteriums zum Teil insofern redundant sind, als dass der Überdeckungsgrad selbst nach dem Verwerfen einiger Testfälle aufrechterhalten werden kann. Anschließend wird eine spezialisierte Heuristik angewandt [TG05], um möglichst viele „überflüssige“ Testfälle aus der ursprünglich ermittelten Testmenge wieder zu verwerfen [Ton04] - eine Aufgabe, die im Allgemeinen selbst als NP-vollständig gilt.

Im Unterschied dazu besteht der im Rahmen dieser Arbeit präsentierte Ansatz in einer Kombination beider Aspekte und ermöglicht die unmittelbare Generierung bereits optimaler (genauer: minimaler) Testdatensätze mittels multimodaler Evolutionärer Verfahren. Es besteht im Wesentlichen aus zwei Teilstrategien, welche im Folgenden als *globale Optimierung* beziehungsweise *lokale Optimierung* bezeichnet werden. Die globale Optimierung verfolgt dabei gleichzeitig beide Ziele: Die Generierung zusätzlich notwendiger Testdaten und die Reduktion der Testmenge um unnötige Testfälle. Aufgabe der lokalen Optimierung hingegen ist die Unterstützung der globalen Optimierung bei der Ermittlung „schwieriger“ Testfälle, also solcher, die lediglich mit einer geringen Wahrscheinlichkeit durch die Heuristik der globalen Optimierung innerhalb ak-

zeptabler Zeit entdeckt werden. Die lokale Optimierung selbst ist nicht an der Identifikation und Verwerfung redundanter Testfälle beteiligt, weshalb das übergeordnete Ziel bereits mit der globalen Optimierung alleine erreicht werden kann.

Anmerkung: Die im Folgenden vorgestellten Konzepte wurden exemplarisch in einem Werkzeug namens •gEAr[1] für die Programmiersprache JAVA™ umgesetzt. Um die theoretischen Aspekte des Verfahrens mit anschaulichen Beispielen zu beleuchten, wird daher an geeigneten Stellen die tatsächliche Umsetzung in •gEAr für JAVA™ erläutert. Damit der Umfang der Betrachtungen den Rahmen dieser Arbeit nicht sprengt, werden die Ansätze hauptsächlich bezüglich der Testkriterienfamilie der Datenflussüberdeckung skizziert, wobei weitere strukturelle Kriterien leicht entsprechend berücksichtigt werden können.

5.1 Globale Optimierung

Im Gegensatz zu den im Kapitel 2.3.2 (ab Seite 23) skizzierten Ansätzen erfordert die hier beschriebene globale Optimierung weder eine symbolische Ausführung des Programms wie bei [MLK03, Grz04], noch eine vorangehende statische Analyse des Kontroll- und Datenflusses des Testobjekts wie in [JSE96, MMS98, Ton04]. Für den Einsatz eines multi-objektiven Evolutionären Verfahrens sind jedoch geeignete Bewertungsfunktionen notwendig, in diesem Falle ist es die Größe der Testfallmenge einerseits sowie die Anzahl der tatsächlich überdeckten und vom vorgegebenen Testkriterium geforderten Entitäten (wie zum Beispiel alle Verzweigungen) andererseits. Während die Herleitung der ersteren trivial ist – dazu müssen schließlich nur die generierten Testfälle gezählt werden – stellt die Bestimmung der letzteren einen ungleich höheren Aufwand dar.

5.1.1 Dynamische Analyse

Um für einen gegebenen Testfall bestimmen zu können, welche Entitäten bei Ausführung des Programms mit diesem Testfall überdeckt werden, ist eine sogenannte *dynamische Analyse* notwendig. Dabei wird das zu testende Programm oder der mit einem Testtreiber sowie Stubs versehene Programmausschnitt unter wohldefinierten Bedingungen ausgeführt. Während der Ausführung müssen jedoch alle relevanten Informationen protokolliert werden, um anschließend eine Aussage bezüglich der Überdeckung des Testfalls zu erzielen. Diese Protokollierung kann auf unterschiedliche Weise umgesetzt werden, wobei man grob *invasive* und *nicht-invasive* Strategien unterscheidet.

Nicht-invasive Laufzeitprotokollierung

Das Ziel moderner Programmiersprachen ist es, möglichst plattformunabhängige Software erstellen zu können, das heißt, der Programmcode wird nur einmal in der endgültigen Form ge-

[1] *DOTgEAr: Dataflow oriented testcase generation with Evolutionary Algorithms*

schrieben und soll anschließend unter verschiedenen Betriebssystemen sowie auf unterschiedlichen Prozessorarchitekturen ablauffähig sein. Um dies zu erreichen, wird der Sourcecode nicht wie in der Frühzeit der Sprachen C oder C++ zu einem prozessor- und betriebssystemspezifischen Binärcode übersetzt und gebunden. Stattdessen wird eine Zwischendarstellung namens *Byte-Code* bei Java beziehungsweise *Intermediate Language (IL)* bei C# und anderen .NET-Sprachen generiert, die von einer spezifischen Laufzeitumgebung, der sogenannten *Virtuellen Maschine (VM)*, interpretiert wird.

Aufgrund dieser Architektur besteht eine nicht-invasive Technik darin, die Virtuelle Maschine zur Laufzeitüberwachung eines ausgeführten Programms einzusetzen. Bei JAVA™ kann beispielsweise über eine der Schnittstellen der *Java Platform Debugger Architecture (JPDA)* auf die Laufzeitumgebung eingewirkt und diese angewiesen werden, bei Zugriffen auf ein bestimmtes Feld eine Nachricht an das überwachende Programm zu senden, welches den Zugriff dann protokolliert. Der Vorteil eines solchen Vorgehens ist, dass keine Instrumentierung des Codes (wie im Folgenden beschrieben) und somit keine Modifikation am Programm selbst notwendig ist. Allerdings ist die über diese Schnittstelle angebotene Funktionalität für die hier auftretenden Aufgaben recht beschränkt: Eine genaue Protokollierung des Kontrollflusses sowie Zugriffe auf lokale Variablen innerhalb einer Methode sind nur mit großem Aufwand mittels des Einzelschrittmodus oder Breakpoints möglich, was die Gesamtausführungszeit stark negativ beeinflussen würde.

Invasive Laufzeitprotokollierung (Instrumentierung)

Alternativ bietet sich aus der Klasse der invasiven Techniken die *Instrumentierung* an. Dabei werden entweder in den Quellcode oder in den Byte-Code sogenannte *Proben* eingesetzt. Das sind Anweisungen, deren Ausführung mit bestimmten Ereignissen gekoppelt ist, welche zur Laufzeit protokolliert werden müssen. Der Vorteil der Instrumentierung auf Byte-Code-Ebene liegt in der Einfachheit der Byte-Code-Sprache, wodurch sich die Instrumentierung relativ leicht gestaltet. Darüber hinaus wird mit einem Testfallsatz, welcher aufgrund der Informationen der Byte-Code-Instrumentierung erstellt wurde, zugleich auch der verwendete Compiler validiert. Der größte Nachteil einer solchen Instrumentierung ist jedoch, dass die dabei protokollierte Information nicht für alle betrachteten Testkriterien genügend Rückschlüsse auf die tatsächliche Überdeckung zulässt. Zum Beispiel ist im Assembler-ähnlichen JAVA™-Byte-Code nicht mehr erkennbar, wo die Auswertung einer Bedingung (einer ursprünglichen `if`-Anweisung) beginnt, womit die Bedingungsüberdeckung nicht realisierbar ist. Aus diesem Grund wurde bei der Implementierung des Werkzeugs .gEAr die Instrumentierung auf der Ebene des Quellcodes vorgezogen.

Die allerwichtigste Voraussetzung für eine erfolgreiche Instrumentierung ist die Vermeidung jeglicher Seiteneffekte, das heißt es muss sichergestellt werden, dass das Programm vor und nach der Instrumentierung unter allen Bedingungen das gleiche Verhalten aufweist. Leider ist dies im Bezug auf das Laufzeitverhalten nicht möglich, da jede eingefügte Probe auch Rechenzeit erfordert.

Das Datenflussdiagramm in Abbildung 5.1 zeigt den gesamten Prozess der Vorverarbeitung, welcher vor der eigentlichen Testdatengenerierung stattfindet. In objektorientierten Programmiersprachen besteht ein Softwarepaket im Allgemeinen aus mehreren Klassen, von denen aber

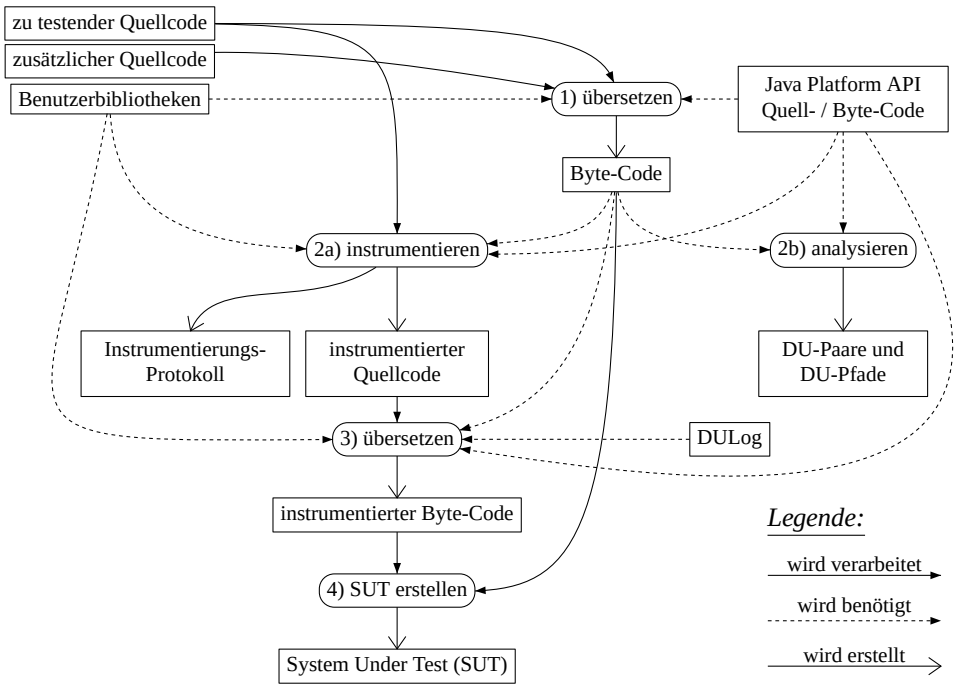


Abbildung 5.1: Analyse und Instrumentierung des zu testenden Systems (SUT)

in einem Testschritt womöglich nur eine Auswahl (im Weiteren als *Testobjekt* bezeichnet) hinsichtlich einer bestimmten Überdeckung getestet werden soll. Dies trifft insbesondere in der Modultestphase zu (siehe Kapitel 2.2.1), da hier eventuell noch gar nicht alle zur Ausführung eines Testobjekts notwendigen Module verfügbar sind und die dafür eingesetzten Treiber und Stümpfe schließlich nicht „mitgetestet“ werden sollen. Aus diesem Grund wird bei •gEAr die gesamte ausführbare Software (im Folgenden *System Under Test* oder kurz *SUT*) eines Projektes in tatsächlich zu testendem Quellcode sowie zur Ausführung zusätzlich notwendige Quellcodedateien und Benutzerbibliotheken unterteilt.

In einem ersten Schritt wird zunächst der vollständige Quellcode übersetzt, da Instrumentierung und Analyse unter anderem auf Typinformationen angewiesen sind und diese idealerweise über den *Reflection*-Mechanismus der Programmiersprache ermitteln, welcher selbst den Byte-Code erfordert.

Anschließend wird in einem zweiten Schritt lediglich der zu testende Quellcodeausschnitt instrumentiert. Dabei benötigt der Instrumentierer zusätzliche Erkenntnisse über die innerhalb des Testobjekts verwendeten, jedoch aus externen Bibliotheken (Benutzerbibliotheken oder die von der Programmierumgebung bereitgestellte *Platform API*) importierten Datentypen. Der Instrumentierer selbst generiert dabei den um Protokollaufrufe (Proben) angereicherten Quellcode einerseits sowie ein Instrumentierungsprotokoll andererseits. Das Instrumentierungsprotokoll enthält alle statisch zur Instrumentierungszeit ermittelbaren Informationen zu den einzelnen Ereignissen, die zur Laufzeit während der Ausführung des instrumentierten Programms auftreten und

vermerkt werden. Zum Datensatz einer Probe gehören jeweils ein eindeutiger Schlüssel (Identifikator), die Bezeichnung des Ereignisses und der Ort, an dem die Probe eingefügt wurde – letzteres besteht aus eindeutigem Klassen- oder Methodenbezeichner, vollständigem Dateinamen sowie Zeilen- und Spaltennummer und ermöglicht dem Tester eine genaue Lokalisierung der überdeckten Entitäten im Quellcode. Dieses Instrumentierungsprotokoll ermöglicht die effiziente Erstellung besonders schlanker Laufzeitprotokolle, da während der Ausführung nur noch der Identifikator des Ereignisses und die ausschließlich zur Laufzeit verfügbaren, dynamischen Informationen protokolliert werden müssen.

Nach der Instrumentierung wird der angereicherte Quellcode im dritten Schritt ebenfalls übersetzt und schließlich im letzten Schritt zusammen mit den nicht-instrumentierten Programmanteilen zu einer ausführbaren Applikation zusammengefasst.

5.1.2 Instrumentierung für die Datenflussüberdeckung

Die in •gEAr umgesetzte vollautomatische Instrumentierung zur Datenflussüberdeckung ist für die Verarbeitung objekt-orientierter Programme in der Sprache JAVA™ vorgesehen und berücksichtigt die besonderen Anforderungen solcher Systeme [Ost04, OD04, Ost05, OS06][2]. Die Programmiersprache JAVA™ enthält auch klassische Konstrukte imperativer Sprachen. Daher ist das hier vorgestellte Verfahren umfassender als eine gleichartige Technik zur Instrumentierung prozeduraler Programme, wie zum Beispiel in der Sprache C, weshalb sich das gesamte Verfahren der Testdatengenerierung aus dieser Arbeit leicht auf klassische Sprachen übertragen lässt.

Eine der Besonderheiten der Sprachfamilie um Java macht es notwendig, die Variablenzugriffe nicht nur nach „schreibend“ (def) oder „lesend“ (use) zu unterscheiden, sondern auch die Art der jeweiligen Variablen zu berücksichtigen. Jede der sogenannten *statischen Variablen* (auch *Klassenvariablen* genannt) gibt es zur Ausführungszeit nur einmal im Speicher. Im Gegensatz dazu gibt es für jede Instanz einer Klasse auch eine eigene „Instanz“ eines sogenannten *Feldes* (daher auch *Instanzvariable*) dieser Klasse. Somit können sich hinter dem gleichen Variablennamen unterschiedliche Felder verschiedener Objekte verbergen – ein Phänomen, welches dem Problem des *pointer aliasing* aus Kapitel 3.4.6 ähnelt. Eine *lokale Variable*, die entweder einen Parameter einer Methode darstellt oder innerhalb der Methode als Hilfsvariable deklariert wurde, kann zur Laufzeit mehrfach „instantiiert“ werden – nämlich bei jedem Methodenaufruf oder im Falle nebenläufiger Ausführung (multi-threading). Eine besondere Form der Datenflussoperation stellen auch Matrizenzugriffe dar, da jede Position innerhalb einer beliebig dimensionierten Matrix (*Array*) ein eigenes Objekt darstellt, welches aber durch den gleichen Namen der Matrix-Variablen und einen differenzierenden Index angesprochen wird. Darüber hinaus gehören zum Sprachschatz von JAVA™ die bereits von C bekannten Pre-/Post-Inkrement/Dekrement-Operatoren „++“ beziehungsweise „--“, die eine kombinierte def/use- oder use/def-Operation darstellen und daher ebenfalls gesondert behandelt werden müssen.

Zur Protokollierung der datenflussrelevanten Ereignisse in •gEAr während der Ausführung eines instrumentierten Programms wird der Applikation ein weiteres Paket namens *DULog* (Abbildung 5.1 beziehungsweise Abbildung 5.3) zur Seite gestellt. Alle Proben in der instrumen-

[2] Erste Ansätze wurden skizzenhaft in [Ost04, OD04, Ost05, OS06] publiziert.

tierten Software repräsentieren Aufrufe statischer Methoden dieses Subsystems, welche die zu überwachenden datenflussrelevanten Anweisungen umschließen. Bei ihrer Ausführung zusammen mit den relevanten Anweisungen übermitteln sie die zur Wiederherstellung der Datenflüsse notwendigen Informationen, welche von *DULog* zur späteren Verwendung in einem Ausführungsprotokoll gespeichert werden.

Die Instrumentierung selbst wurde bei •gEAr auf Basis des freien *ANTLR*[3]-Frameworks umgesetzt. Dieser Parser-Generator erstellt zunächst aus der Grammatik der Sprache JAVA™ (hier in Version 1.4) einen Lexer und einen Parser für JAVA™-Quellcode, welche ihrerseits ebenfalls als JAVA™-Klassen realisiert sind. Damit wird der zu instrumentierende Quellcode eingelesen und in einen abstrakten Syntaxbaum (*AST: Abstract Syntax Tree*) übersetzt. Zusätzlich zur vorhin genannten Quellcode-Grammatik gibt es eine zweite Grammatik, welche jedoch zum Parsen und Verarbeiten von ASTs konzipiert ist und über das Erkennen von programmiersprachlichen Konstrukten in JAVA™-ASTs hinaus auch sogenannte Aktionen zur Instrumentierung bereithält. Aus dieser AST-Grammatik generiert ANTLR einen sogenannten Baumparser, welcher ebenfalls in JAVA™ realisiert ist. Der aus einem Quellcodefragment abgeleitete AST wird demnach mit dem Baumparser Knoten für Knoten durchlaufen, wobei aufgrund der Aktionen alle für die Protokollierung und Rekonstruktion des Datenflusses relevanten Knoten, wie im Folgenden dargestellt, um die notwendigen Proben ergänzt werden.

Eine detaillierte Beschreibung der datenflussorientierten Instrumentierung von JAVA™ würde den Rahmen dieser Arbeit sprengen und bereits ein kleines Forschungsprojekt für sich darstellen [OD04]. Dennoch seien anhand von Listing A.1 (Seite 237) und Listing A.2 (Seite 238) sowie des zugehörigen Instrumentierungsprotokolls aus Tabelle A.1 (Seite 240) die wichtigsten Konzepte dieses Verfahrens skizziert. Listing A.1 zeigt die ursprüngliche, also nicht-instrumentierte Fassung des Testobjekts `DataflowExample`, während Listing A.2 das Ergebnis[4] der Instrumentierung darstellt.

Lokale Variablen

Variablen, die entweder innerhalb einer Methode oder als formale Parameter einer Methode deklariert wurden, heißen lokale Variablen. Um einen lesenden Zugriff (*use*) auf eine solche lokale Variable zur Laufzeit zu protokollieren, verfügt die Protokollbibliothek `DULog` über jeweils eine Methode der Form

`public static` ⟨*type*⟩ `useLocal(int pos,` ⟨*type*⟩ `o)`

für jeden der primitiven Datentypen[5] von JAVA™ sowie eine für den generischen (zu allen Objekten kompatiblen) Datentyp `Object` anstelle von ⟨*type*⟩. Diese Methoden vermerken ihren Aufruf zur Ausführungszeit durch Festhalten des ihnen übergebenen Schlüssels `"pos"` in der

[3] ANother Tool for Language Recognition, `http://www.antlr.org/`

[4] Zur besseren Lesbarkeit wurden einerseits der instrumentierte Quellcode durch zusätzliche Umbrüche neu angeordnet (*pretty-printed*) und andererseits die vollständige Packagebezeichnung der Protokollbibliotheksklasse `de.fau.cs.swe.sa.dynamicdataflowanalysis.rt.DULog` zu `DULog` gekürzt.

[5] boolean, byte, char, short, int, long, float, double

Protokolldatei und liefern ihren Parameter `o` als Rückgabewert zurück. Während der Instrumentierung wird jedes Vorkommen eines lesenden Zugriffs auf eine lokale Variable *v* durch den Aufruf dieser Methode ersetzt sowie ein Eintrag zum entsprechenden Schlüssel "`pos`" im Instrumentierungsprotokoll angelegt. Dabei tritt im instrumentierten Quellcode die Variable *v* anstelle des formalen Parameters `o` der Protokollmethode beziehungsweise der eindeutige Schlüssel aus dem Instrumentierungsprotokoll anstelle von "`pos`".

Im Beispiel aus Listing A.1 enthält der Konstruktor ab Zeile 17 die lokalen Variablen `aValue`, `anotherValue` sowie `index`. Das Verwendung der Variablen `anotherValue` in Zeile 22 wird vom Instrumentierer durch "`(int)DULog.useLocal(34,anotherValue)`" ersetzt, wie in Listing A.2 (Zeile 56) dargestellt. Zum zugehörigen Schlüssel „34“ wird in der Instrumentierungsprotokolldatei der entsprechende Eintrag aus Tabelle A.1 angelegt.

Definitionen (*defs*) lokaler Variablen, welche nicht in Kombination mit einem Lesezugriff wie bei Post-Increment-Operatoren auftreten, werden mittels entsprechend typisierter Varianten von Methoden der Familie

`public static` ⟨*type*⟩ `defLocal(int pos,` ⟨*type*⟩ `o)`

aus der Protokollbibliothek `DULog` vermerkt. Um die Semantik des ursprünglichen Programms beizubehalten und dennoch instrumentierten Code nach gültiger JAVA™-Syntax zu generieren, wird bei einer Wertzuweisung nicht die Variable selbst vom Protokollausdruck umschlossen, sondern der Ausdruck, dessen Auswertung der Variablen zugewiesen werden soll.

Ein typisches Beispiel in Listing A.1 ist die Initialisierung des Schleifenzählers `index` in Zeile 19. Aus dem ursprünglichen Ausdruck "`int index = 2;`" wird in der instrumentierten Fassung "`int index=(int)DULog.defLocal(25,2);`" (Zeile 48 in Listing A.2). Dabei ist der zweite Parameter (hier „2“) der Methode stets der vollständige Ausdruck der rechten Seite der Zuweisung, während "`pos`" (hier belegt durch „25“) den Bezug zum Eintrag im Instrumentierungsprotokoll (Tabelle A.1) darstellt.

Einer Sonderbehandlung bedürfen die Pre-/Post-Inkrement/Dekrement-Operatoren „`++`“ beziehungsweise „`--`“, da sie mit einer Kombination aus lesendem und schreibendem Zugriff gleichzusetzen sind, weshalb sie im Folgenden als *useDef* abgekürzt werden. Für Ausdrücke dieser Art enthält `DULog` eine Reihe von Methoden vom Typ

`public static` ⟨*type*⟩ `useDefLocal(int pos,` ⟨*type*⟩ `o)`

bereit, welche je nach Datentyp immer den gesamten Ausdruck umschließen.

Zur Veranschaulichung betrachte man erneut die Zeile 19 des bisher angesprochenen Programmbeispiels (Listing A.1). Hier wird der Ausdruck "`index++`" vom Instrumentierer durch "`DULog.useDefLocal(30,index++)`" (Zeile 51 in Listing A.2) ersetzt und entsprechend wie in Tabelle A.1 vermerkt.

Ähnlich den Inkrement/Dekrement-Operatoren bietet JAVA™ die Möglichkeit, weitere Ausdrücke abzukürzen. Dies ist zum Beispiel bei arithmetischen Operationen der Fall, bei denen die Variable, der das Ergebnis der Operation zugewiesen werden soll, selbst ein Operand ist. Dabei wird der Wert einer Variablen zunächst ausgelesen (*use*) und eventuell mit anderen Operanden verarbeitet; anschließend wird das Ergebnis der gleichen Variablen zugewiesen (*def*). Zu diesem Zweck umfasst `DULog` eine Reihe von Hilfsmethoden der Form

```
public static ⟨type⟩ useLocal(int pos)
public static ⟨type⟩ bin(Object dummy, ⟨type⟩ ret)
```

Im Gegensatz zu den bereits vorgestellten Varianten der entsprechenden Methode `useLocal`, protokollieren diese Hilfsmethoden nur das Auftreten des mit "`pos`" verbundenen Ereignisses. Die überladenen Methoden der Form "`bin(Object dummy, ⟨type⟩ ret)`" geben lediglich ihren zweiten Parameter als Rückgabewert zurück und dienen sonst als Hilfskonstrukte, unter anderem zur Instrumentierung abgekürzter Ausdrücke.

In Listing A.1 enthält Zeile 20 ein Beispiel für einen abgekürzten arithmetischen Ausdruck. Zur Protokollierung des kombinierten Lese-/Schreibzugriffes auf die Variable `anotherValue`, wird dieser Ausdruck wie in Listing A.2 (Zeile 53) dargestellt instrumentiert. Die jeweiligen Ereignisse „31“, „32“ und „33“ sind der Tabelle A.1 zu entnehmen. Wird diese instrumentierte Programmzeile ausgeführt, dann tritt gemäß der ursprünglichen Semantik zunächst Ereignis „32“ (Verwendung der Variablen `anotherValue`) auf, gefolgt von Ereignis „31“ (Verwendung der Variablen `index`) und schließlich von Ereignis „33“ (Definition der Variablen `anotherValue`).

Statische (Klassen-)Variablen

Die in JAVA™ mit dem Schlüsselwort `static` deklarierten Felder sind sogenannte Klassenvariablen, da zur Ausführungszeit nur jeweils eine Instanz solcher Variablen pro virtueller Maschine im Speicher existiert. Dementsprechend muss keine Instanz der Klasse erstellt werden, um auf solche Variablen zuzugreifen. Aufgrund dieser Eigenart können diese Variablen ähnlich wie lokale Variablen behandelt werden. Um die Datenflusskriterien auch auf beliebige Variablenarten getrennt anwenden zu können, zum Beispiel nur lokale Variablen bei der Überdeckung zu berücksichtigen, werden Ereignisse mit Bezug zum Datenfluss durch statische Variablen auch in der Instrumentierungsprotokolldatei gesondert gekennzeichnet. In der Protokollbibliothek `DULog` gibt es eigens für Klassenvariablen entsprechende Protokollmethoden:

```
public static ⟨type⟩ useStatic(int pos, ⟨type⟩ o)
public static ⟨type⟩ defStatic(int pos, ⟨type⟩ o)
public static ⟨type⟩ useDefStatic(int pos, ⟨type⟩ o)
public static ⟨type⟩ useStatic(int pos)
```

Auch die Anwendung dieser Methoden gestaltet sich analog zu den entsprechenden Funktionen für lokale Variablen. Listing A.1 enthält in Zeile 14 eine Verwendung des Klassenfeldes "`DataflowExample.CONSTANT`", welche bei der Instrumentierung durch die in Listing A.2 (Zeile 42) dargestellte Probe "`(int)DULog.useStatic(20,DataflowExample.CONSTANT)`" ersetzt wird.

Die Protokollierung der Definition einer Klassenvariable kann mit der zugehörigen Protokollmethode wie bei der Überwachung der Definition lokaler Felder geschehen. Falls die Definition jedoch nicht mit einer Deklaration der Variablen zusammenfällt (im Gegensatz zum obigen Beispiel, nämlich der Initialisierung des Schleifenzählers `index` in Zeile 19/Listing A.1), so kann die Protokollmethode auch den gesamten Ausdruck umschließen. Dies zeigen exemplarisch Zeile 38

in Listing A.1 beziehungsweise Zeile 77 in Listing A.2. Dabei wird die Definition der Variablen `DataflowExample.CONSTANT` im Ausdruck "`DataflowExample.CONSTANT = 4711`" zu "`DULog.defStatic(52,DataflowExample.CONSTANT=4711)`" instrumentiert.

Völlig analog zur Behandlung entsprechender Konstrukte bei lokalen Variablen wird auch mit kombinierten use/def-Operatoren bei statischen Feldern verfahren. Zeile 30 in Listing A.1 enthält eine Post-Inkrement-Operation "`CONSTANT++`", welche im instrumentierten Quellcode durch "`DULog.useDefStatic(43,CONSTANT++)`" (Zeile 67, Listing A.2) ersetzt wurde.

Wird eine statische Variable in einem abgekürzten arithmetischen Ausdruck sowohl gelesen als auch geschrieben (zum Beispiel "`DataflowExample.CONSTANT /= 2`"), so erfolgt die Instrumentierung dieses Ereignisses ebenfalls analog zum Verfahren mit lokalen Variablen, das heißt unter Verwendung der bereits erwähnten Methode "`public static ⟨type⟩ bin(Object dummy, ⟨type⟩ ret)`" sowie "`public static ⟨type⟩ useStatic(int pos)`" in diesem Fall.

Instanzvariablen

Ungleich schwieriger ist die Protokollierung des Datenflusses basierend auf Instanzvariablen, also den nicht-statischen Feldern. Das in Kapitel 3.4.6 behandelte Problem des *pointer aliasing* und seine Übertragung auf Referenzvariablen, wie sie in JAVA™ anstelle der aus C/C++ bekannten Zeiger treten, läßt sich auch am Beispiel aus Listing A.1 dokumentieren. In Zeile 37 sowie in Zeile 39 des Programmbeispiels wird je eine Instanz der Klasse `DataflowExample` erstellt und den beiden lokalen Variablen `de1` beziehungsweise `de2` zugewiesen. Zur besseren Unterscheidung seien die beiden Objekte, entsprechend der Programmzeile, in der sie instantiiert wurden, als $DataflowExample_{37}$ respektive $DataflowExample_{39}$ bezeichnet. Die erneute Definition des Feldes `de2` erfolgt in Abhängigkeit vom Wahrheitsgehalt des Prädikats in Zeile 40: Ist die Bedingung falsch, so referenziert `de2` nach Zeile 42 weiterhin das Objekt $DataflowExample_{39}$; ist sie jedoch wahr, so verbirgt sich hinter der Variablen `de2` das gleiche Objekt $DataflowExample_{37}$ wie hinter dem Namen `de1`.

Welche der beiden Möglichkeiten tatsächlich zutrifft, lässt sich erst zur Ausführungszeit feststellen. Dennoch muss die schon vorab erfolgende Instrumentierung so gestaltet sein, dass der von einem Testfall tatsächlich verursachte Datenfluss rekonstruiert werden kann. Betrachtet man die Verwendung der Instanzvariablen `fieldTwo` in Zeile 43, so lässt sich statisch (zur Instrumentierungszeit) „lediglich“ ein Zugriff auf das Feld `fieldTwo` der lokalen Variablen `de2` feststellen.

Ein Testfall t_1, bei dessen Ausführung die Zeile 41 *nicht* überdeckt wird und somit die Variable `de2` das in Zeile 39 instantiierte Objekt $DataflowExample_{39}$ referenziert, überdeckt bezüglich Feld `fieldTwo` ein *def/use*-Paar, dessen Definition in Zeile 22 die zugehörige Verwendung in Zeile 43 erreicht. Dies liegt daran, dass die Instanz $DataflowExample_{39}$ durch Aufruf des Konstruktors ab Zeile 17 initialisiert wird, womit die letzte Definition des Feldes `fieldTwo` dieses Objekts in Zeile 22 erfolgt.

Im Gegensatz dazu würde ein Testfall t_2 einen ganz anderen Datenfluss überdecken, sofern er bei seiner Ausführung die Zeile 41 erreicht. In diesem Fall wird das *def/use*-Paar überdeckt, dessen letzte Definition in Zeile 5 erfolgt und die Verwendung in Zeile 43 erreicht. Grund dafür ist die Tatsache, dass das Objekt $DataflowExample_{37}$, welches sich nun während des Feldzugriffs in Zeile 43 tatsächlich hinter der Bezeichnung `de2` verbirgt, über den Konstruktor ab

Zeile 7 instantiiert wurde. Da die scheinbare Definition in Zeile 10 aufgrund einer Ausnahme nicht erfolgt, ist die letzte „gültige“ Definition des Feldes `fieldTwo` diejenige in Zeile 5. Hierbei handelt es sich um eine Besonderheit von JAVA™: Anders als in C/C++ werden statische Felder und Instanzvariablen (hier `fieldTwo`) von der virtuellen Maschine mit Standardwerten (hier „0“) vorbelegt, auch wenn keine explizite Initialisierung durch den Programmierer (wie bei Feld `fieldOne` mit `null` in Zeile 4) vorgesehen ist.

Aus obigem Beispiel ist ersichtlich, dass allein die Protokollierung eines lesenden Zugriffs auf eine Variable namens „`de2.fieldTwo`“ in Zeile 43 nicht genügt, um den tatsächlichen Datenfluss während einer Programmausführung zu rekonstruieren. Vielmehr muss zusätzlich bekannt sein, auf *welches* Objekt sich der Name `de2` zur Laufzeit tatsächlich bezieht, um dieser Verwendung die richtige Definition zuordnen zu können. Eine Lösung dieser JAVA™-Variante des *pointer aliasing*-Problems besteht darin, jedem Objekt, welches während der Ausführung des zu testenden Programms instantiiert wird, eine eindeutige Instanzkennung zuzuordnen. Wird diese Kennung bei jedem Feldzugriff ebenfalls protokolliert, können auf diese Weise die verschiedenen Feldinstanzen entsprechend ihrer Eigentümerobjekte unterschieden werden.

In •gEAr wurde dieser Lösungsweg aus Gründen der Performance-Optimierung auf zwei verschiedene Arten umgesetzt. Alle Klassen des zu testenden Programms, deren Datenfluss überwacht werden soll und deren Quellcode daher zu instrumentieren ist, werden um ein Feld namens `___instanceId` sowie eine Methode `___getInstanceId()` wie in Zeile 2 beziehungsweise Zeile 3 des instrumentierten Programms in Listing A.2 erweitert. Sobald eine auf diese Weise instrumentierte Klasse instantiiert wird, erhält das entsprechende Objekt von der Protokollbibliothek `DULog` eine eindeutige Kennung.

Die obige Lösung ist zwar schnell und einfach zu realisieren, bedauerlicherweise lassen sich damit aber nicht alle Instanzen kennzeichnen. Dies gilt für Klassen, die gar nicht instrumentiert werden sollen, da sie nicht Bestandteil des Testobjekts sind, jedoch insbesondere für Klassen, deren Quellcode gar nicht zur Verfügung steht, zum Beispiel weil es sich dabei um *off-the-shelf*-Komponenten oder um Bibliotheksklassen der JAVA™-Umgebung handelt. Für Instanzen dieser Art enthält `DULog` eine Sonderbehandlung: Wird innerhalb des instrumentierten Quellcodes erstmals auf ein solches Objekt zugegriffen, so wird dieses Objekt zusammen mit einer neuen und damit eindeutigen Kennung in einer Hash-Tabelle abgelegt. Bei jedem weiteren Zugriff auf ein Objekt dieser Art kann dann die entsprechende Kennung aufgrund der Hash-Tabelle ermittelt werden.

Um die beiden Lösungsstrategien individuell unterscheiden zu können, implementiert jede instrumentierte und daher mit dem speziellen Feld `___instanceId` versehene Klasse die Schnittstelle `InstanceId`, wie dies exemplarisch in Zeile 1 (Listing A.2) dargestellt ist. Falls die Instanzkennung eines Objektes protokolliert werden soll, so prüft `DULog` zunächst, ob das Objekt die Schnittstelle `InstanceId` aufweist. Falls dies zutrifft, wird die Kennung direkt dem Objekt entnommen, ansonsten wird sie aufgrund der zweiten Strategie verwaltet.

Wichtig beim zweiten Lösungsansatz ist die Verwendung einer speziellen Hash-Tabelle. Wird ein nicht-instrumentiertes Objekt o_{ni} in eine klassische Hash-Tabelle (z.B. basierend auf der Bibliotheksklasse `java.util.Hashtable`) eingefügt, so wird die automatische Speicherbereinigung (*garbage collector*) von JAVA™ das Objekt o_{ni} auch bei auftretendem Speichermangel nicht tilgen können, selbst wenn (beziehungsweise gerade weil) der Eintrag in der Tabelle

die letzte Referenz auf o_{ni} ist, wodurch auch die Destruktoren zugehöriger Klassen nie ausgeführt werden. Auf diese Weise verhält sich das instrumentierte Programm semantisch nicht mehr äquivalent zur nicht-instrumentierten Variante. Zu diesem Zweck hält JAVA™ die Klasse `java.util.WeakHashMap` bereit, deren Referenzen vom *garbage collector* ignoriert werden und deren Einträge auch automatisch entsprechend der Speicherbereinigung angepasst werden. Die Verwendung dieser Hash-Tabelle ruft jedoch ein weiteres Problem hervor: Zur Berechnung der Schlüssel bedient sich `java.util.WeakHashMap` der Methode `hashCode()` des zu verwaltenden Objektes, welche vom Programmierer beliebig an die eigenen Bedürfnisse angepasst werden kann. Dabei kann es vorkommen, dass semantisch äquivalente, jedoch tatsächlich nicht identische Objekte den gleichen Hash-Code haben und somit von der Hash-Tabelle fälschlich als identisch behandelt werden. Eine Lösung verspricht hier die JAVA™-eigene Bibliotheksklasse `java.util.IdentityHashMap`, welche die Identität zweier Schlüsselobjekte nicht aufgrund ihres Hash-Codes sondern aufgrund eines Referenzvergleichs (basierend auf der Speicheradresse der Objekte) bestimmt. Da diese `java.util.IdentityHashMap` ihrerseits die Speicherbereinigung stört, muss hier eine Kombination beider Verfahrensweisen umgesetzt werden.

Um einen Zugriff auf ein Feld eines Objektes zu protokollieren, bedarf es nach obiger Beschreibung der eindeutigen Kennung des Objektes. Daher muss bei der Instrumentierung dafür gesorgt werden, dass die Protokollbibliothek `DULog` zusätzlich zur Bezeichnung des Ereignisses, welches den Feldzugriff im Instrumentierungsprotokoll beschreibt, auch Zugang zum Eigentümerobjekt des Feldes erhält.

Handelt es sich beim zu protokollierenden Ausdruck um eine lesende Verwendung (*use*) der Form `objektReferenz.feld`, wobei nicht weiter auf eine Instanzvariable oder eine Methode von `feld` zugegriffen werden soll, so wird zur Instrumentierung des Zugriffs auf `feld` die Methode

```
public static Object useField(int pos, Object lhs)
```

aus `DULog` verwendet, wobei der Teilausdruck `objektReferenz` anstelle des Parameters `lhs` tritt. Bei Aufruf der Methode werden im Laufzeitprotokoll die Kennung `"pos"` des Ereignisses, wie es im Instrumentierungsprotokoll abgelegt wurde, sowie die Kennung des Objektes `lhs`, welche nach einer der obigen Strategien ermittelt wird, vermerkt. Anschließend wird das Objekt `lhs` zurückgegeben, so dass der Zugriff auf das Feld `feld` des Objekts, auf welches `lhs` und damit `objektReferenz` verweisen, erfolgen kann.

Nach diesem Ansatz wird der Ausdruck `"de2.fieldTwo"` in Zeile 43 aus Listing A.1 instrumentiert, wodurch sich der Code aus Zeile 84 in Listing A.2 ergibt.

Ist der Feldeigentümer `objektReferenz` in einem Ausdruck mit einem beliebigen Zugriff der Form `objektReferenz.feld` jedoch selbst ein Feld, so kann obige Methode nicht zur Protokollierung des lesenden Zugriffs auf das Feld `objektReferenz` verwendet werden, da einerseits die Objektkennung des Eigentümers von `objektReferenz` protokolliert werden muss, die Methode andererseits das Objekt `objektReferenz` (und nicht deren Eigentümer) zurückzuliefern hat, damit der eigentliche Zugriff auf `objektReferenz.feld` erfolgen kann. Dazu enthält `DULog` eine Reihe von Funktionen der Art:

```
public static ⟨type⟩ useField(int pos, Object lhs, ⟨type⟩ o)
```

Ein Beispiel einer solchen Instrumentierung enthält Listing A.2 in Zeile 41, welche aus der Zeile 14 in Listing A.1 hervorgegangen sind. Der Ausdruck "`fieldOne.fieldTwo`" wird dabei so instrumentiert, dass zunächst der lesende Zugriff auf Feld `fieldOne` des aktuellen Objektes `this` protokolliert wird (Zeile 41 in Listing A.2), was mit dem Ereignis „18“ (siehe Tabelle A.1) gleichzusetzen ist. Die Methode `useField` in Zeile 41 gibt *nicht* das Objekt `this` sondern `fieldOne` zurück, welches als Parameter für den Aufruf der Methode `useField` in Zeile 40 verwendet wird. Diese reicht nach Protokollierung des Ereignisses „19“ das gleiche Objekt wieder zurück, so dass der Zugriff auf das Feld `fieldOne.fieldTwo` erfolgen kann.

Bei schreibenden Zugriffen auf Felder (*defs*) muss, ebenso wie bei den lesenden Verwendungen, zusätzlich zur Bezeichnung des Feldes, auch die eindeutige Kennung des Objektes, zu dem das beschriebene Feld gehört, protokolliert werden. Dazu verfügt `DULog` über eine Reihe von Protokollmethoden der Form:

`public static` ⟨*type*⟩ `defField(int pos, Object lhs,` ⟨*type*⟩ `o)`

Die praktische Anwendung dieser Protokollfunktionen erfordert eine Unterscheidung hinsichtlich der Art des zu instrumentierenden Ausdrucks, was ebenfalls am Codebeispiel aus Listing A.1 demonstriert sei. Den einfachsten Fall stellt Zeile 22 des nicht-instrumentierten Quellcodes und dessen instrumentiertes Gegenstück, nämlich Zeile 56 in Listing A.2, dar. Handelt es sich nicht wie im letzten Beispiel um den direkten Zugriff auf ein Feld des aktuellen Objektes ("`fieldTwo = anotherValue`"), sondern wie in Zeile 14 (Listing A.1) um einen Ausdruck der Form "`objektReferenz.feld = ...`", so muss die Protokollierung, wie in Listing A.2 dargestellt, zweistufig erfolgen: Zunächst wird mittels der Methode `DULog.useField` in Zeile 39 die Verwendung des Feldes `fieldOne` vermerkt. Diese Methode gibt das Objekt zurück, welches von `fieldOne` referenziert wird und auf dessen Feld `fieldTwo` schreibend zugegriffen werden soll. Umschlossen wird diese Probe vom eigentlichen Protokollaufruf ab Zeile 38, wobei das Eigentümerobjekt zunächst in der Hilfsvariablen `___t1` zwischengespeichert wird. Diese Hilfsvariable ist dann notwendig, wenn anstelle der "`objektReferenz`" keine einfache Variable, sondern ein komplexer Ausdruck mit eventuellen Seiteneffekten steht, weshalb dieser Ausdruck nur einmal ausgewertet werden darf. Ohne Hilfsvariable müsste der Ausdruck in Zeile 39 (Listing A.2) dupliziert werden, da er sowohl für die Bestimmung der Instanzkennung des Zielobjektes (Eigentümerobjekt des geschriebenen Feldes) als auch für den eigentlichen Zugriff auf das Feld während der Zuweisung notwendig ist – und somit doppelt ausgewertet werden würde.

Ähnlich gestaltet sich auch die Instrumentierung der Pre-/Post-Inkrement/Dekrement-Operatoren im Kontext nicht-statischer Instanz-Felder. Für den Fall, dass ein Feld im Kontext seines Eigentümerobjektes bearbeitet wird, bietet die Protokollbibliothek `DULog` die notwendigen Funktionen der Familie:

`public static` ⟨*type*⟩ `useDefField(int pos, Object lhs,` ⟨*type*⟩ `o)`

Der Ausdruck "`fieldTwo++`" in Zeile 28 (Listing A.1) wird von der Protokollfunktion zu "`DULog.useDefField(41,DataflowExample.this,fieldTwo++)`" (Zeile 65 in Listing A.2) umschlossen.

Wird jedoch das Feld eines Objektes in der Form `objektReferenz.feld` einer solchen Operation unterzogen, so muss zunächst der lesende Zugriff auf das aktuelle Feld `objektReferenz`

protokolliert werden, ehe der *useDef* des zugehörigen Feldes `feld` vermerkt wird. Letzteres geschieht mit Hilfe der Protokollmethode:

```
public static Object useDefField(int pos, Object lhs)
```

Diese speichert die mit dem betrachteten *useDef* verbundene Ereigniskennung sowie die Instanzkennung des Objektes, auf das `lhs` verweist, in der Prokolldatei und gibt anschließend den zweiten Parameter zurück.

Angewandt auf das exemplarische Codefragment "`fieldOne.fieldTwo++;`", in dem die Variable `fieldOne` von einem zur Klasse `Object` kompatiblen Datentyp ⟨*type*⟩ ist, ergibt[6] sich nach der Instrumentierung: "`((`⟨*type*⟩`)DULog.useDefField(`⟨ID_1⟩`,(`⟨*type*⟩`)DULog.useField(`⟨ID_2⟩`,`⟨*type*⟩`.this,fieldOne))).fieldTwo++;`".

Arrays

Lokale Variablen sowie statische und nicht-statische Felder können einfache Variablen eines primitiven Datentyps oder typisierte Objektreferenzen sein. Darüber hinaus gibt es in JAVA™ eine ganz spezielle „Objektart“, nämlich die sogenannten Arrays[7]. Dabei handelt es sich um eine Datenstruktur, welche eine geordnete Liste mehrerer Objekte mit kompatiblen Datentypen darstellt und auf deren Komponenten über einen numerischen Index zugegriffen wird. In vielerlei Hinsicht werden Arrays in JAVA™ wie Objekte behandelt, selbst wenn die Komponenten einen primitiven Datentyp haben. Insbesondere werden Array-Variablen anstelle von Parametern bei Methodenaufrufen als Referenz (genauer „*reference by value*“) übergeben und nicht in Kopie („*by value*“) wie dies bei primitiven Datentypen der Fall ist. Dieser Aspekt ist für die Instrumentierung besonders wichtig, da viele Operationen damit einfacher behandelt werden können.

Die in diesem Abschnitt beschriebene Behandlung von Arrays befasst sich speziell mit lesenden und schreibenden Zugriffen auf einzelne Komponenten eines Arrays. Jeder Zugriff auf eine Komponente einer Array-Variablen hat zwangsweise auch einen lesenden Zugriff auf die Array-Variablen selbst zur Folge. Dabei werden Array-Variablen zunächst wie einfache Variablen entsprechend ihrer Art (lokal, statisch, nicht-statisch) instrumentiert.

Zur Protokollierung von Datenflussereignissen im Zusammenhang mit Arrays enthält die Protokollbibliothek `DULog` eine Reihe öffentlicher Methoden ("`public static`") der Form:

```
int useArrayLength(int pos, ⟨type⟩[] array)
⟨type⟩ useArray(int pos, ⟨type⟩[] array, int index)
⟨type⟩ defArray(int pos, ⟨type⟩[] array, int index, ⟨type⟩ value)
⟨type⟩ preIncArray(int pos, ⟨type⟩[] array, int index)
⟨type⟩ preDecArray(int pos, ⟨type⟩[] array, int index)
⟨type⟩ postIncArray(int pos, ⟨type⟩[] array, int index)
⟨type⟩ postDecArray(int pos, ⟨type⟩[] array, int index)
```

[6] Hier zum Zwecke einer besseren Darstellung in einer generischen Form angegeben.

[7] Im Deutschen wird *Array* typischerweise mit den Begriff *Feld* übersetzt. Hier wird zur besseren Unterscheidung solcher Strukturen von Klassen- oder Instanzvariablen (meist ebenfalls als „Felder“ bezeichnet) die englische Benennung *Array* beibehalten.

Zwar sind Arrays[8] in JAVA™ nativ implementiert, also keine Objekte, denen eine Klasse zugrunde liegt, dennoch verfügen alle „Array-Objekte“ über ein spezielles „Feld“ namens *length*. Zur Protokollierung eines lesenden Zugriffes auf dieses Pseudo-Feld (schreibende Zugriffe sind nicht möglich, werden hier jedoch mit der Instantiierung von Arrays gleichgesetzt), wird der Ausdruck, dessen Auswertung ein Array ergibt, auf dessen Länge zugegriffen werden soll, von der Methode "`useArrayLength`" umschlossen. Diese vermerkt die Kennung "`pos`" des Ereignisses zusammen mit der Instanzkennung des Array-Objekts und gibt die Länge des Arrays zurück.

In Listing A.1 (Zeile 40) wird die Länge des Arrays in der lokalen Variable "`args`" ausgelesen. Die instrumentierte Fassung dieses Programmfragments findet sich in Listing A.2 bei Zeile 81 wieder. Hier ist ebenfalls exemplarisch zu erkennen, wie zunächst der lesende Zugriff auf die Variable und erst anschließend der *use* des Pseudo-Feldes "`length`" protokolliert werden.

Ähnlich geschieht auch die Instrumentierung eines lesenden Zugriffs auf eine Komponente eines Arrays. Dazu werden das Array und der Index der Komponente jeweils getrennt als Parameter der Methode "`useArray`" übergeben. Diese protokolliert die Kennung des Ereignisses ("`pos`"), die Instanzkennung des Arrays sowie den Index der Komponente ("`index`") und gibt das Objekt an der Stelle *index* im Array als Rückgabewert zurück. Der Index ist notwendig, um den Datenfluss eindeutig zu verfolgen, da ansonsten alle Komponenten die gleiche „Kennung“ besitzen, obwohl sie unterschiedliche Speicherplätze belegen.

Trifft der Instrumentierer auf eine Anweisung der Form "`System.out.println(i[3]);`", so wird der Teilausdruck "`i[3]`" als Parameter der Methode "`System.out.println`" zum Beispiel durch "`(int)DULog.useArray(19,(int[])DULog.useStatic(18,i),3)`" ersetzt. Der restliche Teilausdruck wird ebenfalls entsprechend instrumentiert.

Im Falle einer Zuweisung eines Wertes an eine Komponente eines Arrays wird der gesamte Zuweisungsausdruck zerlegt und seine einzelnen Teilausdrücke jeweils als Parameter der Methode "`defArray`" instrumentiert. Zusätzlich zur Protokollierung der eindeutigen Kennung der Komponente (Ereignis "`pos`", Instanzkennung des Arrays und "`index`" der Komponente), führt diese Methode die Zuweisung des Wertes "`value`" an "`array[index]`" selbst durch und gibt den zugewiesenen Wert zurück.

Der Ausdruck "`i[3] = 5;`" sieht beispielsweise nach der Verarbeitung durch den Instrumentierer wie folgt aus: "`DULog.defArray(12,(int[])DULog.useStatic(11,i),3,(int)5);`".

Im Gegensatz zu den "`useDef`"-Methoden bei lokalen Variablen und Feldern erfordert die Instrumentierung der Pre-/Post-Inkrement/Dekrement-Operatoren bei Array-Komponenten jeweils eigene Methoden, da eine weitere Indirektion über den Index der Komponente protokolliert werden muss. Demnach wird ein entsprechender Ausdruck durch einen Aufruf der zugehörigen Protokollmethode ("`preIncArray`", "`preDecArray`", "`postIncArray`" oder "`postDecArray`") ersetzt.

Beispielsweise tritt anstelle des Ausdrucks "`--i[3];`" im instrumentierten Programmcode der Ausdruck "`DULog.preDecArray(24,(int[])DULog.useStatic(23,i),3);`".

[8] Nicht zu verwechseln mit der Klasse "`java.lang.reflect.Array`".

Initialisierung von Klassen und Feldern

Eine besondere Betrachtung verdient die Instrumentierung der Klassen- beziehungsweise Instanzinitialisierungen. Der Unterschied zwischen lokalen Variablen einerseits sowie statischen und nicht-statischen Feldern andererseits liegt darin, dass lokale Variablen zusätzlich zu ihrer Deklaration auch stets explizit definiert werden müssen, ehe sie in einer prädikativen oder berechnenden Verwendung gelesen werden können. Letzteres gilt trivialerweise für formale Parameter von Methoden, die beim Aufruf der Methode mit dem Wert des entsprechenden Aktualparameters belegt werden. Darüber hinaus muss diese Vorschrift für innerhalb einer Methode lokal deklarierte Variablen berücksichtigt werden, ansonsten ließe sich eine entsprechend fehlerhafte Klasse gar nicht erst übersetzen. Demnach kann jede protokollierte Verwendung einer lokalen Variablen stets einer zugehörigen Definition dieser Variablen zugeordnet werden, welche in einem während der Ausführung zeitlich vorausgehenden Protokolleintrag erfasst wurde. Bei Feldern muss die letzte Aussage nicht mehr zwangsweise zutreffen. Da Felder nicht explizit im Programmcode initialisiert werden müssen, führt die Virtuelle Maschine eine solche Initialisierung selbst durch – ein Ereignis, welches aufgrund einer fehlenden Definition im Quellcode nicht unmittelbar instrumentiert und damit protokolliert werden kann.

Wird eine Klasse während der Programmausführung erstmals von der Virtuellen Maschine geladen, werden noch vor dem ersten Zugriff des restlichen Programms auf diese Klasse alle statischen Felder zunächst mit Standardwerten („0“ für numerische Variablen und „null“ bei Objektreferenztypen) belegt und anschließend eine eventuell vorhandene Programmiererdefinition ausgeführt. Einen ähnlichen Ablauf gibt es auch bei der Instantiierung einer Klasse: Dabei wird zunächst jedes Feld mit einem Standardwert vorbelegt, anschließend die Initialisierungsanweisungen im deklarativen Teil der Klasse evaluiert und erst abschließend der Konstruktor der Klasse selbst ausgeführt. Exemplarisch zeigt das Programmbeispiel in Listing A.3 dieses Verhalten. Führt man das Programm aus, erhält man erwartungsgemäß die Ausgabe "`sInit1:0 fInit1:0 sInit2:91058 fInit2:91126`".

Obwohl eine Verwendung von Variablen unmittelbar nach der Initialisierung durch die Virtuelle Maschine und noch vor der Programmierdefinition sehr ungewöhnlich und daher selten zu erwarten ist, wird sie durch die Sprachsyntax und -semantik von JAVA™ nicht ausgeschlossen. Aus diesem Grunde muss eine Instrumentierung zur Überwachung des Datenflusses eine entsprechende Sonderbehandlung vorsehen, um die Semantik des instrumentierten Programms nicht zu verändern und ihren Ablauf zur Ausführungszeit trotzdem originalgetreu wiederzugeben. Zu diesem Zwecke verfügt die Protokollbibliothek `DULog` über die Methodenfamilie

```
public static void enterSInit(int pos)
public static void reenterSInit(int pos)
public static void leaveSInit(int pos)
public static void sInitCompleted(int pos)
```

für die Überwachung der Initialisierung statischer Variablen, sowie analog über die Methodenfamilie

```
public static void enterInit(int pos)
public static void reenterInit(int pos)
```

```
public static void leaveInit(int pos)
public static void initCompleted(int pos)
```

für nicht-statische Felder. Zusätzlich wird jede instrumentierte Klasse um zwei Hilfsmethoden der Form

```
protected static void ___doStaticZeroInit()
protected void ___doZeroInit()
```

ergänzt, die das Verhalten der Virtuellen Maschine bei der Initialisierung mit Standardwerten nachbildend protokolliert.

Der Einsatz dieser Methoden durch den Instrumentierer sei am Beispiel der bereits bekannten Klasse `DataflowExample` aus Listing A.1 erläutert. Diese enthält in Zeile 2 ein statisches Feld namens `CONSTANT`. Wird die Klasse in einem laufenden Programm benötigt, so wird sie vom *Classloader* in den Speicher geladen, das Feld wird zunächst mit dem Wert „0“ vorbelegt und anschließend wird die dargestellte Zuweisung ausgeführt. Um das beschriebene Verhalten auch nach der Instrumentierung semantisch identisch zu reproduzieren, werden die statischen Initialisierungssequenzen wie in Listing A.2 durchgeführt. Dazu wird zunächst der Codeblock ab Zeile 9 eingefügt, welcher als erster beim Laden der Klasse ausgeführt wird und die Hilfsmethode "`___doStaticZeroInit()`" in Zeile 89 aufruft. Auf diese Weise wird zunächst die „Definition“ von `CONSTANT` durch die Virtuelle Maschine protokolliert. Die sich anschließende Programmiererinitialisierung dieser statischen Variablen wird durch den Codeblock ab Zeile 15 erfasst. Um den Kontrollfluss korrekt nachzuvollziehen, werden alle Initialisierungen statischer Variablen von je einem Aufruf der Protokollmethoden "`enterSInit(int pos)`" (Ereignis „3“ entsprechend Tabelle A.1) in Zeile 9 sowie "`sInitCompleted(int pos)`" (Ereignis „4“) in Zeile 92 umschlossen.

Ähnlich gestaltet sich auch die Instrumentierung zur Protokollierung aller Ereignisse, die mit der Initialisierung eines Objektes bei der Instantiierung einer Klasse zusammenhängen. Als Beispiel sei die Behandlung des Feldes `fieldOne` aus Zeile 4 (Listing A.1) verfolgt. Bei der Instantiierung der Klasse `DataflowExample` werden zunächst die beiden Hilfskonstrukte Zeile 2 und Zeile 3 (Listing A.2) ausgewertet, welche aber für die Laufzeitüberwachung selbst eingefügt wurden und daher nicht als Teil des Kontroll- oder Datenflusses berücksichtigt werden. Die eigentliche Protokollierung setzt mit Zeile 5 ein, in welcher die Methode "`___doZeroInit()`" aus Zeile 90 aufgerufen wird. Dabei wird die Initialisierung der beiden nicht-statischen Felder mit Standardwerten durch die Virtuelle Maschine vermerkt (Ereignisse „8“ und „9“ aus Tabelle A.1). Anschließend wird aufgrund des Codeblocks ab Zeile 21 die vom Programmierer vorgesehene Initialisierung protokolliert. Abschließend wird mittels der Anweisung in Zeile 91 das Ende des Abschnitts in der Protokolldatei vermerkt, welcher die Instanzinitialisierung betrifft. Da der Programmierer für das zweite Feld `fieldTwo` keine explizite Definition vorgesehen hat, erübrigt sich dafür ein Codeblock ähnlich dem ab Zeile 21 mit einem Aufruf der Protokollfunktion "`reenterInit(int pos)`".

Prädikate und Verzweigungen

In Kapitel 3.4.1 werden lesende Zugriffe auf Variablen hinsichtlich ihres Zwecks unterschieden, je nachdem, ob diese Variablen in einem *berechnenden* (Definition 3.14 auf Seite 53) oder *prädikativen* (Definition 3.15) Kontext verwendet werden. Auf dieser Unterscheidung baut die erste Familie der Datenflusskriterien nach Rapps und Weyuker [RW82, RW85] auf (Kapitel 3.4.2). Spätere Erweiterungen oder Neuformulierungen der den Datenfluss betreffenden Kriterienhierarchie verzichteten auf diese Unterscheidung, forderten aber im Gegenzug die Überdeckung aller aus einem Knoten ausgehenden Kanten, sofern der Knoten eine Verwendung einer Variablen in einem Prädikat aufweist.

Solange Bedingungen in einem prädikativen Ausdruck ausschließlich Konstanten und Variablen miteinander in Relation setzen, wie dies in den Beispielen bei [RW82, RW85] vorkommt, ist eine Unterscheidung in *c-use* und *p-use* unproblematisch. Moderne Programmiersprachen erlauben jedoch auch die Verwendung von Methodenaufrufen innerhalb eines Prädikates. Eine auf diese Weise aufgerufene Methode kann jedoch selbst eine ganze Abfolge verschiedener Anweisungen enthalten, darunter weitere Variablendefinitionen oder gar Methodenaufrufe sowie insbesondere auch lesende Zugriffe auf Variablen. Enthält eine solche Methode einen (scheinbar) berechnenden Zugriff auf eine Variable, zum Beispiel als Operand im Rahmen einer Addition, so muss dieser Zugriff als prädikative Verwendung umdeklariert werden, falls die Methode im Kontext einer Bedingungsauswertung aufgerufen wurde.

Einen Extremfall dieses Dilemmas stellt der Quellcode in Listing A.4 dar. Jede JAVA™-Applikation startet gewöhnlich mit einer `main()`-Methode. Wenn diese wie hier in Zeile 3 zum Initiieren der eigentlichen Funktionalität dient und letztere in einem Prädikat kapselt, müssten prinzipiell *alle* Variablenverwendungen als prädikativ interpretiert werden.

Somit steht zur Instrumentierungszeit im Allgemeinen noch nicht fest, ob ein lesender Zugriff an einer bestimmten Stelle im Code berechnend oder prädikativ ist. Aus diesem Grunde wurde bei der Protokollierung der Datenflussereignisse bezüglich einzelner Variablen bewusst auf die Unterscheidung in *c-uses* und *p-uses* verzichtet. Um dennoch Testfälle anhand der Kriterienfamilie nach Rapps und Weyuker generieren und optimieren zu können, erhalten Prädikate eine besondere Instrumentierung. Da eine Bedingung aus mehreren Teilbedingungen aufgebaut sein kann, welche ihrerseits aufgrund von Methodenaufrufen einen komplexen Kontrollfluss umfassen können, wird jedes Prädikat mit Hilfe der beiden Protokollmethoden

```
public static int newPredicate(int pos)
public static boolean predResult(int pos, int predId, boolean result)
```

aus `DULog` instrumentiert. Dies wird dadurch erreicht, dass der gesamte prädikative Ausdruck, seinerseits natürlich entsprechend instrumentiert, als dritter Parameter (`result`) der Methode `"predResult"` eingesetzt wird. Nach Auswertung des Ausdrucks wird diese Methode ausgeführt, welche den erfolgreichen Abschluss der Bedingungsauswertung zusammen mit dem Wahrheitsgehalt des Prädikats im Protokoll vermerkt. Der Wahrheitsgehalt ist deshalb besonders wichtig, da die Datenflusskriterien die Überdeckung beider Kanten einer Verzweigung fordern - bei Rapps und Weyuker sind die *p-uses* sogar diesen Kanten zugeordnet. Bevor jedoch die erste Teilbedingung ausgewertet werden darf, muss der Beginn der Prädikatsauswertung ebenfalls im

Protokoll vermerkt werden, damit alle zur Bedingung gehörenden Ereignisse nach der Ausführung des Programms auch eindeutig dieser Bedingung zugeordnet werden können. Dazu wird die Methode "`newPredicate`" anstelle des zweiten Parameters `predId` der Methode "`predResult`" eingesetzt. Da die Parameter einer Methode in JAVA™ immer von links nach rechts ausgewertet werden, wird die Methode "`newPredicate`" stets vor der eigentlichen Bedingung ausgeführt. Diese Methode erzeugt für die jeweils aktuelle Prädikatsauswertung eine eindeutige Kennung `predId`, damit wiederholte Auswertungen der gleichen Bedingung unterschieden werden können. Sie vermerkt im Ausführungsprotokoll den Start der Prädikatsauswertung zusammen mit dieser Kennung (`predId`) und der Kennung des Prädikats im Code ("`pos`") und gibt `predId` an die sie umfassende Methode "`predResult`" weiter. Auf diese Weise kann "`predResult`" den Abschluss der aktuellen „Instanz“ mit der Kennung `predId` eines bestimmten Prädikates protokollieren.

Die Klasse `DataflowExample` aus Listing A.1 enthält zwei Beispiele eines solchen Prädikates. Eines davon ist Teil einer `IF`-Anweisung (Zeile 40) und das zweite versteckt sich als Abbruchbedingung in der `FOR`-Schleife (Zeile 19). Die entsprechend instrumentierten Anweisungen findet man in Listing A.2 (Zeile 80 beziehungsweise Zeile 49).

Einen Sonderfall einer „Verzweigungsanweisung“ stellt das `switch/case`-Konstrukt dar. Im Gegensatz zu einer `IF`-Anweisung, bei der das Ergebnis der Prädikatsauswertung binär ist und es daher nur zwei mögliche Ergebnisse beziehungsweise zwei Folgeanweisungen geben kann, stellt ein `switch/case`-Konstrukt eine Mehrfachverzweigung dar. Dabei kann der „Parameter“ der `switch`-Anweisung, also der Ausdruck, dessen Wert ausgewertet und aufgrund dessen einer der alternativen Kontrollflüsse ausgewählt wird, ebenso als eine Form eines „Prädikats“ interpretiert werden – schließlich kann jeder `switch/case`-Block durch eine Abfolge von ineinander geschachtelten `IF`-Verzweigungen ersetzt werden. Zur Instrumentierung eines `switch/case`-Ausdrucks werden die folgenden Methoden der Protokollbibliothek `DULog` verwendet:

```
public static int newSwitchPredicate(int pos)
public static int switchPredResult(int pos, int predId, int result)
public static void switchPredEquivalent(int pos, int equivClass)
```

Mittels "`newSwitchPredicate`" und "`switchPredResult`" wird, analog zum Verfahren bei binären Prädikaten, der „Parameter“ der `switch`-Anweisung umschlossen, so dass sowohl die tatsächliche Auswertung einer solchen „Bedingung“ als auch ihr Ergebnis protokolliert werden kann. Um darüber hinaus die während der Testausführung tatsächlich angesprungene Alternative nachvollziehen zu können, erhält jeder der `case`-Blöcke einen Aufruf der Methode "`switchPredEquivalent`" unmittelbar vor der ersten Anweisung des entsprechenden Codeblocks. Dabei werden die einzelnen „Zweige“ jeweils mit einer eindeutigen numerischen Kennung versehen, beginnend beim `default`-Zweig mit der Kennung 0. Falls der Programmierer keine solche `default`-Alternative vorgesehen hat, so wird künstlich eine hinzu instrumentiert.

Das Codebeispiel in Listing A.1 enthält ein `switch/case`-Konstrukt ab Zeile 26. Der gemäß obiger Beschreibung instrumentierte Quellcode ist in Listing A.2 ab Zeile 62 zu finden. Dabei zeigt Zeile 62 wie der „Parameter“ der `switch`-Anweisung umschlossen wird und Zeile 64 sowie Zeile 66, auf welche Weise der überdeckte Zweig protokolliert wird. Schließlich ist ab Zeile 68 eine künstlich eingefügte `default`-Alternative zu sehen.

Wichtig in diesem Zusammenhang und ebenfalls am Beispiel aus Listing A.1 beziehungsweise Listing A.2 zu erkennen ist, dass ein `case`-Block nicht notwendigerweise mit einem `break`-Befehl enden muss, der einen unbedingten Sprung zur ersten Anweisung nach dem gesamten `switch/case`-Konstrukt einleitet. Fehlt ein solches `break` am Ende eines `case`-Blocks, so wird der darauf folgende `case`-Block anschließend ebenfalls ausgeführt. Weil die Instrumentierung dieses Verhalten nicht stören darf, können im Protokoll der Laufzeitüberwachung durchaus auch mehrere `"switchPredEquivalent"`-Einträge der gleichen `switch/case`-Instanz auftreten. Nachträglich kann der tatsächlich angesprungene Zweig dennoch eindeutig rekonstruiert werden, da der entsprechende `"switchPredEquivalent"`-Eintrag unmittelbar nach dem zugehörigen `"switchPredResult"`-Eintrag im Protokoll steht.

Methodenaufrufe

Mit den vorangehend beschriebenen Ansätzen zur Instrumentierung von JAVA™-Quellcode lässt sich der Datenfluss während der Ausführung eines Programms bereits verfolgen und anschließend soweit rekonstruieren, dass eine hinreichend genaue Bewertung der Überdeckung zum Zwecke der Testdatengenerierung möglich ist. Dennoch gibt es in objekt-orientierten Programmiersprachen typische Implementierungsempfehlungen, die einer Sonderbehandlung im Kontext der datenflussorientierten Instrumentierung bedürfen.

Ein Ergebnis der sogenannten Kapselung in objekt-orientierten Programmen ist die Tendenz, möglichst alle (vorzugsweise nicht-statischen) Felder einer Klasse vor dem direkten Zugriff aus der Umgebung soweit wie möglich zu schützen. Dabei werden alle Klassen, entsprechend ihrer Verwandschaft mit der betrachteten Klasse, jeweils unterschiedlichen Zugriffsschutzzonen zugeteilt und diesen Zonen, mittels der Schlüsselwörter *public*, *protected* sowie *private* vor der Felddeklaration, entsprechende Zugriffsrechte eingeräumt beziehungsweise verweigert. Wird ein Feld als *private* deklariert, kann es ausschließlich vom Code aus der zugehörigen Klasse selbst bearbeitet werden. Wird keines der Schlüsselwörter angewandt, können zusätzlich zur Klasse selbst auch alle Klassen des gleichen *Package* auf das Feld zugreifen. Erhält das Feld die Auszeichnung *protected*, so können darüber hinaus auch Unterklassen auf dieses Feld zugreifen. Nur ein Feld, welches mit dem Schlüsselwort *public* deklariert wurde, kann von jeder beliebigen Klasse gelesen oder geschrieben werden.

Im Allgemeinen würde man nun alle Felder vorzugsweise als *private* oder höchstens *protected* deklarieren und sie somit dem direkten Zugriff aller fremden Klassen entziehen. Um dennoch aus der „Außenwelt“ Werte in diese Variablen fließen zu lassen, stellt man entsprechenden Feldern sogenannte *getter*- und *setter*-Methoden zur Seite. Erstere erlauben es dem Namen nach indirekt über eine entsprechende Methode lesend auf das Feld zuzugreifen, während Letztere zum schreibenden Zugriff gedacht sind. Was auf den ersten Blick kontraproduktiv erscheint, macht dennoch Sinn: Vor allem die *setter*-Methoden können den Schreibzugriff kontrollieren und so das zugehörige Objekt vor einem inkonsistenten Zustand bewahren. Das JavaBeans-Konzept setzt diesen Ansatz konsequent mit dem Ziel um, die Wiederverwendung von JAVA™-Komponenten in einer Komponentenarchitektur so weit wie möglich zu vereinfachen.

Problematisch ist diese Vorgehensweise für die Datenflussprotokollierung in Bezug auf statische und nicht-statische Felder. Bei konsequenter Anwendung von *getter*- und *setter*-Methoden

sind die Definitionen und Verwendungen der Felder nicht mehr über den gesamten Programmcode verteilt, also erkennbar an der Stelle im Quellcode, an der der Wert der Variablen ausgelesen beziehungsweise gesetzt wird. Stattdessen konzentrieren sich alle *defs* des gleichen Feldes auf die zugehörige *setter*-Methode, während alle *uses* analog in die entsprechenden *getter*-Methoden wandern. Im Extremfall gibt es dadurch nur noch jeweils eine einzige Stelle im Programmcode, an der ein bestimmtes Feld gelesen beziehungsweise geschrieben wird – und somit nur noch ein einziges Datenflusspaar.

Bezogen auf Feld "`fieldTwo`" aus Listing A.1 würden alle expliziten *uses* aus Zeile 14, Zeile 28, Zeile 33 sowie Zeile 43 zu einem einzigen *use* in einer dedizierten Programmzeile innerhalb dieser Klasse kollabieren. Anstelle aller expliziten *uses* tritt jeweils ein Aufruf der entsprechenden *getter*-Methode.

Wird eine *setter*- oder *getter*-Methode von unterschiedlichen Stellen im Programmcode aus aufgerufen, so repräsentiert die jeweilige Aufrufstelle selbst ein Unterscheidungsmerkmal für die ansonsten gleichen *defs* beziehungsweise *uses*. Verfolgt man diese Überlegungen rekursiv, so bietet es sich an, den jeweils zum Zeitpunkt eines Feldzugriffs aktuellen Methodenaufrufstack als präzisestes Differenzierungsmerkmal einer Definition oder Verwendung hinzuzunehmen. Dies entspricht dem vollständigen Abrollen des kompakten *Java Interclass Graphs* (siehe Kapitel 3.1) einer zu testenden Komponente oder Applikation.

Um dieses Ziel zu erreichen, ist eine Protokollierung des Kontrollflusses auf der Granularitätsebene einzelner Methoden notwendig. Dabei muss allerdings berücksichtigt werden, dass die in •gEAr umgesetzte Instrumentierung nicht zwangsläufig den gesamten, während der Ausführung erreichbaren Code instrumentieren muss (oder kann[9]). Insbesondere können aus dem zu testenden und daher instrumentierten Quellcodebereich auch Bibliotheksfunktionen aufgerufen werden, die nicht instrumentiert sind und daher außerhalb des Zugriffs der Laufzeitprotokollierung liegen. Aus diesem Grund werden Methodenaufrufe zweigleisig behandelt: Zunächst wird im instrumentierten Programmteil der Aufruf selbst protokolliert. Zusätzlich wird das Betreten und Verlassen jeder Methode ebenfalls im Laufzeitprotokoll vermerkt, vorausgesetzt die entsprechende Methode gehört zum instrumentierten Programmbereich. Auf diese Weise ist aus der Laufzeitüberwachung zumindest der Sprung zu einem potentiell uninstrumentierten Codebereich rekonstruierbar.

Zum Zwecke der Instrumentierung eines Methodenaufrufs verfügt die Protokollklasse `DULog` über die Methodenfamilie *„CallPoint“ (`cp`)*:

```
public static ⟨type⟩ cp(int pos, ⟨type⟩ lastParameter)
public static Object cp(int pos)
```

Ist eine Methode aufzurufen, die mindestens einen formalen Parameter hat, so wird der letzte (rechte) Aktualparameter an der Aufrufstelle von der ersten der beiden "`cp`"-Varianten umschlossen. Auf diese Weise wird der Aktualparameter zunächst zum Parameter "`lastParameter`" der Methode "`cp`" und diese tritt selbst anstelle des Parameters. Gelangt der Kontrollfluss während

[9] Jeder Instrumentierungsversuch endet zwangsläufig bei nativen Methoden, die gar nicht als Quelltext oder Bytecode vorliegen.

der Ausführung zu einem auf diese Weise instrumentierten Methodenaufruf, so werden die Parameter der aufzurufenden Methode zunächst von links nach rechts ausgewertet, so dass als allerletztes vor dem unmittelbaren Sprung in die Zielmethode die Protokollmethode selbst "`cp`" ausgeführt wird. Diese vermerkt die Kennung "`pos`" des Ereignisses im Protokoll und gibt das Ergebnis der Auswertung des letzten Aktualparameters zurück, so dass er seinerseits der eigentlich aufzurufenden Methode übergeben werden kann.

Listing A.1 enthält in Zeile 43 einen Aufruf der Methode "`getFieldTwo(int)`". Aufgrund der Instrumentierung wird daraus der Ausdruck "`getFieldTwo((int)DULog.cp(65,1))`" (Listing A.2, Zeile 85).

Etwas umständlicher muss der Aufruf einer Methode ohne formale Parameter instrumentiert werden. Um die Aufrufsemantik und -reihenfolge beibehalten zu können, wird eine Kombination aus obiger Methode "`public static Object cp(int pos)`" und einer der bereits bekannten Protokollfunktionen "`public static` ⟨*type*⟩ `bin(Object dummy,` ⟨*type*⟩ `ret)`" eingesetzt, welche lediglich ihren zweiten Parameter zurückgeben. Dabei tritt der Aufruf der *CallPoint*-Methode anstelle des Parameters "`dummy`" und die aufzurufende Methode anstelle des zweiten Parameters "`ret`".

Auf diese Weise wird ein Methodenaufruf der Form "`iFunc()`" vom Instrumentierer durch den Ausdruck "`(int)DULog.bin(DULog.cp(86),iFunc())`" ersetzt.

Vergleichsweise trivial gestaltet sich die Überwachung des Kontrollflusses bezüglich Betreten und Verlassen einfacher Methoden – ausgenommen davon sind Konstruktoren, deren Sonderbehandlung im nächsten Absatz näher beschrieben wird. Dazu hält `DULog` die beiden Methoden

```
public static void enter(int pos)
public static void leave(int pos)
```

bereit. Ein Aufruf der ersten Protokollfunktion wird jeder instrumentierten Methode vor der allerersten Anweisung selbiger injiziert. Um sicher zu stellen, dass das Verlassen einer Methode auch dann verzeichnet wird, falls innerhalb der Methode eine nicht behandelte Ausnahme auftritt, welche zum sofortigen Abbruch der Methodenabarbeitung führt, wird der gesamte Rumpf der ursprünglichen Methode von einem "`try/finally`"-Konstrukt umschlossen. Der "`finally`"-Codeblock enthält in der instrumentierten Fassung des Quelltextes lediglich den Aufruf der Methode "`leave`".

Exemplarisch sei dieses Vorgehen an der Methode "`public int getFieldTwo(int)`" aus Listing A.1 (ab Zeile 25) demonstriert. Die instrumentierte Fassung ist ab Zeile 61 in Listing A.2 zu sehen. Dabei enthält Zeile 61 den Aufruf der Protokollfunktion "`enter`", während der Aufruf von "`leave`" in Zeile 70 den Abschluss des "`try/finally`"-Konstrukts bildet.

Konstruktoraufrufe

Im Gegensatz zu einfachen Klassen- oder Instanzmethoden, erfordern Konstruktoren eine Sonderbehandlung, wenn es um die Protokollierung der Instantiierung von Objekten oder des Kontrollflusses hinsichtlich Betreten und Verlassen eines Konstruktors geht. Ursache hierfür ist, dass ein Konstruktor einer Klasse mittels "`this(`⟨*Parameterliste*⟩`)`" oder "`super(`⟨*Parameterliste*⟩`)`"

einen anderen Konstruktor der gleichen Klasse beziehungsweise einen Konstruktor der Superklasse aufrufen kann, wobei über die „⟨*Parameterliste*⟩“, welche auch leer sein mag, eine Auswahl eines eventuell überladenen Konstruktors möglich ist. Problematisch für die Instrumentierung sind die Forderungen der JAVA™-Sprachdefinition, wonach es nur höchstens einen der beiden Aufrufe (*this/super*) geben darf und der entsprechende Aufruf die erste Anweisung im Konstruktor sein muss. Die letzte Anforderung verhindert die bei einfachen Methoden angewandte Vorgehensweise, einen Protokollaufruf der Form "`enter`" als erste Anweisung im Konstruktor einzufügen.

Da die Instantiierung einer Klasse im Wesentlichen vergleichbar mit einem Methodenaufruf ist, jedoch nicht zwangsweise zum Aufruf eines expliziten, vom Programmierer implementierten Konstruktors führt, gibt es in der Protokollbibliothek `DULog` die Methoden:

```
public static Object newCall(int pos, Object lhs)
public static Object newCallCompleted(int pos, int dummy, Object o)
```

Trifft der Instrumentierer auf einen Ausdruck, dessen Auswertung zur Instantiierung einer Klasse führt, so wird diese Anweisung durch einen Aufruf der Protokollfunktion "`newCallCompleted`" ersetzt. Der Ausdruck selbst wird zum letzten Parameter der Funktion. Da die eigentliche Instantiierung die Überdeckung eines komplexen Kontroll- und Datenflusses nach sich ziehen kann, wird der Beginn der Operation durch einen Aufruf der Methode "`newCall`" (als zweiter Parameter "`dummy`" der Methode "`newCallCompleted`") im Protokoll eingeleitet, während abschließend die Ausführung von "`newCallCompleted`" die Ereignissequenz abrundet.

Die Klasse "`DataflowExample`" aus Listing A.1 wird in Zeile 13 instantiiert. Das Ergebnis der Transformation durch den Instrumentierer zeigt Listing A.2 in Zeile 37.

Da Arrays zwar ähnlich wie Objekte behandelt werden, ihnen aber keine Klasse mit Konstruktoren zugrunde liegt, enthält `DULog` eine dedizierte Methode zur Protokollierung von Array-Instantiierungen:

```
public static Object newArray(int pos, int depth, Object array)
```

Der Parameter "`depth`" nimmt dabei die Dimension des Arrays auf, während der letzte Parameter "`array`" mit der ursprünglichen Instantiierungsanweisung belegt wird. Falls es sich beim zu instantiierenden Array um eine n-dimensionale ($n \geq 2$) Matrix handelt, kann jede Komponente der ersten Dimension selbst als $(n-1)$-dimensionale Matrix betrachtet werden.

Sei beispielsweise "`i`" eine 3-dimensionale Matrix, deklariert mittels "`int[][][] i = new int[10][20][]`". Dann sind "`i[0]`" ..."`i[9]`" jeweils paarweise verschiedene 2-dimensionale Matrizen, also vom Typ "`int[][]`" – wobei für die dritte Dimension in diesem Fall noch kein Speicher reserviert wurde, das heißt: $\forall 0 \leq x < 10,\ 0 \leq y < 20$ ist "`i[x][y]`" jeweils eine Referenz auf ein 1-dimensionales Array, welches seinerseits einzelne *int*-Werte aufnehmen kann. Da eine Instantiierung eines multidimensionalen Feldes damit implizit auch eine Instantiierung aller Felder mit geringerer Dimension bedeutet, bildet die Protokollfunktion "`newArray`" dieses Verhalten nach und protokolliert für jede Dimension die jeweilige Instanzkennung des niedriger dimensionierten Sub-Arrays.

Ein Ausdruck der Form "`new int[10][20][]`" wird bei der Instrumentierung beispielsweise durch den semantisch äquivalenten Ausdruck "`(int[][][])DULog.newArray(5,3,new int[10][20][])`" ersetzt.

Bezüglich der Laufzeitüberwachung von Konstruktorausführungen müssen drei Fälle unterschieden werden:

1. Der betrachtete Konstruktor ruft keinen weiteren Konstruktor auf, weder einen der eigenen Klasse noch einen der Superklasse.

2. Der betrachtete Konstruktor ruft zwar mittels "`this()`" beziehungsweise "`super()`" einen weiteren Konstruktor auf, jedoch haben diese aufgerufenen Konstruktoren keine formalen Parameter.

3. Der betrachtete Konstruktor ruft mit Hilfe eines der Ausdrücke "`this(`⟨*Parameterliste*⟩`)`" beziehungsweise "`super(`⟨*Parameterliste*⟩`)`" einen weiteren Konstruktor mit formalen Parametern auf.

Im ersten Fall ist keine besondere Behandlung notwendig, so dass eine Instrumentierung mittels der von den einfachen Methoden übernommenen Protokollfunktionen "`enter`" und "`leave`" möglich ist.

Im zweiten Fall ist nur zu beachten, dass die Überwachungsfunktion "`enter`" nicht als erste Anweisung sondern erst unmittelbar nach einem vorhandenen "`this()`" oder "`super()`" stehen darf. Zwar wird der Kontrollfluss nun nicht mehr absolut exakt protokolliert, da zunächst der betrachtete Konstruktor tatsächlich betreten wird, ehe der Aufruf weitergeleitet wird. Weil aber die Ablaufkontrolle unmittelbar nach Betreten des betrachteten Konstruktors direkt an einen anderen Konstruktor übergeben wird, ohne dass weitere Ereignisse dazwischen auftreten können, genügt diese Betrachtung zum Zwecke der datenflussorientierten Testdatengenerierung und Testmengenoptimierung.

Schwieriger ist hingegen die Instrumentierung eines Konstruktors aus dem dritten Fall. Zwar darf vor "`this(`⟨*Parameterliste*⟩`)`" beziehungsweise "`super(`⟨*Parameterliste*⟩`)`" keine weitere Anweisung im Code stehen, jedoch können die Aktualparameter dieser Aufrufe durchaus das Ergebnis komplexerer Ausdrücke oder gar weiterer Methodenaufrufe sein. Infolge dessen ergeht die Kontrolle nicht unmittelbar an den aufgerufenen Konstruktor, sondern an die Auswertung der Aktualparameter, was einen komplexen Kontrollfluss mit weiteren datenflussrelevanten Ereignissen nach sich ziehen kann.

Als Beispiel diene ein Konstruktor, dessen erste Anweisung "`super(f(), i, g());`" lautet, wobei "`f()`" und "`g()`" zwei statische Methoden sind, während "`i`" ein Parameter des betrachteten Konstruktors selbst ist. Die Instrumentierung muss sicherstellen, dass das Betreten dieses Konstruktors im Laufzeitprotokoll vermerkt wird, ehe die statische Methode "`f()`" ausgeführt wird, da alle Ereignisse, die aufgrund der Auswertung dieser drei Parameter auftreten, bereits als Teil der Abarbeitung des Konstruktors zu zählen sind.

Um dies zu erreichen, verfügt die Protokollbibliothek `DULog` über die Methode

```
public static Object earlyCtorEnter(int pos)
```

welche zusammen mit einer Funktion der bereits bekannten Methodenfamilie

`public static ⟨type⟩ bin(Object dummy, ⟨type⟩ ret)`

hier zum Einsatz kommt. Dabei wird der erste Aktualparameter aus der ⟨*Parameterliste*⟩ im jeweiligen Ausdruck "`this(`⟨*Parameterliste*⟩`)`" beziehungsweise "`super(`⟨*Parameterliste*⟩`)`" von der Methode "`bin`" so umfasst, dass er zum Parameter "`ret`" wird und die Funktion "`bin`" selbst anstelle des Aktualparameters tritt. Die eigentliche Protokollierung des „vorzeitigen Betretens des Konstruktors“ (*early constructor enter: earlyCtorEnter*) erfolgt durch die gleichnamige Methode, deren Aufruf anstelle des Parameters "`dummy`" der Methode "`bin`" eingesetzt wird – wodurch sie zuerst ausgewertet wird.

Durch die Instrumentierung ensteht aus dem erwähnten Ausdruck "`super(f(), i, g());`" (abgesehen von den jeweiligen Ereigniskennungen ⟨*ID*⟩): "`super((int) DULog.bin (DULog.earlyCtorEnter(`⟨ID_1⟩`), DULog.bin(DULog.cp(`⟨ID_2⟩`),f())),(int)DULog.useLocal(`⟨ID_3⟩`,i),(int)DULog.bin(DULog.cp(`⟨ID_4⟩`),g()));`". Dabei wird noch vor dem Aufruf der Methode "`f()`" das Ereignis *earlyCtorEnter* vermerkt, welches mit dem später im Protokoll auftretenden *enter* zur Rekonstruktion des tatsächlichen Kontrollflusses zusammengeführt wird. Ein einfacheres Beispiel enthält auch Listing A.1 in Zeile 8 sowie das entsprechend instrumentierte Gegenstück in Zeile 29 aus Listing A.2.

Parameter von Methoden und Konstruktoren

Ergänzend zur Behandlung von Methoden- und Konstruktoraufrufen ist an dieser Stelle die Beschreibung der Instrumentierung des Quellcodes zur Protokollierung der Definition von Funktionsparametern angebracht. Wird eine Methode oder ein Konstruktor mit einer nicht-leeren Parameterliste aufgerufen, so findet beim Betreten solcher Funktionsblöcke implizit eine Reihe von *defs* lokaler Variablen statt, nämlich die Zuweisung der Aktualparameter an die formalen Parameter der jeweiligen Methode.

Zur Instrumentierung dieser Ereignisse bedarf es keiner spezialisierten Protokollmethode in `DULog`, sondern lediglich der Ergänzung des statischen Instrumentierungsprotokolls um die relevanten Einträge. Letzteres enthält unter anderem zu diesem Zweck eine vierte Spalte, wie in Tabelle A.1 dargestellt, welche weitere Informationen zu einem Ereignis beherbergen kann.

Das Programmbeispiel aus Listing A.1 enthält drei Methoden mit einer nicht-leeren Parameterliste: "`DataflowExample(int)`" (Zeile 17), "`int getFieldTwo(int)`" (Zeile 25) und "`static void main(String[])`" (Zeile 36). Das Betreten der jeweiligen Methoden beziehungsweise des Konstruktors wird entsprechend Listing A.2 mit den Ereignissen „36“, „47“ sowie „67“ identifiziert. Zu jedem dieser Ereignisse enthält die Spalte vier der Tabelle A.1 die Liste der Parameter der jeweiligen Funktion. Wird eine der Funktionen betreten und das entsprechende Ereignis im Protokoll vermerkt, so können aufgrund des Instrumentierungsprotokolls die jeweils zugehörigen Definitionen der Parametervariablen rekonstruiert werden. Umfasst die Parameterliste einer Methode mehr als eine Variable, so werden alle Variablennamen, in der Reihenfolge ihres Auftretens als formale Parameter und durch das Sonderzeichen „#“ getrennt, als Zusatzinformation im Instrumentierungsprotokoll aufgeführt.

Ausnahmebehandlung

Um den Kontrollfluss auch im Falle einer auftretenden Ausnahme (*Exception*) korrekt rekonstruieren zu können, für welche eine Ausnahmebehandlungsroutine vorgesehen ist, enthält die Protokollbibliothek `DULog` die Methode

```
public static void exceptHandlerCall(int pos)
```

Ein individueller Aufruf dieser Methode mit einem jeweiligen Ereigniskennzeichen wird während der Instrumentierung zu Beginn eines jeden, durch das Schlüsselwort *catch* eingeleiteten Ausnahmebehandlungsblocks eingefügt.

Die Beispiel-Klasse `DataflowExample` (Listing A.1) enthält in Zeile 11 den Start einer Ausnahmebehandlung. Die vom Instrumentierer hinzugefügte Probe in Listing A.2 (Zeile 34) zeigt, wie mittels "`exceptHandlerCall`" das Betreten dieses *catch*-Blocks protokolliert und darüber hinaus die Zuweisung (*def)* der geworfenen Ausnahme an den Parameter des *catch*-Abschnitts als Definition einer lokalen Variablen ("`DULog.defLocal(12)`") behandelt wird.

5.1.3 Instrumentierung für die Bedingungsüberdeckung

Die erste Version des Werkzeugs •gEAr zur automatischen Generierung optimaler Testdaten wurde zunächst für das strukturelle Testen von JAVA™-Programmen nach dem Kriterium der Verzweigungsüberdeckung und ausgewählter datenflussorientierter Kriterien umgesetzt. Dazu wurde zunächst die ab Seite 125 beschriebene Instrumentierung eingesetzt. Die ersten experimentellen Einsätze von •gEAr zeigten jedoch, dass sich die Verfahren zur Instrumentierung des Quellcodes und zur Generierung optimaler Testdaten leicht auf weitere Überdeckungskriterien ausdehnen lässt. Dabei eignet sich das Konzept nicht nur für rein am Kontrollfluss orientierte strukturelle Überdeckungen. Denkbar ist auch der Einsatz dieses Verfahrens zur Unterstützung struktureller Grenzwerttests, welche zur Klasse der „funktionsorientierten Qualitätsprüfverfahren“ (siehe Abbildung 2.3) zählen, oder dem in Kapitel 3.6 vorgestellten Mutationstesten.

Obwohl oder gerade weil die in Kapitel 3.3 vorgestellten Kriterien aus der Familie der Bedingungsüberdeckung orthogonal zu den klassischen kontroll- und datenflussorientierten Teststrategien sind, was sich in der Subsumptionsrelation aus Abbildung 3.9 widerspiegelt, bietet es sich an, die in •gEAr eingesetzten Verfahren auch zur Generierung optimaler Testdaten zur Bedingungsüberdeckung von JAVA™-Code einzusetzen. Die zur Überwachung aller Ereignisse im Zusammenhang mit Bedingungsauswertungen notwendige Instrumentierung des Quellcodes erfolgt dabei in Anlehnung an die Protokollierung des Datenflusses, wie sie in Kapitel 5.1.2 ab Seite 125 beschrieben wurde. Auch hierbei wird das freie *ANTLR*-Framework eingesetzt, um den JAVA™-Quellcode zu parsen und in einen abstrakten Syntaxbaum zu transformieren. Die zur Bearbeitung eines JAVA™-ASTs konzipierte Baumgrammatik modifiziert in diesem Falle das Programm so, dass die eingefügten Proben jeweils Aufrufe an die Protokollbibliothek `Logger`[10] darstellen. Diese Protokollklasse vermerkt die Ausführung aller relevanten Anweisun-

[10] Zur besseren Lesbarkeit wurden einerseits der instrumentierte Quellcode in Listing A.6 durch zusätzliche Umbrüche neu angeordnet (*pretty-printed*) und andererseits die vollständige Packagebezeichnung der Protokollbibliotheksklasse `de.fau.cs.swe.sa.conditionCoverage.Logger` zu `Logger` gekürzt.

gen, insbesondere die Ergebnisse der Auswertung atomarer und nicht-atomarer Bedingungen, so dass die zur Bewertung der von einem Testfall erreichten Bedingungsüberdeckung notwendigen Ereignisse rekonstruiert werden können.

Wie bei der Beschreibung des datenflussorientierten Instrumentierers sei auch hier auf ein umfassendes und durchgängiges JAVA™-Beispiel zurückgegriffen. Der ursprüngliche Quellcode des Programms ist dabei im Anhang in Listing A.5 (Seite 242) dargestellt. Das Ergebnis der Instrumentierung zeigt Listing A.6 (Seite 243). Das zugehörige Instrumentierungsprotokoll ist in Tabelle A.2 (Seite 245) wiedergegeben.

Explizite Prädikate

Das in Kapitel 3.3 skizzierte Kriterium der ***Minimalen Mehrfach-Bedingungsüberdeckung*** (*minimal multiple condition coverage*) erfordert die Ermittlung einer Testfallmenge, bei deren Ausführung jede Teilbedingung mindestens einmal sowohl den Wert *wahr* als auch den Wert *falsch* annimmt – dies gilt sowohl für alle atomaren und alle nicht-atomaren Teilbedingungen als auch insbesondere für den gesamten Bedingungsausdruck. Im Gegenzug genügt eine Testfallmenge dem Kriterium der ***Einfachen Bedingungsüberdeckung*** (*simple condition coverage*), falls bei Ausführung des Programms mit den entsprechenden Testfällen lediglich jede atomare Teilbedingung die beiden möglichen Wahrheitswerte annimmt. Um einen Testfall hinsichtlich der von ihm erreichten Überdeckung anhand eines beliebigen Bedingungsüberdeckungskriteriums bewerten zu können, ist eine Instrumentierung erforderlich, welche jede Granularitätsstufe innerhalb eines Prädikates entsprechend unterscheiden kann. Aus diesem Grund kennt der für •gEAr vorgesehene Quellcode-Instrumentierer die folgenden Bedingungsarten:

- *p (primitive)*: primitives atomares Prädikat, zum Beispiel eine einzelne Boole'sche Variable oder der Aufruf einer Methode mit Boole'schem Rückgabewert;
- *a (atomic)*: atomares nicht-primitives Prädikat, typischerweise auf einer Boole'schen Relation basierend, wie zum Beispiel die Vergleichsoperatoren „<“ oder „==“;
- *c (combined)*: zusammengesetztes Prädikat, deren Teilbedingungen selbst entweder zusammengesetzt oder atomar beziehungsweise primitiv sein können;
- *b (branch)*: gesamter Bedingungsausdruck, dessen Auswertung zum Beispiel als Operand einer `IF`-Anweisung über den zu verfolgenden Pfad entscheidet.

Um die tatsächlich während einer Testausführung erzielten Auswertungsergebnisse einzelner Bedingungsausdrücke entsprechend dem gewünschten Überdeckungskriterium nachträglich rekonstruieren zu können, müssen sowohl alle primitiven und atomaren Teilbedingungen als auch alle zusammengesetzten Teilprädikate jeweils getrennt überwacht und daher entsprechend instrumentiert werden. Dabei ist bei der Instrumentierung insbesondere darauf zu achten, dass die Auswertungsreihenfolge der Teilbedingungen semantisch äquivalent beibehalten wird. Darüber

hinaus muss auch die vom Compiler entsprechend der Sprachspezifikation verfolgte Auswertungsstrategie aufrechterhalten werden: Teilbedingungen, die aufgrund der so genannten *Kurzschlussauswertung* (*shortcut evaluation* oder *partial evaluation*, siehe Kapitel 3.1) im ursprünglichen Programm nicht ausgewertet würden, dürfen auch in der instrumentierten Fassung nicht ausgewertet werden.

Um obigen Anforderungen gerecht zu werden, verfügt die Protokollbibliothek `Logger` für Boole'sche Teilbedingungen über die Methode:

```
public static boolean _logB(boolean b, long id)
```

Trifft der Instrumentierer auf ein Boole'sches Prädikat, zum Beispiel als Operator eines `IF`- oder `WHILE`-Konstrukts, so wird dieses Prädikat zunächst als Ganzes von einem Aufruf der Protokollmethode so umschlossen, dass das ursprüngliche Prädikat zum ersten Parameter der Methode wird. Gleichzeitig legt der Instrumentierer im Instrumentierungsprotokoll einen Eintrag mit einer eindeutigen Kennung "`id`" an, welche das Ereignis als Auswertung einer Bedingung vom Typ „*b*“ beschreibt, und setzt diese Kennung als zweiten Parameter der Methode "`_logB`" ein. Wird das Prädikat erfolgreich ausgewertet und "`_logB`" somit aufgerufen, vermerkt diese Methode die Ereigniskennung "`id`" zusammen mit dem aktuellen Wahrheitswert des Prädikats und einer Thread-Kennung im Ausführungsprotokoll.

Darüber hinaus stellt jedes Prädikat entweder eine primitive (*p*), eine atomare (*a*) oder eine zusammengesetzte (*c*) Teilbedingung dar, so dass sie rekursiv in Teilbedingungen zerlegt und entsprechend instrumentiert wird. Dabei wird bei der Einfassung zusammengesetzter Teilprädikate auf die Auswertungsreihenfolge der sie verbindenden Operatoren geachtet.

Ein einfaches Beispiel zeigt Listing A.5 in Zeile 25. Die Abbruchbedingung "`i < 7`" der `FOR`-Schleife stellt sowohl ein Verzweigungsprädikat (*b*) als auch eine atomare Teilbedingung dar (*a*). Entsprechend obiger Beschreibung wird dieser Ausdruck im instrumentierten Quellcode durch "`Logger._logB(Logger._logB(i < 7, 16), 17)`" (Listing A.6, Zeile 62) ersetzt. Die zugehörigen Ereigniskennungen „16“ und „17“ sind in Tabelle A.2[11] dargestellt.

Ein etwas komplexeres Beispiel eines zusammengesetzten Prädikats mit verschiedenen Operatoren unterschiedlicher Priorität enthält Listing A.5 in Zeile 11. Die zugehörige instrumentierte Fassung stellt Listing A.6 ab Zeile 22 dar. Das gesamte Prädikat (Typ *b* in Tabelle A.2) wird vom Aufruf der Methode "`_logB`" in Zeile 22 mit der Ereigniskennung „11“ umschlossen. Gleichzeitig handelt es sich um eine zusammengesetzte (*c*) Bedingung, deren Auswertung mittels des Methodenaufrufs in Zeile 23 mit der Kennung „10“ protokolliert wird. Das nicht-atomare Prädikat zerfällt entsprechend der Operatorpriorität[12] in zwei ebenfalls zusammengesetzte Teilbedingungen: "`a < 10 && b > 0`" sowie "`c > 10 && d`". Deren Ausführung wird im Erfolgsfall von den beiden Protokollaufrufen in Zeile 25 (Ereigniskennung „6“) beziehungsweise Zeile 33 (mit "`id`" „9“) vermerkt, welche durch den Operator niedrigerer Priorität „`||`“ verknüpft sind. Ist die erste Teilbedingung ("`a < 10 && b > 0`") wahr, so erübrigt sich laut JAVA™-Sprachdefinition die Auswertung der zweiten Teilbedingung, was auch nach der Instrumentierung semantisch

[11] Hinweis: Die in der Tabelle angegebene Zeilennummer bezieht sich auf den ursprünglichen instrumentierten Quellcode aus Listing A.6, also *vor* dem *pretty-printing*.

[12] In JAVA™ hat der Operator `&&` Vorrang vor `||`.

korrekt erfüllt ist. Weiterhin kann das erste der beiden Teilprädikate in zwei atomare (*a*) Teilbedingungen "`a < 10`" sowie "`b > 0`" zerlegt werden. Deren eventuelle Auswertung wird durch die Aufrufe der Methode "`_logB`" in Zeile 27 (mit "`id`" „4“) beziehungsweise Zeile 29 (Ereigniskennung „5“) protokolliert. Analog zerfällt das zweite Teilprädikat in eine atomare (*a*) und eine primitive (*p*) Teilbedingung "`c > 10`" sowie "`d`", die entsprechend Zeile 35 und Zeile 37 mit den jeweiligen Kennungen „7“ beziehungsweise „8“ instrumentiert werden.

Zusätzlich zu den beiden Wahrheitswerten *wahr* und *falsch* kann eine (Teil-)Bedingung auch einen „undefinierten“ Wert annehmen, selbst wenn sie trotz *partial evaluation* tatsächlich auszuwerten ist. Dies ist dann der Fall, wenn während der Auswertung der Bedingung eine Ausnahme (*exception*) auftritt und dadurch die begonnene Auswertung abgebrochen wird. Um nach einem Testlauf feststellen zu können, ob ein Prädikat vollständig, sprich ohne Ausnahme, ausgewertet wurde, verfügt `Logger` über die Methoden:

```
public static boolean _catchException(long start, boolean b, long end)
public static long _startExpression(long id)
public static long _endExpression(long id)
public static void _handleException(long row)
```

Die Methode "`_catchException`" umschließt dabei das zu überwachende und seinerseits instrumentierte Prädikat, welches an Stelle des zweiten Parameters "`b`" tritt und dessen Wert zurückgegeben wird. Die beiden Hilfsmethoden "`_startExpression`" und "`_endExpression`" werden jeweils entsprechend anstelle der Parameter "`start`" beziehungsweise "`end`" der Methode "`_catchException`" eingesetzt. Ihre Aufgabe ist es, den Beginn und das erfolgreiche Ende einer Prädikatsauswertung zu protokollieren. Dazu übergibt ihnen der Instrumentierer anstelle ihrer Parameter "`id`" jeweils die Ereignisskennung des Gesamtprädikats. Ein Aufruf der Methode "`_handleException`" nach jedem *catch*-Block sowie am Ende jeder instrumentierten Methode stellt sicher, dass im Falle einer Ausnahme innerhalb einer Prädikatsauswertung, der zugehörige "`_startExpression`"-Eintrag im Laufzeitprotokoll mit einem "`_handleException`"-Ereignis anstelle des erwarteten "`_endExpression`"-Eintrags abgeschlossen wird.

Die Verwendung dieser Protokollmethoden wird in Listing A.6 durch den Codeblock ab Zeile 20 (insbesondere Zeile 21, Zeile 43 sowie Zeile 83) anschaulich demonstriert, welcher aus Zeile 11 in Listing A.5 hervorgegangen ist.

Ternärer Operator

Das Ergebnis der Auswertung eines Ausdrucks mit einem ternären Operator der Form "$\langle op1 \rangle$? $\langle op2 \rangle$: $\langle op3 \rangle$", dessen erster Operand $\langle op1 \rangle$ ein beliebiger Ausdruck mit einem Boole'schen Wert sein muss, ergibt je nach Wahrheitswert von $\langle op1 \rangle$ entweder den Wert von $\langle op2 \rangle$, falls $\langle op1 \rangle$ zu *wahr* ausgewertet wird, oder von $\langle op3 \rangle$ sonst. Insofern ist ein solches Konstrukt vergleichbar mit einer abgekürzten `IF`-Verzweigung. Daher erhält das „Prädikat“ des ternären Operators (hier dargestellt durch $\langle op1 \rangle$) keine besondere Behandlung gegenüber expliziten Prädikaten in Konstrukten wie `IF`, `WHILE` oder `FOR`.

Ein komplexeres Beispiel mit einer kaskadierten Verwendung des ternären Operators enthält Listing A.5 in Zeile 26. Wie ab Zeile 64 in Listing A.6 zu erkennen, wird jeder Boole'sche

Ausdruck, welcher einen ersten Operanden (⟨*op1*⟩) eines ternären Operator repräsentiert, entsprechend so behandelt, wie vorangehend für explizite Prädikate beschrieben.

Switch/Case-Konstrukte

Zwar stellt der Vergleichsparameter eines SWITCH/CASE-Konstruktes kein Prädikat im herkömmlichen Sinne dar, dennoch handelt es sich um eine Verzweigungsstruktur, die den tatsächlichen Kontrollfluss zur Laufzeit des Programms in Abhängigkeit von seinem Vergleichsparameter beeinflusst. Aufgrund seiner Semantik kann jedes SWITCH/CASE-Konstrukt durch eine Sequenz einfacher IF-Verzweigungen ausgedrückt werden, wobei jeweils der Vergleichsparameter mit der Sprungmarke im CASE-Block verglichen wird. Aufgrund dieser Sichtweise können der Vergleichsparameter zusammen mit den Sprungmarken im weitesten Sinne ebenfalls als eine Abfolge von „Bedingungen“ betrachtet werden – was auch maßgeblich zur Erfüllung der Subsumptionsbeziehung zwischen *Bedingungs-/Entscheidungsüberdeckung* und *Verzweigungsüberdeckungstest* (siehe Abbildung 3.9) beiträgt. Zu diesem Zweck verfügt die Protokollbibliothek Logger für die Methodenfamilie:

```
public static final ⟨type⟩ _enterSwitch(⟨type⟩ l, long id)
public static void _logC(long id)
public static void _leftSwitch(long id)
```

wobei es je eine Methode "_enterSwitch" für die ganzzahligen Datentypen *byte*, *char*, *short*, *int* und *long* anstelle von ⟨*type*⟩ gibt. Bei der Instrumentierung wird zunächst der Vergleichsparameter der SWITCH-Anweisung von der Methode "_enterSwitch" umschlossen, so dass dieser anstelle des Parameters "l" tritt. Erreicht die Programmausführung das auf diese Weise instrumentierte SWITCH/CASE-Konstrukt, so wird die Methode "_enterSwitch" nach erfolgreicher Auswertung des Vergleichsparameters ausgeführt, welche den Start eines solchen SWITCH/CASE-Blocks mit der Ereigniskennung "id" protokolliert und den ersten Parameter zurückgibt, so dass dieser für den Vergleich mit den einzelnen Sprungmarken verwendet werden kann. Welcher der „CASE“-Zweige tatsächlich angesprungen wurde, lässt sich nachträglich auf Basis der Methode "_logC" rekonstruieren. Je ein solcher Aufruf mit einer eindeutigen (individuellen) Kennung "id" wird zu Beginn eines jeden CASE-Blocks eingefügt. Trifft keine der vorhandenen CASEs zu und gibt es keinen default-Zweig, so schließt ein Aufruf der Methode "_leftSwitch" unmittelbar nach dem gesamten SWITCH/CASE-Block die durch "_enterSwitch" im Protokoll eingeleitete Auswertung des Blocks.

Das Programmbeispiel aus Listing A.5 enthält ab Zeile 16 ein SWITCH/CASE-Konstrukt mit dem Vergleichsparameter "a". Wie in Listing A.6 dargestellt, umschließt ein Aufruf der Methode "_enterSwitch" diesen Vergleichsparameter (Zeile 48). Die Ausführung der einzelnen CASE-Blöcke wird durch die jeweiligen Aufrufe der Methode "_logC" in Zeile 50, Zeile 53 und Zeile 57 protokolliert. Falls mehrere CASE-Abschnitte sequentiell abgearbeitet werden, wie in diesem Beispiel falls "a" den Wert „0“ hat, so wird lediglich der erste Aufruf von "_logC" berücksichtigt, welcher unmittelbar nach einem Protokolleintrag aufgrund der Methode "_enterSwitch" entstanden ist. Hat die Variable "a" während der Ausführung des SWITCH/CASE-Konstruktes hingegen einen Wert, welcher mit keinem der Vergleichsmarken übereinstimmt, so folgt im Aus-

führungsprotokoll auf den Eintrag zu "`_enterSwitch`" unmittelbar ein von "`_leftSwitch`" vermerktes Ereignis (Zeile 60).

Polymorphie und Overloading

Moderne objekt-orientierte Programmiersprachen zeichnen sich unter anderem durch die Konzepte *Polymorphie* (siehe auch Kapitel 3.1) und *Überladen von Methoden* (*overloading*) aus.

Hinter der *Polymorphie* steht der Gedanke, eine beliebige Variable v in einem Programm mit einem bestimmten Objektreferenztyp (Klasse) C zu deklarieren, dieser Variablen v jedoch Objekte zuzuweisen, welche entweder Instanzen dieser Klasse C oder einer beliebigen Unterklasse ($\tilde{C}$) von C sind und welche daher zwar unterschiedlichen Datentyps sein können, jedoch typkompatibel zu C sein müssen. Enthält darüber hinaus die Superklasse C eine Methode $m()$, die in einer der Unterklassen von einer gleichnamigen Methode $\tilde{m}()$ mit identischer Signatur überschrieben wird, und enthält das Programm einen Aufruf der Form $v.m()$, so ist statisch nicht feststellbar, welche der beiden Methoden $m()$ beziehungsweise $\tilde{m}()$ tatsächlich ausgeführt werden wird, da erst zur Ausführungszeit feststeht, welche Art von Instanz die Variable referenziert. Abbildung 3.3 (Kapitel 3.1, Seite 40) stellt eine Klassenhierarchie dar. In einer sie nutzenden Applikation können Variablen vom Datentyp `Obst` deklariert werden, dieser Variablen können jedoch beliebige Instanzen einer der drei Klassen `Obst`, `Apfel` oder `Birne` zugewiesen werden. Entsprechend ist erst während der Ausführung der Applikation bekannt, ob die Methode `vergleicheMit(Object o)` der Klasse `Obst` oder der Klasse `Apfel` auszuführen ist.

Im Gegenzug bezeichnet *Overloading* ein Konzept, bei dem innerhalb einer Klasse mehrere Methoden mit gleichem Namen jedoch unterschiedlicher Parameterliste definiert werden. Dabei können die Parameterlisten der beiden Methoden auch gleich lang sein, müssen dann jedoch mindestens einen Parameter enthalten, der jeweils einen anderen Datentyp aufweist, die Methoden müssen also eine unterschiedliche Signatur haben.

Wird in einer JAVA™-Applikation von Polymorphie oder Overloading Gebrauch gemacht, so kann die „Auswahl“ der tatsächlich auszuführenden Methode durch den Compiler oder die Virtuelle Maschine im weitesten Sinne ebenfalls als eine Form einer „bedingten“ Sprunganweisung betrachtet werden. Ebenso kann beispielsweise die Entscheidung hinsichtlich der aufzurufenden Methode einer potentiell polymorphen Variablen durch eine Sequenz von `IF`-Verzweigungen simuliert werden, welche mittels des `instanceof`-Operators das gerade referenzierte Objekt hinsichtlich seines Datentyps untersucht. Aus diesem Grund können die klassischen Kriterien der Bedingungsüberdeckung dahingehend erweitert werden, dass diese die Ausführung aller potentiellen Kandidaten eines polymorphen Methodenaufrufs während des Tests fordern.

Das Beispiel aus Listing A.5 enthält in Zeile 38 und Zeile 41 zwei überladene Methoden sowie je einen Aufruf dieser Methoden in Zeile 32 beziehungsweise Zeile 33. Erreicht der Kontrollfluss einen der beiden Aufrufe, so muss entschieden werden, welche der beiden gleichnamigen Methoden aufzurufen sind[13].

Analog verhält es sich mit den beiden polymorphen Variablen `x` und `y` sowie den beiden Aufrufen der gleichnamigen Methode "`polymorphic()`" in Zeile 34 respektive Zeile 35. Die

[13] Streng genommen fällt der Compiler diese Entscheidung bereits zur Übersetzungszeit.

Klasse `A` deklariert eine Methode namens "`polymorphic()`" in Zeile 50 und die von ihr abgeleitete Klasse `B` überschreibt diese Methode in Zeile 56. Die beiden Variablen `x` und `y` sind zwar beide vom Typ `A`, jedoch referenziert `x` eine Instanz der Klasse `A` (Zeile 29) während `y` auf ein Objekt vom Typ `B` (Zeile 30) verweist. Während der Ausführung der Methodenaufrufe in Zeile 34 beziehungsweise Zeile 35 muss zur Laufzeit entschieden werden, zu welcher Methode der Kontrollfluss abzuzweigen hat.

Betrachtet man die vorangehend geschilderten Entscheidungen jeweils als eine besondere Form der Bedingung, so lässt sich eine „einfache Bedingungsüberdeckung“ dadurch erzielen, dass alle überladenen beziehungsweise alle bezüglich Polymorphie äquivalenten Methoden während des Testens mindestens einmal ausgeführt werden. Präzisieren kann man die „Bedingungsüberdeckung“ im Falle der Polymorphie aufgrund einer statischen Analyse, welche die potentiellen Typen T aller Objekte identifiziert, die eine polymorphe Variable "`v`" an der Stelle eines Methodenaufrufs der Form "`v.m(`⟨*Parameterliste*⟩`)`" referenzieren kann, indem man die „Überdeckung“ selbiger fordert – das heißt, für alle T muss die Testfallmenge mindestens einen Testfall enthalten, so dass "`v`" zum Zeitpunkt des Aufrufs "`v.m(`⟨*Parameterliste*⟩`)`" auf ein Objekt vom Typ T verweist. Prototypisch wurde in der Protokollbibliothek `Logger` eine Unterstützung für die erste Variante vorgesehen und dazu die Methoden

```
public static void _enterMethod(int id)
public static void _leftMethod(int id)
```

umgesetzt. Je ein Aufruf der Methode "`_enterMethod`" wird zu Beginn eines jeden Methodenrumpfs eingesetzt, wobei der aus Kapitel 5.1.2 bekannte Sonderfall einer Konstruktordelegation mittels "`this(`⟨*Parameterliste*⟩`)`" oder "`super(`⟨*Parameterliste*⟩`)`" entsprechend berücksichtigt wird. Darüber hinaus ergänzt der Instrumentierer jede Methode zusätzlich mit einem Aufruf von "`_leftMethod`". Auf diese Weise ist es möglich, die Ausführung jeder Methode zu protokollieren und damit Rückschlüsse auf die Überdeckung im Sinne der Polymorphie beziehungsweise des Overloadings zu ziehen.

Bezogen auf das Beispiel aus Listing A.5 und den beiden Methoden "`polymorphic()`" in den Klassen `A` (Zeile 50) und `B` (Zeile 56) ist die Instrumentierung zur Überwachung der Methodenausführungen in Listing A.6 jeweils in Zeile 116 sowie Zeile 118 beziehungsweise Zeile 123 sowie Zeile 125 dargestellt.

5.1.4 Ausführung und Überdeckungsbestimmung

Wie bereits Eingangs zu Beginn von Kapitel 5.1.1 angedeutet, ist eine Ausführung des instrumentierten Programms notwendig, um die von einem Testfall überdeckten Entitäten entsprechend dem geforderten Testkriteriums exakt rekonstruieren zu können. Da sich dieser Prozess unabhängig vom Überdeckungskriterium gestalten lässt und die Aufbereitung der Ausführungsprotokolle für die Datenflusskriterien ungleich anspruchsvoller ist als die entsprechende für die Bedingungsüberdeckung, sei an dieser Stelle exemplarisch die erstere detaillierter vorgestellt.

Nachdem der zu testende (und somit zu überwachende) Programmausschnitt wie in Abbildung 5.1 instrumentiert und mit den restlichen Programmbestandteilen zu einer ausführbaren

Applikation (*SUT: System Under Test*) zusammengestellt wurde, kann ein Testfall nun jederzeit ausgeführt werden. Dazu sollten jedoch noch wichtige Aspekte der Architektur bedacht werden, die sich auf die Stabilität und die Performanz der Testfalloptimierung auswirken.

Da •gEAr zunächst für die Generierung von Testdaten für JAVA™-Programme konzipiert und darüber hinaus selbst in JAVA™implementiert ist, bietet es sich an, die zu testende Applikation mit jedem Testfall direkt aus •gEAr heraus und somit in der gleichen Virtuellen Maschine (VM) zu starten. Alternativ könnte jedoch für jede Testfallausführung eine eigene VM instantiiert werden. Vorteile der ersten Variante sind zweifellos eine höhere Startgeschwindigkeit, da die Initiierung einer Virtuellen Maschine selbst einen gewissen Overhead erzeugt, ehe das eigentliche Testprogramm ausgeführt werden kann. Aufgrund ausgereifter Puffermechanismen moderner Betriebssysteme relativiert sich dieser Vorteil zunehmend. Darüber hinaus leidet dieses direkte Vorgehen unter drastischen Nachteilen. So muss beachtet werden, dass die Initialisierung statischer Variablen nur einmalig beim Laden der Klasse in die Virtuelle Maschine durchgeführt wird, weshalb die Ausführung weiterer Testfälle eine gezielte und potentiell fehleranfällige Rücksetzung solcher Klassenfelder auf den Initialzustand erfordert. Desweiteren kann die Ausführung eines fehlerhaften Testobjektes in einer eigenen VM leichter abgebrochen werden, falls dieses nicht innerhalb vorgegebener Zeit selbständig terminiert. Schließlich birgt das Ausführen des Testsubjekts in der gleichen Virtuellen Maschine die Gefahr, dass ein Fehler im SUT oder in der Maschine zum Absturz der gesamten Applikation einschließlich des Testgenerators führt. Desweiteren kann. Aus diesen Betrachtungen wurde in •gEAr die zweite Alternative verfolgt: Jeder Testfall wird in einer eigenen Instanz der Virtuellen Maschine unabhängig von anderen Testfällen durchgeführt.

Ablauf der Testausführung

Da Evolutionäre Verfahren mit einer größeren Population von „Individuen“ hantieren, entstehen in jeder Generation mehrere unterschiedliche Testfälle, die individuell ausgeführt werden müssen, um die von ihnen jeweils erreichte Überdeckung zu messen. Im Allgemeinen beansprucht die Ausführung des zu testenden Systems den größten Anteil an der Laufzeit der Testdatengenerierung und -optimierung, gerade bei größeren Testobjekten und nicht zuletzt aufgrund der Instrumentierung. Da die verschiedenen zu bewertenden Testfälle unabhängig voneinander sind, bietet es sich an, die Ausführung zu parallelisieren.

Die grundsätzliche Architektur der Komponenten und ihre Interaktion bei der parallelen Testausführung ist dabei in Abbildung 5.2 dargestellt. Auf einem dedizierten Rechner läuft die eigentliche Instanz der sogenannten •gEAr-*Workbench*, also der Zentralknoten, auf dem das SUT instrumentiert und für die Testausführung vorbereitet wurde. Auf beliebigen geeigneten Rechnern[14] kann jeweils mindestens ein sogenannter *Remote Execution Manager (REE)* installiert werden. Dies ist ein Dämondienst, welcher im Hintergrund läuft, idealerweise mit dem Rechner automatisch startet und die Anfragen von beliebigen *Workbenches* entgegennimmt und delegiert.

Wird eine Testdatenerstellung an der *Workbench* angestoßen, so werden zunächst ein *Local Execution Manager* und eine *Optimisation Engine* instantiiert. Letztere umfasst das heuristi-

[14]Mindestens ein *Remote Execution Manager* kann auch auf dem Zentralknoten ausgeführt werden.

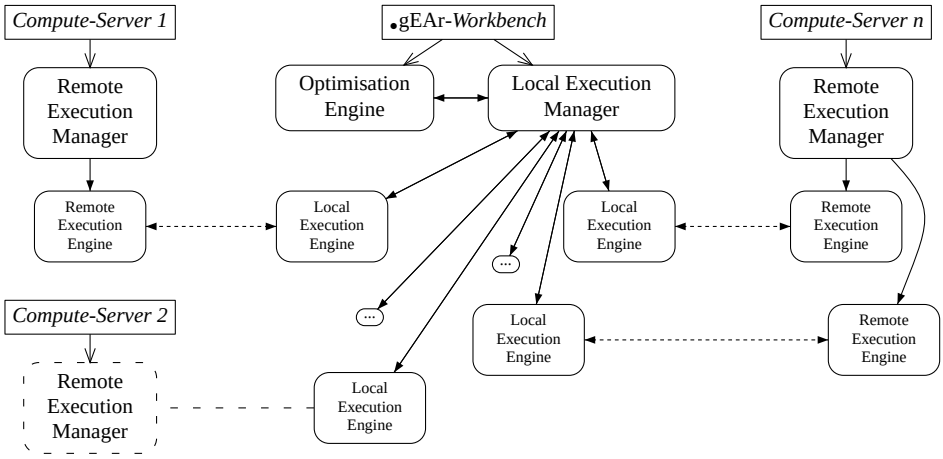


Abbildung 5.2: Architektur und Datenkommunikation der verteilten Testausführung in •gEAr

sche Verfahren zur Testdatengenerierung, während ersterer für die Abarbeitung der jeweils noch nicht ausgeführten Testfälle verantwortlich ist. Um dieser Aufgabe gerecht zu werden, instantiiert der *Local Execution Manager* unmittelbar nach seinem Start je eine *Local Execution Engine (LEE)* für jeden angegebenen Testausführungsknoten – wobei leistungsfähige *Compute-Server* auch mehrfach aufgeführt werden können. Jede *Local Execution Engine* baut eine Verbindung zum ihr zugewiesenen *Remote Execution Manager* auf, woraufhin dieser bei erfolgreicher Authentifizierung eine *Remote Execution Engine* instantiiert. Fortan kommunizieren nur noch die *Local Execution Engines* mit ihren zugehörigen *Remote Execution Engines*. Kann eine der *Local Execution Engines* keinen *Remote Execution Manager* kontaktieren, so versucht sie dies in regelmäßigen Abständen erneut. Gleiches gilt auch, wenn ein *Compute-Server* während einer Testausführung abstürzt oder ausgeschaltet wird: Dabei gibt die entsprechende *Local Execution Engine* den angenommen Testfall wieder zurück an den *Local Execution Manager*, damit der Testfall einer anderen *Local Execution Engine* übergeben werden kann – wodurch ein hohes Maß an Fehlertoleranz und eine optimale Lastverteilung während der Testfallabarbeitung erreicht wird.

Der grundsätzliche Ablauf der dynamischen Testfallanalyse ist in Abbildung 5.3 skizziert. Wie vorangehend erläutert, baut jede *Local Execution Engine* zunächst eine Netzwerkverbindung zu einer *Remote Execution Engine* noch ehe der erste Testfall zur Ausführung ansteht. Dabei übermittelt die *LEE* gleich nach erfolgreicher Kopplung das fertig instrumentierte *System Under Test* und das zugehörige Instrumentierungsprotokoll. Weil beide Informationen unabhängig von einer individuellen Testausführung sind, werden sie von der *REE* jeweils lokal gespeichert und verbleiben dort bis zum Beenden oder Abbrechen der Verbindung zwischen *LEE* und ihrer *REE*. Hat die *Optimisation Engine* einen neuen Testfall erstellt und steht dieser somit zur Ausführung an, so wird er zunächst dem *Local Execution Manager* übergeben, welcher ihn an die nächste freie und ausführungsbereite *LEE* übergibt. Da *Optimisation Engine* und *Local Execution Manager* in nebenläufigen Threads ausgeführt werden, kann der erste Testfall

bereits ausgeführt werden, während die *Optimisation Engine* noch weitere Testfälle generiert. Die empfangende *LEE* sendet den Testfall an die *REE*, wo er in einer eigenen Virtuellen Maschine ausgeführt wird. Das vom Protokollsubsystem dabei erstellte Ausführungsprotokoll wird von der *REE* eingelesen und mit der statischen Information aus dem Instrumentierungsprotokoll zusammengeführt und somit zur vollständigen Überdeckungsinformation aufgearbeitet. Die auf diese Weise erstellte Datenstruktur wird anschließend von der *REE* an die *LEE* übermittelt. Daraufhin wird das Ausführungsprotokoll auf dem *Compute-Server* gelöscht; die *Local Execution Engine* markiert den Testfall als erfolgreich ausgeführt, ergänzt seine Datenstruktur um die zugehörige Überdeckungsinformation und meldet dies dem *Local Execution Manager* zusammen mit der Bereitschaft dieser *LEE* zur Ausführung des nächsten Testfalls.

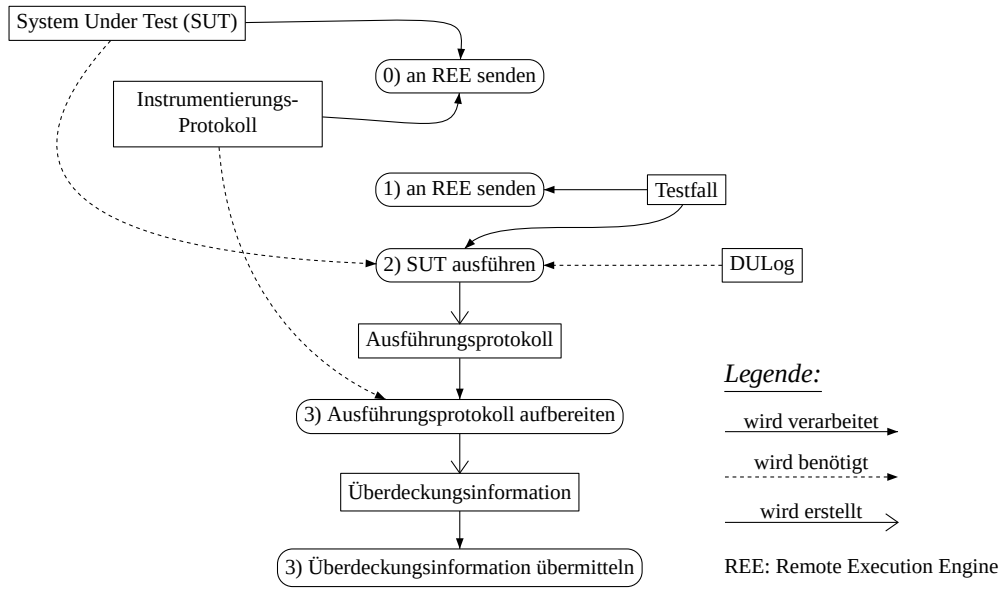


Abbildung 5.3: Ausführung des instrumentierten zu testenden Systems

Terminiert eine Instanz des zu testenden Systems *SUT* nicht innerhalb einer vorgegebenen maximalen Zeitspanne, zum Beispiel infolge eines Programmierfehlers, so bricht die *REE* die Ausführung selbständig ab, indem sie die entsprechende Virtuelle Maschine beendet. Um den Absturz oder das gezielte Herunterfahren eines *Compute-Servers* feststellen zu können, nutzen *LEE* und *REE* ein Heartbeat-Protokoll: Beantwortet die *REE* eine Anfrage auf Lebenszeichen nicht rechtzeitig, so gilt sie als unwiederbringlich verloren. Daraufhin bricht die *LEE* die Verbindung ab, meldet dies dem *Local Execution Manager* und versucht fortan zyklisch, eine neue Verbindung zum zugehörigen *Remote Execution Manager* aufzubauen. Wird auf diese Weise eine eventuell laufende Testausführung abgebrochen, so übergibt der *Local Execution Manager* den entsprechende Testfall einer anderen, freien und betriebsbereiten *LEE*. Nachdem alle dynamisch zu analysierenden Testfälle abgearbeitet wurden, teilt der *Local Execution Manager* dies der *Optimisation Engine* mit, woraufhin diese ihre Arbeit erneut aufnehmen kann.

Rekonstruktion des Datenflusses

Die in Kapitel 5.1.2 und Kapitel 5.1.3 vorgestellte Instrumentierung dient dazu, die während der Ausführung des zu testenden Programms mit einem Testfall tatsächlich überdeckten Entitäten zusammen mit dem abgearbeiteten Kontrollfluss soweit zu rekonstruieren, dass der entsprechende Testfall hinsichtlich der von ihm erzielten Überdeckung quantitativ bewertet werden kann. Wird eine JAVA™-Applikation, die teilweise von •gEAr entsprechend Kapitel 5.1.2 mit Proben zur Datenflussrekonstruktion versehen wurde, analog obiger Darstellung von einer *Remote Execution Engine* ausgeführt, so führt jede Auswertung einer Probe zur Ergänzung eines *Ausführungsprotokolls* mit einem zugehörigen Datensatz.

Um die Protokollierung während der Programmausführung möglichst effizient zu gestalten, wurde bereits in Kapitel 5.1.2 angedeutet, dass jegliche, statisch zur Instrumentierungszeit ermittelbare Information in einem Instrumentierungsprotokoll erfasst wird. Das Ausführungsprotokoll enthält nur noch Verweise auf Einträge in dem zugehörigen Instrumentierungsprotokoll sowie weitere Daten, die lediglich zur Ausführungszeit ermittelt werden können. Demnach besteht jeder Datensatz, also jeder von einer Probe angelegte Protokolleintrag, im Falle der datenflussorientierten Instrumentierung aus genau vier ganzen[15] Zahlen:

1. Ereigniskennung (*ID*): Diese identifiziert eindeutig das Ereignis, so dass es aufgrund des Instrumentierungsprotokolls mit den statischen Informationen angereichert werden kann. Zusätzlich gibt es Kennungen für spezielle Laufzeitereignisse, wie zum Beispiel die Erzeugung eines neuen Threads oder das Ende des Protokolls.

2. Threadkennung (*T*): Falls das *System Under Test* mit mehreren Threads umgesetzt wurde, können hier die Ereignisse einzelner Threads unterschieden werden. Darüber hinaus wird jeder Klasseninitialisierung ein eigener Thread zugeordnet, da diese von der Virtuellen Maschine selbst vorgenommen wird und nicht in einem der Programmthreads ausgeführt wird.

3. Instanzkennung (*I*): Bei Feldzugriffen identifiziert die Instanzkennung eindeutig das Objekt, auf dessen Feld zugegriffen wird. Im Falle von Prädikaten erhält jede Auswertung eine eigene „Instanzkennung“.

4. Daten (*D*): Enthält zusätzliche Informationen, die jeweils ereignispezifisch sind. Wird zum Beispiel das Ende einer Prädikatsauswertung protokolliert, so enthält dieses Feld das Ergebnis: 0 für *falsch* und 1 für *wahr*.

Wird das instrumentierte Programm aus Listing A.2 ausgeführt, welches ursprünglich auf den Code aus Listing A.1 zurückgeht, so entsteht zunächst das entsprechend aufbereitete Ausführungsprotokoll aus Tabelle A.3. Die erste, mit „Nr.“ überschriebene Spalte wurde nachträglich hinzugefügt und ist nicht Bestandteil des eigentlichen Protokolls. Die Protokolldatei enthält in binärem Format und ohne Zeilenumbrüche lediglich die Informationen aus den darauf folgenden

[15] Dabei wird der JAVA™-Datentyp *int* mit je 32 Bit verwendet.

vier Spalten. Die sich anschließenden Spalten wurden aufgrund des „Primärschlüssels“ Ereigniskennung (*ID*) aus dem Instrumentierungprotokoll ergänzt.

Nachdem als erstes von der Virtuellen Maschine ein neuer Thread (Nr. 1) zur Ausführung des Hauptprogramms initiiert wurde, muss als nächstes die Hauptklasse mit der *main()*-Methode geladen und initialisiert werden, wofür ein zweiter Thread (Nr. 2) instantiiert wird. Die Klasseninitialisierung (Nr. 3–6) besteht hier lediglich aus der Zuweisung je eines Wertes an die Variable `"DataflowExample.CONSTANT"`, wobei die doppelte Definition eine Folge der zweistufigen Initialisierung ist (siehe Kapitel 5.1.2 ab Seite 135).

Die Auswertung eines Prädikats ist jeweils durch eine Ereignissequenz im Protokoll erkennbar, wie zum Beispiel Nr. 19–22 oder Nr. 35–38. Eine solche Sequenz beginnt mit einem Startereignis (Nr. 19), welches den Beginn der Prädikatsauswertung einleitet, und einem Ergebnisereignis (Nr. 22), welches im Datenfeld (*D*) den Wahrheitswert des Ausdrucks enthält. Ähnlich verhält es sich mit der Ausführung eines `switch/case`-Konstrukts (Nr. 158–161), wobei das Ereignis zum Betreten des angesprungenen `case`-Zweiges (Nr. 161) unmittelbar auf das Ereignis zum Ende der Auswertung des `switch`-Arguments (Nr. 160) folgt.

Aufgrund der in Kapitel 5.1.2 (ab Seite 141) geschilderten Problematik der Instrumentierung von Konstruktoren mit `"this(`⟨*Parameterliste*⟩`)"`- oder `"super(`⟨*Parameterliste*⟩`)"`-Aufrufen, kann das Ausführungsprotokoll nicht unmittelbar verwendet werden. Vielmehr erfordert diese Besonderheit eine Umsortierung der Ereignisse aus der Protokolldatei, um den tatsächlichen Kontroll- und damit auch Datenfluss korrekt rekonstruieren zu können – eine Aufgabe, welche ebenfalls von der entsprechenden *Remote Execution Engine* wahrgenommen wird.

Ein Beispiel für dieses Vorgehen ist die Instantiierung der Klasse `DataflowExample` in Zeile 37 aus Listing A.1 beziehungsweise Zeile 76 aus Listing A.2. Der bei Ausführung dieser Anweisung erzeugte Protokollblock erstreckt sich in Tabelle A.3 von Nr. 8 bis Nr. 94. Zunächst wird aufgrund des Aufrufs `"this(int)"` das vorläufige Betreten des Konstruktors vermerkt (Nr. 9). Das tatsächliche Betreten des Konstruktors wird mittels der Protokollmethode `"enter"` jedoch erst bei Nr. 42 vermerkt. Der gesamte dazwischen liegende Protokollblock (Nr. 10–41) muss daher soweit verschoben werden, dass die Ereigniskette ab Nr. 10 auf Nr. 42 folgt. Diese Betrachtung setzt sich rekursiv fort und gilt analog auch für den Eintrag Nr. 16. Die Instanzinitialisierung in Nr. 11–15 ist eigentlich eine Folge des Ereignisses Nr. 16 und muss daher entsprechend ebenfalls nach hinten verschoben werden. Aufgrund der Korrektur der Konstruktoraufrufsequenzen entfällt Nr. 9, an dessen Stelle Nr. 42 tritt. Die darauf folgenden weiteren Ereignisse sind dann: Nr. 16, Nr. 10–15, Nr. 17–41, Nr. 43ff.

Zusätzlich zur Umsortierung der Ereignisse aufgrund der Konstruktordelegation ist ein zweiter Nachbearbeitungsschritt notwendig, der für die Rekonstruktion des Datenflusses essentiell ist. Wird eine Methode betreten, so erfahren ihre formalen Parameter jeweils eine Definition, was einem *def* einer lokalen Variable entspricht (siehe Kapitel 5.1.2, Seite 144). Da sich diese Ereignisse nicht direkt instrumentieren lassen, werden die entsprechenden Zuweisungen indirekt im Instrumentierungsprotokoll vermerkt, wie in Tabelle A.1 für die Ereigniskennungen 36 und 67. Bei der Bearbeitung des Ausführungsprotokolls müssen diese impliziten Definitionen in explizite Ereignisse umgewandelt werden. Erreicht die dynamische Analyse einen Methodenaufruf, so ergänzt sie unmittelbar im Anschluss an dieses Ereignis jeweils einen Eintrag für jeden formalen Parameter.

Nach der vorangehend beschriebenen Überarbeitung des Protokolls kann nun die Ermittlung der überdeckten Datenflusspaare erfolgen. Dazu wird das umgeordnete Protokoll sequentiell durchlaufen, was verhältnismäßig rechenaufwändig ist und daher ebenfalls in der *Remote Execution Engine* durchgeführt wird. Trifft die dynamische Analyse auf die Definition einer Variablen, so wird diese Definition als die zurzeit aktive vermerkt. Jede folgende Verwendung der *gleichen* Variablen (bei Feldern zusätzlich durch die Instanzkennung des umfassenden Objektes unterschieden) wird der gerade aktiven Definition zugeordnet. Eine nachfolgende Definition derselben Variable ersetzt demnach die jeweils aktive. Als Beispiel betrachte man die lokale Variable "`anotherValue`" welche in Zeile 18 (Listing A.1) deklariert und definiert wird, was im Protokoll bei Nr. 17 (Tabelle A.3) vermerkt ist. Diese Definition erreicht eine Verwendung in Zeile 20 (Nr. 23), was anhand des Protokolls durch einen bezüglich dieser Variablen definitionsfreien „Teilpfad“ entlang Nr. 18 bis Nr. 22 erkennbar ist. Da die Anweisung in Zeile 20 zugleich eine Definition der gleichen Variable zur Folge hat (Nr. 25), wird diese neue *def* aktiviert. Aufgrund der Schleife erreicht diese Definition nun eine Verwendung in der gleichen Zeile (Nr. 31). Da der Schleifenrumpf mehrfach iteriert wird, entstehen „weitere“ Datenflusspaare, zum Beispiel Nr. 33/39 oder aufgrund einer späteren erneuten Ausführung dieser Methode in Nr. 61/67 sowie 69/75, welche jedoch ohne Berücksichtigung des Kontrollflusses alle äquivalent zum *def/use*-Paar aus Nr. 25/31 sind. Schließlich erreicht die letzte Definition in Zeile 20 nach Verlassen der Schleife eine Verwendung in Zeile 22 (Nr. 39).

Da eine JAVA™-Applikation nie allumfassend instrumentiert wird, kann es vorkommen, dass die dynamische Analyse im Ausführungsprotokoll einen lesenden Zugriff auf eine Variable vorfindet, zu der keine Definition als aktiv vermerkt ist. Dies ist zum Beispiel bei einer Programmausgabe mittels der Anweisung "`System.out.println()`" der Fall. Ursache dafür ist, dass hier das statische Feld "`System.out`" bei der Initialisierung der Klasse "`System`" durch die Virtuelle Maschine und damit außerhalb des Zugriffs durch die Protokollierung definiert wird. In diesem Fall wird eine künstliche Definition eingefügt, um weitere Verwendungen dieses Feldes vermerken zu können. Etwas abmildern lässt sich dieses Informationsdefizit, falls eine nichtinstrumentierten Klasse im Kontext des instrumentierten Programmteils instantiiert wird. Dabei umschließt der Instrumentierer dieses Ereignis mit je einem Aufruf der Methoden "`newCall`" beziehungsweise "`newCallCompleted`" (siehe Kapitel 5.1.2, Seite 141). War das Laden der Klasse erfolgreich, so gibt es zwischen den beiden Protokollereignissen auch einen speziellen Eintrag namens *NewInstance* (zum Beispiel Nr. 10 in Tabelle A.3), der die Instantiierung der entsprechenden Klasse zusammen mit der ihr zugewiesenen Instanzkennung darstellt. Aufgrund dieser Information kann die dynamische Analyse eine „pauschale“ künstliche Definition anlegen, die als Anlaufstelle für alle folgenden Verwendungen von beliebigen nicht-statischen Feldern des Objektes mit der gleichen Kennung gilt. Dieser Betrachtung liegt die Annahme zugrunde, dass die Initialisierung der Instanzvariablen bei der Instantiierung der Klasse zugleich die letzte (bekannte) Definition dieser Felder darstellt.

Ob die Verwendung einer Variablen prädikativ oder berechnend ist, kann auf Basis des Ausführungsprotokolls wie folgt ermittelt werden. Trifft die dynamische Analyse bei der sequentiellen Betrachtung der Protokolleinträge auf den Beginn einer Prädikatsauswertung, so wird dies beispielsweise durch Erhöhen eines Zählers oder durch Setzen der Kennung auf einen Stack vermerkt. Entsprechend wird ein solcher Zähler wieder herabgesetzt beziehungsweise der letzte

Eintrag vom Stack genommen, sobald das entsprechende Ende der gleichen Prädikatsauswertung im Protokoll erreicht wird. Dieses Vorgehen ist notwendig, da die Auswertung von Bedingungen aufgrund von Methodenaufrufen in den Prädikaten auch verschachtelt erfolgen kann. Ist die Verwendung einer Variablen zu verarbeiten und mindestens eine Prädikatsauswertung aktiv (der Prädikatezähler größer Null oder Prädikatestack nicht leer), so kann dieser *use* als *p-use* betrachtet werden, ansonsten entsprechend als *c-use*. Ein Beispiel hierfür bietet die Variable "`index`" in Zeile 19 (Listing A.1). Die Definition der Variablen in Zeile 19 erreicht zwei Verwendungen in Zeile 19 sowie eine in Zeile 20. Die letzte Verwendung in Zeile 19 (im Ausdruck "`index++`") sowie diejenige in Zeile 20 sind *c-uses*, da sie außerhalb jeder Prädikatsauswertung ausgeführt werden. Anders die erste Verwendung in Zeile 19 (Ausdruck "`index <= aValue`"): Hierbei handelt es sich um einen *p-use*, was laut Tabelle A.3 anhand der Ereignisse Nr. 19 und Nr. 22 und der dazwischen liegenden Verwendung mit Nr. 20 festgestellt werden kann. Entsprechendes gilt auch für `switch/case`-Verzweigungen, wie in Zeile 26. Dabei erfährt die Variable "`selection`" eine prädikative Verwendung (Nr. 159 in Tabelle A.3), wie aufgrund der Schachtelung durch Nr. 158 beziehungsweise Nr. 160 erkennbar.

Besondere Betrachtungen der Datenflussrekonstruktion

Die Komplexität der Datenflussrekonstruktion auf Basis der dynamischen Analyse, nach einer Instrumentierung des Quellcodes ohne statische Analyse des Datenflusses, verdeutlicht eine detailliertere Betrachtung des Beispiels der Variablen "`anotherValue`" im Konstruktor "`public DataflowExample(int aValue)`" (Listing A.1). Die Definition dieser Variablen in Zeile 18 wird in Zeile 20 überschrieben. Dass es sich um die gleiche Variable handelt, kann anhand des voll-qualifizierten Namens, bestehend aus Package-, Klassen- und eindeutigem Methodenbezeichner (hier "`int DataflowExample.DataflowExample(int).anotherValue`"), festgestellt werden. Aufgrund der gleichen Information kann die Verwendung der Variablen in Zeile 22 entweder der Definition in Zeile 18 oder derjeningen in Zeile 20 zugeordnet werden, je nachdem ob der Schleifenrumpf betreten wird oder nicht. Nun wird der betrachtete Konstruktor jedoch mehrfach aufgerufen (Zeile 8, Zeile 13 und Zeile 39), dennoch ist eine erneute Überdeckung der gleichen *def/use*-Paare nicht erneut zu zählen. Erreicht der Kontrollfluss bei einer wiederholten Ausführung des Konstruktors die erste Definition erneut, so muss die dynamische Analyse erkennen, dass diese bereits in einem früheren Durchlauf vermerkt wurde. Dazu genügt jedoch nicht mehr die Verwaltung dieser Definition anhand ihres voll-qualifizierten Namens alleine, da es in dieser Methode zwei Definitionen der gleichen Variable an verschiedenen Stellen gibt. Vielmehr erfordert dies die Verwendung eines zweiten Schlüssels zur Identifikation der Definition, wozu sich eine Kombination aus dem voll-qualifizierten Namen und der eindeutigen Ereigniskennung aus dem Instrumentierungsprotokoll anbietet. Gleiches gilt natürlich auch für die entsprechenden Verwendungen, wobei hier zusätzlich noch eine Diskriminierung hinsichtlich der eventuell gerade aktiven Prädikatsauswertungen und deren Ergebnis notwendig ist, um die beiden *p-uses* entsprechend dem Wahrheitswert des Prädikats unterscheiden zu können.

Während die Behandlung statischer Felder und lokaler Variablen durch die dynamische Analyse gleichartig nach vorangehender Darstellung erfolgt, bedarf die Rekonstruktion des Datenflusses aufgrund nicht-statischer Felder eines zweistufigen Prozesses mit einer erweiterten Ver-

waltung. Damit die Verwendung einer Instanzvariablen eindeutig der Definition des *identischen* und nicht nur namensgleichen Feldes zugeordnet werden kann, müssen die gerade aktiven Definitionen und die betrachtete Verwendung aufgrund eines zusammengesetzten Schlüssels verglichen werden, der sowohl den voll-qualifizierten Namen der Instanzvariablen als auch die Instanzkennung des zugehörigen Objektes berücksichtigt. Um die Überdeckung des gleichen *def/use*-Paares (gleiche Ereigniskennung und gleicher Ort der zugehörigen Anweisungen im Code) in verschiedenen Instanzen der gleichen Klasse nicht mehrfach zu werten, müssen die entsprechenden *def/use*-Paare, im Anschluss an die vollständige Untersuchung des Ausführungsprotokolls, *ohne* Berücksichtigung der Instanzkennung wieder zusammengeführt werden.

Bei Zugriffen auf einzelne Komponenten eines Arrays muss zusätzlich und in beiden Fällen der Index der Komponente im Array als Unterscheidungsmerkmal hinzugezogen werden.

In Kapitel 5.1.2 wurde ab Seite 139 die Auswirkung der Kapselung von Feldern mittels *getter*- und *setter*-Methoden auf den Datenfluss beschrieben. Wird der Zugriff auf Klassen- und Instanzvariablen weitgehend über diese Funktionen abgewickelt und der Kontext des Aufrufs dieser Methoden nicht berücksichtigt, so entsteht ein häufig überdeckter aber lokal beschränkter „Datenfluss“ von der Definition in der *setter*-Methode zur Verwendung in der *getter*-Funktion. Um den Datenfluss etwas diversifizierter zu betrachten, wurde in •gEAr eine optionale Erweiterung des Konzeptes der Datenflusspaarung entwickelt und umgesetzt, welche sich an dem Gedanken der Kontext-Sensitivität (siehe Kapitel 3.4.6) orientiert.

Bei der sequentiellen Bearbeitung des Ausführungsprotokolls durch die dynamische Analyse wird der Kontrollflussbaum schrittweise mitgeführt. Trifft die Analyse auf ein Methodenaufruf-Ereignis (*CallPoint*, siehe Kapitel 5.1.2 ab Seite 139), welches von einem Protokolleintrag zum Betreten der Methode (*enter*) gefolgt wird, so wird die Kennung des ersten Ereignisses auf einem Kontrollflussstapel abgelegt. Wird eine instrumentierte Methode von einer nicht-instrumentierten Methode aus aufgerufen, so fehlt ein entsprechender *CallPoint*-Eintrag im Protokoll, weshalb dann ersatzweise die Kennung des *enter*-Ereignisses verwendet wird. Der Methodenaufrufbaum wird im letzteren Fall zwar weiterhin im Rahmen der möglichen Genauigkeit[16] vollständig rekonstruiert, der kontext-sensitive Datenfluss hingegen möglicherweise nicht mehr ganz korrekt: Beispielsweise bei einer Sequenz von Aufrufen der gleichen Methode würde jeder Aufruf eine eigene *CallPoint*-Kennung erhalten, ohne diese ist jedoch nur das wiederholte Betreten der gleichen Methode und damit stets der gleiche Kontrollfluss-Stack erkennbar. Ebenfalls auf dem Kontrollfluss-Stack werden die Ereignisse zum Betreten einer Klassen- oder Instanzinitialisierung abgelegt. Entsprechend wird bei Verlassen einer Methode oder Initialisierungssequenz der zugehörige Eintrag vom Kontrollflussstapel getilgt. Auf diese Weise stellt ein Querschnitt durch den Kontrollfluss-Stack stets den aktuellen Aufrufkontext der jeweils ausgeführten Methode dar. Dieser Aufrufkontext wird nun zusätzlich in den Verwaltungsschlüssel für Definitionen und Verwendungen aufgenommen, das heißt, falls ein *def* d_j (oder *use* u_n) der gleichen Variablen in einem anderen Aufrufkontext auftritt als d_i (respektive u_m), so handelt es dabei um eine andere Definition (beziehungsweise Verwendung).

Obige Gedankengänge lassen sich auch auf die Auswertung von Prädikaten übertragen. Da die prädikativen Verwendungen von Variablen bei der klassischen Datenflusskriterienfamilie von

[16] Je nach Vollständigkeit der Instrumentierung

Rapps/Weyuker den Kanten des Kontrollflusses zugeordnet werden (siehe Definition 3.15), entstehen für die gleiche Variable im Kontext eines Prädikates, je nach Wahrheitsgehalt der Bedingung, zwei verschiedene Verwendungen. Kommt es jedoch beispielsweise bei der Auswertung der Bedingung b_i einer `IF`-Anweisung zu einem Aufruf einer Methode `m`, die ihrerseits ebenfalls eine `IF`-Verzweigung mit einer Bedingung b_k enthält[17], so findet die Verwendung einer Variablen v in b_k im Kontext zweier geschachtelter Prädikate statt. Weil bei der Auswertung von b_i und b_k jeweils alle möglichen Wahrheitswerte und deren Kombinationen möglich sind, muss die prädikative Verwendung von v auch mit dem kombinierten Wahrheitsgehalt beider Prädikate identifiziert und somit diskriminiert werden, wodurch nun bis zu *vier* unterschiedliche *p-uses* der identischen Variable v an der gleichen Stelle im Programmcode existieren können. Tatsächlich kann sich die Zahl der *p-uses* sogar noch weiter erhöhen, wenn die aufgerufene Methode `m` aus einem anderen Kontext heraus ohne umschließendes Prädikat aufgerufen wird.

Um diese unterschiedlichen prädikativen Kontexte berücksichtigen zu können, wurden in •gEAr beide Betrachtungsweisen verfolgt, so dass dem Tester die freie Auswahl des Detailgrads der Überdeckung überlassen wird. Sofern sich während der Rekonstruktion des Datenflusses durch die dynamische Analyse mindestens ein Prädikat in Auswertung befindet, so wird in der einfachen Variante lediglich der Wahrheitsgehalt der innersten auszuwertenden Bedingung zusammen mit der Kennung des Prädikats (aus dem Startereignis der Prädikatsauswertung) als Unterscheidungsmerkmal der *p-uses* verwendet. Hingegen wird beim sogenannten *deep branch tracing*, ähnlich wie beim vorangehend beschriebenen Kontrollfluss-Stack, ein Prädikate-Stapel mitgeführt. Sobald eine neue Prädikatsauswertung beginnt, wird die Kennung des Startereignisses, zusammen mit dem Wahrheitsgehalt der Auswertung, auf diesem Stapel abgelegt und entsprechend bei Erreichen des Endereignisses wieder entfernt. Falls es sich bei dem aktuellen „Prädikat“ um den Operanden eines `switch`-Ausdrucks handelt, wird anstelle des Wahrheitsgehaltes die Kennung der Äquivalenzklasse des tatsächlich angesprungenen `case`-Falls verwendet. Der jeweils aktuelle „Querschnitt“ durch den Prädikate-Stack gibt für eine Variable den jeweiligen „Zweig“ an, dem der entsprechende *p-use* angehört. Falls es im Laufe der Programmausführung zu keiner geschachtelten Prädikatsauswertung kommt, so verhalten sich die beiden Varianten selbstverständlich identisch.

Zusammenführung der Datenflussüberdeckungsinformation

Das Besondere an dem hier vorgestellten Verfahren ist, dass die sogenannte *Globale Optimierung* auf Basis multi-objektiver Metaheuristiken nach der Identifikation optimaler Testdatensätze strebt. Diese Aufgabe wird dabei als „Suche“ nach einer Testfallmenge interpretiert, die eine möglichst hohe Überdeckung mit minimaler Anzahl Testfälle erreicht. Die objektive Bewertung der von einem „Individuum“ erzielten Überdeckung geschieht somit nicht auf der Granularitätsebene eines einzelnen Testfalls, sondern auf der einer gesamten Testfallmenge, bestehend aus mehreren einzelnen Testfällen. Da aber jeder Testfall getrennt und unabhängig von den anderen ausgeführt werden muss, liegen die Ausführungsprotokolle und damit die (datenflussbezogenen) Überdeckungsinformationen zunächst auch nur für jeden Testfall einzeln vor. Daher wurde die

[17] Was sich beliebig fortsetzen lässt, da hier nun wiederum ein Methodenaufruf stehen kann.

Rekonstruktion des Datenflusses so optimiert, dass das Verfahren zur Superposition einzelner Überdeckungen zu einer Gesamtüberdeckung möglich wird.

Betrachtet man nur den Konstruktor "`public DataflowExample(int aValue)`" aus dem Code-Beispiel in Listing A.1 ab Zeile 17 sowie das Kriterium *all-uses* bezüglich der lokalen Variablen "`anotherValue`", so müssen zur Erfüllung des Kriteriums mindestens zwei Testfälle erstellt werden:

- Einer davon durchläuft den Rumpf der `FOR`-Schleife mindestens einmal, wodurch er das Paar bestehend aus der Definition in Zeile 18 und der Verwendung in Zeile 20 überdeckt, infolge dessen jedoch nicht in der Lage ist, die von der Definition in Zeile 18 erreichbare Verwendung in Zeile 22 zu überdecken.

- Ein Weiterer umgeht die Schleife und erreicht zwar nicht die Verwendung in Zeile 20, dafür jedoch diejenige in Zeile 22.

Eine Testfallmenge, bestehend aus obigen beiden Testfällen, erreicht hingegen eine vollständige Überdeckung bezüglich dieser Variablen. Dazu muss die Überdeckungsinformation entsprechend korrekt zusammengeführt werden. Die vorangehend beschriebene Rekonstruktion des Datenflusses nach einer Testausführung (dynamische Analyse) ergibt eine Liste unterschiedlicher Definitionen. Jede Datenstruktur, die je einer Definition assoziiert ist, umfasst ihrerseits eine Liste unterschiedlicher Verwendungen, welche jeweils von dieser Definition erreicht werden. Um die Zusammenführung solcher Listen effizient zu gestalten, bietet sich an ihrer Stelle der Einsatz von Hash-Tabellen an. Dabei wird jede Definitionsdatenstruktur mit einem eindeutigen globalen Schlüssel abgelegt, der so beschaffen ist, dass er unabhängig von einem bestimmten Testfall ist. Gleiches gilt auch für jede Datenstruktur die eine Verwendung repräsentiert.

Im Falle der Definition statischer Felder ist dieser Schlüssel eine Zeichenkette bestehend aus dem voll-qualifizierten Namen der Variablen[18], gefolgt von einem trennenden Sonderzeichen, einem Kennzeichen, dass es sich um eine statische Variable handelt, erneut einem Trennzeichen, dem optionalen Aufrufkontext, einem weiteren Trennzeichen und schließlich der Ereigniskennung laut Instrumentierungsprotokoll. Letzteres unterscheidet die verschiedenen Definitionen der gleichen Variablen im identischen Aufrufkontext (sofern dieser überhaupt berücksichtigt wird). Der Hash-Schlüssel für die Verwendung einer statischen Variablen setzt sich aus dem vollqualifizierten Namen der Variablen, dem trennenden Sonderzeichen, einem Kennzeichen, dass es sich um eine statische Variable handelt, erneut einem Trennzeichen, dem Querschnitt der aktiven Prädikatsauswertungen und deren Wahrheitswert, einem weiteren Trennzeichen, dem optionalen Aufrufkontext, dem letzten Trennzeichen und der Ereigniskennung laut Instrumentierungsprotokoll zusammen. Ein kombiniertes Ereignis, welches eine Verwendung und eine anschließende Definition symbolisiert (*useDef*, siehe Kapitel 5.1.2), wird in diese zwei entsprechenden Ereignisse getrennt, welche jedoch die Ereigniskennung gemeinsam haben.

Völlig gleichartig sind die entsprechenden Hash-Schlüssel für die Verwaltung der Datenflussereignisse, die lokalen Variablen betreffend, aufgebaut. Lediglich das Kennzeichen weist diese Datenstruktur nicht als Verwendung oder Definition einer statischen Variable sondern einer lokalen Variable aus.

[18] Kann prinzipiell weggelassen werden, da die Ereigniskennung bereits eindeutig ist.

Während der Rekonstruktion des Datenflusses für einen Testfall werden in einem ersten Durchlauf die Definitionen und Verwendungen nicht-statischer Felder zunächst noch zusätzlich anhand der Instanz-Kennung des Eigentümerobjektes diskriminiert. Ein zweiter Durchlauf führt diese verschiedenen Definitionen und Verwendungen wieder zusammen, ohne dabei die entsprechende Instanzkennung zu berücksichtigen. In der endgültigen Zusammenfassung der *def/use*-Paare von Instanzvariablen werden somit wieder ähnlich aufgebaute Hash-Schlüssel verwendet wie bei lokalen Variablen und statischen Feldern. Der Verwaltungsschlüssel für eine Definition beginnt mit dem voll-qualifizierten Namen der Variablen, gefolgt von dem trennenden Sonderzeichen, einem Kennzeichen, welches die Variable als nicht-statisches Feld ausweist, erneut einem Trennzeichen, dem optionalen Aufrufkontext, einem weiteren Trennzeichen und schließlich der Ereigniskennung laut Instrumentierungsprotokoll. Analog setzt sich der Hash-Schlüssel einer Verwendung aus dem voll-qualifizierten Namen der Variablen, dem trennenden Sonderzeichen, einem Kennzeichen, dass es sich um eine Instanzvariable handelt, erneut einem Trennzeichen, dem Querschnitt der aktiven Prädikatsauswertungen, einem weiteren Trennzeichen, dem optionalen Aufrufkontext, dem letzten Trennzeichen und der Ereigniskennung laut Instrumentierungsprotokoll zusammen.

Um einen weiteren Eintrag reicher werden die entsprechenden Hash-Schlüssel bei Definitionen und Verwendungen von Array-Komponenten, da hier zusätzlich der Index der Komponente im Array in den Schlüssel codiert wird. Somit beginnt der Schlüssel einer Array-Definitionsdatenstruktur mit dem voll-qualifizierten Namen des Arrays und setzt sich mit einem trennenden Sonderzeichen, dem Index der definierten Komponente, erneut einem Trennzeichen, einem Kennzeichen, dass es sich um einen Array-Zugriff handelt, einem weiteren Trennzeichen, dem optionalen Aufrufkontext, einem letzten Trennzeichen und schließlich der Ereigniskennung laut Instrumentierungsprotokoll fort. Entsprechend besteht auch der Hash-Schlüssel der Verwendung einer Array-Komponente aus dem voll-qualifizierten Namen des Arrays, einem trennenden Sonderzeichen, dem Index der verwendeten Komponente, erneut einem Trennzeichen, einem Kennzeichen, dass es sich um einen Array-Zugriff handelt, einem weiteren Trennzeichen, dem Querschnitt der aktiven Prädikatsauswertungen, einem zusätzlichen Trennzeichen, dem optionalen Aufrufkontext, einem letzten Trennzeichen und schließlich der Ereigniskennung laut Instrumentierungsprotokoll. Da jeder lesende und schreibende Zugriff auf eine Komponente eines Arrays automatisch auch eine Verwendung der Array-Variablen einschließt, unabhängig davon, ob es eine lokale Variable oder ein Feld ist, werden die zusätzlichen *def/uses* einzelner Komponenten eines Array nur optional als solche getrennt gezählt. Auch optional kann die Verwendung des speziellen Array-Feldes `"length"` als eigenständiger *use* betrachtet werden, deren zugehörige Definition mit dem Ereignis *newArray* (siehe Kapitel 5.1.2, Seite 141) zusammenfällt.

Wurden nun verschiedene Testfälle unabhängig voneinander ausgeführt und liegen deren *def/use*-Paare in der vorangehend beschriebenen Form vor, so ist die Superposition dieser Datenflussüberdeckungen trivial. Dabei ist jedoch zu beachten, dass die Hash-Listen nicht einfach ineinander kopiert werden dürfen, da sonst die Überdeckungsinformation eines Testfalls ungültig wird. Letzteres äußert sich insbesondere dann, wenn ein Testfall Bestandteil mehrerer Testfallmengen ist. Unter diesen Umständen wird der Testfall nur einmal ausgeführt und die somit erhaltenen Überdeckungsdaten werden so lange wiederverwendet, bis der Testfall geändert wurde und daher neu ausgeführt werden muss. Bei der Ermittlung der Datenflussüberdeckung einer gesam-

ten Testfallmenge T wird zunächst eine leere Definitionen-Hash-Liste D_{hl}^{T} instantiiert, welcher sequentiell die Definitionen (zusammen mit deren erreichten Verwendungen) aus den Listen $D_{hl}^{t_i}$ jedes einzelnen Testfalls t_i hinzugefügt werden. Hat ein Testfall t_m ein *def/use*-Paar überdeckt, deren Definition mit einem Schlüssel $s_d^{t_m}$ verwaltet wird und enthält D_{hl}^{T} noch keine Definition mit diesem Schlüssel, so wird D_{hl}^{T} eine Kopie der Definitionsdatenstruktur unter dem Schlüssel $s_d^{t_m}$ hinzugefügt. Enthält jedoch ein Testfall t_k eine Definition d, welche mit dem gleichen Schlüssel $s_d^{t_k}$ in $D_{hl}^{t_k}$ abgelegt wurde, wie eine bereits von einem anderen Testfall t_j aus deren $D_{hl}^{t_j}$ übernommene und zu D_{hl}^{T} hinzugefügte Datenstruktur, dann werden die von d erreichten Verwendungen, der bereits in D_{hl}^{T} enthaltenen Definition mit dem gleichen Schlüssel $s_d^{t_k}$, hinzugefügt. Auch dabei werden Duplikate erkannt und eliminiert. Da die von einer Definition erreichten Verwendungen ebenfalls in einer Hash-Liste in der Datenstruktur der Definition verwaltet werden, lassen sich äquivalente *def/use*-Paarungen zweier äquivalenter Definitionen aus verschiedenen Testfällen anhand der entsprechenden Hash-Schlüssel jeder Verwendung erkennen.

Anmerkung zur Verzweigungsüberdeckung: Nach dem gleichen Verfahren werden auch die von einem Testfall überdeckten Verzweigungen verwaltet beziehungsweise zur quantitativen Verzweigungsüberdeckung einer Testfallmenge zusammengeführt. Dazu liefert die in Kapitel 5.1.2 beschriebene Instrumentierung, dabei insbesondere die ab Seite 137 dargestellte Behandlung von Prädikaten, genügend Daten, um auch eine dynamische Analyse der Verzweigungsüberdeckung ohne zusätzlichen Aufwand aufzusetzen. Zu diesem Zweck wird, ähnlich wie bei den Definitionen, eine Hash-Tabelle aller von einem Testfall tatsächlich überdeckten „Kanten“ aus dem Ausführungsprotokoll rekonstruiert. Die jeweiligen Schlüssel bestehen bei binären Prädikaten aus der Ereigniskennung der Probe `"predResult"` und dem Wahrheitswert der jeweiligen Auswertung, wobei beide durch ein Sonderzeichen getrennt werden. Im Falle von `switch/case`-Konstrukten tritt die Kennung der `"switchPredResult"`-Probe anstelle derjenigen von `"predResult"` sowie die Äquivalenzklasse des tatsächlich angesprungenen `case`-Zweiges (aus der entsprechenden `"switchPredEquivalent"`-Probe) anstelle des Wahrheitswertes von vorhin.

5.1.5 Testfall- und Testdatenrepräsentation

Kapitel 5.1.4 beschreibt zwar die Ausführung eines Testfalls und die sich anschließende dynamische Analyse der dadurch erreichten Überdeckung, lässt jedoch einen, für die praktische Umsetzung einer automatischen Testdatengenerierung und -optimierung mittels multi-objektiver Metaheuristiken, zentralen Aspekt noch völlig offen: Die Definition eines Testfalls im Sinne der Testdaten. Angesichts der möglichen Vielfalt an Darstellungsoptionen erfordert dieses Themengebiet eine eigenständige Betrachtung, insbesondere vor dem Hintergrund der Forderungen aus Kapitel 4.5.1 nach einer möglichst optimierten Codierung der Individuen zum Zwecke einer effizienten Verarbeitung durch die eingesetzte Metaheuristik.

In Kapitel 3.1 wurde bereits mit Definition 3.2 eine sehr allgemeine Beschreibung des Konzeptes „Testfall“ vorgestellt. Nun lässt sich die *Menge der erwarteten Ergebnisse* sowie die *erwarteten Nachbedingungen* nicht ohne weiteres automatisch ermitteln, da dieser Prozess kognitiv

und von der Erwartungshaltung des zukünftigen Nutzers abhängt. Zwar kann ein Testfall nachträglich, zum Beispiel aufgrund der Spezifikation, mit dieser Information erweitert werden, für die automatische Generierung von Testdaten zum Zwecke einer (strukturellen) Überdeckung ist sie nicht erforderlich. Darüber hinaus muss ein Verfahren, wie es in dieser Arbeit beschrieben ist, möglichst unabhängig von einer bestimmten Applikation sein, weshalb die Herstellung eines initialen Zustands, der die *notwendigen Vorbedingungen* erfüllt, nicht Bestandteil der Testdatengenerierung sein kann.

Demnach verbleiben noch die beiden Aspekte *Menge der Eingabewerte* sowie *Prüfanweisung* – was zu Beginn von Kapitel 3.1 als *Testdaten* und *Testszenario* beschrieben wurde. Während sich die *Testdaten* nach dieser Definition intuitiv erfassen und als Individuum einer Metaheuristik darstellen lassen, erfordert das *Testszenario* hier eine besondere Behandlung. Zu bedenken ist dabei stets, dass für die Verwendung einer Heuristik eine wiederholte und vollautomatische Ausführung des zu testenden Systems essentiell ist.

Je nachdem, auf welcher Granularitätsebene beziehungsweise für welche Art von Software Unterstützung hinsichtlich der automatischen Testgenerierung geboten werden soll, unterscheiden sich die Darstellungsformen für Szenarien gravierend. Gilt es, eine vollständige Applikation mit graphischer Benutzeroberfläche (*GUI: Graphical User Interface*) zu testen, so besteht ein Szenario aus einer Abfolge von Ereignissen, die ein Benutzer durch die GUI an der Software auszulösen vermag. Dazu gehören Mausbewegungen und die Bedienung der Maustasten ebenso wie Tastaturanschläge[19] – sowie insbesondere deren Reihenfolge und zeitlicher Zusammenhang. Die größte Schwierigkeit bereitet dabei die automatische Ansteuerung der GUI. Scheinbar einfacher ist der Test einer Applikation, welche lediglich über eine textuelle Konsole mit dem Benutzer interagiert – doch auch dabei ist die Kenntnis der Aufrufparameter des Programm und des Protokolls zur „Konversation“ zwischen simuliertem Benutzer (automatischem Tester) und Applikation notwendig. Ist hingegen lediglich eine Softwarekomponente zu testen, wie zum Beispiel die Funktionalität eines Packages oder gar einer einzelnen Klasse, so beschreibt ein entsprechendes Szenario, welche Methoden in welcher Reihenfolge aufzurufen sind. Noch gar nicht berücksichtigt sind dabei die „Hardware-nahen“ Applikationen, wie zum Beispiel bei eingebetteten Systemen, die im Rahmen eines sogenannten *„Hardware-in-the-Loop (HIL)“*-Tests ausgeführt werden.

Grundsätzlich ist das in dieser Arbeit vorgestellte Verfahren unabhängig von der Art des *System under test*. Um diese Behauptung zu stützen, wird in Kapitel 5.3, zusätzlich zu dem im Folgenden dargestellten Ansatz, eine Methode vorgeschlagen, wie automatisch für •gEAr geeignete Testtreiber generiert werden können, mit deren Hilfe sich beliebige Softwarekomponenten ansprechen und somit gezielt zugehörige Testdaten hinsichtlich struktureller Testkriterien generieren und optimieren lassen.

Da sich komplexe Datenstrukturen nur schwer für die Codierung als Individuen einer Metaheuristik eignen und außerdem eine möglichst problemunabhängige Darstellung des Konzeptes „Testfall“ anzustreben ist, wurde für die prototypische Umsetzung des Verfahrens im Werkzeug •gEAr eine wohldefinierte Schnittstelle zu einer zu testenden Software gewählt: Die Parameterliste beim Aufruf einer Applikation. Auf diese Weise können prinzipiell alle Arten von Software-

[19] *Welche* Taste dabei zu drücken ist, gehört nach dieser Betrachtung eher zu den „Testdaten“.

systemen angesprochen werden. Trivialerweise gilt dies für einfache Konsolenapplikationen, welche eine Reihe von Parametern entgegennehmen und das Ergebnis nach deren Verarbeitung ausgeben – dies ist das meistgenutzte Konzept typischer Unix/Linux-Werkzeuge. Hierbei gibt es kein „Testszenario“ im klassischen Sinn, sondern lediglich Testdaten, die das Verhalten des *SUT* indirekt steuern. Soll hingegen eine einzelne Komponente getestet werden, wie typischerweise während des Unit- oder Integrationstests, so kann ein Teil der von •gEAr bereitgestellten Testdaten eines Testfalls als „Steuerdaten“ interpretiert werden, wonach sich zum Beispiel die Reihenfolge der Methodenaufrufe oder Instantiierungsaktionen richtet – siehe dazu Kapitel 5.3. Mit dem gleichen Ansatz lässt sich über eine zwischengeschaltete Hilfsapplikation, welche die Test- und Steuerdaten entgegennimmt und geeignet weiterleitet, auch eine GUI-orientierte Software oder eines der vielen, am Markt verfügbaren GUI-Testwerkzeuge ansprechen.

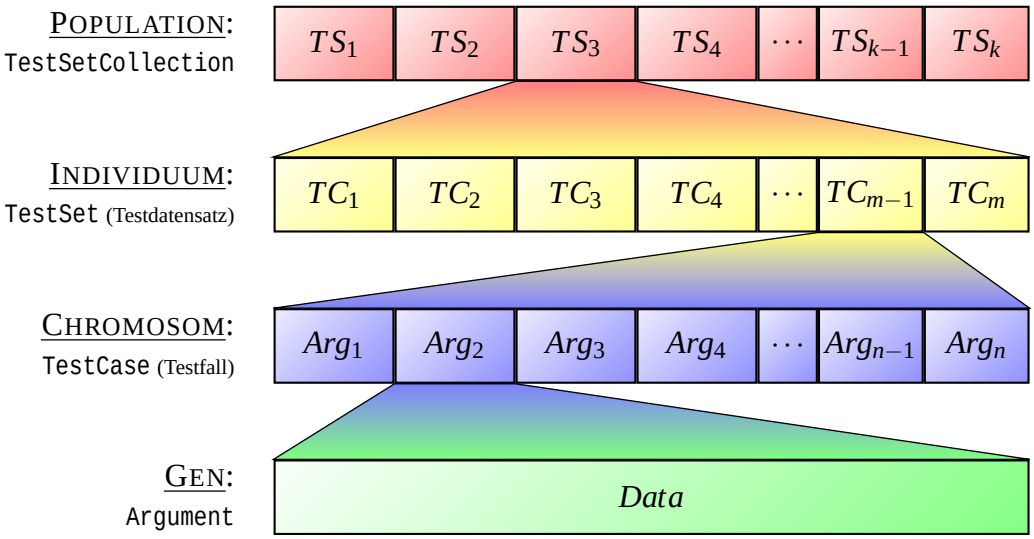


Abbildung 5.4: Datenstruktur bei der globalen Optimierung von Testdaten

Die in •gEAr daher verwendete Datenarchitektur ist in Abbildung 5.4 dargestellt. Demnach ist ein Testfall eine Abfolge primitiver Daten (*Data* in der Abbildung), den sogenannten *Argumenten* (Arg_i) der zu testenden Applikation. In der Terminologie Genetischer Algorithmen, wie sie im Rahmen der globalen Optimierung eingesetzt werden, handelt es sich bei den Argumenten um die *Gene* eines *Chromosoms*, letzteres stellt dabei einen Testfall (*TestCase*, *TC*), dar.

Jede Anwendung hat eine individuelle Signatur, das heißt, jedes Programm erwartet eine bestimmte Anzahl von Parametern eines bestimmten Datentyps. Aus diesem Grund wurde eine einfache Methode zur Schnittstellenbeschreibung entwickelt, die es dem Tester erlaubt, eine solche Signatur für die automatische Testdatengenerierung selbständig zusammen zu stellen. Diese ist dergestalt, dass auch entsprechend geeignete Testtreiber bequem angesprochen und somit Testfälle für beliebige Softwarekomponenten generiert und optimiert werden können. Dazu spezifiziert der Tester eine minimale und eine maximale Anzahl Parameter, im Rahmen derer sich die später automatisch erfolgende Ermittlung der entsprechenden Daten bewegen darf, weshalb auch

Testszenarien beliebiger Länge möglich sind (siehe Kapitel 5.3). Zusätzlich muss er mindestens für das erste Argument einen Datentyp und einen Wertebereich angeben. Zwar kann der Tester beliebig viele der ersten Argumente auf diese Weise spezifizieren, ist die maximale Anzahl der erlaubten Parameter jedoch größer als die Anzahl der angegebenen Argumentbeschreibungen, so wird die letzte Beschreibung implizit entsprechend oft wiederholt, was ebenfalls im Hinblick auf die automatische Testtreibergenerierung aus Kapitel 5.3 von Bedeutung ist. Weil •gEAr prototypisch für JAVA™ umgesetzt wurde, bietet es die primitiven Datentypen *long*, *double* sowie die speziellen Typen *String* und *Enumeration*. Bei geeigneter Einschränkung des Wertebereichs können somit alle in JAVA™ darstellbaren ganzen Zahlen mittels *long* sowie alle möglichen Gleitkommazahlen durch *double* ausgedrückt werden. Zum Beispiel lässt sich ein Parameter vom Typ *byte* durch einen *long*-Parameter mit Wertebereich $[-128, 127]$ darstellen. Wenngleich *String* kein primitiver JAVA™-Datentyp ist, bietet es sich dennoch an, auch Argumente in Form von Zeichenketten zu generieren. Deshalb kann ein Tester auch den Datentyp String verwenden, muss dabei jedoch die minimale und maximale Länge der möglichen Zeichenketten einschränken. Optional kann die Auswahl der zu verwendenden Zeichen ebenfalls beschränkt werden. Für spezielle Argumente ist es manchmal nützlich, einen von mehreren fest vorgegebene Werten automatisch auswählen zu lassen. Dies gilt zum Beispiel für komplexe Zeichenketten mit einem festen Aufbau, wie bei Internet-Adressen oder Ortsnamen, für die eine zufällige Generierung aufgrund einer freieren *String*-Deklaration sehr unwahrscheinlich ist. Dafür bietet sich der Datentyp *Enumeration* an, bei dem der Tester zusätzlich die Menge der möglichen Werte des entsprechend typisierten Parameters spezifizieren muss.

Auf diese Weise aufgebaute Argumentlisten der Form $TC_h := \left[Arg_i^h,\ 1 \leq i \leq n\right]$ stellen somit jeweils einen eindeutigen Testfall TC_h dar. Eine Testausführung beginnt daher beim Aufrufen der Applikation mit dieser Parameterliste, was im Falle von JAVA™ dem Aufruf der initialen Methode "`public static void main(String[] args)`" gleichkommt. Ziel der *Globalen Optimierung* ist jedoch die Generierung einer optimalen Testdatenmenge, also einer minimalen Anzahl unterschiedlicher Testfälle, die zusammen eine maximale Überdeckung erreichen. Aus diesem Grunde ist eine „Lösung“ des gestellten Problems nicht ein einzelner Testfall, sondern eine Testfallmenge oder ein *Testdatensatz* $TS_g := \{TC_h^g,\ 1 \leq h \leq m\}$. In Abbildung 5.4 als *TestSet* beziehungsweise kurz TS_g bezeichnet, entspricht diese Datenstruktur im Sinne Evolutionärer Verfahren einem *Individuum*.

Die Metaheuristiken aus der Familie der Genetischen Algorithmen wenden ihre Operatoren auf eine ganze Population mehrerer Individuen an, was in der Datenstruktur aus Abbildung 5.4 durch die oberste Ebene (*TestSetCollection*) repräsentiert wird. Die „Population“ stellt im Rahmen der Globalen Optimierung demnach eine Menge von Testfallmengen, hier $\{TS_g,\ 1 \leq g \leq k\}$, dar.

Man beachte, dass die in Abbildung 5.4 gezeigten Indizes k, m und n lediglich auf eine Oberschranke hinweisen. Tatsächlich ist das Verfahren so flexibel, dass es sich, wie bereits angedeutet, beliebig anpassen lässt, unter anderem durch Angabe einer minimalen und maximalen Anzahl gewünschter Testfälle in einem Testdatensatz.

5.1.6 Spezialisierung der Metaheuristiken

Auf Basis der in Kapitel 5.1.5 beziehungsweise Abbildung 5.4 dargestellten Datenstruktur kann nun die Spezialisierung der in Kapitel 4 zunächst allgemein beschriebenen Metaheuristiken zum Zwecke der *Globalen Optimierung* von Testdatensätzen erfolgen. Im Rahmen des Prototypen •gEAr wurden vier dieser Heuristiken umgesetzt, um sie miteinander vergleichen zu können; das Verfahren schließt jedoch nicht den Einsatz weiterer Metaheuristiken aus. Es handelt sich dabei um folgende:

- Random: *Random Search* (siehe Kapitel 4.2 ab Seite 85),
- SimAnn: *Simulated Annealing* (siehe Kapitel 4.4 ab Seite 87),
- MOA: *Multi-objektive Aggregation* (siehe Kapitel 4.5.4 ab Seite 112) sowie
- NSGA: *Nondominated Sorting Genetic Algorithm* (siehe Kapitel 4.5.4 ab Seite 114).

Bei den eingesetzten Varianten der ersten drei Algorithmen (Random, SimAnn und MOA) handelt es sich um pseudo-multi-objektive Verfahren. Das heißt, sie berücksichtigen nicht alle Bewertungsaspekte eines Individuums unabhängig voneinander, hier zum Beispiel erzielte Überdeckung und Größe einer Testfallmenge, sondern bestimmen vielmehr anhand einer gewichteten Summe eine einzige Bewertung eines jeden Testfalls, um mehrere potentielle Lösungen des gestellten Problems zu vergleichen. Aus diesem Grund ist auch die Angabe der Gewichte durch den Tester notwendig und zugleich essentiell. NSGA ermittelt im Gegenzug mit der sogenannten *Pareto-Front* eine Vielzahl unterschiedlicher Testfallmengen, von denen jede einzelne optimal im Sinne aller objektiven Bewertungen ist. Für eine datenflussorientierte Generierung optimaler Testdaten stellen alle Testfallmengen entlang der Pareto-Front aus Abbildung 6.3 den bestmöglichen Kompromiss zwischen Anzahl der Testfälle und damit erreichbarer struktureller Überdeckung dar. Mit Festlegung der Gewichte bei den ersten drei Optimierungsverfahren schränkt der Tester sich bei Random, SimAnn und MOA auf die Ermittlung einer einzigen Testfallmenge entlang dieser Front ein. Welche davon tatsächlich erzielt wird, hängt ausschließlich vom Verhältnis der Gewichte untereinander ab: Überwiegt das Gewicht der Überdeckung, so legt der Tester den Schwerpunkt auf eine möglichst hohe Überdeckung, auf Kosten einer potentiell größeren Testfallmenge.

Verfahrensbedingt umfasst die „Population“ bei den ersten beiden Heuristiken jederzeit lediglich ein einziges Individuum. Die Größe der Population bei MOA und NSGA kann der Tester bei Bedarf anpassen, •gEAr nutzt standardmäßig jedoch Werte entsprechend den Empfehlungen aus Kapitel 4.5.1.

Anmerkung Zugunsten einer einfacheren Darstellung wird im Folgenden zunächst nur von der „Anzahl überdeckter Entitäten“ hinsichtlich eines einzigen Überdeckungskriteriums ausgegangen. Das Verfahren kann jedoch leicht um die Betrachtung mehrerer Objektiven erweitert werden, so dass es automatisch Testfälle gleichzeitig hinsichtlich mehrerer, teils auch orthogonaler Kriterien generieren und optimieren kann.

Random Search

Die auf *Random Search* basierende „Optimisation Engine“ (Abbildung 5.2) beginnt zunächst mit der Generierung einer zufälligen initialen Lösung (siehe Abbildung 4.1). Dazu wird im Rahmen der vom Tester vorgegebenen Grenzen eine Testfallmenge mit einer zufällig gewählten Anzahl Testfälle erstellt, wobei die Anzahl der Argumente jedes einzelnen Testfalls ebenfalls zufällig gewählt wird, genauso wie deren Daten aus dem entsprechend spezifizierten Wertebereich. Dabei liegt der zufälligen Auswahl, wie bei allen umgesetzten Algorithmen, eine uniforme Verteilung der entsprechenden Zufallszahlen zugrunde. Die somit entstandenen Testfälle werden jeweils, wie in Kapitel 5.1.4 (ab Seite 152) beschrieben, dem „Local Execution Manager“ übergeben, welcher sie zur Ausführung an einen verfügbaren „Compute-Server“ weiterreicht. Dort wird die Applikation mit dem jeweiligen Testfall ausgeführt und mittels der dynamischen Analyse werden die davon tatsächlich überdeckten Entitäten bestimmt. Um welche Entitäten es sich jeweils handelt, hängt von der Auswahl des Überdeckungskriteriums durch den Tester ab, zum Beispiel die einzelnen Verzweigungen bei der Verzweigungsüberdeckung oder die *def/use*-Paare für das *all-uses*-Kriterium. Wurden auf diese Weise alle Testfälle ausgeführt, so werden die jeweils von den einzelnen Testfällen überdeckten Entitäten entsprechend Kapitel 5.1.4 (ab Seite 160) zusammengeführt.

Nachdem nun bekannt ist, wie viele Testfälle der aktuelle Testdatensatz umfasst und wie viele Entitäten einer bestimmten Art er überdecken konnte, kann prinzipiell eine Gesamtbewertung der Testfallmenge vorgenommen werden. Da eine Testfallmenge umso „besser“ ist, je mehr Entitäten sie überdeckt, aber je weniger Testfälle sie umfasst, muss die Größe der Menge indirekt proportional in die gewichtete Summe eingehen. Würde man ihren Kehrwert verwenden, so ginge die Größe exponentiell in die Berechnung der Güte ein, was den „Selektionsdruck“ auf unerwünscht Weise negativ beeinflusst. Weil das Verfahren bewusst ohne eine statische Analyse des Programms entwickelt wurde, kann a priori nicht bestimmt werden, wie viele Entitäten überhaupt überdeckt werden können[20] und wie viele Testfälle höchstens dazu notwendig sind. Aus diesem Grund wird die Berechnung der „Testmengengüte“ zunächst noch zurückgestellt.

Im nächsten Schritt wird, analog zur Beschreibung vorhin, eine zweite Testfallmenge zufällig erzeugt. Auch ihre Testfälle werden ausgeführt und der dynamischen Analyse unterzogen. Ebenfalls wird die von der ganzen Menge jeweils erreichte Überdeckung ermittelt.

Nun gilt es, die beiden Testfallmengen hinsichtlich ihrer „Qualität“ zu vergleichen und die „schlechtere“ der beiden zu verwerfen. Dazu muss berücksichtigt werden, dass die beiden Komponenten *Größe* und *Anzahl überdeckter Entitäten* der Testmengengüte im Allgemeinen stark unterschiedliche Wertebereiche aufweisen. Daher ist die Verwendung des sogenannten *Windowing* zusammen mit einer Normierung auf den Wertebereich $[0,1]$ hier ratsam (siehe Kapitel 4.5.1 ab Seite 94) – ein Verfahren, das die Beiträge aller Objektiven einzeln und von allen zu vergleichenden Testfallmengen erfordert und geeignet umrechnet. Nachdem die jeweiligen Beiträge ermittelt und angepasst wurden, wird jeder Testfallmenge eine „Fitness“ durch gewichtete Summierung der Teilbeiträge zugewiesen:

Seien s_i und s_j die Anzahlen der Testfälle der beiden Testfallmengen T_i beziehungsweise T_j

[20] Eine Information, die auch eine statische Analyse, unter anderem aufgrund des Problems der *Feasibility*, höchstens durch eine obere Schranke abschätzen, jedoch im Allgemeinen nicht genau bestimmen kann.

sowie c_i und c_j die Anzahl der von T_i respektive T_j überdeckten Entitäten. Darüber hinaus seien w_c und w_s die vom Tester vorgegebenen Gewichte für die Überdeckung beziehungsweise die Testfallgröße. Zunächst werden s_i und s_j je nach Konfiguration dem Windowing unterzogen und anschließend normiert:

$$\hat{s_i} := \begin{cases} \frac{s_i - \min(s_i,s_j)}{\max(s_i,s_j) - \min(s_i,s_j)}, & \text{falls Windowing erwünscht und } \max(s_i,s_j) \neq \min(s_i,s_j); \\ \frac{s_i}{\max(s_i,s_j)}, & \text{falls kein Windowing erwünscht und } \max(s_i,s_j) \neq 0; \\ 0, & \text{sonst.} \end{cases}$$

$$\hat{s_j} := \begin{cases} \frac{s_j - \min(s_i,s_j)}{\max(s_i,s_j) - \min(s_i,s_j)}, & \text{falls Windowing erwünscht und } \max(s_i,s_j) \neq \min(s_i,s_j); \\ \frac{s_j}{\max(s_i,s_j)}, & \text{falls kein Windowing erwünscht und } \max(s_i,s_j) \neq 0; \\ 0, & \text{sonst.} \end{cases}$$

Gleiches gilt auch für die Anzahlen c_i und c_j der überdeckten Entitäten:

$$\hat{c_i} := \begin{cases} \frac{c_i - \min(c_i,c_j)}{\max(c_i,c_j) - \min(c_i,c_j)}, & \text{falls Windowing erwünscht und } \max(c_i,c_j) \neq \min(c_i,c_j); \\ \frac{c_i}{\max(c_i,c_j)}, & \text{falls kein Windowing erwünscht und } \max(c_i,c_j) \neq 0; \\ 1, & \text{sonst.} \end{cases}$$

$$\hat{c_j} := \begin{cases} \frac{c_j - \min(c_i,c_j)}{\max(c_i,c_j) - \min(c_i,c_j)}, & \text{falls Windowing erwünscht und } \max(c_i,c_j) \neq \min(c_i,c_j); \\ \frac{c_j}{\max(c_i,c_j)}, & \text{falls kein Windowing erwünscht und } \max(c_i,c_j) \neq 0; \\ 1, & \text{sonst.} \end{cases}$$

Um die Größe der Testfallmenge gegenläufig in die Fitness eingehen zu lassen, wird noch folgende Umrechnung durchgeführt:

$$\tilde{s_i} := 1 - \hat{s_i},\ \tilde{s_j} := 1 - \hat{s_j}$$

Somit ergeben sich für die relative Güte der Testfallmengen zum Zwecke des Vergleiches folgende Fitness-Werte:

$$f_i := w_c \cdot \hat{c_i} + w_s \cdot \tilde{s_i},\ f_j := w_c \cdot \hat{c_j} + w_s \cdot \tilde{s_j}\ \textit{mit}\ f_i, f_j \in [0, w_c + w_s]$$

Die Testfallmenge mit der größeren Fitness wird beibehalten, während die andere verworfen wird. Falls das Abbruchkriterium für das Verfahren noch nicht erfüllt ist (siehe Kapitel 5.1.8), wird nun ein weiterer Testdatensatz zufällig generiert, ausgeführt sowie dynamisch analysiert und der gesamte Algorithmus *Random Search* wiederholt sich ab der Berechnung der Vergleichsfitness.

Windowing und Normierung erwecken den Anschein, als würden die Teilbewertungen aller Testfallmengen in jede einzelne Fitnessberechnung eingehen. Wichtig und hier auch erfüllt ist, dass die relative Ordnung der Testdatensätze induziert durch die Gesamtfitness stets die gleiche ist, unabhängig davon, ob Windowing oder Normierung eingesetzt werden (Kapitel 4.5.1).

Simulated Annealing

Ähnlich wie Random Search verläuft die Testdatengenerierung und -optimierung auch bei Simulated Annealing. Der Algorithmus unterscheidet sich lediglich in zwei wesentlichen Aspekten:

1. Jede weitere Testfallmenge (außer natürlich der ersten) wird nicht zufällig erstellt, sondern jeweils aufgrund der bisher als vorläufiger Lösung identifizierten Menge konstruiert.

2. Ist die zuletzt generierte Testdatenmenge schlechter als eine zuvor ermittelte, so wird sie nicht unbedingt verworfen.

Wie bei Random Search auch, wird zunächst eine zufällige initiale Testfallmenge generiert, deren Testfälle ausgeführt sowie die Überdeckung letzterer rekonstruiert und anschließend die von der gesamten Testfallmenge überdeckten Entitäten zusammengeführt werden. Der zweite und jeder weitere Testdatensatz wird nun aus der „Umgebung“ der aktuellen Testfallmenge gewählt (siehe Abbildung 4.2). Dazu wird im Rahmen von •gEAr die „Umgebung“ einer Testfallmenge operationell durch die Mutationsoperatoren der ebenfalls eingesetzten Evolutionären Verfahren definiert. Das bedeutet demnach, dass der jeweils neue nächste Testdatensatz ein „Mutant“ des aktuell als Lösung identifizierten ist.

Da die Generierung und Optimierung auf der Granularitätsebene einer gesamten Testfallmenge ansetzt, bieten sich auch für die Mutationsoperatoren mehrere Angriffsflächen an. Demzufolge gibt es folgende Mutationsoperatoren:

- Hinzufügen eines zufällig generierten Testfalls, sofern die vom Tester vorgegebene maximale Anzahl der Testfälle pro Testdatensatz noch nicht erreicht ist.

- Entfernen eines zufällig ausgewählten Testfalls, sofern die vom Tester vorgegebene minimale Anzahl der Testfälle noch nicht erreicht ist.

- Mutieren eines zufällig ausgewählten Testfalls durch:

  – Hinzufügen eines zufällig gewählten Argumentes aus dem „Verfügungsbereich“, sofern die vom Tester vorgegebene maximale Anzahl der Argumente pro Testfall noch nicht erreicht ist.

  – Entfernen eines zufällig gewählten Argumentes aus dem „Verfügungsbereich“, sofern die vom Tester vorgegebene minimale Anzahl der Argumente noch nicht erreicht ist.

  – Mutieren eines zufällig ausgewählten Argumentes.

Der sogenannte „Verfügungsbereich“ einer Argumentliste umfasst diejenigen Argumente, die implizit durch Wiederholung der letzten Argumentspezifikation entstehen. Auf diese Weise wird sichergestellt, dass die angegebenen Datentypen der ersten, spezifizierten Argumente nicht durch die Mutation verletzt werden. Eine Verletzung könnte auftreten, wenn ein zufälliges Argument vor dem Verfügungsbereich gelöscht oder hinzugefügt wird und dadurch die rechts der Mutationsposition stehenden Argumente entsprechend um eine Stelle versetzt werden müssten – wodurch die versetzten Argumente nicht mehr notwendigerweise der für ihre neue Stelle spezifizierten Typisierung gehorchen würden.

Die Mutation eines Argumentes erfolgt je nach Datentyp auf unterschiedliche Weise. Für numerische Argumente (*double* und *long*) wird anstelle einer normalverteilten Zufallsgröße (wie in Kapitel 4.5.1 ab Seite 102 erläutert) eine dreiecksverteilte[21] Zufallsvariable mit Erwartungswert $\mu = 0$ verwendet, welche zum jeweils aktuellen Wert hinzuaddiert wird; es sei denn, der für dieses Argument erlaubte Wertebereich würde dabei verlassen werden, wodurch dann der entsprechend erreichte Grenzwert eingesetzt wird. Der maximale positive beziehungsweise minimale negative Betrag dieser Zufallsvariablen ergibt sich aus dem Produkt der maximalen Spannweite des für dieses Argument spezifizierten Wertebereichs und einer vom Tester anpassbaren „Varianz“ aus dem Bereich $(0,1]$. Bei Argumenten vom Typ *String* wird eine neue Zeichenkette entsprechend der Spezifikation des Arguments zufällig ausgewählt. Entsprechend wird auch bei Argumenten vom Typ *Enumeration* der aktuelle Wert durch einen der vorgegebenen Werte zufällig und aufgrund einer uniformen Verteilung ersetzt.

Nachdem durch Mutation des aktuellen Testdatensatzes ein neuer Vergleichskandidat ermittelt wurde, wird dieser anschließend ebenfalls durch den „Local Execution Manager“ nach und nach ausgeführt und somit die vom neuen Testdatensatz erzielte Überdeckung festgestellt. Da nun wie bei Random Search ebenfalls zwei Testfälle vorliegen, deren Fitnesswerte f_i und f_j miteinander zu vergleichen sind, wird die jeweilige Fitness auch bei Simulated Annealing auf die gleiche Weise errechnet. Ist die neue Testfallmenge entsprechend der Fitness besser als die bisher als Lösung festgehaltene, so wird die alte verworfen und von der neuen ersetzt. Ist die Güte des neuen Testdatensatzes jedoch kleiner, so wird dieser mit der Wahrscheinlichkeit

$$P\left(f_i,\ f_j,\ T\right) = \begin{cases} e^{-\frac{|f_i - f_j|}{T}}, & falls\ T > 0; \\ 0, & sonst. \end{cases}$$

trotzdem angenommen (siehe Kapitel 4.4). Für die sogenannte „Temperatur“ T kann der Tester bei •gEAr einen beliebigen initialen Wert vorsehen. Nach jeder weiteren ausgewerteten Testfallmenge sinkt T entsprechend einer geometrischen Reihe der Form $T \leftarrow T \cdot (1-\delta)$, wobei der Tester den Faktor $\delta \in (0,1)$ ebenfalls frei wählen kann.

Multi-objektive Aggregation

Anders als Random Search und Simulated Annealing gehört *Multi-objektive Aggregation* zu den Algorithmen aus der Familie Evolutionärer Verfahren. Daher enthält die „Population“ nun nicht mehr nur jeweils einen Testdatensatz, sondern je nach Vorgabe des Testers gleichzeitig mehrere (typischerweise um die 40) Testfallmengen.

Entsprechend Abbildung 4.3 beginnt dieser Algorithmus ebenfalls mit der Generierung zufälliger Testfälle beziehungsweise Testfallmengen. Nach Ausführung aller Testdatensätze und Ermittlung der jeweils überdeckten Entitäten auf der Granularitätsebene eines einzelnen Testfalls beziehungsweise jedes Testdatensatzes erfolgt die Bewertung jedes Individuums. Anders als bei

[21] Im Allgemeinen kann auch die Normalverteilung zugrunde gelegt werden (siehe Kapitel 4.5.1, Seite 102). Hier findet die Mutation jedoch innerhalb fester Grenzen statt, weshalb der Wertebereich einer (aus mathematischer Sicht) normalverteilten Zufallsvariablen stark eingeschränkt werden müsste und diese damit ohnehin nicht mehr „normalverteilt“ wäre.

Random und SimAnn werden bei MOA nicht nur zwei Testfallmengen miteinander verglichen, um den besseren Testdatensatz zu ermitteln. Vielmehr erfordert hier der genetische Operator *Selektion* ein relatives Vergleichsmaß, um die Wahrscheinlichkeit für die Auswahl eines Individuums proportional zu seiner relativen Fitness innerhalb der Population berechnen zu können. Daher erfolgt die Ermittlung der Fitness eines Individuums in der Population entsprechend nach folgendem Verfahren:

Sei $TSC := \{TS_1, TS_2, \ldots, TS_k\}$ die Menge aller Testfallmengen TS_i der aktuellen Population (*TestSetCollection*) TSC. Darüber hinaus sei s_i $(1 \leq i \leq k)$ die Anzahl der Testfälle im Testdatensatz TS_i sowie c_i $(1 \leq i \leq k)$ die Anzahl der von TS_i überdeckten Entitäten. Diese absoluten Werte werden zunächst einem (optionalen) Windowing unterzogen und anschließend auf das Intervall $[0,\ 1]$ normiert:

$$\hat{s_i} := \begin{cases} \frac{s_i - \min(s_1,s_2,\ldots,s_k)}{\max(s_1,s_2,\ldots,s_k) - \min(s_1,s_2,\ldots,s_k)}, & \text{falls Windowing erwünscht und } \max(s_1,s_2,\ldots,s_k) \neq \min(s_1,s_2,\ldots,s_k); \\ \frac{s_i}{\max(s_1,s_2,\ldots,s_k)}, & \text{falls kein Windowing erwünscht und } \max(s_1,s_2,\ldots,s_k) \neq 0; \\ 0, & \text{sonst.} \end{cases}$$

$$\hat{c_i} := \begin{cases} \frac{c_i - \min(c_1,c_2,\ldots,c_k)}{\max(c_1,c_2,\ldots,c_k) - \min(c_1,c_2,\ldots,c_k)}, & \text{falls Windowing erwünscht und } \max(c_1,c_2,\ldots,c_k) \neq \min(c_1,c_2,\ldots,c_k); \\ \hat{c_i} := \frac{c_i}{\max(c_1,c_2,\ldots,c_k)}, & \text{falls kein Windowing erwünscht und } \max(c_1,c_2,\ldots,c_k) \neq 0; \\ 1, & \text{sonst.} \end{cases}$$

Anschließend erfolgt die vorangehend beschriebene „Inversion“ der Größe eines jeden Testfalls als Teilbeitrag zu dessen Fitness:

$$\tilde{s_i} := 1 - \hat{s_i}$$

Wie bei Random und SimAnn berechnet sich die gesamte relative Fitness eines einzelnen Testdatensatzes auch bei MOA als gewichtete Summe der entsprechend umgerechneten Teilbeiträge:

$$f_i := w_c \cdot \hat{c_i} + w_s \cdot \tilde{s_i},\ mit\ f_i \in [0, w_c + w_s]$$

Da die in •gEAr prototypisch implementierte Variante des MOA generationenbasiert ist, wird zunächst eine leere neue Population angelegt. Im nächsten Schritt wird ein, ebenfalls je nach Vorgabe des Benutzers, mehr oder weniger großer Anteil der aktuellen Population mit den jeweils besten Testdatensätzen unverändert in die neue Generation übernommen (*Elitismus*), um eine monoton wachsende Qualität des besten Testdatensatzes garantieren zu können. Die zum Vervollständigen der nächsten Generation noch fehlenden Individuen werden nun mittels der genetischen Operatoren Selektion, Rekombination und Mutation ermittelt. Grundlage für die Selektion ist die vorhin berechnete Fitness f_i jedes Testdatensatzes TS_i der aktuellen Generation. Für vergleichende Bewertungen bietet •gEAr die drei wichtigsten Auswahlverfahren „Roulette Wheel“, „Rank“ und „Tournament“ an, wobei entweder der Benutzer eine feste Wahl trifft oder diese der automatischen Rekonfiguration des Werkzeuges überlässt.

Nachdem die Selektion zwei Eltern-Testdatensätze ausgewählt hat, werden diese dem Crossover zugeführt. Dazu umfasst •gEAr die drei bekannten Mechanismen „1-point-“, „2-point-“ und „uniform crossover“ (Kapitel 4.5.1 ab Seite 100) sowie eine experimentelle Variante namens

„uniform chaotic crossover“. Da die Anzahl der Testfälle in einem Testdatensatz innerhalb vorgegebener Grenzen variabel ist und darüber hinaus identische[22] Testfälle in einer Testfallmenge vermieden werden sollten, wird der „1-point-crossover“ durch einen algorithmischen Trick gelöst: Nachdem die Größe des zukünftigen Kind-Testdatensatzes zufällig gewählt wurde, werden die Testfälle der „Mutter“ in aufsteigender und die des „Vaters“ in absteigender Reihenfolge durchlaufen. Dabei wird jeweils abwechselnd von Mutter und Vater der nächste, im Kind-Chromosom noch nicht enthaltene Testfall des jeweiligen Elternteils übernommen. Dadurch ergibt sich je ein „natürlicher“ Kreuzungspunkt für beide Elternchromosomen. Ähnlich arbeitet auch der „2-point-crossover“-Operator, wobei das Mutterchromosom abwechselnd von beiden „Enden“ her durchlaufen wird, das Chromosom des Vaters hingegen von einem zufällig gewählten Punkt aus abwechselnd in beide Richtungen zum Ende hin. Vergleichsweise einfach ist der „uniform-crossover“-Operator: Dabei werden die Testfälle der Elternchromosomen abwechselnd übernommen, es sei denn, der entsprechende Testfall ist bereits im Kindchromosom enthalten, wobei eine „Umschaltung“ zwischen Mutter und Vater durch die Crossover-Wahrscheinlichkeit eingeleitet wird. Die Variante „uniform chaotic crossover“ des Kreuzungsoperators durchmischt zunächst zufällig alle paarweise nicht identischen Testfälle der beiden Elternteile zu einer allumfassenden Testfallmenge und eliminiert anschließend ebenfalls zufällig so viele Testfälle, bis die Größe des Kindchromosoms unter die zufällig, aber innerhalb der vorgegebenen Grenzen gewählte Anzahl an Testfällen fällt.

Nachdem das neue Kindchromosom durch Rekombination erstellt wurde, wird es mit einer vom Benutzer vorgegebenen Wahrscheinlichkeit einer Mutation unterzogen. Dieser Operator ist identisch mit dem vorangehend für Simulated Annealing beschriebenen.

Sobald die neue Generation durch wiederholte Selektion, Rekombination und Mutation vervollständigt wurde, ersetzt diese die alte Population vollkommen. Der gesamte Algorithmus wiederholt sich nun ab der Fitness-Bewertung der aktuellen Population solange, bis das Abbruchkriterium (siehe Kapitel 5.1.8) erfüllt ist. Da ein Testfall durch den Crossover-Operator nicht modifiziert wird, sondern lediglich durch die Mutation, kann ein erheblicher Anteil der Ausführungszeit und damit der Rechenressourcen eingespart werden, indem lediglich die neuen oder geänderten Testfälle dem „Local Execution Manager“ zur dynamischen Analyse übergeben werden. Im Allgemeinen nicht umgangen werden kann hingegen die Zusammenführung der Überdeckungsinformationen aller Testfälle eines jeden einzelnen Testdatensatzes.

Nondominated Sorting Genetic Algorithm (NSGA)

Auf den gleichen Varianten der genetischen Operatoren Selektion, Rekombination und Mutation wie MOA baut auch die in •gEAr implementierte Version des *Nondominated Sorting Genetic Algorithm (NSGA)*. Lediglich die Bestimmung der Fitness und die Anwendung des „Elitismus“ unterscheidet die beiden Evolutionären Verfahren voneinander.

Wie bereits in Kapitel 4.5.4 ab Seite 114 beschrieben, erfolgt die Bestimmung der Fitness jedes einzelnen Individuums (Testdatensatzes) nach dem sogenannten *Pareto-Ranking* aus Ab-

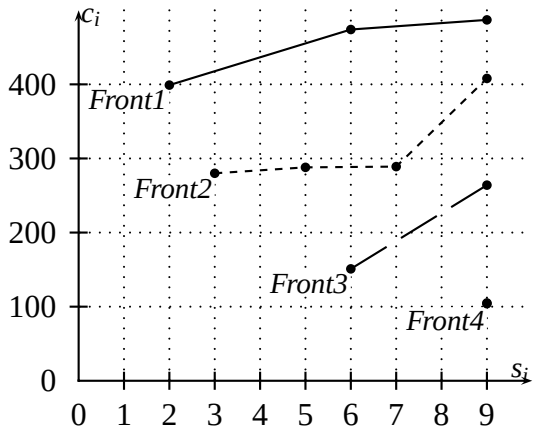
[22] „Identisch“ hier im Sinne der Objektreferenz. Zwar könnten auch (semantisch) gleiche Testfälle nachträglich durch einen Korrekturoperator entfernt werden, jedoch kann der erhebliche Zusatzaufwand bei Evolutionären Algorithmen sogar kontraproduktiv sein.

bildung 4.8. Nachdem die Anzahlen c_i der von jeder Testfallmenge $TS_i \in \{TS_1, TS_2, \ldots, TS_k\}$ überdeckten Entitäten ermittelt wurde und die Größe s_i jedes Testdatensatzes TS_i bekannt ist, kann zwar zunächst wie bei MOA eine Normalisierung der Werte nach einem optionalen Windowing erfolgen, dies ist jedoch für NSGA nicht erforderlich.

Das *Pareto-Ranking* identifiziert iterativ jeweils die Menge aller Testdatensätze, die innerhalb der noch verbliebenen Population nicht dominiert werden (Pareto-Front), extrahiert sie aus der Population und weist allen ihren Testfallmengen die gleiche, entsprechend ihres Rankings proportional fallende Fitness zu. Eine Testfallmenge T_i dominiert eine andere Testfallmenge T_j (geschrieben $T_i \succ T_j$), wenn sie hinsichtlich mindestens eines der Aspekte (hier: Größe und Überdeckung) höherwertiger und den übrigen Aspekten nach zumindest gleichwertig ist, also $T_i \succ T_j \Leftrightarrow \left[s_i \leq s_j \wedge c_i \geq c_j \wedge \left(s_i < s_j \vee c_i > c_j\right)\right]$.

i	s_i	c_i	*Front1*	*Front2*	*Front3*	*Front4*
1	9	487	⊗	–	–	–
2	7	289		⊗	–	–
3	9	104				⊗
4	9	408		⊗	–	–
5	3	280		⊗	–	–
6	6	151			⊗	–
7	9	264			⊗	–
8	5	288		⊗	–	–
9	2	399	⊗	–	–	–
10	6	474	⊗	–	–	–
Fitness			4	3	2	1

(a) Zeitlicher Ablauf



(b) Graphische Darstellung

Abbildung 5.5: Exemplarische Darstellung des *Pareto-Rankings*

Abbildung 5.5 zeigt exemplarisch, wie die einzelnen Pareto-Fronten einer Population von Testdatensätzen iterativ extrahiert werden. Dabei stelle jede Zeile eine Testfallmenge mit entsprechender Anzahl Testfälle s_i (zweite Spalte) und Anzahl überdeckter Entitäten c_i (dritte Spalte) dar. Die mit *„Front 1“* überschriebene vierte Spalte kennzeichnet alle im ersten Durchlauf identifizierten Testdatensätze, die in der gesamten Population nicht dominiert werden. Demnach wird diesen Testfallmengen 1, 9 und 10 jeweils der gleiche initiale Fitnesswert zugeordnet, welcher üblicherweise dem Ranking der entsprechenden Front entspricht, hier also zunächst 4. Im nächsten Schritt werden die Testfallmengen der ersten Front aus der Population entfernt und innerhalb der verbliebenen Testfallmengen erneut alle nicht-dominierten Testdatensätze identifiziert. Im Beispiel sind das die Testfallmengen 2, 4, 5 und 8, welche zusammen die *„Front 2“* bilden und denen die Fitness 3 zugewiesen wird. Nachdem diese ebenfalls der Population entnommen wurden, entsteht auf die gleiche Art und Weise die nächste „Schale“, nämlich *„Front 3“*, mit den Testdatensätzen 6 und 7, denen die Fitness 2 entsprechend ihrem Ranking zugeordnet wird. Schließlich verbleibt nur noch eine Testfallmenge in der Population, welche die letzte Front bildet und daher die kleinste Fitness zugewiesen bekommt.

Bei dieser Art der Fitnessvergabe macht es wenig Sinn, einen beliebigen Anteil der Population mit den besten Individuen in die nächste Generation retten zu wollen, da die Individuen innerhalb einer Front nicht untereinander vergleichbar sind. Daher wurde für •gEAr ein sogenannter *Pareto-Elitismus* entwickelt. Dieser übernimmt jeweils die gesamte erste Pareto-Schale einer Population unverändert in die nächste Generation. Auf diese Weise bleibt stets die vollständige Front der bisher als optimal identifizierten Lösungen erhalten.

Bis auf die Bestimmung der Fitness jeder einzelnen Testfallmenge und der Anwendung des Elitismus gleicht der sonstige Ablauf des Nondominated Sorting Genetic Algorithm dem klassischer Evolutionärer Verfahren (Abbildung 4.3), insbesondere also auch dem des MOA.

5.1.7 Erweiterung zur multikriteriellen Optimierung

Die Spezialisierung der Metaheuristiken, wie sie in Kapitel 5.1.6 beschrieben ist, lässt zusammen mit dem Konzept der globalen Optimierung eine besonders interessante Erweiterung zu, die im Rahmen der Software-Verifikation zu einer Erhöhung der Fehleraufdeckungsquote einer Testfallmenge führt, welche im Folgenden unter dem Namen *multikriterielle Testdatengenerierung und -optimierung* motiviert und beschrieben wird.

Betrachtet man beispielsweise die einfache Verzweigung mit einem Boole'schen Prädikat `"if (age > 18)"`, so genügt eine Testfallmenge dem Kriterium der Verzweigungsüberdeckung bezüglich dieser Anweisung, falls die relevante Variable bei der Ausführung der Testfälle zum Beispiel die Werte 13 und 29 annimmt. Nicht wesentlich restriktiver ist das *all-uses*-Kriterium im Hinblick auf die Wahl der Variablenwerte, lediglich alle Paare aus Definitionen der Variablen `age` und der davon erreichbaren Verwendung in dieser Verzweigungsanweisung sind zu überdecken. Dennoch könnte das Prädikat den sehr verbreiteten Programmierfehler enthalten, bei dem bestimmte Grenzfälle nicht bedacht werden, so dass die korrekte Anweisung hier `"if (age >= 18)"` lauten müsste. Demnach bliebe ein solcher Fehler in der Verifikationsphase bei Verwendung eines der klassischen Testkriterien unerkannt.

Zusammenfassend kann man präzisieren, dass die datenflussorientierten Überdeckungskriterien zwar den *Fluss* der Daten mittels Variablen durch ein Programm betrachten, jedoch keinerlei Anforderungen bezüglich der *Daten* selbst erheben. Doch in Anbetracht der typischen Programmierfehler ähnlich dem oben dargestellten, ist es zwingend ratsam, Testfälle nicht nur nach einer bestimmten Teststrategie zu entwickeln, sondern verschiedene Aspekte eines Testobjektes zu berücksichtigen. Solange die vorgegebenen Testkriterien einander subsumieren (siehe Kapitel 3.5), genügt es, das jeweils umfassendste zu verfolgen, um alle anderen ebenfalls zu erfüllen. Sind die Kriterien jedoch orthogonal[23], so wird die Testdatengenerierung ungleich schwieriger. Eine Lösung besteht darin, jeweils eine eigene Testfallmenge im Hinblick auf jedes orthogonale Kriterium für sich zu entwickeln. Die manuelle Erstellung solcher Testfallmengen ist dabei einfach, weil nur jeweils ein Kriterium gleichzeitig zu beachten ist, jedoch entsteht dabei im Allgemeinen eine Vielzahl von teils „redundanten"[24] Testfällen, die die meist manuelle Verifikation der Ergebnisse jedes Testlaufs unnötig aufwändig machen.

[23] Orthogonale Kriterien sind unvergleichbar, das heißt, es besteht keine Subsumptionsrelation zwischen ihnen.

[24] Unterschiedliche Testfälle können zum Beispiel redundant sein, wenn sie den gleichen Pfad durchlaufen.

Aufgrund dieser Betrachtungen bietet es sich an, die ohnehin automatische Generierung und Optimierung von Testdaten entsprechend mit der Aufgabe zu betrauen, bei der Ermittlung von Testfällen mehrere Kriterien gleichzeitig zu verfolgen. Auf diese Weise wird eine hohe Überdeckung hinsichtlich verschiedener Kriterien erzielt, jedoch ohne dabei „redundante“ Testfälle ausführen und prüfen zu müssen. Zu diesem Zweck sind lediglich eine dem Kriterium entsprechende dynamische Analyse umzusetzen (beispielsweise analog Kapitel 5.1.2) sowie die im Kapitel 5.1.6 vorgestellten Bewertungsfunktionen geeignet anzupassen. Letzteres kann unabhängig von der Teststrategie auch generisch erfolgen.

Beim Einsatz von pseudo-multi-objektiven Metaheuristiken wie *Random Search*, *Simulated Annealing* oder *Multi-objektive Aggregation* müssen zusätzlich zur Gewichtung w_s für die Testdatensatzgröße auch für jedes Kriterium $K_1, K_2, \ldots, K_n$ geeignete Gewichtungen $w_1, w_2, \ldots, w_n$ angegeben werden. Sind s_i die Anzahl der Testfälle im Testdatensatz TS_i sowie c_i die Anzahl der von TS_i überdeckten Entitäten und $\hat{c}_i$ beziehungsweise $\tilde{s}_i$ die durch Windowing, Normalisierung und Inversion (nur $\tilde{s}_i$) abgeleiteten Teilbeiträge zur Fitness, so kann ein Testdatensatz TS_i aufgrund folgender Formel bewertet werden:

$$f_i := w_s \cdot \tilde{s}_i + \sum_{j=1}^{n} w_j \cdot \hat{c}_j$$

Im Gegensatz dazu können moderne multi-objektive Metaheuristiken, wie der in dieser Arbeit verwendete *Nondominated Sorting Genetic Algorithm (NSGA)*, bei der Lösung solcher Fragestellungen ihre wahren Stärken ausspielen. Allen voran ist keine Angabe einzelner Gewichtungen für jedes Kriterium schon im Vorfeld der Testdatenermittlung notwendig. Vielmehr kann das Testwerkzeug dem Tester eine ganze Reihe unterschiedlicher Testfallmengen mit jeweils spezifischen Schwerpunkten bieten, idealerweise zusammen mit deren jeweiligen Fehleraufdeckungspotentialen, so dass der Tester den für die betrachtete Anwendung idealen Testdatensatz in Abhängigkeit von den zur Verfügung stehenden Mitteln zur Verifikation/Validierung auswählen kann. Für den Einsatz des *NSGA* ist dabei lediglich das Konzept der Dominanzrelation ($\succ$) entsprechend anzupassen. Sind $K_1, K_2, \ldots, K_n$ Testkriterien, T_i und T_j zwei Testfallmengen aus der aktuellen Population $TSC := \{TS_1, TS_2, \ldots, TS_k\}$, s_i und s_j die Anzahlen der Testfälle der beiden Testfallmengen T_i beziehungsweise T_j sowie c_i^p und c_j^p die Anzahl der von T_i respektive T_j entsprechend Kriterium K_p überdeckten Entitäten, so gilt:

$$T_i \succ T_j \Leftrightarrow \left\{ s_i \leq s_j \wedge \bigwedge_{p=1}^{n} \left(c_i^p \geq c_j^p \right) \wedge \left[s_i < s_j \vee \bigvee_{p=1}^{n} \left(c_i^p > c_j^p \right) \right] \right\}$$

Anknüpfend an den einleitenden, zur multikriteriellen Testdatengenerierung motivierenden Fehler (`"if (age > 18)"` statt `"if (age >= 18)"`) sei an dieser Stelle ein typisches Beispiel einer zu den Datenflusskriterien orthogonalen Teststrategie skizziert, der sogenannte *strukturelle Äquivalenzklassentest*. Der klassische Äquivalenzklassentest und seine Erweiterung, der sogenannte Grenzwerttest, basieren auf einer *black-box*-artigen Sicht auf das zu testende System und gehören daher zu den funktionsorientierten Qualitätsprüfverfahren, wie in Abbildung 2.3 dargestellt. Dabei unterteilt man den Eingaberaum des Testobjekts entsprechend der Spezifikation so

in „Äquivalenzklassen“, dass das zu testende System bei jeder Eingabe aus der gleichen Klasse ein aus Sicht der Spezifikation äquivalentes Verhalten aufweist.

Im Gegensatz dazu wird die Äquivalenzklassenpartitionierung beim strukturellen Äquivalenzklassentest ausschließlich auf Basis des Programmquellcodes beziehungsweise des Kontrollflusses definiert. Jede atomare Bedingung, in der mindestens eine Variable vorkommt, unterteilt dabei den möglichen Wertebereich der entsprechenden Variable im Kontext des Prädikats in mehrere Äquivalenzklassen. Die Verwendung eines falschen Operators hat eine Verschiebung der Partitionierung zu Folge, das heißt, im Vergleich zum korrekten Prädikat sind einzelne Elemente einer Äquivalenzklasse nun in eine andere Klasse übergetreten.

Die Beispielanweisung "`if (age > 18)`" zerlegt den Wertebereich der Variablen `age` vom Typ *int* in zwei Äquivalenzklassen: $A_{false} := \left[-2^{31}; 18\right]$ sowie $A_{true} := \left[19; 2^{31}-1\right]$. Falls stattdessen die Anweisung "`if (age >= 18)`" richtig gewesen wäre, hätten folgende Äquivalenzklassen gegolten: $\bar{A}_{false} := \left[-2^{31}; 17\right]$ sowie $\bar{A}_{true} := \left[18; 2^{31}-1\right]$. Demnach kam es zu einer Verschiebung der Partitionierung, wodurch das Element 18 die Klasse gewechselt hat. Da sich das Programm bezüglich dieser Verzweigung für alle Werte $age \leq 17$ beziehungsweise $age \geq 19$ unabhängig vom Operator jeweils äquivalent verhält, kann nur ein Testfall die Verwendung des falschen Operators entdecken, unter dessen Ausführung die Variable `age` genau den Wert 18 einnimmt.

Anstatt des relationalen Operators `>=` wurde im Beispiel „versehentlich“ der Operator `>` verwendet. Ebenso hätte jedoch auch ein beliebiger anderer der Operatoren `<`, `<=`, `==` oder `!=` fälschlich eingesetzt werden können. In jedem Fall wäre die Ermittlung einer Testfallmenge hilfreich, die die „Grenzfälle“ testet, also bei deren Ausführung die Variable `age` jeweils einen Wert deutlich unterhalb der Grenze 18 (zum Beispiel „-13“), unmittelbar unterhalb der Grenze 18 (hier „17“), unmittelbar oberhalb der Grenze 18 (im Beispiel „19“) sowie deutlich oberhalb der Grenze 18 (hier „29“) einnimmt.

Analog dazu lässt sich für eine Vielzahl von Operatoren sowie ihren ähnlichen und daher potentiell zu verwechselnden Gegenstücken eine vergleichbare Verschiebung der strukturellen Äquivalenzklassenpartitionierung feststellen und damit können entsprechende Grenzfallüberdeckungskriterien definiert werden. In Tabelle 3.6 (Kapitel 3.6) wurde exemplarisch eine Auswahl „typischer Programmierfehler“ vorgestellt, wie sie im Rahmen des Mutationstestens zur Fehlerinjektion und damit zur Bewertung der Güte von Testfällen eingesetzt werden und welche sich hier als Kandidaten für eine genauere Untersuchung anbieten. Ein umfassendes Testkriterium dieser Art ist orthogonal zu den datenflussorientierten Kriterien und ergänzt dieses in idealer Weise. Entsprechend „kombinierte“ Testfälle, die beiden Kriterienfamilien zugleich gerecht werden, berücksichtigen sowohl den Fluss der Daten als auch die Daten selbst.

5.1.8 Terminierung

Alle in Kapitel 5.1.6 beschriebenen Metaheuristiken haben eine Gemeinsamkeit: Sie wiederholen die Kernschritte ihres Algorithmus „Erstellen neuer Testdatensätze“, „Ausführen und Bewerten der neuen/gänderten Testfallmengen“ und „Vergleich der Qualität aktueller Testdatensätze“ prinzipiell ad infinitum - beziehungsweise bis ein sogenanntes *Abbruchkriterium* erfüllt ist (Abbildung 4.1, Abbildung 4.2 und Abbildung 4.3).

Selbstredend wäre das ideale Abbruchkriterium auf Basis des Testziels zu definieren. Dies wäre im Falle der *Globalen Optimierung* mit pseudo-multi-objektiven Verfahren genau dann erfüllt, wenn entsprechend dem vorgegebenen Gewichtungsverhältnis ein optimaler Testdatensatz identifiziert würde. Ein Testdatensatz wäre dann optimal, wenn er die höchstmögliche Überdeckung mit der kleinstmöglichen Testfallmenge erreicht, das heißt, wenn es keinen anderen Testdatensatz gibt, dessen Verhältnis aus entsprechend gewichteter Anzahl überdeckter Entitäten zur Anzahl der Testfälle größer ist. Entsprechendes gilt auch für echt multi-objektive Metaheuristiken (wie MOA): Das Abbruchkriterium wäre genau dann erfüllt, wenn die optimale Pareto-Front ermittelt wurde, das heißt, wenn alle Testfallmengen identifiziert wurden, für die das Verhältnis aus der Anzahl der durch sie überdeckten Entitäten zur Anzahl ihrer Testfälle jeweils maximal ist.

Die Auswertung eines solch gearteten Kriteriums erfordert jedoch a priori Kenntnisse über den Zusammenhang zwischen der Anzahl der Testfälle in einem Testdatensatz und der von den jeweiligen Testfällen des Testdatensatzes maximal überdeckbaren Anzahl struktureller Entitäten, entsprechend dem vorgegebenen Testkriterium. Für die automatische und problemunabhängige Ermittlung dieser Informationen kommen theoretisch nur zwei Ansätze in Frage: Die statische Analyse des Quellcodes oder die dynamische Analyse aller denkbaren Testfälle. Letzterer scheidet aus, da die Ausführung aller möglichen Testfälle aufgrund der Größe dieses „Suchraumes“ im Allgemeinen nicht zu bewältigen ist. Die statische Analyse vermag zwar für einfachere Teststrategien, wie zum Beispiel Verzweigungsüberdeckung (Kapitel 3.2) oder *all-uses* (Kapitel 3.4.2), alle zu überdeckenden Entitäten zu identifizieren, scheitert jedoch bereits an dieser Aufgabe aufgrund des Problems der *Feasibility* (siehe Definition 3.11) bei speziellen oder erweiterten teilpfadorientierten Kriterien wie *all-DU-paths+*. Darüber hinaus ist jedoch auch die Identifikation aller Pfade notwendig, die von einzelnen Testfällen so überdeckt werden können, dass sie in unterschiedlichen Kombinationen jeweils im obigen Sinne optimalen Testfallmengen entsprechen. Eine rein statische, graphentheoretische Analyse scheitert im Allgemeinen ebenfalls an der *Feasibility*, während eine ergänzende symbolische Ausführung aufgrund des NP-vollständigen Problems und gerade bei der Betrachtung einzelner Komponenten als SUT kaum realistisch ist.

Aufgrund obiger Betrachtungen ist bereits die Definition eines Abbruchkriteriums für ein vereinfachtes Ziel zum Scheitern verurteilt, wonach lediglich eine Testfallmenge zu identifizieren ist, die die größtmögliche Überdeckung ohne Berücksichtigung des Umfangs einer solchen Menge erreicht. Daher sind andere Abbruchbedingungen notwendig, die unabhängig vom optimalen, zu erwartenden Ergebnis sind. Für •gEAr wurden zwei solcher Bedingungen vorgesehen, die zusammen oder einzeln eingesetzt werden können. Dazu kann der Tester eine maximale „Generationenanzahl“ angeben, die die Heuristik nacheinander auswerten darf, und/oder eine maximale Ausführungszeit für das Werkzeug vorgeben, nach der die Optimierung abgebrochen wird. Bei Verwendung beider Kriterien greift das zuerst zutreffende. Selbstredend kann der Anwender die Testdatengenerierung und -optimierung auch jederzeit manuell anhalten. Der Vorteil eines solchen Verfahrens zur Testfalloptimierung ist gerade, dass jederzeit im Laufe der Ausführung eine Zwischenlösung abgerufen werden kann, die relativ früh eine bereits hohe Qualität aufweist.

Der Nachteil einer direkten oder indirekten Beschränkung der dem Werkzeug zur Verfügung gestellten Optimierungszeit nach obigem Muster ist, dass dieses Abbruchkriterium nicht

den jeweils gegenwärtigen Zustand der Metaheuristik berücksichtigt, sprich ihre Fähigkeit, in eventuell noch annehmbarer Zeit signifikant bessere Ergebnisse zu erzielen. Eine Möglichkeit, solche Informationen ebenfalls in das Abbruchkriterium einfließen zu lassen, besteht darin, die Fitness des jeweils besten Individuums einer Generation über die Ausführungszeit der Heuristik hinweg zu verfolgen. Typischerweise ergeben sich bei Anwendung Evolutionärer Verfahren für klassische, monoobjektive Aufgabenstellungen Verläufe der Art aus Abbildung 6.1 mit zunächst großer und später abnehmender Steigung. Ursache für diesen „Sättigungseffekt“ ist die zunehmend kleiner werdende Wahrscheinlichkeit, in einer weiteren Generation eine noch bessere Lösung zu finden - was durch eine zunehmende Konvergenz der Individuen einer Population zu einem Optimum noch begünstigt wird. Anschaulich begründen lässt sich hier die abnehmende Verbesserungswahrscheinlichkeit damit, dass die noch unbekannten Testfälle, die nicht-überdeckte Entitäten zu überdecken vermögen, eine mit zunehmender Überdeckung und damit Zeit schwindende Wahrscheinlichkeit aufweisen, aufgrund einer zufälligen (bei Random) oder evolutionären Generierung ermittelt zu werden. Ein empirisches Abbruchkriterium würde dieses Verhalten berücksichtigen und für einen Abbruch der Optimierung sorgen, sofern in absehbarer Zeit voraussichtlich mit keiner signifikanten Steigerung der bisherigen Lösungsqualität zu rechnen ist. In [Ost01b] wurde dieser Ansatz genutzt, um innerhalb eines hybriden Optimierungsverfahrens vorübergehend nicht-evolutionäre Operatoren anzuwenden. Falls über eine bestimmte Anzahl Generationen hinweg keine Verbesserung der Fitness des jeweils besten Individuums einer Generation festgestellt wurde, so wurde dabei der Genetische Algorithmus angehalten und rekonfiguriert sowie eine lokale Optimierung eingeleitet, nach deren Abschluss erneut zum Genetischen Algorithmus zurückgeschaltet wurde.

Eine erwähnenswerte Alternative zur Verfolgung der Fitnesswerte über mehrere Generationen hinweg stellt bei Evolutionären Verfahren die sogenannte *Cluster-Analyse* dar [OVW98]. Diese betrachtet lediglich die Verteilung der Individuen innerhalb der jeweils aktuellen Population. Dazu werden die Wertebereiche aller n Gene der Individuen auf das Intervall $[0,1]$ projiziert und somit normalisiert, wodurch die gesamte Population nun stets innerhalb eines Hyperwürfels der Kantenlänge 1 in einem n-dimensionalen kartesischen Koordinatensystem liegt. Aufgrund einer euklidischen Metrik wird anschließend paarweise der Abstand zwischen den Individuen in diesem Einheitshyperwürfel berechnet. Auf Basis eines Vergleichsabstandes d_{check} (der sogenannten „check distance“) werden die Individuen schließlich in Cluster eingeteilt, wobei je zwei Individuen zum gleichen Cluster gehören, sofern ihr euklidischer Abstand kleiner als d_{check} ist. Soll die Cluster-Analyse als Abbruchkriterium angewandt werden, so wird das Clustering für die gleiche Generation iterativ beginnend bei $d_{check} = \sqrt{n}$ (Diagonale des n-dimensionalen Hyperwürfels) mit zunehmend kleiner werdendem Vergleichsabstand durchgeführt, also zum Beispiel jeweils mit $d_{check} = \sqrt{n}/2,\ \sqrt{n}/3,\ \sqrt{n}/4,\ \ldots$: Bei $d_{check} = \sqrt{n}$ gibt es zunächst immer nur ein Cluster, welches alle Individuen umfasst, jedoch mit abnehmendem d_{check} zerfällt die Population im Allgemeinen in immer mehr Cluster, bis im Extremfall jedes Individuum ein eigenes Cluster bildet. Da Evolutionäre Verfahren für die zunehmende Konvergenz ihrer Individuen zu den identifizierten Optima bekannt sind, weisen bereits konvergierte Populationen selbst bei sehr kleinen Abständen d_{check} (wenn überhaupt) eventuell mehrere Culster auf, deren durchschnittliche Fitnesswerte jeweils annähernd gleich sind. Demnach kann die Optimierung abgebrochen werden, sobald die Population aufgrund der Cluster-Analyse als bereits konvergiert gilt. Der Vorteil die-

ses Clusterings ist, dass auch schwierige Suchräume korrekt interpretiert werden, zum Beispiel wenn die Optima entlang eines langgezogenen aber schmalen Bereichs des Hyperwürfels liegen.

5.2 Lokale Optimierung

Wenngleich die in Kapitel 5.1 vorgeschlagene Spezialisierung multi-objektiver Metaheuristiken mit verhältnismäßig geringem Aufwand im Allgemeinen schnell und qualitativ hochwertige Testfallmengen zu generieren und zu optimieren vermag (Kapitel 6), kann die Ermittlung des optimalen Testdatensatzes mit dem idealen Verhältnis von Anzahl überdeckter Entitäten entsprechend dem gewünschten Testkriterium zur Anzahl dafür notwendiger Testfälle nicht garantiert werden. Die Ursache dafür liegt, wie in Kapitel 5.1.8 angedeutet, in der Natur der Metaheuristiken: Anstatt den Raum aller möglichen Testfallmengen systematisch zu untersuchen, bedienen sich diese Algorithmen heuristischer, also zufallsgetriebener Verfahren, um die Konvergenz zu einer idealen Lösung zu erreichen. Zwar suchen Evolutionäre Verfahren einen Kompromiss zwischen *Erkundung (exploration)* und *Ausbeutung (exploitation)*, wie in Kapitel 4.5.1 beschrieben, dennoch kann es vorkommen, dass relevante Entitäten nicht mit geeigneten Testfällen überdeckt werden konnten, da diese Testfälle „zufällig“ nicht untersucht wurden.

Betrachtet man das scheinbar triviale Beispiel einer JAVA™-Methode, welche eine einzige ganzzahlige (*int*) Eingabe verarbeitet, so gibt es allein dafür bereits 4.294.967.296 verschiedene Testfälle. Enthält diese Methode nur eine einzige Verzweigung der Form "`if (x == 7)`", so ist die Wahrscheinlichkeit, genau denjenigen Testfall „zufällig“ unter einer uniformen Verteilung des Eingabewertes *x* auf Anhieb zu generieren, welcher diese Bedingung *wahr* werden lässt, mit $p_{wahr} = 1/2^{32} \approx 0,00000000023$ etwa 31-mal kleiner als einmalig sechs Richtige mit Superzahl im deutschen Lotto zu tippen ($p_{Lotto} = 1/139838160 \approx 0,00000000715$). Verbergen sich im Codeblock entlang des *wahr*-Zweiges nun auch noch eine Vielzahl nach der Teststrategie zu überdeckender Entitäten, so wäre eine Testfallmenge ohne einen solch unwahrscheinlichen Testfall qualitativ bedeutend schlechter.

Aus diesem Grund empfiehlt sich eine Erweiterung der Globalen Optimierung aus Kapitel 5.1, was im Rahmen der vorliegenden Arbeit in Form einer Hybridisierung vorgeschlagen wird. Das Gesamtverfahren startet demnach mit der *Globalen Optimierung* aus Kapitel 5.1 und zielt auf die Generierung einer optimalen Testfallmenge im Rahmen ihrer durch die Heuristik eingeschränkten Möglichkeiten. Wird eine der spezialisierten Abbruchbedingungen für Evolutionäre Verfahren aus Kapitel 5.1.8 (zum Beispiel auf Basis der Cluster-Analyse oder der Fitnessverfolgung) erfüllt, also wird die Population als zunächst hinreichend konvergent erkannt, so besteht immer noch die Hoffnung, dass bestimmte Pfade im *System Under Test* noch nicht überdeckt wurden. Um diese fehlende Überdeckung zu erreichen, kann nun eine Phase der *Lokalen Optimierung* eingeleitet werden, die gezielt zur Identifikation eines[25] geeigneten Testfalls dient, welcher den noch nicht überdeckten Pfad zur Ausführung bringt. Nachdem ein solcher Testfall ermittelt wurde, kann dieser allen aktuellen Testfallmengen (jedem Individuum der Population) hinzugefügt werden, woraufhin idealerweise wieder eine Phase der *Globalen Optimierung* folgt.

[25] Eventuell auch mehrerer Testfälle, falls weitere Entitäten noch nicht überdeckt werden konnten.

Damit kann letztere entscheiden, ob dieser zusätzliche Testfall eine Verbesserung des Verhältnisses zwischen Anzahl überdeckter Entitäten und Anzahl dazu eingesetzter Testfälle leisten kann oder doch eher verworfen werden sollte. Insbesondere bringt dieser Testfall neues „genetisches Material“ in die Population, was der *Globalen Optimierung* die Möglichkeit gibt, die vorhandenen Testfälle weiter zu optimieren.

5.2.1 Statische Analyse

Für die Umsetzung einer *Lokalen Optimierung* ist die Kenntnis aller Entitäten notwendig, die entsprechend dem gewünschten Testkriterium zu überdecken sind. Dies kann nur eine statische Analyse des Quelltextes oder des Bytecodes der zu testenden Applikation leisten. Da diese Analyse für verschiedene Überdeckungskriterien signifikant unterschiedlich ausfällt, werden im Folgenden exemplarisch mehrere Varianten näher beschrieben. Die einfachste Analyse erfolgt hinsichtlich der Verzweigungsüberdeckung, da hierbei pro Bedingungsanweisung lediglich ein eng begrenzter Bereich im Kontrollflussgraphen betrachtet werden muss. Ungleich schwieriger ist die Ermittlung der zu überdeckenden Entitäten hinsichtlich verschiedener datenflussorientierter Kriterien, weil dabei Definition und Verwendungen einer Variablen prinzipiell beliebig im datenflussannotierten Kontrollflussgraphen auftreten können.

Darüber hinaus dient eine statische Analyse auch noch einem weiteren Zweck. Die in Kapitel 5.1.1 dargestellte *Dynamische Analyse* liefert lediglich eine absolute Aussage darüber, wie viele Entitäten von einem Testfall oder Testdatensatz tatsächlich überdeckt wurden. Zwar genügt diese Information für das Verfahren der Globalen Optimierung, jedoch erlaubt sie keine Bewertung des jeweils erzielten *relativen* Überdeckungsgrades[26]. Erst wenn aufgrund einer statischen Analyse ermittelt wurde, wie viele Entitäten zur Erfüllung des Kriterium überdeckt werden müssen, steht fest, wie vollständig die strukturelle Abdeckung durch einen aktuellen Testfall erfolgt ist. Zu bedenken ist jedoch dabei, dass eine vollständige statische Analyse wegen des Problems der *Feasibility* im Allgemeinen dazu neigt, mehr zu überdeckende Entitäten zu ermitteln, als mit Testfällen überhaupt abgedeckt werden können – das heißt, das Kriterium erscheint dann trotz „Erreichen des Machbaren“ als nicht erfüllt.

Verzweigungsüberdeckung

Die statische Analyse des Quellcodes hinsichtlich der zu überdeckenden Verzweigungen im Kontrollfluss gestaltet sich aufgrund der Instrumentierung zum Zwecke der dynamischen Analyse für die Datenflussüberdeckung aus Kapitel 5.1.2 (ab Seite 125) verhältnismäßig einfach und ohne nennenswerten zusätzlichen Aufwand. Wie in Kapitel 5.1.2 ab Seite 137 dargestellt, wird jedes auszuwertende binäre Prädikat im Quelltext des zu testenden Systems, zum Beispiel als Operand einer `IF`- oder `WHILE`-Anweisung, durch den Instrumentierer vom Methodenpaar `"newPredicate"` und `"predResult"` entsprechend umschlossen. Dabei wird im Instrumentierungsprotokoll ein Eintrag mit der Ereigniskennung `"pos"` abgelegt, welches den gesamten Bedingungsausdruck zusammen mit seiner Position im Quellcode beschreibt. Entsprechendes gilt

[26] Typischerweise eine prozentuale Angabe, als Verhältnis der Anzahl der tatsächlich überdeckten zur Anzahl der zu überdeckenden Entitäten.

auch für Mehrfachverzweigungen aufgrund von `switch/case`-Anweisungen, welche mit Hilfe der Proben "`newSwitchPredicate`", "`switchPredResult`" sowie "`switchPredEquivalent`" instrumentiert werden. Für die statische Analyse zur Ermittlung aller Verzweigungen genügt es somit, das Instrumentierungsprotokoll des Datenflussinstrumentierers hinsichtlich aller Vorkommen der Ereignisse zu untersuchen, welche im Zusammenhang mit den Proben "`predResult`" beziehungsweise "`switchPredEquivalent`" stehen.

Das Kriterium der Verzweigungsüberdeckung kann dann als erfüllt betrachtet werden, wenn alle "`predResult`"-Ereignisse sowohl mit dem Wahrheitswert *wahr* als auch mit dem Wert *falsch* in den Ausführungsprotokollen der Testfälle eines Testdatensatzes vorkommen, denn dann wurde die Bedingung einer jeden Verzweigung sowohl zu wahr als auch zu falsch ausgewertet und entsprechend jeder Zweig überdeckt. Analog dazu muss jede der `case`-Optionen mindestens einmal zutreffen und direkt angesprungen werden, weshalb nicht das Ereignis "`switchPredResult`" hier von Belang ist sondern "`switchPredEquivalent`". Demnach muss das relevante Ereignis "`switchPredEquivalent`" unmittelbar auf das Ereignis "`switchPredResult`" im Ausführungsprotokoll folgen, um als Beitrag zur Überdeckung gewertet zu werden, sonst wäre der entsprechende `case`-Fall lediglich aufgrund eines vorhergehenden abgearbeitet worden.

Für das Testobjekt `DataflowExample` aus Listing A.1 ergibt sich nach der Instrumentierung (Listing A.2) das Protokoll aus Tabelle A.1. Demnach gibt es die beiden binären Verzweigungen mit den Ereigniskennungen 29 und 59 aufgrund der Anweisungen in Zeile 19 beziehungsweise Zeile 40 (Listing A.1) sowie die drei Ziele der Mehrfachverzweigung in Zeile 26 mit den Kennungen 42, 44 und 45.

All-defs-Überdeckung

Je nachdem, zu welchem Zweck die statische Analyse des *System Under Test* im Falle des *all-defs*-Kriteriums dient, kann diese Aufgabe auf mehrere Arten mit unterschiedlich hohem Aufwand umgesetzt werden. Für die *Lokale Optimierung* sind nicht nur die Anzahl der Definitionen sowie ihre jeweilige Position im Programm relevant, sondern zusätzlich auch alle von jeweils einer Definition erreichbaren Verwendungen. Da es sich darüber hinaus um eine Sonderform der sogenannten „Knoten-Knoten-orientierten“ Verfahren handelt [Bar00], erfordert dies eine Ermittlung aller *def/use*-Paare wie für das *all-uses*-Kriterium im folgenden Abschnitt.

Um jedoch zusätzlich zur absoluten Aussage über die Anzahl überdeckter Entitäten im Rahmen der *Globalen Optimierung* auch ein relatives Überdeckungsmaß berechnen zu können, ist über die dynamische Analyse hinaus lediglich eine Untersuchung des Instrumentierungsprotokolls notwendig, wie dies im vorangehenden Abschnitt für die Verzweigungsüberdeckung beschrieben wurde. Die dynamische Analyse eines Testdatensatzes ermittelt bereits alle tatsächlich überdeckten *def/use*-Paare und damit insbesondere alle überdeckten Definitionen im Sinne der *all-defs*-Teststrategie. Was demnach noch zur Berechnung des Überdeckungsmaßes fehlt, ist die Kenntnis der Anzahl aller im instrumentierten Teil des SUT auftretenden Definitionen. Diese lässt sich jedoch einfach dem statischen Protokoll der datenflussorientierten Instrumentierung entnehmen.

Jede Definition einer Variable wird durch den Instrumentierer in •gEAr entsprechend Kapitel 5.1.2 von einer Probe umschlossen. Für lokale Variablen werden zu diesem Zweck die

Methoden "`useLocal`" und "`useDefLocal`" (Seite 126) sowie "`enter`" zur Erfassung der Definition von formalen Methodenparametern (Seite 144) verwendet. Entsprechend werden Wertzuweisungen an statische Variablen mit Proben der Form "`defStatic`" und "`useDefStatic`" (Seite 128) sowie Definitionen nicht-statischer Felder mit Hilfe der Methoden "`defField`" und "`useDefField`" (Seite 129) instrumentiert. Eine umfangreichere Behandlung durch den Instrumentierer erfahren die *defs* einzelner Komponenten eines Arrays, wozu die Methodenfamilie mit "`defArray`", "`preIncArray`", "`preDecArray`", "`postIncArray`" und "`postDecArray`" (Seite 133) eingesetzt wird. Um die Anzahl aller Definitionen und den Ort im Quellcode jedes *defs* einer beliebigen Variable zu ermitteln, genügt es, alle Ereignisse aus dem Instrumentierungsprotokoll zu extrahieren, welche mit den oben genannten Ereignissen im Zusammenhang stehen.

ID	*Betroffenes Element*	*Zusätzlich betroffene Elemente*	*Zeile*
defLocal:			
12	java.lang.Exception exception		11
24	int DataflowExample.DataflowExample(int).anotherValue		18
25	int DataflowExample.DataflowExample(int).index		19
33	int DataflowExample.DataflowExample(int).anotherValue		20
51	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		37
55	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		39
61	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		41
66	int DataflowExample.main([Ljava.lang.String;).temp		43
useDefLocal:			
30	int DataflowExample.DataflowExample(int).index		19
enter (mit ctorEnter und earlyCtorEnter):			
22	public DataflowExample()		7
36	public DataflowExample(int)	int DataflowExample.DataflowExample(int).aValue	17
47	public int DataflowExample.getFieldTwo(int)	int DataflowExample.getFieldTwo(int).selection	25
67	public static void DataflowExample.main(java.lang.String[])	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args	36
defStatic:			
5	public static int DataflowExample.CONSTANT		2
6	public static int DataflowExample.CONSTANT		2
52	public static int DataflowExample.CONSTANT		38
useDefStatic:			
43	public static int DataflowExample.CONSTANT		30
defField:			
7	public DataflowExample DataflowExample.fieldOne		4
8	public DataflowExample DataflowExample.fieldOne		4
9	public int DataflowExample.fieldTwo		5
11	public int DataflowExample.fieldTwo		10
16	public DataflowExample DataflowExample.fieldOne		13
21	public int DataflowExample.fieldTwo		14
35	public int DataflowExample.fieldTwo		22
useDefField:			
41	public int DataflowExample.fieldTwo		28

Tabelle 5.1: Übersicht aller Definitionen im Codebeispiel *DataflowExample*

Angewandt auf das Codebeispiel aus Listing A.1 lassen sich mit diesem Verfahren aus dem Instrumentierungsprotokoll in Tabelle A.1 die in Tabelle 5.1 aufgeführten Definitionen[27] extrahieren. Identifiziert die dynamische Analyse, aufgrund der Rekonstruktion und Zusammenführung der überdeckten datenflussrelevanten Entitäten (entsprechend Kapitel 5.1.4) aus den Ausführungsprotokollen eines oder mehrerer Testfälle, zu jeder Definition mindestens eine davon erreichte Verwendung, so gilt das *all-defs*-Kriterium mit einem Überdeckungsmaß von $C_{all-defs} =$ 100% als erfüllt.

[27] Eine Unterscheidung bezüglich ihrer Art ist für die statische Analyse nicht von Bedeutung und dient hier nur der besseren Übersicht.

All-uses- und All-DU-paths-Überdeckung

Die Kriterien *all-defs* und *all-uses* gehören zu den sogenannten „Knoten-Knoten-orientierten" Verfahren [Bar00], das heißt, bezogen auf den datenflussannotierten Kontrollflussgraphen bestehen die zu überdeckenden Entitäten aus einem Knoten n_d mit der Definition einer Variablen *v* und einem weiteren Knoten n_u, welcher die von der Definition erreichbare Verwendung enthält, sofern es einen bezüglich *v* definitionsfreien Teilpfad von n_d zu n_u gibt. Im Gegensatz dazu zählt man die *all-DU-paths*-Teststrategie zu den „Knoten-Weg-orientierten" Verfahren, da eine zu überdeckende Entität aus dem Knoten n_d mit der Definition einer Variablen *v* und einem bestimmten, bezüglich *v* definitionsfreien Teilpfad (Weg) $p_{n_d}^{n_u}$ von n_d zu einem Knoten n_u, welcher eine Verwendung der Variablen *v* beherbergt.

Für eine *Lokale Optimierung* bezüglich des *all-defs*-Kriteriums, also die Generierung eines dedizierten Testfalls zur Überdeckung mindestens eines *def/use*-Paares für eine gegebene Definition, ist nicht nur die Kenntnis der Existenz eines Pfades von dem Knoten mit der Definition zu irgend einer Verwendung erforderlich, sondern idealerweise die Bestimmung aller Teilpfade von der betrachteten Definition zu allen von ihr erreichbaren Verwendungen. Entsprechendes gilt auch für das *all-uses*-Kriterium. Problematisch wird jedoch die Ermittlung aller Teilpfade zwischen den *defs* und den davon erreichbaren *uses* einer Variable insbesondere dann, wenn entlang eines Teilpfades eine Schleife im Kontrollflussgraphen auftritt. In diesem Fall gibt es aufgrund der statischen graphentheoretischen Betrachtung im Allgemeinen unendlich viele Teilpfade, welche eine Definition mit einer ihrer Verwendungen verbinden – jeder zusätzliche Schleifendurchlauf impliziert einen weiteren Teilpfad.

Aufgrund obiger Betrachtungen und zur Einschränkung des ohnehin hohen Rechen- und Zeitaufwandes zur automatischen Ausführung einer statischen Analyse für diese drei Datenflusskriterien wurde in •gEAr ein Verfahren für eine umfassende Analyse des Datenflusses umgesetzt, welche die erforderlichen Informationen für alle drei Teststrategien gemeinsam ermittelt. Dabei werden nicht *alle* relevanten Teilpfade ermittelt, sondern lediglich die für *all-DU-paths* erforderlichen schleifenfreien Teilpfade. Um die technischen Möglichkeiten zu demonstrieren, wurde diese statische Analyse in •gEAr auf Basis des JAVA™-Bytecodes anstelle des Quellcodes umgesetzt [OS06][28].

In der jüngeren Forschung und der zugehörigen Literatur gibt es eine Reihe unterschiedlicher Ansätze zur statischen Analyse des Datenflusses [PLR94, HR94, CRL99, CR01, LPH01, SP03]. Diese unterscheiden sich jedoch teilweise gravierend in ihrem Anwendungsgebiet und ihrer Analysegenauigkeit. So sind manche nur für prozedurale Sprachen geeignet, während andere lediglich für einzelne Klassen einer objektorientierten Applikation anwendbar sind. Oftmals schränken diese Ansätze auch den zulässigen Sprachschatz ein. Das wichtigste Unterscheidungsmerkmal der verschiedenen Verfahren ist ihre Präzision bezüglich der korrekten Auflösung potentieller Referenzgleichheiten, dem sogenannten *pointer aliasing*, also insbesondere die Sensitivität der Datenflussinformation bezüglich Kontrollfluss und Methodenaufrufkontext (siehe dazu auch Kapitel 3.4.6).

Das Verfahren in [HR94] ist für das datenflussorientierte Testen einer einzelnen Klasse kon-

[28] Der in [OS06] nur knapp umrissene Lösungsansatz wird in diesem Kapitel der Dissertation im Detail behandelt.

zipiert. Dabei wird zwar der „interprozedurale“ Datenfluss über die Methodengrenzen hinweg berücksichtigt, jedoch direkt nur innerhalb einer Klasse. Indirekt wird die Verwendung einer Klasse und die Interaktion selbiger mit anderen Klassen durch einen generischen Treiber simuliert, welcher mehrere Aufrufe der Methoden dieser Klasse in einer beliebigen Reihenfolge erlaubt. Aus diesem Grund ist dieses Verfahren als kontext-insensitiv zu betrachten. Der Ansatz basiert auf dem Verfahren von [PLR94] zur Ermittlung einer Abschätzung der durch *pointer aliasing* induzierten Beziehungen zwischen den Zeigervariablen in polynomialer Zeit, welches seinerseits für die Programmiersprache C entwickelt wurde. Demzufolge erfordert die Methode nach [HR94] keine Betrachtung der für die vollständige statische Analyse von beliebig-granularen JAVA™-Komponenten oder -Applikationen relevanten Aspekte wie Vererbung, Polymorphie, dynamisches Binden und Ausnahmebehandlung.

Einen umfassenderen Ansatz präsentiert [SP03] welcher im Gegensatz zu demjenigen in [HR94] zwar kontext-sensitiv ist, jedoch die Flusssensitivität zugunsten einer effizienteren Ausführung einschränkt. Dazu erstellt der vorgestellte Algorithmus einen sogenannten *Annotated Points-to Escape graph (APE)*, eine Erweiterung des klassischen Graphen zur Darstellung von *„pointer aliasing“*-Beziehungen (*Points-to Escape graph (PE)*). Die Knoten eines APE stellen Datenobjekte (also auch primitive Daten, nicht nur Instanzen von Klassen) dar, während die Kanten eine *points-to*-Beziehung darstellen. Die Annotationen an den Kanten beschreiben den Kontext der Programmausführung, in welchem eine bestimmte Beziehung hergestellt oder modifiziert wird. Dieser Kontext entspricht dem in Kapitel 5.1.4 (ab Seite 158) dargestellten Methodenaufrufkontext, wobei in den Darstellung aus [SP03] anstelle des Methodenaufrufs die Programmzeile der relevanten Anweisung tritt. Nachdem je ein APE für jede Methode aller Klassen konstruiert wurde, welcher „lediglich“ die kontext-sensitiven *„points-to“*-Beziehungen innerhalb der Methode als ganzes und nicht für jede Anweisung individuell darstellt, weshalb das Verfahren in gewisser Hinsicht fluss-insensitiv ist, werden diese APEs entsprechend der Aufrufreihenfolge innerhalb jeder Methode (beginnend bei einer vorgegebenen „Einstiegsmethode“) zusammengeführt.

Für eine statische Analyse des Datenflusses als Grundlage für die *Lokale Optimierung* und zur Bestimmung eines relativen Überdeckungsmaßes in •gEAr wurde ein Verfahren entwickelt, welches im Wesentlichen auf dem Ansatz von Ramkrishna Chatterjee und Barbara G. Ryder [CR01] basiert, welcher seinerseits auf [CRL99, Cha00] zurückgeht. Der Algorithmus von Chatterjee/Ryder ist sowohl fluss- als auch kontextsensitiv und weist daher eine hohe Präzision bei der Ermittlung der *points-to*-Beziehungen auf. Darüber hinaus ist er insofern modular, als dass zur Analysezeit stets nur wenige Methoden der untersuchten Applikation zeitnah bearbeitet werden, weshalb auch nur diese im Speicher vorgehalten werden müssen, was die Analyse sehr ressourcenschonend gestaltet. Weiterhin macht die Methode von Chatterjee/Ryder nur wenige Einschränkungen bezüglich der zugelassenen Sprachelemente von JAVA™. Unter anderem schließt sie die Verwendung von Threads aus und erfordert dass "`finalize()`"-Methoden keine Modifikationen an bestehenden Referenzen durchführen sowie dass die Initialisierung statischer Felder nicht von der Reihenfolge abhängt, in der die Klassen von der Virtuellen Maschine geladen werden. Da die Ausführung eines Programms bei Verwendung des *Reflection*-Mechanismus von JAVA™ nicht mehr sinnvoll statisch analysiert werden kann, wird der Einsatz von *Reflection* in von •gEAr zu analysierenden Programmteilen explizit ausgeschlossen. Beispielsweise kann

mittels *Reflection* eine Klasse instantiiert und damit eine ihrer Methoden aufgerufen werden, die zur Übersetzungszeit noch gar nicht implementiert und somit völlig unbekannt ist, weshalb sie auch nicht analysiert werden kann.

Da eine vollständige Beschreibung des für .gEAr entwickelten, spezialisierten Verfahrens nach [CR01] den Rahmen dieser Arbeit sprengen würde, sei hier nur das grundsätzliche Vorgehen skizziert und für weiterführende Informationen auf die entsprechenden Quellen verwiesen [Pol05, CR01]. Die Eingabe an diesen Algorithmus besteht aus einer Reihe zu analysierender JAVA™-Klassen in Form ihres Bytecodes. Zu ihrer Verarbeitung wird die sogenannte *BCEL*[29], eine ebenfalls freie Bibliothek als Gegenstück zu *ANTLR* aus Kapitel 5.1.2, eingesetzt. Die eigentliche statische Analyse des Datenflusses zusammen mit der Bestimmung der *points-to*-Relationen erfolgt auf Basis einer symbolischen Ausführung.

Mittels BCEL werden die Klassen zunächst eingelesen und anschließend wird ein *Interprozeduraler Kontrollflussgraph* (kurz *ICFG*) ähnlich dem *Java Interclass Graph (JIG)* aus Kapitel 3.1 konstruiert. Im Gegensatz zu einem JIG werden bei einem ICFG die *Aufrufkanten* und `exception`-Kanten nicht explizit modelliert, wie dies beim JIG aus Abbildung 3.4 der Fall ist. Auf diese Weise wird die jeweils relevante Kante dem aktuellen Kontext entsprechend während der symbolischen Ausführung ausgewählt und es werden nicht *alle* möglichen Kanten verfolgt, was die Kontextsensitivität des Verfahrens beeinträchtigen würde.

Um den Datenfluss trotz *pointer aliasing* so präzise wie möglich zu analysieren, wird der ICFG beginnend mit einer beliebigen Methode symbolisch ausgeführt. Zwar kann die Analyse mit jeder Methode beginnen, die natürliche Programmausführung in JAVA™ startet jedoch bekanntlich mit der `main()`-Methode. Jedes Datenobjekt (ob primitiv oder Klasseninstanz), welches zur „symbolischen Ausführungszeit“ erzeugt wird, erhält eine Kennzeichnung basierend auf seinem Erzeugungsort (voll-qualifizierter Methodenname und Offset der Bytecode-Anweisung)[30] und wird fortan als symbolischer Wert dem Kontrollfluss entlang propagiert. Die aktuelle „Belegung“ aller Variablen sowie der Speicherbereiche (Stack und Heap) der Virtuellen Maschine repräsentieren jeweils einen „Zustand“. Erreicht die symbolische Ausführung eine Verzweigung, so wird dieser Zustand dupliziert und jede Kante wird mit einer eigenen Kopie des Zustands weiter verfolgt, wodurch die Flusssensitivität gewahrt bleibt. Wird der Kontrollfluss aus verschiedenen Zweigen wieder zusammengeführt, so wird dies auch mit den entsprechend propagierten Zuständen nachvollzogen. Wurde entlang eines Teilpfades eine *points-to*-Beziehung zwischen den Variablen *x* und *y* identifiziert und entlang eines anderen Teilpfades eine zwischen *x* und *z*, so gehören nach der Vereinigung der beiden Teilpfade beide Variablen *y* und *z* zur *Alias*-Menge der Variablen *x*.

Aufgrund der Propagierung des Datenflusses beziehungsweise der *Aliasing*-Information über die verschiedenen Teilpfade, kann es vorkommen, dass bestimmte Knoten wiederholt ausgeführt werden müssen, sobald sich entlang eines weiteren Teilpfades, der diesen Knoten erreicht, neue datenflussrelevante Zusammenhänge ergeben haben. Um die Wahrscheinlichkeit einer unnötig häufigen Neuauswertung eines Knotens zu minimieren, wird der ICFG in einer erweiterten

[29] Byte Code Engineering Library, `http://jakarta.apache.org/bcel/`

[30] Zur Verbesserung der Genauigkeit können hier optional auch der aktuelle Aufrufstack oder darüber hinaus gehende Informationen hinzugezogen werden. Je umfangreicher und präziser die Kennzeichnung, desto komplexer wird die Analyse – im Extremfall terminiert sie nicht.

„Breitensuche“ abgearbeitet. Problematisch wird diese jedoch im Falle von Schleifen, da beim Betreten des Schleifenrumpfes mindestens eine eingehende Kante von einem noch auszuwertenden Knoten (welcher typischerweise die letzte Anweisung des Schleifenrumpfes beherbergt) kommt. Um diese Zusammenhänge korrekt erkennen und entsprechend behandeln zu können, wird zunächst eine Dominanzmatrix aufgrund der Kantenrelation des ICFG ermittelt: Ein Knoten n_i dominiert einen Knoten n_k, falls jeder Teilpfad vom Startknoten zum Knoten n_k durch den Knoten n_i verläuft. Erreicht die symbolische Ausführung einen Knoten n_j, so wird dieser nur dann analysiert, falls alle seine von n_i nicht dominierten Vorgänger bereits untersucht wurden.

Da ein Schleifenrumpf im Allgemeinen mehrere unterschiedliche Teilpfade umfasst, was insbesondere bei ineinander geschachtelten Schleifen eine kombinatorische Explosion zu verursachen vermag, kann die über einen Teilpfad propagierte und modifizierte *points-to*-Relation für einen anderen Teilpfad der Schleife relevant werden und umgekehrt. Aus diesem Grund ist jeder Knoten des ICFG solange wiederholt auszuwerten, wie sich die symbolischen Zustände seiner Vorgängerknoten ändern. Dieses Verfahren ist in der Literatur unter dem Begriff *Fixpunktiteration* bekannt. Da die Anzahl der Variablen endlich ist, terminiert auch die Fixpunktiteration stets, *sofern* die Anzahl der zu erzeugenden symbolischen Werte endlich ist. Dies schränkt zugleich die erreichbare Genauigkeit der Analyse ein, da beispielsweise bei einer Instantiierung einer Klasse innerhalb einer Schleife prinzipiell beliebig viele (symbolische) Werte erzeugt werden können, es sei denn, man unterscheidet nicht alle eindeutig – was zum Beispiel durch Kennzeichnung alleine aufgrund des Erzeugungsortes erreicht wird, jedoch eine verhältnismäßig ungenaue Erfassung der *points-to*-Beziehungen zur Folge hat.

Wie eine „Verzweigung“ und damit kontextsensitiv wird von der symbolischen Ausführung auch ein dynamisch zu bindender Methodenaufruf behandelt. Je nachdem, welche *Aliasing*-Beziehungen es aktuell für die betrachtete Aufrufinstanz gibt, also welche Typen die symbolischen Werte der potentiellen Aufrufkandidaten haben, werden alle in Frage kommenden Methodenimplementierungen getrennt symbolisch analysiert. Ähnlich wird auch eine ausnahmeauslösende Anweisung behandelt, wobei der anschließend zu analysierende Codeblock in Abhängigkeit vom Typ der jeweils geworfenen Ausnahme ausgewählt wird.

Der vorangehend beschriebene Algorithmus dient zunächst nur dem Zweck, die *points-to*-Relationen für alle Knoten zu ermitteln. Mit einer einfachen Modifikation ist es jedoch möglich, das Verfahren in einem einzigen Durchlauf auch zur Aufbereitung der *def/use*-Paare zu verwenden. Erreicht die symbolische Ausführung eine Definition einer Variablen, so wird der ihr zugewiesene symbolische Wert mit einem zusätzlichen Kennzeichen annotiert, nämlich mit dem eindeutigen Ort der Definition (kurz: *defSite*) dieser Variablen, ebenfalls in Form des voll-qualifizierten Methodennamens[31] und des Offsets der Zuweisungsanweisung im Bytecode. Sobald die symbolische Ausführung eine Verwendung des entsprechenden symbolischen Wertes erreicht, wird die aktuelle Bytecode-Anweisung als *useSite* interpretiert und zusammen mit der *defSite* des Wertes zu einem *def/use*- oder kurz DU-Paar zusammengesetzt.

Diese Zusammenhänge und die von diesem Verfahren ermittelten DU-Paare seien am Codebeispiel "`PointerAliasingExample`" aus Listing 5.1 exemplarisch erläutert. Die in der folgen-

[31] Klassen- und Instanzinitialisierungssequenzen werden im Bytecode wie eine eigenständige Methode mit einem speziellen Namen behandelt.

```
1  public class PointerAliasingExample
2  public int i = 0;     ↪ Bytecode-Offsets: 5̃,6̃
3
4  public static void main(String[] args) {
5    PointerAliasingExample pae1 = new PointerAliasingExample();     ↪ Bytecode-Offsets: 0,3,4,7
6    pae1.i = 10;     ↪ Bytecode-Offsets: 8,9,11
7    PointerAliasingExample pae2 = new PointerAliasingExample();     ↪ Bytecode-Offsets: 14,17,18,21
8    if (pae1.i < 0) {     ↪ Bytecode-Offsets: 22,23,26
9      pae2 = pae1;     ↪ Bytecode-Offsets: 29,30
10     pae2.i = 10;     ↪ Bytecode-Offsets: 31,32,34
11   }
12   pae2.i++;     ↪ Bytecode-Offsets: 37,38,39,42,43,44
13 }
14 }
```

Listing 5.1: Quellcode des Beispiels *PointerAliasingExample*

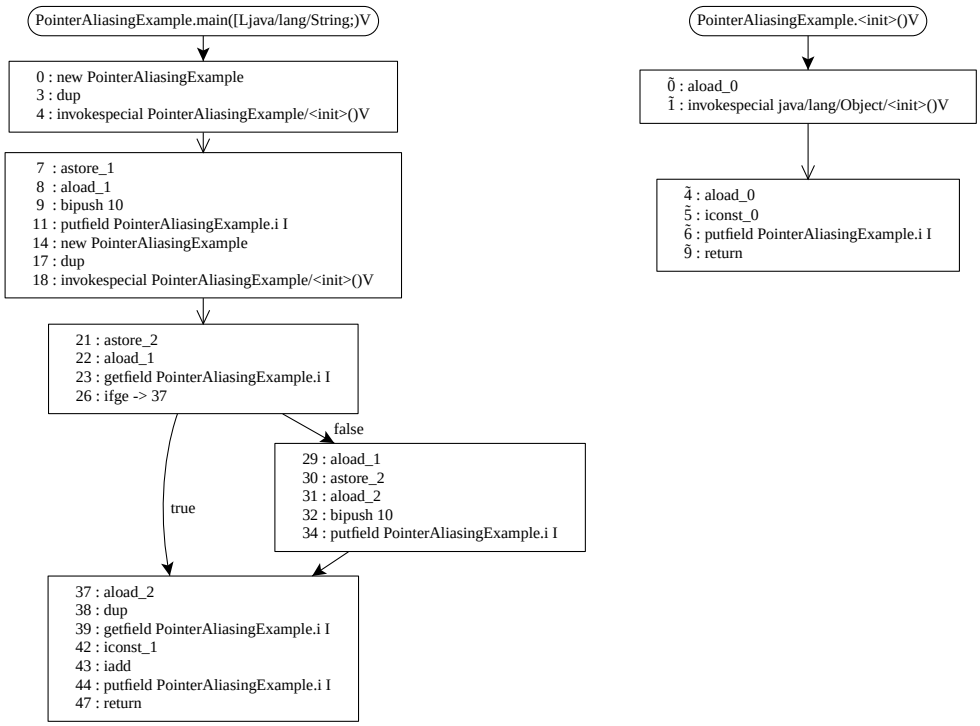


Abbildung 5.6: (I)CFG basierend auf dem Bytecode des Beispiels *PointerAliasingExample*

den Beschreibung auftretenden Verweise auf Codezeilen beziehen sich auf dieses Codebeispiel. Der zu diesem JAVA™-Quellcode gehörende Kontrollflussgraph (ICFG) auf Basis des entsprechenden JAVA™-Bytecodes ist in Abbildung 5.6 dargestellt und alle folgenden Angaben zu (Bytecode-)Offsets beziehen sich darauf. Die Ergebnisse der statischen Analyse zur Bestimmung der DU-Paare zeigt Tabelle A.4, auf deren (Zeilen-)Nummer jeweils verwiesen wird.

In Zeile 5 wird ein Objekt der Klasse "`PointerAliasingExample`" instantiiert. Da dieses Ereignis bei Offset 0 auftritt, wird das somit entstandene Objekt während der symbolischen Analyse als *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0* bezeichnet. Nach der Instantiierung wird das Objekt der Variablen `pae1` zugewiesen (Offset 7), weshalb dieser Definition die *defSite PointerAliasingExample.main([Ljava/lang/String;)V#7* annotiert wird. Da der ursprüngliche Name dieser lokalen Variablen im Bytecode fehlt, sofern die Klasse ohne Debugging-Informationen übersetzt wurde, wird sie in Tabelle A.4 unter dem Pseudonamen "`<noname>(#1)`"[32] geführt. In Zeile 6 wird das Feld `i` des Objektes definiert, welches von `pae1` referenziert wird. Um diese Zuweisung vornehmen zu können, erfolgt zunächst jedoch ein lesender Zugriff auf die Variable `pae1`. Diese Verwendung bei Offset 8 erhält als *useSite* die Bezeichnung *PointerAliasingExample.main([Ljava/lang/String;)V#8*. Da hierbei aus Sicht des Bytecodes auf die Variable "`<noname>(#1)`" und damit auf den symbolischen Wert *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0* zugegriffen wird, welcher mit der *defSite*-Annotation *PointerAliasingExample.main([Ljava/lang/String;)V#7* versehen wurde, kann das *def/use*-Paar bei Nummer 4 (Tabelle A.4) abgeleitet werden.

Ähnliches gilt auch für die Verwendung dieser Variablen im Prädikat in Zeile 8 beziehungsweise dem zugehörigen Offset 22. Aufgrund dessen wird ein weiteres *def/use*-Paar bezüglich der Variablen "`<noname>(#1)`" identifiziert, welches bei Nummer 7 dargestellt ist. Betrachtet man den ICFG aus Abbildung 5.6, so erkennt man, dass der Beginn einer Prädikatsauswertung im Bytecode nicht mehr erkennbar ist, weshalb die Verwendung dieses DU-Paars nicht als prädikativ klassifiziert werden kann – im Gegensatz zu einer statischen oder dynamischen Analyse auf Basis des Quellcodes (siehe Kapitel 5.1.2 auf Seite 137). Die Definition der Variablen "`<noname>(#1)`" in Zeile 5 erreicht eine letzte Verwendung in Zeile 9 (Offset 29), was zum *def/use*-Paar bei Nummer 8 führt.

Entsprechend zu Zeile 5 wird in Zeile 7 (im zugehörigen Bytecode bei Offset 14) die Klasse "`PointerAliasingExample`" erneut instantiiert, was zur Erstellung eines symbolischen Wertes namens *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14* führt, welcher der Variablen "`<noname>(#2)`" (ursprünglich `pae2`) unter der *defSite*-Annotation *PointerAliasingExample.main([Ljava/lang/String;)V#21* zugewiesen wird. Da eine statische Analyse aufgrund einer symbolischen Ausführung im Allgemeinen gar nicht[33] und nur in besonderen Fälle mit hohem Aufwand den Wahrheitsgehalt von Bedingungen in Prädikaten auszuwerten vermag, kann für die `IF`-Anweisung in Zeile 8 keine Aussage darüber getroffen werden, welcher der Zweige zur Laufzeit tatsächlich ausgeführt werden wird. Daher muss die statische Analyse beide Möglichkeiten berücksichtigen. Falls die Bedingung falsch ist, so wird im ICFG aus Abbil-

[32] „#1“ bezeichnet dabei ihren Index in der Tabelle der lokalen Variablen. Der Parameter *args* der Methode erhält hier den Index 0.

[33] Zum Beispiel, wenn die Zuweisung in Zeile 6 nicht mit einer Konstanten, sondern aufgrund eines Eingabeparameters aus `args` erfolgt.

dung 5.6 die *true*-Kante überdeckt, da die Bedingung im Bytecode negiert dargestellt ist (Offset 26). In diesem Fall erreicht die Definition der Variablen `pae2` eine Verwendung in Zeile 12 (Offset 37). Dieses DU-Paar bezüglich der im Bytecode als "`<noname>(#2)`" identifizierten Variablen ist bei Nummer 5 mit der *defSite PointerAliasingExample.main([Ljava/lang/String;)V#37* vermerkt. Ehe jedoch der Knoten des ICFG mit der Anweisungssequenz ab Offset 37 analysiert wird, werden zunächst alle nicht-dominierten Vorgängerknoten untersucht, in diesem Falle also der Knoten mit den Anweisungen 29 bis 34. Darin erfährt die Variable `pae2` eine weitere Definition bei Offset 30 (Zeile 9). Diese Definition mit der *defSite*-Annotation *PointerAliasingExample.main([Ljava/lang/String;)V#30* erreicht unmittelbar in Zeile 10 (Offset 31) eine Verwendung unter der *useSite*-Annotation *PointerAliasingExample.main([Ljava/lang/String;)V#31*. Das von der Bytecode-Variablen "`<noname>(#2)`" dabei referenzierte Objekt ist jedoch der symbolische Wert aus Offset 0, welcher zunächst der Variablen `pae1` zugewiesen wurde, daher der entsprechend lautende DU-Eintrag bei Nummer 3. Anschließend erreicht auch der Kontrollfluss über die *false*-Kante des ICFG den letzten Knoten ab Offset 37, was zur erneuten Verwendung der Variablen `pae2` ("`<noname>(#2)`") in Zeile 12 führt. Doch nachdem die letzte Definition der Variablen nun bei Offset 30 liegt, entsteht das *def/use*-Paar aus Nummer 9 zusätzlich zu dem bereits genannten aus Nummer 5.

Zusammenfassend ist also festzuhalten, dass die Propagierung des Datenflusses durch die Variable `pae2` hin zur Verwendung in Zeile 12 (Offset 37) auf zwei verschiedene Definitionen zurückgeführt werden kann: Diejenige in Zeile 7 (Offset 21), welche zum DU-Paar Nummer 5 führt und diejenige in Zeile 9 (Offset 30), aus der das *def/use*-Paar Nummer 9 hervorgeht. Diese Paare ermittelte die symbolische Ausführung aufgrund der Zusammenführung der beiden Analysezustände nach Offset 26 und nach Offset 34 entlang der Offset-Kanten (26,37) beziehungsweise (34,37) im ICFG.

Schwieriger wird die statische Analyse des Datenflusses aufgrund des nicht-statischen Feldes `PointerAliasingExample.i`. Interessant ist dabei insbesondere die Verwendung in Zeile 12 (Offset 39). Die Tabelle A.4 weist für diesen *use* die drei unterschiedlichen *def/use*-Paare mit den Nummern 6, 10 und 14 aus, obwohl durch „scharfes Hinsehen“ nur zwei, nämlich Nummer 6 und Nummer 14, tatsächlich möglich sind. Ursache für diese Diskrepanz ist eine Einschränkung der Flusssensitivität dieser Analyse zugunsten einer endlichen Ausführungszeit der symbolischen Auswertung. Wie vorangehend beschrieben, werden die Analysezustände aus den beiden alternativen Zweigen (entlang *true* beziehungsweise *false* im ICFG) unmittelbar vor Erreichen der Anweisung bei Offset 37 zusammengeführt. Dabei ermittelt die symbolische Ausführung, dass die Variable `pae2` entweder auf das in Zeile 5 (Offset 0) oder das in in Zeile 7 (Offset 14) instantiierte Objekt verweist, ersteres als Folge der Anweisung in Zeile 9. Um welche der beiden Möglichkeiten es sich jeweils handelt, hängt lediglich davon ab, welcher Teilpfad vom Anfang des Programms bis zur aktuell betrachteten Instanz der Anweisung in Zeile 12 tatsächlich ausgeführt wurde. Um diese Unterscheidung immer treffen zu können, müsste die symbolische Ausführung jeden möglichen Pfad durch das Programm einzeln analysieren – was zwar eine absolut flusssensitive Analyse zur Folge hat, die im Allgemeinen jedoch nicht terminiert. Die Aggregation der Information und die Anwendung einer Fixpunktiteration, wie im vorliegenden Fall, machen eine brauchbare statische Analyse des Datenflusses erst möglich, jedoch auf Kosten einer eingeschränkten Flusssensitivität.

Aufgrund der Zusammenführung der Zustände ist in Zeile 12 lediglich bekannt, dass `pae2` entweder auf den bei Offset 0 (Zeile 5) „instantiierten“ symbolischen Wert *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0* oder auf den Wert *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14* verweist. Die letzte bekannte Definition des nicht-statischen Feldes `i` der Instanz aus Offset 14 ist diejenige aufgrund der Initialisierungssequenz bei Offset $\tilde{6}$, daher das *def/use*-Paar Nummer 6 aus Tabelle A.4. Da die symbolische Ausführung nicht berücksichtigt, ob die Referenz von `pae2` auf den symbolischen Wert *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0* nur entlang des Teilpfades durch den ICFG-Knoten ab Offset 29 zustande kam und entlang des anderen Teilpfades *nicht* gültig ist, muss sie annehmen, dass die Definition des Feldes `i` in Zeile 10 (Offset 34) nicht die einzige, zuletzt gültige ist – was lediglich das DU-Paar Nummer 14 hervorgebracht hätte. Vielmehr betrachtet sie die Definition in Zeile 6 (Offset 11) als ebenfalls potentiell gültig bei Offset 39 (nämlich unter Umgehung der erneuten Definition bei Offset 34 entlang der *true*-Kante), was zur Ermittlung des *def/use*-Paares Nummer 10 führt.

Im Gegensatz zu rein flussinsensitiven Verfahren bewahrt die für •gEAr vorgesehene Strategie zur statischen *Points-to-* und Datenflussanalyse aufgrund einer symbolischen Ausführung eine gewisse Flusssensitivität, was sich mit einer leicht modifizierten Variante des Codes aus Listing 5.1 demonstrieren lässt, bei der anstelle der Bedingung `"pae1.i < 0"` in Zeile 8 die neue Bedingung `"pae2.i < 0"` tritt. Ein vollkommen flussinsensitives Verfahren würde die *Alias*-Beziehung zwischen `pae1` und `pae2` nicht erst ab Zeile 9 als solche interpretieren, sondern für die gesamte Methode. Demnach wäre die tatsächliche Instanz (eine der beiden aus Offset 0 oder Offset 14) hinter `pae2` beim lesenden Zugriff auf das Feld `i` in Zeile 8 nicht mehr zu unterscheiden, weshalb die Definition des Feldes `i` in Zeile 6 (Offset 11) mit der Verwendung in Zeile 8 (Offset 23) ebenso assoziiert werden würde wie die Definition des Feldes `i` in Zeile 2 (Offset $\tilde{6}$).

Nachdem zunächst wie vorangehend beschrieben alle *def/use*-Paare identifiziert wurden, können in einem zweiten Schritt alle zugehörigen DU-Teilpfade (siehe Definition 3.25) ermittelt werden. Dazu vermerkt die symbolische Ausführung während der DU-Paarung nicht nur die *defSite* und die zugehörige *useSite*, sondern auch den (Pseudo-)Namen der Variablen und ihren jeweiligen symbolischen Wert, wie dies auch in Tabelle A.4 dargestellt ist. Zur Herleitung der DU-Teilpfade wird der ICFG jeweils ausgehend von jeder vorangehend ermittelten *useSite* jeder Variablen *v* rückwärts, also in umgekehrter topologischer Reihenfolge der Knoten, traversiert. Dabei werden von einem Knoten n_j alle zu n_j führenden Kanten (n_i, n_j) in ihrer umgekehrten Richtung verfolgt, die somit von dem jeweils aktuell betrachteten Knoten n_j zu einem Vorgängerknoten n_i führen, dessen Analyseendzustand noch immer eine Zuordnung des gleichen symbolischen Wertes zur betrachteten Variable aufweist. Diese Form des sogenannten *Back-Tracking*-Verfahrens endet schließlich jeweils bei einem Knoten, dem die *defSite* zugeordnet ist.

Bezogen auf das Beispiel aus Listing 5.1 und dem zugehörigen ICFG in Abbildung 5.6 sei das Verfahren kurz an der *useSite* in Zeile 12 bei Offset 37 exemplarisch demonstriert. Laut Tabelle A.4 verweist die Variable `"<noname>(#2)"` (`pae2`) potentiell auf die symbolischen Objekte *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14* (Nr. 5) oder *PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0* (Nr. 9). Bezüglich `"<noname>(#2)"` und ihrer Referenz auf das erstere symbolische Objekt beginnt das *Back-Tracking* im letzten Knoten. Von den beiden Vorgängerknoten dieses Abschlussknotens kommt

jedoch nur derjenige mit der Bytecode-Anweisungsfolge 21 bis 26 in Frage, da der andere (Offset 29 bis 34) nur noch das *Aliasing* von "<noname>(#2)" und dem letzteren Objekt (aufgrund der Anweisung in Zeile 9, also Offsets 29 und 30) kennt. Das *Back-Tracking* endet schließlich auch bei diesem Vorgängerknoten mit der Anweisungsfolge 21 bis 26, da die *defSite* mit dem Offset 21 assoziiert ist.

5.2.2 Umsetzung der lokalen Optimierung

Aufgrund statischer Analysen, ähnlich denen aus Kapitel 5.2.1, können in Abhängigkeit vom vorgegebenen Testkriterium alle zur Erfüllung des Kriteriums notwendigerweise zu überdeckenden Entitäten identifiziert werden. Dabei ist nicht nur ihre Anzahl sondern insbesondere auch ihre Art und Lage im getesteten Programm(fragment) auschlaggebend. Nachdem anhand einer dynamischen Analyse nach Kapitel 5.1.4 ermittelt wurde, welche Entitäten bereits überdeckt sind, gilt es nun im Rahmen der lokalen Optimierung die noch nicht überdeckten Entitäten gezielt zu verfolgen. Dazu wurden in verschiedenen Forschungsprojekten bereits diverse Ansatzmöglichkeiten untersucht. Diese reichen von einer rein statischen Ermittlung notwendiger Testfälle und -daten mittels symbolischer Ausführung [Grz04] bis hin zum Einsatz von metaheuristischen Verfahren. In Kapitel 2.3.2 wurden unterschiedliche Methoden skizziert, wobei diese meist zum Ziel hatten, durch wiederholte Anwendung für jede zu überdeckende Entität einen eigenen Testfall zu erzielen und dadurch eine Testfallmenge aufzubauen.

Schwerpunkt dieser Arbeit ist insbesondere die Generierung *optimaler* Testfallmengen, also die Reduzierung des Validierungsaufwands durch Ermittlung minimaler Testfallmengen mit maximaler Überdeckung. Darüber hinaus ermöglicht das Verfahren auch die ebenso automatische Generierung von Testdaten, welche eine Maximierung mehrerer Überdeckungsgrade nach unterschiedlichen Teststrategien verfolgt, also der gleichzeitigen Erfüllung mehrerer Testkriterien entgegen strebt. Aus diesem Grund ist die lokale Optimierung im Kontext der globalen Optimierung aus Kapitel 5.1 lediglich als eine unterstützende Maßnahme zu sehen - und als solche auch nicht erforderlich, sondern lediglich der Performanz des gesamten Ansatzes zuträglich.

Da eine automatische Testdatenermittlung aufgrund deterministischer Ansätze, zum Beispiel mittels statischer Analysen und/oder symbolischer Ausführung, bislang im Allgemeinen nicht erfolgreich sind, nicht zuletzt aufgrund der statisch selten ermittelbaren Anzahl von Schleifeniterationen, haben sich Verfahren als besonders effektiv hervorgetan, welche ähnlich der globalen Optimierung aus Kapitel 5.1 auf spezialisierte Metaheuristiken fußen. Da es diesbezüglich schon einige wissenschaftliche Untersuchungen mit ansehnlichem Ergebnis gab, sei hier nur kurz der Ansatz der Forschungsgruppe von Dr. Joachim Wegener (DaimlerChrysler AG, Berlin) [WBP02, Bar00, McM04] skizziert, welches sich im Rahmen von •gEAr besonders anbietet.

Zunächst erfordert der Einsatz Evolutionärer Verfahren zur lokalen Optimierung eine Klassifikation der Testkriterien nach Art der zu überdeckenden Entität (Testziel) [Bar00]. Demnach gibt es sogenannte:

- *Knotenorientierte Verfahren:* Diese erfordern pro Entität die Ausführung eines bestimmten Knotens im Kontrollflussgraphen. Typischer Vertreter dieser Gattung ist die Anweisungsüberdeckung (Kapitel 3.2.1).

- *Pfadorientierte Verfahren:* Jede zu überdeckende Entität stellt einen Pfad im Kontrollflussgraphen dar. Zu diesen Verfahren zählen die verschiedenen Varianten der strukturierten Pfadüberdeckung (Kapitel 3.2.3).

- *Knoten-Wegorientierte Verfahren:* Zur Überdeckung einer Entität ist hier die Ausführung eines bestimmten Teilpfades notwendig, also die Ausführung eines Testfalls, bei dem zunächst ein bestimmter Knoten über einen beliebigen Teilpfad erreicht wird und anschließend ein spezieller Teilpfad ausgehend von diesem Knoten verfolgt wird. Hierzu zählt insbesondere das *all-DU-paths*-Kriterium (Kapitel 3.4.2).

- *Knoten-Knotenorientierte Verfahren:* Diese erfordern für jede Entität die Überdeckung eines Pfades, welcher je zwei dedizierte Knoten in einer bestimmten Reihenfolge enthält, wobei die Teilpfade vom Startknoten zum ersten Knoten sowie vom ersten zum zweiten Knoten ebenso beliebig sind, wie der Teilpfad vom zweiten Knoten zum Ende der Programmausführung. Zu dieser Kategorie zählt das *all-uses*-Kriterium (Kapitel 3.4.2).

Im weitesten Sinne wird das Kriterium der Verzweigungsüberdeckung (Kapitel 3.2.2) ebenfalls zur Klasse der *knotenorientierten Verfahren* gezählt, wobei zusätzliche Forderungen zu erfüllen sind, während das *all-defs*-Kriterium (Kapitel 3.4.2) eine Zwitterform zwischen *knoten-knotenorientierten* und *knoten-wegorientierten Verfahren* einnimmt [Bar00].

Aufgrund dieser Klassifikation kann für jedes Kriterium eine individuelle Bewertungsfunktion definiert werden, welche jedem Testfall eine „Güte“ zuordnet. Zum besseren Verständnis sei vorweggenommen, dass die „Population“ eines zum Zwecke der lokalen Optimierung spezialisierten Evolutionären Verfahrens aus einzelnen Testfällen besteht, im Gegensatz zur globalen Optimierung, wo ein Individuum eine ganze Testfallmenge repräsentiert (siehe auch Abbildung 5.4). Eine generische und zunächst von der Kriterienklasse unabhängige Fitnessfunktion besteht nach [WBP02] aus zwei Grundbeiträgen: Der sogenannten *Annäherungsstufe* (*approximation level*) und der *lokalen Abstandsfunktion* (*local distance*).

Die *Annäherungsstufe* spiegelt dabei die Anzahl der Verzweigungen wider, die zwischen dem zu erreichenden Knoten und den von einem Testfall bereits überdeckten Knoten liegen. Dabei werden jedoch nur diejenigen Verzweigungen berücksichtigt, die mindestens eine Kante umfassen, deren Überdeckung unwiderruflich zum Verfehlen des gewünschten Knotens im Kontrollflussgraphen führen. Im Gegenzug bewertet die *lokale Abstandsfunktion* für die jeweils letzte relevante aber in unerwünschter Weise überdeckte Verzweigung, wie „weit“ der betrachtete Testfall die notwendige Bedingung verfehlt hat, um den anderen Zweig zur Ausführung zu bringen. Für die *lokale Abstandsfunktion* gibt es eine Reihe spezialisierter Berechnungsvorschriften in Abhängigkeit von der Art des Prädikats und dem erwünschten Auswertungsergebnis [Bar00, MMS98]. Das Evolutionäre Verfahren wird bei diesem Ansatz so spezialisiert, dass es eine Minimierung der Fitness anstrebt. Wurde ein Individuum mit Fitness 0 identifiziert, so überdeckt dieser Testfall genau die gewünschte Entität.

Während für *knotenorientierte Verfahren* bereits die *Annäherungsstufe* alleine als Bewertungsgrundlage genügt, erfordern die anderen Klassen eine detailliertere Betrachtung. Für *pfadorientierte Verfahren* kann eine Fitnessfunktion beispielsweise auf Grundlage der jeweils fehlerhaft überdeckten Verzweigungen definiert werden. Alternativ ist auch die Bewertung der Länge

des bereits überdeckten Teilpfades beginnend beim Startknoten hilfreich. Bei *knoten-wegorientierten Verfahren* wird zunächst die Annäherung an den ersten Zielknoten verfolgt sowie entsprechend bewertet und erst bei Erreichen dieses Knotens kann eine Verfeinerung der Bewertung aufgrund obiger Betrachtungen für *pfadorientierte Verfahren* erfolgen. Schließlich beruht die Fitnessberechnung der Testfälle bei *knoten-wegorientierten Verfahren* auf der Betrachtung zweier Annäherungsstufen: Einer für die Annäherung an den ersten Zielknoten und einer zweiten, die die Annäherungsstufe zwischen den beiden Zielknoten berücksichtigt.

Da im Rahmen der statischen Analyse bereits ein Kontrollflussgraph des zu testenden Systems erstellt und analysiert wurde, können die jeweiligen *Annäherungsstufen* aufgrund der dynamischen Überdeckung leicht ermittelt werden. Zur Berechnung der *lokalen Abstandsfunktionen* ist eine dedizierte Instrumentierung mit zugehöriger dynamischer Analyse notwendig, die die verschiedenen Werte der in Prädikaten verwendeten Variablen protokolliert und entsprechend ihrer Verknüpfung aufbereitet. Auf den für •gEAr vorgesehenen Prototypen einer solchen Instrumentierung und dynamischen Analyse sei hier lediglich verwiesen [Tob04].

Als Abbruchkriterium eignet sich im Kontext der lokalen Optimierung natürlich das Erreichen einer Fitness von 0 für mindestens einen Testfall, also die Überdeckung der gewünschten Entität. Wie bereits in vorangehenden Kapiteln angedeutet, kann eine statische Analyse im Allgemeinen nicht entscheiden, ob eine identifizierte Entität überhaupt überdeckt werden kann. Falls es keinen Testfall gibt, der einen zur Überdeckung einer identifizierten Entität notwendigen Pfad auszuführen vermag, muss sichergestellt werden, dass eine lokale Optimierung auf Basis metaheuristischer Verfahren dennoch rechtzeitig (aber nicht zu früh) abgebrochen wird und ein anderes Testziel (eine andere Entität) gewählt wird. Sind jedoch alle noch offenen Testziele erfolglos heuristisch untersucht worden, so kann daraufhin auch die globale Optimierung terminiert werden.

Zwar stellt die lokale Optimierung mittels Evolutionärer Verfahren eine Verbesserung der Globalen Optimierung dar, dennoch kann auch sie durch eine ungünstige Wahl der Bewertungsfunktionen leicht in die Irre geleitet werden. Um ihre Performanz und Stabilität zu erhöhen, gibt es eine Vielzahl unterschiedlicher Ansätze. Nennenswert sind insbesondere der sogenannte *chaining approach* [McM04] und das *flag removal* nach [HHH+02]. Grundidee des *chaining approach* ist es, die Kanten des Kontrollflussgraphen für die jeweils aktuell betrachtete und zu überdeckende Entität in *kritische*, *semi-kritische* und *irrelevante* Kanten zu unterscheiden. Die Heuristik wird dann so spezialisiert, dass sie kritische Kanten gezielt meidet und die Anzahl der überdeckten semi-kritischen Kanten minimiert. Dieses Verfahren ist besonders für *knotenorientierte Verfahren* geeignet und berücksichtigt bei der Klassifizierung der Kanten auch den Datenfluss. Darüber hinaus passt es sich dynamisch den auftretenden Schwierigkeiten entlang aktuell überdeckter (Teil)Pfade an. Ziel des sogenannten *flag removal* ist es, die für Evolutionäre Verfahren kritischen Prädikate basierend auf Boole'schen Variablen zu entschärfen. Gerne versuchen Programmierer, komplexe Bedingungen aufzuspalten und die Teilbedingungen in verschiedenen Programmzeilen zunächst unabhängig voneinander auszuwerten. Die Ergebnisse dieser Teilbedingungen werden in temporären Zwischenvariablen gespeichert und im eigentlichen Prädikat nur noch mittels logischer Operatoren in Relation gesetzt. Die in [Bar00] vorgeschlagenen und in [WBP02] propagierten *lokalen Abstandsfunktionen* weisen in Falle Boole'scher *flags* jedoch weitläufige „Plateaus" auf, die einem Evolutionären Verfahren keinerlei Selektionsdruck bieten.

5.3 Automatische Testtreibergenerierung

Die Umsetzung eines Verfahrens zur automatischen Testdatengenerierung in einem Werkzeug macht nur dann Sinn, wenn sie unabhängig vom zu testenden System erfolgt. Wie in Kapitel 5.1.5 angedeutet, sind die zu untersuchenden *Systems Under Test* (SUT) sehr vielfältig, genauso wie ihre Schnittstellen. Um ein Werkzeug wie •gEAr so weit wie möglich unabhängig von einem bestimmten SUT zu gestalten, wurde eine standardisierte sowie allgemein verfügbare oder zumindest leicht zu erstellende Schnittstelle zum SUT gewählt: Die Einstiegsmethode `main` einer jeden JAVA™-Standalone-Applikation. Stellt das Testobjekt eine vollständige Applikation ohne weitere Benutzerinteraktion dar, so kann •gEAr bereits ohne weitere Anpassungen verwendet werden. Falls jedoch beispielsweise eine graphische Benutzerschnittstelle vom Testgenerator anzusprechen ist, erfordert dies weitere Werkzeuge, welche eventuell über eine dazwischen geschaltete Hilfsapplikation mit den Testdaten versorgt wird.

Am häufigsten hingegen werden strukturelle Testkriterien eher in den frühen Verifikationsphasen Modultest oder Integrationstest angewendet. Zu diesem Zeitpunkt gibt es eine vollständig ausführbare Applikation üblicherweise noch gar nicht. Vielmehr werden hier einzelne Methoden, Klassen oder Komponenten (zum Beispiel *packages*) getrennt von ihrer zukünftigen Einsatzumgebung getestet. Dies erfordert den Einsatz spezieller Hilfsroutinen, nämlich sogenannter Treiber (*driver*). Typische Aufgabe solcher Treiber für JAVA™-Komponenten ist es zunächst, alle für die Ausführung des Testobjektes erforderlichen Klassen entsprechend des Bedarfs zu instantiieren und in einen wohldefinierten Zustand zu versetzen. Diese Menge bezeichnet man üblicherweise als *Fixture* und wird in der *setup*-Phase angelegt. Ebenfalls Aufgabe dieses initialen *setups* ist es beispielsweise, alle notwendigen Dateien zu öffnen sowie Datenbank- und Netzwerkverbindungen herzustellen. Anschließend führt der Testtreiber eine Reihe von Operationen durch, die im Zusammenhang mit dem eigentlichen Test stehen, wie zum Beispiel das Aufrufen der zu testenden Methoden in einer wohldefinierten Reihenfolge. Während oder am Ende dieser zweiten Phase kann dem Treiber auch die Aufgabe übertragen werden, bestimmte Ergebnisse oder Zustände zu überprüfen (*assertions*). Nach Durchführung des eigentlichen Tests gilt es noch „aufzuräumen“, das heißt zum Beispiel offene Dateien zu schließen und Netzwerkverbindungen zu trennen, was in die Phase des sogenannten *teardown* fällt.

Um die Erstellung solcher Testtreiber zu vereinfachen und über verschiedene Programmiersprachen hinweg zu harmonisieren, gibt es eine Reihe verschiedener Ansätze. Der wohl berühmteste aus dieser Kategorie mündete im sogenannten *xUnit*-Framework (siehe Kapitel 2.3), von dem es jeweils sprachenspezifische Varianten gibt, darunter *JUnit*[34] für JAVA™. Das im vorangehenden Absatz beschriebene Kernkonzept von *JUnit* ist jedoch für eine allgemeine Verwendung im Rahmen von •gEAr zu starr. Ursache dafür ist, dass ein *JUnit*-Test vollständig ablauffähig sein muss, das heißt insbesondere, dass bereits alle Testszenarien zusammen mit den erforderlichen Testdaten darin kodiert sein müssen. Im Gegensatz dazu muss ein automatischer Testgenerator auf Basis metaheuristischer Verfahren eine Vielzahl verschiedener Testabläufe jeweils mit eigenen Datensätzen ausführen können, um diese einer dynamischen Analyse und damit einer Bewertung zu unterziehen.

[34] `www.junit.org`

Aus diesem Grund wurde für den Einsatz in •gEAr ein zweistufiges Testtreiberkonzept entwickelt und so umgesetzt, dass die Testtreibergenerierung bei Bedarf ebenfalls automatisch ausgeführt werden kann. In der ersten Stufe wird ein generischer Testtreiber erstellt, welcher vollständig durch die vom Testfallgenerator übermittelten Daten gesteuert wird. Dabei legt der Testdatengenerator sowohl die Szenarien fest, als auch die dabei zu verwendenden Daten. Dieses Verfahren ähnelt dem in [HR94] dargestellten Ansatz zur statischen Datenflussanalyse einzelner Klassen, ist jedoch so umfassend, dass der Test auf einer beliebigen Granularitätsstufe angesetzt werden kann, von einer einzelnen Methode, über eine einzelne Klasse bis hin zu einem beliebig großen Klassenverbund. Da dieser Testtreiber sehr generisch sein muss, enthält er weder Testszenarien noch Testdaten, welche stattdessen getrennt vom Treiber in Form eines Datenstroms beziehungsweise unterschiedlich langer Argumentlisten vorliegen. Da ein solch generischer Testtreiber selbst zusammen mit dem Datenstrom kaum noch von einem menschlichen Tester manuell nachvollzogen oder gar geändert werden kann, werden diese beiden Artefakte in der zweiten Stufe zusammengeführt und in einem *JUnit*-konformen Testfall übersichtlich abgelegt.

Wie bereits in Kapitel 5.1.5 erläutert, sollte für die Kodierung der Individuen eines Evolutionären Verfahrens eine geeignete Struktur mit Bedacht gewählt werden. Unabhängig von einem speziellen SUT und dennoch intuitiv wurde daher für •gEAr die Repräsentation eines Testfalls in Form einer Argumentliste mit einzelnen Daten primitiven Typs gewählt, wie auch in Abbildung 5.4 dargestellt. Falls das Testobjekt lediglich eine nicht-statische Methode m einer beliebigen Klasse k ist, so muss vor dem Aufruf der Methode m zunächst eine Instanz der Klasse k erstellt werden, welche Ziel des Mehthodenaufrufs ist. Erschwerend kommt hinzu, dass die Methode möglicherweise eine Reihe unterschiedlicher Parameter aufweist, welche jeweils nicht nur von einem primitiven Typ sein müssen. In diesem Fall sind zusätzlich für jeden nicht-primitiven Parameter entsprechende Objekte zu instantiieren. Bei der Instantiierung dieser Objekte oder der Klasse k müssen eventuell vorhandene Konstruktoren aufgerufen werden, welche ihrerseits ebenfalls weitere Objekte als Parameter entgegen nehmen – dabei kann es vorkommen, dass es keinen *default*-Konstruktor (mit leerer Parameterliste) gibt; ein Aufruf der Konstruktoren oder Methoden mit Null-Referenzen testet das SUT jedoch nicht ausreichend. Erschwerend kommt hinzu, dass das Verhalten einer aufgerufenen Methode vom Zustand des umfassenden Objektes ebenso abhängt wie vom Zustand der als Parameter übergebenen Objekte. Aus diesem Grund muss ein generischer Testtreiber nicht nur die einmalige Instantiierung notwendiger Klassen vornehmen, ehe die zu testende Methode aufgerufen wird, sondern zusätzlich weitere Methoden der Ziel- oder Parameterobjekte vorab aufrufen können oder zumindest deren Zustand direkt oder indirekt vorweg beeinflussen.

Um den Zustand eines Objektes zu beeinflussen, ganz gleich ob er das Ziel eines Methodenaufrufs ist oder als Parameter für einen solchen dient, können die Zustandsvariablen (Felder) in manchen Fällen direkt geändert werden, zum Beispiel bei Feldern, die mit dem Schlüsselwort *public* deklariert wurden und unter Umständen auch bei *protected*-Variablen oder Feldern ohne besondere Auszeichnung. Typisch und ein Zeugnis guten Programmierstils ist es jedoch eher, den Zustand indirekt über den Aufruf der Methoden des Objektes zu beeinflussen. Ob nun ein Methodenaufruf durch den Testtreiber als Teil des Tests selbst oder lediglich zum Generieren eines *Fixture* dient, spielt für das Verfahren aus •gEAr letztlich keine Rolle mehr, weshalb diese Unterscheidung aufgegeben wird. Daher ist die im Folgenden betrachtete Testeinheit stets eine

ganze Klasse – das heißt, alle ihre Methoden müssen in einer noch festzulegenden Weise aufgerufen werden können. Eventuell zusätzliche Klassen sind auf die gleiche Art und Weise zu behandeln, sofern sie für den Aufruf irgendwelcher Methoden relevant sind.

Aufgrund der Polymorphie können Methoden nicht nur Parameter des deklarierten Typs entgegennehmen, sondern auch Instanzen aller entsprechenden Unterklassen. Dies ist besonders dann zu berücksichtigen, wenn der deklarierte Datentyp eines Parameters keine instantiierbare Klasse darstellt, sei es weil es sich beim Typ um eine Schnittstellendefinition (*Interface*) handelt oder weil die Klasse abstrakt ist. Ein generischer Testtreiber muss daher auch die Möglichkeit vorsehen, anstelle eines Objektes vom deklarierten Typ auch Objekte beliebiger Unterklassen als Parameter einer Methode übergeben zu können. Bedauerlicherweise aber nicht unerwartet bietet JAVA™ keine Funktionalität an, beispielsweise über den *Reflection*-Mechanismus alle Unterklassen einer Klasse abzufragen – sehr wohl können jedoch alle Superklassen in Erfahrung gebracht werden. Dies ist nicht weiter verwunderlich, da die Unterklassen zur Übersetzungszeit noch nicht feststehen müssen, daher also auch noch nicht bekannt sind. Um dieses Problem zu lösen, startet die Testtreibergenerierung zunächst mit dem Aufbau einer vollständigen Klassenhierarchie aller bekannten Klassen, welche sowohl die Beziehungen der vom Benutzer bereitgestellten Klassen des SUT als auch die von der JAVA™-Laufzeitbibliothek angebotenen Klassen berücksichtigt. Diese Klassenhierarchie wird dabei iterativ und *bottom-up* aufgebaut, da zu jeder Klasse lediglich die Superklasse und alle implementierten Schnittstellen (Interfaces) bekannt sind.

Ermittlung der ausführungsrelevanten Entitäten — Testtreiber-Generator

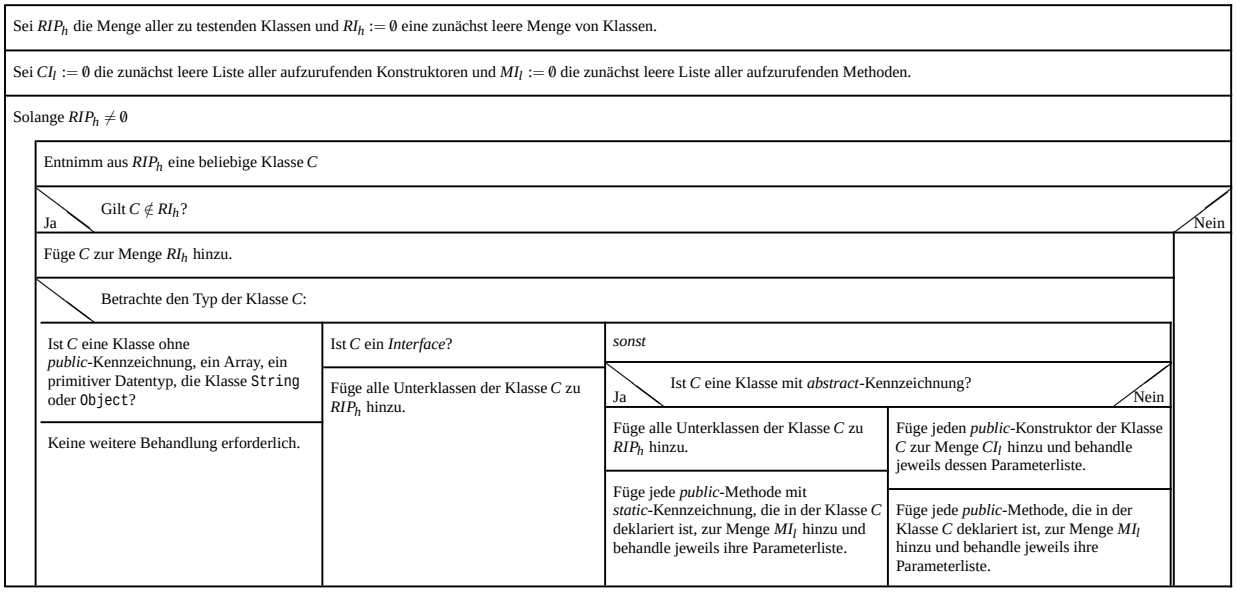


Abbildung 5.7: Verfahren zur Ermittlung der für den Test relevanten Entitäten

Nachdem nun alle bekannten Vererbungsbeziehungen erfasst sind, werden ebenfalls iterativ alle aufzurufenden Konstruktoren und Methoden zusammengetragen und analysiert sowie dabei die notwendigen Variablen und Klasseninstanzen ebenfalls festgehalten. Dieser Prozess ist in Abbildung 5.7 dargestellt und hat zum Ziel, folgende Informationen zu ermitteln:

- MI_l: die Liste aller Methoden, die vom Testtreiber aufgerufen werden können und sollen.
- CI_l: die Liste aller Konstruktoren, die vom Testtreiber aufgerufen werden können und sollen.
- RI_h: die Menge aller Datentypen, von denen mindestens eine Instanz und damit eine entsprechend typisierte Variable im Testtreiber benötigt wird, zusammen mit der tatsächlichen Anzahl der mindestens erforderlichen Instanzen. Falls eine Methode mehrere Parameter vom gleichen Typ erwartet, gibt deren Anzahl die minimal benötigte Anzahl entsprechender Instanzen für den Aufruf der Methode wieder. Pro Datentyp muss diese minimale Anzahl über alle Methoden hinweg bestimmt und deren Maximum erfasst werden. Die Datentypen können dabei sowohl Klassen als auch primitive Datentypen sein, die vom *Reflection*-Mechanismus von JAVA™ ebenfalls durch ein entsprechendes Klassenobjekt repräsentiert werden. Arrays unterschiedlicher Dimensionen stellen dabei jeweils unterschiedliche Datentypen dar.

Die in der Menge RI_h gesammelten Informationen sind einerseits für die automatische Ermittlung der Schnittstellenbeschreibung relevant, wie sie nach der Generierung des Testtreibers an den Testdatengenerator weitergereicht wird. Andererseits sind sie erforderlich, um im Testtreiber ein Objekt-Management zu etablieren, das im Laufe der Testausführung instantiierte Objekte aufnimmt und bei Bedarf an aufgerufene Methoden und Konstruktoren entsprechend ihrer Signatur übergibt. Ersteres geschieht dadurch, dass für alle ermittelten primitiven Datentypen einschließlich des Types *String* entsprechend ihrer Multiplizität effektive Testdaten vom Datengenerator erwartet und entgegengenommen werden.

Beginnend bei den zu testenden Klassen in der Arbeitsmenge RIP_h werden jeweils klassenweise alle öffentlichen Methoden und Konstruktoren betrachtet, die in öffentlichen Klassen deklariert sind. Jede untersuchte Klasse wird dabei der Menge RIP_h entnommen und der Liste RI_h hinzugefügt. Da alle Klassen implizit stets auch Unterklassen von *Object* sind und die Klasse *String* ohnehin als „primitiv“ im Sinne des Testdatengenerators betrachtet wird, werden diese speziellen Datentypen explizit nicht der üblichen Instantiierung zugeführt beziehungsweise mit entsprechenden Methodenaufrufen versehen. Handelt es sich bei einem in RIP_h erfassten Datentyp um ein *Interface*, so werden alle seine implementierenden und daher typkompatiblen Unterklassen zu RIP_h hinzugefügt, da entsprechende Instanzen anstelle der mit dem *Interface*-Typ deklarierten formalen Parameter treten können. Gleiches gilt auch für abstrakte Klassen, da diese nicht instantiiert werden können. Darüber hinaus werden alle öffentlichen statischen Methoden abstrakter Klassen einer genaueren Untersuchung zugeführt, nachdem sie jeweils der Liste MI_l angehängt wurden. Im Falle nicht-abstrakter Klassen geschieht das gleich mit allen öffentlichen Methoden, wobei zusätzlich auch alle öffentlichen Konstruktoren einer genaueren Untersuchung zugeführt werden, nachdem sie zur Liste CI_l hinzugefügt wurden.

Jede aufgerufene Funktion, unabhängig davon ob es sich um einen Konstruktor oder eine Methode handelt, hat einen deklarierten Rückgabetyp. Im Falle eines Konstruktors ist dies die zugehörige Klasse, während diejenigen Methoden, die keinen Wert explizit mittels einer *return*-Anweisung zurückgeben, den Rückgabetyp *void* aufweisen. Im Gegensatz zu Konstruktoren und statischen Methoden erfordert der Aufruf einer nicht-statischen Methode ein Zielobjekt, also eine Instanz, die eine solche Methode beherbergt - sei es direkt, indem sie in der entsprechenden Klasse deklariert ist, oder indirekt durch Vererbung. Zusammen mit den Datentypen aller formalen Parametern bilden somit Rückgabetyp und Zielobjekttyp die Basis für die genauere Untersuchung einzelner Konstruktoren und Methoden. Da ein Datentyp in dieser Basis mehrfach vorkommen kann, wird zunächst die Anzahl der Vorkommen gezählt und entsprechend vermerkt – falls der Datentyp bereits betrachtet wurde und daher eine vorläufige Mindestanzahl bekannt ist, wird lediglich das Maximum der bisherigen Mindestanzahl und der für die betrachtete Methode ermittelten Anzahl notwendiger Instanzen vermerkt. Abgesehen von dem Rückgabewert ist von jedem anderen Datentyp entsprechend mindestens eine Instanz zum Aufruf der gerade analysierten Methode erforderlich. Daher werden diese zusätzlichen Datentypen jeweils der Arbeitsmenge RIP_h hinzugefügt und einer vollständigen Analyse unterzogen, es sei denn, sie wurden bereits analysiert, in welchem Falle sie bereits in RI_h erfasst sind.

Bevor die Testtreiber-Klasse nun tatsächlich angelegt wird, muss zunächst noch eine „Aufräumaktion“ durchgeführt werden, welche die gesammelten Informationen in einen konsistenten Zustand versetzt. Dabei werden aus der Liste CI_l alle Konstruktoren wieder entfernt, für deren Aufruf nicht alle Parameter als instantiierbar identifiziert wurden. Entsprechendes gilt bezüglich Parameter und Zieltyp auch für alle Methoden der Liste MI_l. Ein solcher Fall tritt beispielsweise dann auf, wenn beim Modultest noch nicht alle Klassen implementiert sind, so dass der deklarierte Datentyp eines Parameters lediglich ein *Interface* ohne instantiierbare implementierende Unterklassen ist.

In Kapitel 5.1.5 wurde bereits die in •gEAr eingesetzte Datenstruktur dargestellt. Ein Testfall besteht dabei aus einer Anordnung einzelner Datenfelder primitiven Typs (oder vom Typ *String*) und ist in seiner Länge dynamisch, daher wird diese Abfolge im Weiteren auch als Datenstrom bezeichnet. Der Tester beziehungsweise der Testtreibergenerator muss lediglich die Datentypen und den zugelassenen Wertebereich der zu Beginn des Datenstroms stehenden Argumente spezifizieren, wobei der letzte angegebene Datentyp als Muster für alle folgenden Datenfelder eingesetzt wird. Genau diese Art der Schnittstellenbeschreibung zum *System Under Test*, hier indirekt durch den Testtreiber repräsentiert, macht sich der Testtreibergenerator zunutze. Den initialen Teil des Datenstroms interpretiert der Generator als Abfolge primitiver Testdaten zur Übergabe an das Testobjekt, wo dies vom Datentyp möglich ist. Alle weiteren Einträge im Datenstrom werden als Teil des Testszenarios interpretiert und dienen zur Auswahl der jeweils als nächsten durchzuführenden Aktion: Zum Beispiel dem Aufruf einer Methode, der Instantiierung einer Klasse oder der Rotation von Parametern.

Da das Verhalten eines Testobjekts von der Länge der zu behandelnden Arrays abhängen kann, stellt die Länge der im Testtreiber verwalteten Arrays bereits ein Testdatum dar. Da dem Testtreibergenerator Anzahl und Dimensionen der Arrays bekannt ist, kann er für jede Dimension jedes Arrays jeweils eine ganze Zahl am Anfang des Datenstroms reservieren und entsprechend im „*setup*“ als erstes verarbeiten. Aufgrund der vorangehend ermittelten Liste RI_h und

einer vorgegebenen maximalen Array-Größe kann der Generator für jeden benötigten primitiven Datentyp[35] entsprechend viele Positionen im Datenstrom vorsehen. Der Treibergenerator legt zum Objektmanagement im Treiber jeweils typisierte Variablen entsprechend der in RI_h erfassten Multiplizitäten und Array-Dimensionen an. Alle „Objekte“ mit primitivem Datentyp werden zunächst aus dem Datenstrom gelesen und als initiale Werte in den entsprechenden Variablen abgelegt, während später alle Objekte (insbesondere die mit nicht-primitivem Datentyp) entweder dem Aufruf von Konstruktoren entstammen oder Rückgabewerte von Methoden sind. Die Verarbeitung der Szenarien-Informationen aus dem zweiten Teil des Testdatenstroms wird vom Testtreibergenerator durch ein umfassendes `switch/case`-Konstrukt gelöst. Während der Ausführung eines Testfalls wird somit im Rahmen einer Schleife für jedes Datum (vom Typ *int*) im Strom jeweils entschieden, welche Aktion damit verknüpft ist. Zu diesen Aktionen gehören Methoden- und Konstruktoraufrufe ebenso wie die Initialisierung von Arrays und eine sogenannte *Rotationsfunktion*.

Diese „Rotationsfunktion“ dient dem Zweck, die beim Aufruf von Funktionen übergebenen Parameter zu variieren. Falls von einem Datentyp mehrere Instanzen im Testtreiber verwaltet werden (was ebenfalls in einem Array geschieht), weil für einen Methodenaufruf mehrere gleichzeitig benötigt werden, so sorgt diese Rotation dafür, dass die Elemente im Array um eine Position wie in einem Ring durchgerückt werden. Liegt zwischen zwei Aufrufen der gleichen Methode eine Rotation, so wird der zweite Aufruf mit anderen Parametern durchgeführt als der erste – möglicherweise gelangt damit sogar ein Rückgabewert einer Methode zu einem Einsatz als Parameter.

Wenngleich die Klasse aus Listing A.1 eine eigene *main*-Methode hat und daher bereits eine selbstständig ablauffähige Applikation darstellt, so kann sie dennoch als Komponente in einem anderen Kontext betrachtet werden. Wendet man die vorangehend beschriebene automatische Testtreibergenerierung auf dieses Beispiel an, so ergibt sich der •gEAr-spezifische Testtreiber aus Listing A.7. Die vom Treibergenerator an den Testdatengenerator übermittelte Schnittstellenspezifikation sieht folgende Informationen vor:

- Die Anzahl der Argumente muss zwischen 16 und 44 liegen. Dabei stellen die ersten neun den initialen Datenstrom dar. Die Angabe soll erreichen, dass im Mittel jede der sieben Aktionen etwa ein- bis fünfmal pro Testfall initiiert wird.

- Der initiale Datenstrom muss in dieser Reihenfolge enthalten: Einen *long*-Wert aus dem Wertebereich $[0,6]$ für die Array-Längen, sechs *Strings* sowie zwei *long*-Werte mit dem Wertebereich von *int* ($\left[-2^{31}, 2^{31}-1\right]$).

- Das Testszenario basiert auf einem letzten Eintrag vom Typ *long* mit dem Wertebereich $[0,7]$, welcher pro Testfall implizit sieben- bis 35-mal wiederholt wird.

Auf Basis des vorangehend beschriebenen Prototypen für •gEAr entstand im Rahmen eines untergeordneten Forschungsteilprojektes eine erweiterte Variante dieses Testtreibergenerators [Ost06]. Eine Verbesserung besteht darin, dass damit auch die Ausführung von Methoden mit *protected*-Kennzeichnung möglich wird. Dies erreicht der Testtreibergenerator dadurch, dass

[35] Hier: boolean, byte, char, short, int, long, float, double, String

nicht nur eine einzelne Testtreiberklasse generiert wird, sondern für jede Klasse mit einer *protected*-Methode eine zusätzliche Treiberhilfsklasse, welche ins gleiche *Package* wie die angesprochene Klasse abgelegt wird. Da die Testtreibergenerierung einen rekursiven Prozess zur Ermittlung aller notwendigen Instanzen und ihrer Zustandsmodifikatoren einsetzt, kann ein generierter Testtreiber bereits für ein kleines SUT so groß werden, dass der JAVA™-Compiler ihn nicht mehr zu übersetzen vermag. Aus diesem Grund wird das Testszenario nicht mehr in einer einzelnen allumfassenden Treiberklasse aufgelöst, sondern in Abschnitte zerlegt und in den jeweiligen Hilfsklassen für jede Klasse des Testobjektes untergebracht. Ebenfalls in die jeweiligen Hilfsklassen ausgelagert wurde der sogenannte „Instanzenpool“, also das Objektmanagement des Treibers. Die erweiterte Variante des Testtreibergenerators erstellt nicht nur den Testtreiber automatisch, sondern im gleichen Durchlauf auch einen *JUnit*-Testtreibergenerator, weshalb dieser Teil des Werkzeugs als „Testtreibergeneratorgenerator“ bezeichnet wird [Ost06]. Letzterer nimmt lediglich den Testdatenstrom entgegen, sobald dieser nach seiner Generierung und Optimierung feststeht, und generiert einen *JUnit*-Test, dessen Verhalten dem des .gEAr-spezifischen Testtreibers, unter Ausführung mit dem gleichen Testdatenstrom, äquivalent ist.

Kapitel 6

Evaluierung der vorgestellten Methode

„Of all my programming bugs, 80% are syntax errors.
Of the remaining 20%, 80% are trivial logical errors.
Of the remaining 4%, 80% are pointer errors.
And the remaining 0.8% are hard.“
Marc Donner, IBM Watson Research Center

Um die praktische Relevanz der im Rahmen dieser Arbeit entwickelten und beschriebenen Methode zur automatischen Testdatengenerierung und -optimierung zu evaluieren sowie um die Stärken und Erweiterungsmöglichkeiten des Ansatzes auszuloten, wurde das Verfahren implementiert und auf unterschiedliche experimentelle Beispiele angewandt. Das entstandene Werkzeug wurde zunächst für die datenflussorientierte Testdatengenerierung objekt-orientierter JAVA™-Programme konzipiert, weshalb es das Akronym •gEAr (lies *„DOTgEAr“*: Dataflow Oriented Testcase-generation with Evolutionary Algorithms) erhalten und beibehalten hat. Nachdem sich jedoch früh herausgestellt hat, dass auch andere strukturelle wie funktionale Testkriterien ebenfalls von einem solchen Verfahren unterstützt werden, ist das Werkzeug nach und nach gewachsen und umfasst derzeit unter anderem ein Modul für die Verzweigungsüberdeckung (Kapitel 3.2.2) und das Mutationstesten (Kapitel 3.6).

•gEAr ist selbst vollständig in der Sprache JAVA™ programmiert. Einen ersten Eindruck von der vielfältigen Funktionalität des Tools vermittelt eine Reihe von Screenshots in Abbildung C.1. Die Komplexität des umgesetzten Werkzeugs spiegelt Tabelle C.1 wider. Obwohl es noch nicht in der vollständig ausgebauten Version vorliegt, umfasst es derzeit bereits 98998 Programmcode-Zeilen in 806 Klassen – darin nicht eingerechnet sind Hilfsbibliotheken (zum Beispiel *BCEL*) und automatisch generierte Klassen (wie die von *ANTLR* erstellten Lexer und Parser).

Um nebst theoretischer Betrachtungen und Bewertungen des Ansatzes auch experimentelle Daten in die Evaluierung einbeziehen zu können, wurde •gEAr mit der automatischen Testdatengenerierung für unterschiedliche Beispiel-Projekte betraut. Eine Übersicht dieser Beispiele zeigt Tabelle 6.1. Die darin aufgeführten Testobjekte reichen von 82 Codezeilen in einer Klasse bis hin zu 5439 Zeilen in 27 Klassen. Während die Projekte mit den Kennungen (ID) 1 bis 4 eigens entwickelte, vollständig ablauffähige Applikationen darstellen, wurden die Testobjekte 5 und 6

ID	Projekt	AK	ASZ	GSB	AGM
1	BigFloat	3	540	17526	1528
2	Dijkstra	2	141	4080	220
3	Hanoi	1	38	1279	227
4	Huffman	2	298	8931	623
5	JDK sort	1	82	2639	852
6	JDK logging	27	5439	113046	1970

AK: Anzahl der Klassen, aus denen das Projekt besteht
ASZ: Anzahl der Source-Code-Zeilen des Projektes (LOC)
GSB: Größe des Projekt-Source-Codes in Byte
AGM: Anzahl generierter Mutanten (einschließlich evtl. äquivalenter)

Tabelle 6.1: Vorstellung der experimentell untersuchten Beispiel-Projekte

ohne Modifikationen aus dem Quelltext der API[1]-Standardbibliothek von JAVA™ extrahiert und mit einem •gEAr-spezifischen Testtreiber versehen.

Das Projekt *BigFloat* stellt eine Komponente dar, welche eine ähnliche Funktionalität bietet wie die Klasse `BigDecimal` des JAVA™-eigenen `java.math`-Packages. Intern verwaltet die Hauptklasse Gleitkommazahlen beliebiger Genauigkeit mittels doppelt verketteter Listen, deren Knoten je eine Stelle der „Zahl“ repräsentieren – die letzteren beiden Datenstrukturen sind in eigenen Klassen umgesetzt und ergänzen somit die Hauptklasse. Das Codebeispiel *Dijkstra* ist eine Umsetzung des Algorithmus nach Dijkstra zur Ermittlung kürzester Pfade zwischen einem Startknoten und allen anderen Knoten in einem Graphen. Neben der Hauptklasse enthält diese Komponente eine Hilfsklasse, deren Instanzen jeweils Knoten des Graphen darstellen. Das verhältnismäßig kleine Beispiel *Hanoi* ist eine Implementierung des Problems der „Türme von Hanoi“ und enthält ein komplexes Prädikat mit sieben atomaren Teilbedingungen, was für die vollständige Datenflussüberdeckung nach den Kriterien *all-p-uses* oder *all-uses* eine Herausforderung darstellt. Die Funktionalität der Komponente *Huffman* besteht in der Kompression beliebiger Zeichenketten nach dem Huffman-Algorithmus. Der dazu notwendige Baum wird mit Instanzen einer Hilfsklasse aufgebaut und verwaltet. Das Projekt *JDK sort* enthält einen optimierten Sortieralgorithmus für ganze Zahlen, welcher in Teilen auf dem rekursiven Quicksort-Verfahren nach Hoare basiert. Die Komponente *JDK logging* ist identisch mit dem Package `java.util.logging` der JAVA™-API.

Die mit „AGM“ überschriebene Spalte der Tabelle 6.1 enthält die Anzahl der Mutanten, welche aus dem Quellcode des jeweiligen Testobjektes hervorgegangen sind. Diese Mutanten wurden automatisch mit Hilfe des Werkzeugs *µJava*[2] [OMK04, MOK05] so erstellt, dass sie sich jeweils durch einen einzigen „Fehler“ von der ursprünglichen Version unterscheiden. Dabei wurden sowohl klassische als auch objekt-orientierte Mutationsoperatoren angewendet. Entsteht während der Erstellung der Mutanten ein veränderter Quelltext, welcher syntaktisch fehlerhaft ist und sich daher nicht übersetzen lässt, so wird dieser von *µJava* wieder verworfen und ist

[1] API: *Application Programming Interface*, JDK: *Java Development Kit*
[2] `http://ise.gmu.edu/~ofut/mujava/`, George Mason University, Virginia, USA

daher nicht in der Tabelle 6.1 erfasst. Wichtig in diesem Zusammenhang ist, dass *µJava* nicht zu prüfen vermag, ob ein ausführbarer Mutant funktional äquivalent zum ursprünglichen Testobjekt ist, weshalb solche Mutanten sehr wohl in der Spalte „AGM“ mitgezählt werden.

6.1 Vergleich verschiedener Optimierungsalgorithmen

Wie bereits im Kapitel 5.1.6 angedeutet, umfasst die prototypische Implementierung von •gEAr derzeit vier unterschiedliche Metaheuristiken, wobei jederzeit weitere Optimierungsalgorithmen leicht hinzugefügt werden können. Es handelt sich dabei um die folgenden:

- Random: *Random Search* (siehe Kapitel 4.2 ab Seite 85),
- SimAnn: *Simulated Annealing* (siehe Kapitel 4.4 ab Seite 87),
- MOA: *Multi-objektive Aggregation* (siehe Kapitel 4.5.4 ab Seite 112) sowie
- NSGA: *Nondominated Sorting Genetic Algorithm* (siehe Kapitel 4.5.4 ab Seite 114).

Zweck dieser redundanten Umsetzung ist es, die grundverschiedenen Verfahren im Hinblick auf ihre Eignung für die automatische Testdatengenerierung zu vergleichen. Dazu wurden diese jeweils einzeln in den verschiedenen Projekten aus Tabelle 6.1 angewandt, wobei sich bezüglich der Anzahl der jeweils überdeckten Entitäten im Wesentlichen stets das gleiche Verhalten offenbart hat, welches exemplarisch in Abbildung 6.1[3] für das Code-Beispiel *BigFloat* sowie in Abbildung 6.2 für das größte Testobjekt *JDK logging* dargestellt ist. Die entsprechenden Diagramme aller anderen Projekte sind der Abbildung B.1 (im Anhang ab Seite 253) zu entnehmen. Bei den Algorithmen, welche auf Basis einer gewichteten Summe optimieren, wurde eine Gewichtung von 0,05:1 für die Anzahl der Testfälle pro Testdatensatz gegenüber der Anzahl der überdeckten *def/use*-Paare gewählt, womit der Selektionsdruck stark zugunsten der Überdeckung konfiguriert wurde. Begründet wird diese Wahl dadurch, dass für sicherheitskritische Applikationen eine möglichst hohe Überdeckung erzielt werden soll, während der Testumfang sekundär ist. Die weiteren, hierbei verwendeten Parameter der Metaheuristiken sind in Tabelle B.1 aufgeführt.

Für den praktischen Einsatz ist zweifelsohne auch die Performanz des Verfahrens von großer Bedeutung. Daher bietet es sich an, den Ressourcen-Aufwand und den Nutzen im Sinne der erzielten Überdeckung für die verschiedenen Verfahren einander gegenüber zu stellen. Da die Generierung und Optimierung voll-automatisiert ausgeführt wird, ist kein weiterer Eingriff durch einen Tester erforderlich, demnach sind nur noch Rechenzeit und Speicherauslastung von Interesse. Dazu sei jedoch angemerkt, dass die Umsetzung des Verfahrens in •gEAr nicht zum Zwecke einer kommerziellen Nutzung erfolgte und daher nicht im Hinblick auf Ressourcenverbrauch optimiert wurde. Zwar hängt die Speichernutzung entscheidend vom Umfang und der Komplexität des Testobjektes ab, dennoch blieb sie mit maximal 200 Megabyte über alle Beispiele hinweg weit unterhalb des Arbeitsspeichers üblicher Arbeitsplatzrechner – und dies obwohl •gEAr selbst in JAVA™ implementiert und damit erwartungsgemäß speicherintensiver ist als vergleichbare Implementierungen in maschinennäheren Sprachen wie zum Beispiel C.

[3] Diese Abbildung wurde in [OS06] publiziert und kurz erläutert.

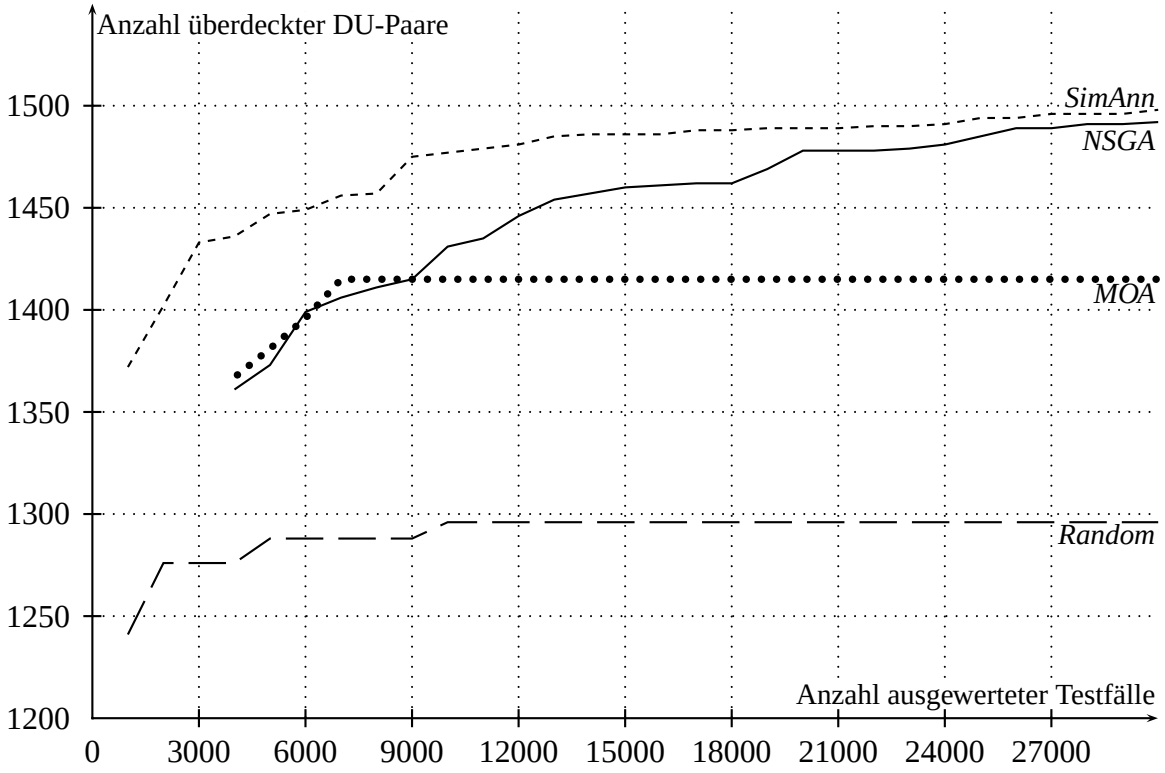


Abbildung 6.1: Vergleich der Optimierungsverfahren (Projekt *BigFloat*): Überdeckung

Eine Aussage bezüglich absoluter Rechenzeit gestaltet sich, nicht nur aufgrund der ohnehin langsameren Interpretation des Werkzeugs durch die Virtuelle Maschine von JAVA™ gegenüber einer nativ auf dem Prozessor ausgeführten Umsetzung in C, sogar noch ungleich schwieriger, da .gEAr gerade zum Zwecke einer zügigen Bewertung einzelner Testfälle mit einer verteilten Ausführung von Tests konzipiert wurde, wie dies in Abbildung 5.2 dargestellt (Seite 153) und in Kapitel 5.1.4 beschrieben ist. Die verschiedenen Experimente wurden mit 10 bis 45 dieser sogenannten „Compute Server“ ausgeführt. Ein Teil dieser Server waren dabei dedizierte Rechner eines Linux-Clusters, die meisten hingegen waren übliche Arbeitsplatzrechner. Letztere sollten während der Ausführung der Projekt-Experimente von ihren jeweiligen Benutzern ohne Einschränkungen bedient werden können, das heißt, dass der Testausführung in Abhängigkeit von der Auslastung und Verfügbarkeit der Rechnerstationen unterschiedlich viel Rechenzeit zugeteilt wurde. Daher ist eine Aussage aufgrund der tatsächlichen Laufzeit der Testdatengenerierung und -optimierung nur beschränkt aussagekräftig. Darüber hinaus weisen die verschiedenen Algorithmen über die Zeit hinweg unterschiedliche Lastprofile auf. Evolutionäre Verfahren müssen beim Start der Optimierung eine große Population auswerten und erzeugen daher zu Beginn eine hohe Last, während später, zum Beispiel aufgrund vereinzelter Mutationen, nur noch wenige Testfälle auszuführen sind. Im Gegenzug beginnt *Random Search* mit einer einzelnen Testfallmenge, generiert dafür in jedem weiteren Schritt wieder eine komplette, neue Testfallmenge, welche jeweils vollständig auszuführen ist. Aus oben genannten Gründen zeigen Abbildung 6.1, Abbildung 6.2, Abbildung 6.4 und Abbildung 6.5 auf der x-Achse nicht die reale Kalenderzeit sondern

ein objektiveres Äquivalent: Die Anzahl der zur Erreichung einer entsprechenden Überdeckung jeweils ausgeführten Testfälle insgesamt.

Erwartungsgemäß war *Random Search* in allen Experimenten den anderen Algorithmen weit unterlegen. Bezüglich Projekt *BigFloat* (Abbildung 6.1) konnte dieses Verfahren nach circa 9000 insgesamt ausgeführten Testfällen einen Testdatensatz identifizieren, welcher mit 1296 überdeckten *def/use*-Paaren der bis dahin beste war, dazu jedoch 97 verschiedene Testfälle umfasste. Eine weitere Verbesserung konnte *Random Search* erst nach etwa 50000 evaluierten Testfällen erzielen, wobei die dann generierte Testfallmenge mit 97 Testfällen für 1305 DU-Paaren nicht entscheidend erfolgreicher war. Ein vergleichbares Verhalten zeigte *Random Search* auch für das Projekt *JDK logging* (Abbildung 6.2). Dabei erreichte diese Heuristik nach insgesamt etwa 2700 evaluierten Testfällen lediglich eine Überdeckung von 850 *def/use*-Paaren. Selbst nach 4100 Schritten und 164760 ausgeführten Testfällen konnte dieses Verfahren nicht mehr als 877 Datenflusspaare überdecken.

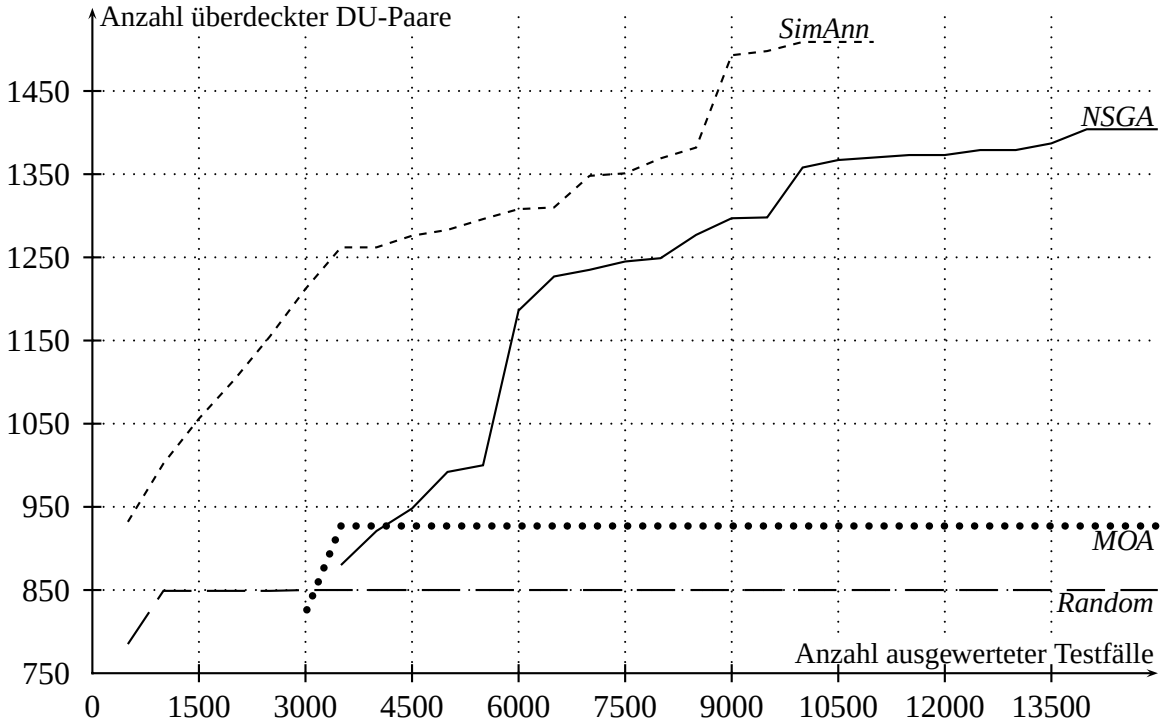


Abbildung 6.2: Vergleich der Optimierungsverfahren (Projekt *JDK logging*): Überdeckung

Deutlich besser abgeschnitten hat in allen Projekten das Evolutionäre Verfahren *Multi-objektive Aggregation* (MOA). Bei einer Populationsgröße von 70 Individuen und einer Vorgabe von 1 bis 100 Testfällen pro Testdatensatz im Projekt *BigFloat* mussten bereits für die erste Generation 3449 Testfälle ausgewertet werden, weshalb die entsprechende Kurve in Abbildung 6.1 auch erst da beginnt. Nach etwa 769 Generationen und ca. 6500 insgesamt ausgewerteten Testfällen „fand“ diese Metaheuristik eine Testfallmenge, welche mit 47 Testfällen 1415 *def/use*-Paare zu überdecken vermochte, womit sich ein „Sättigungseffekt“ eingestellt hat, bei dem über längere Zeit hinweg keine weitere Verbesserung erreicht werden konnte. Eine bessere Testfallmenge

konnte MOA erst in Generation 8778 nach 48841 evaluierten Testfällen finden, welche mit einer Größe von 40 eine DU-Paar-Überdeckung von 1410 schaffte. Von Generation 23616 (130261 Testfälle insgesamt) bis zum Abbruch der Optimierung nach 42689 Generationen und 245391 ausgeführten Testfällen verblieb MOA bei einer Testfallmenge mit 39 Elementen, welche 1439 *def/use*-Paare überdecken. Bezüglich Projekt *JDK logging* (Abbildung 6.2) stellte sich der Sättigungseffekt in Generation 153 nach 3802 insgesamt ausgeführten Testfällen ein, wobei die bis dahin beste Testfallmenge 46 Testfälle umfasst und 927 *def/use*-Paare überdeckt. Erst in Generation 19675 und damit nach 119463 betrachteten Testfällen erreichte ein weiterer Testdatensatz mit 59 Testfällen eine Überdeckung von 1015 DU-Paaren.

Mit Abstand die besten Ergebnisse in den betrachteten Projekten erzielte *Simulated Annealing*. Im Projekt *BigFloat* konnte diese Heuristik mit 17 Testfällen zusammen 1511 *def/use*-Paare überdecken, wozu insgesamt 67300 Testfälle ausgeführt werden mussten. Bereits nach 29685 evaluierten Testfällen insgesamt, betrug die Überdeckung schon 1498 DU-Paare bei lediglich 20 Testfällen pro Testdatensatz. Ähnlich erfolgreich war dieses Verfahren auch im Projekt *JDK logging*, bei welchem sich eine Sättigung nach 9814 Testfällen eingestellt hat, zu einem Zeitpunkt, bei dem mit 40 Testfällen 1509 DU-Paare überdeckt werden konnten. Eine bessere Testfallmenge konnte auch nach Auswertung von 1300 weiteren Testfällen nicht erreicht werden.

Eine besondere Stellung im Kreise der vier betrachteten Heuristiken nimmt NSGA (*Non-dominated Sorting Genetic Algorithm*) ein. Wenngleich NSGA nach Abbildung 6.1 und Abbildung 6.2 scheinbar *Simulated Annealing* geringfügig unterlegen war, so unterscheidet sich diese Variante des Genetischen Algorithmus grundsätzlich von den anderen drei Heuristiken in der Art der Fitnessbewertung und insbesondere in der Darstellung der Ergebnisse. Wie in Kapitel 5.1.6 angedeutet, sind *Random Search*, *Simulated Annealing* und *Multi-objektive Aggregation* insofern pseudo-multi-objektiv, als dass sie eine Testfallmenge lediglich aufgrund einer aggregierten Fitness bewerten, während NSGA beide Objektiven gleichwertig und gleichzeitig berücksichtigt. Aus diesem Grund besteht das Ergebnis einer NSGA-Ausführung aus der gesamten sogenannten *Pareto-Front*[4]. Ein Beispiel einer solchen Front ist für das Projekt *BigFloat* in Abbildung 6.3 gegeben. Jeder Punkt in diesem Diagramm stellt eine Testfallmenge dar. Die im Sinne der Dominanzrelation optimalen (nicht-dominierten) Lösungen sind durch Geradenabschnitte verbunden und jeweils mit der Anzahl der Testfälle und der damit erreichten Überdeckung von *def/use*-Paaren beschriftet.

Während das „Ergebnis“ einer NSGA-Optimierung also eine vollständige Pareto-Front darstellt, besteht die Lösung des Such- und Optimierungsproblems bei den anderen Verfahren lediglich in einer einzigen Testfallmenge. Bei vergleichbarem Optimierungsfortschritt bestünde das Ergebnis der pseudo-multi-objektiven Heuristiken lediglich in einem der Elemente aus der Pareto-Front des NSGA. Aus diesem Grund ist der Vergleich zwischen den Metaheuristiken in Abbildung 6.1 und Abbildung 6.2 nur bedingt aussagekräftig, da für *Random*, *SimAnn* und *MOA* jedem Punkt der Entwicklungskurven nur ein „optimaler“ Testdatensatz zugrunde liegt, während die mit „NSGA“ beschriftete Kurve auf Basis desjenigen Individuums der Pareto-Front entstanden ist, welches in der entsprechenden Generation die größte Überdeckung erzielen konnte. Eine gewisse Vergleichbarkeit ist dadurch gegeben, dass die Gewichtung bei den pseudo-multi-

[4]Die in [Ost05, OS06] kurz beschriebene Pareto-Front wird hier ausführlich behandelt.

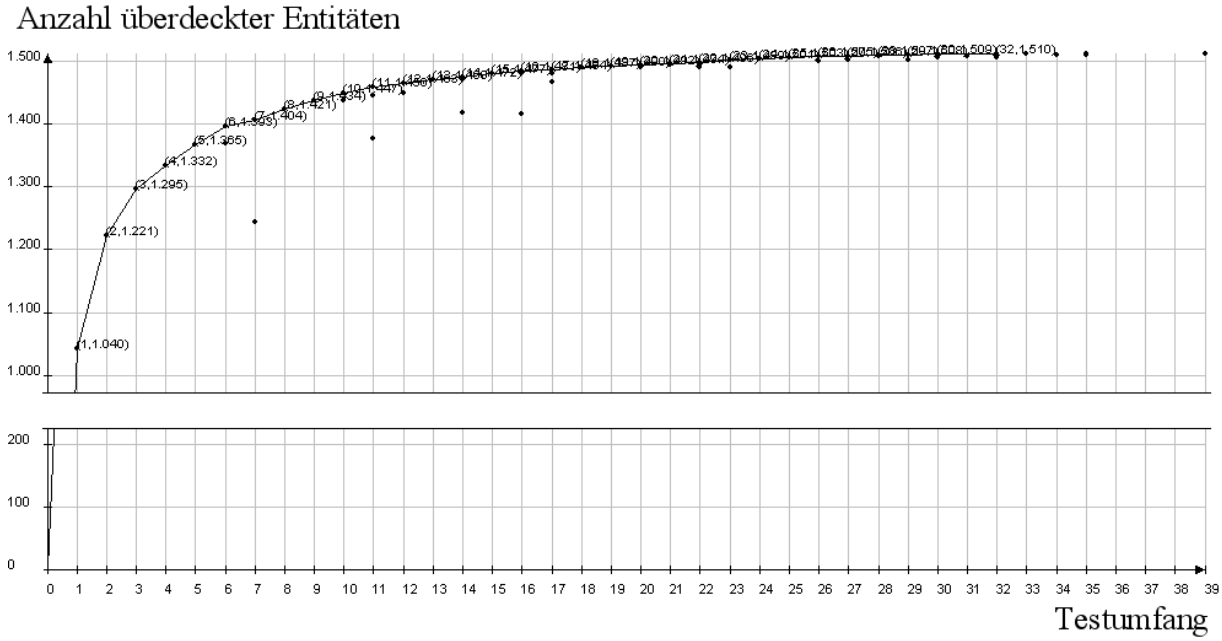


Abbildung 6.3: Pareto-Front (Projekt *BigFloat*) [nach Screenshot *.gEAr*]

objektiven Verfahren mit 0,05:1 für die Anzahl der Testfälle pro Testdatensatz gegenüber der Anzahl der überdeckten *def/use*-Paare ebenfalls zugunsten einer möglichst hohen Überdeckung gewählt wurde.

Bei der Anwendung von NSGA auf das Projekt *BigFloat* (Abbildung 6.1) konnte nach 14898 Generationen und damit 29699 insgesamt ausgeführten und bewerteten Testfällen ein Testdatensatz identifiziert werden, welcher mit 45 Testfällen zusammen 1492 DU-Paare zu überdecken vermag und damit *SimAnn* nur unwesentlich unterlegen ist. Eine Sättigung stellte sich in der Generation 49164 (entspricht 93221 ausgewertete Testfälle), welche bis zur Generation 55000 (105355) anhielt, und deren bestes Ergebnis eine Überdeckung von 1510 *def/use*-Paaren mit 32 Testfällen war – die zugehörige Pareto-Front ist in Abbildung 6.3 gegeben. Etwas ungünstiger fiel das Ergebnis im Projekt *JDK logging* aus, wo NSGA nach der 11839. Generation und demnach der Auswertung von 26222 Testfällen einen Testdatensatz mit 56 Testfällen identifiziert hat, welcher 1449 DU-Paare zu überdecken vermochte. Dieses sub-optimale Ergebnis konnte auch nach 12386 Generationen und insgesamt 27158 ausgewerteten Testfällen nicht weiter verbessert werden.

Zwar erreicht NSGA eine vergleichbare Überdeckung wie SimAnn lediglich mit einem höheren Aufwand an Rechenressourcen, doch dafür bieten die von NSGA ermittelten Ergebnisse dem Tester eine ungleich allgemeinere Sicht auf die Komplexität der Kontroll- oder Datenflussstruktur des Testobjekts sowie eine größere Flexibilität bei der Wahl eines optimalen Testdatensatzes. Dadurch dass der Tester für Random, SimAnn und MOA die Gewichtung der Objektiven (Anzahl der Testfälle pro Testdatensatz sowie Anzahl der überdeckten Entitäten) bereits a priori

vorgeben muss, legt er sich zwangsweise bereits *vor* der Ausführung des Algorithmus bezüglich des Kompromisses zwischen Verifikationsaufwand (infolge des Testumfangs) und der Fehleraufdeckungsquote (soweit diese mit der Überdeckung korreliert) fest. Anders bei Anwendung des NSGA: Erst wenn die Pareto-Front ermittelt wurde, kann der Tester die hinsichtlich des erwähnten Kompromisses ideale Testfallmenge individuell auswählen. Angesichts der Verteilung der Testfallmengen entlang der Front in Abbildung 6.3 ist leicht nachzuvollziehen, dass bei kleineren Testdatensätzen bereits jeder weitere Testfall eine erhebliche Steigerung der Überdeckung bieten kann. Bei größeren Mengen hingegen ist der Zugewinn an Überdeckung (und damit im Allgemeinen auch an Fehleraufdeckung) mit jedem weiteren Testfall weitaus geringer, während der Verifikationsaufwand im Allgemeinen für jeden Testfall als gleich groß zu erwarten ist.

Im Beispiel-Projekt *BigFloat* erzielt bereits ein einziger, hochgradig optimierter Testfall eine Überdeckung von 1040 *def/use*-Paaren. Mit lediglich neun weiteren Testfällen kann bereits eine Überdeckung von 1447 (also 407 zusätzlichen) DU-Paaren erzielt werden. Um jedoch lediglich weitere 63 DU-Paare zu testen, müssten 22 zusätzliche Testfälle (manuell) überprüft werden. Ob sich der zusätzliche Aufwand lohnt, ist sicherlich projekt-spezifisch und kann nicht allgemein beurteilt werden. Unter der Annahme, dass jede im Test nicht überdeckte Entität mit gleicher Wahrscheinlichkeit zu einem gleich-schweren (also für den Software-Hersteller „gleich-teuren“) Versagen führt und dass die Überprüfung jedes einzelnen Testfalls gleich viel Aufwand erfordert, kann die Auswahl eines optimalen Testdatensatzes aus der Pareto-Front durch folgende Betrachtungen unterstützt werden: Sei k_{TF} der (zum Beispiel finanzielle) Aufwand zur Verifikation eines Testfalls, k_E die durch eine nicht-überdeckte Entität verursachten Kosten (in gleichen Einheiten wie k_{TF}) im Betrieb, a_E^{max} die maximale Anzahl der überdeckten Entitäten laut Pareto-Front, $a_{TF}(i)$ die Anzahl der Testfälle im Testdatensatz i der Front und $a_E(i)$ die vom Testdatensatz i überdeckten Entitäten, so ist diejenige Testfallmenge optimal, für die die Kostenfunktion

$$K(i) = k_{TF} \cdot a_{TF}(i) + k_E \cdot (a_E^{max} - a_E(i))$$

minimal ist.

Bei den in Abbildung 6.1 und Abbildung 6.2 dargestellten Kurven handelt es sich um die Entwicklung der absoluten Überdeckung im Sinne der Anzahl relevanter Entitäten entsprechend dem gewählten Testkriterium *all-uses*. Da die aggregierenden Heuristiken mit einer Gewichtung von 0,05:1 zugunsten der Überdeckung ausgeführt wurden, sind diese Diagramme auch von größerer Bedeutung. Dennoch lohnt hier auch ein Blick auf die weiteren Aspekte der Fitnessbewertung einer Testfallmenge. Dazu gehört zunächst der Testumfang, ausgedrückt durch die Anzahl der Testfälle des jeweils bis dahin als optimal identifizierten Testdatensatzes, dessen Entwicklung während der gleichen Ausführung von •gEAr für das Projekt *BigFloat* in Abbildung 6.4 dargestellt ist, auf deren Auswertung auch Abbildung 6.1 basiert. Dabei fällt auf, dass die Kurven (insbesondere für die Heuristik NSGA, weniger ausgeprägt auch für SimAnn) nicht monoton fallen, wie zunächst zu erwarten wäre. Ursache dafür ist, dass ein Zugewinn an Überdeckung vorübergehend auch eine Verschlechterung des Testumfangs zur Folge haben kann, die jedoch aufgrund der gezielt einseitigen Gewichtung zunächst in der jeweiligen Population akzeptiert wird. Man beachte hierbei jedoch, dass die Kurve für die Heuristik NSGA lediglich aufgrund des jeweiligen Individuums mit der größten Überdeckung entlang der Pareto-Front be-

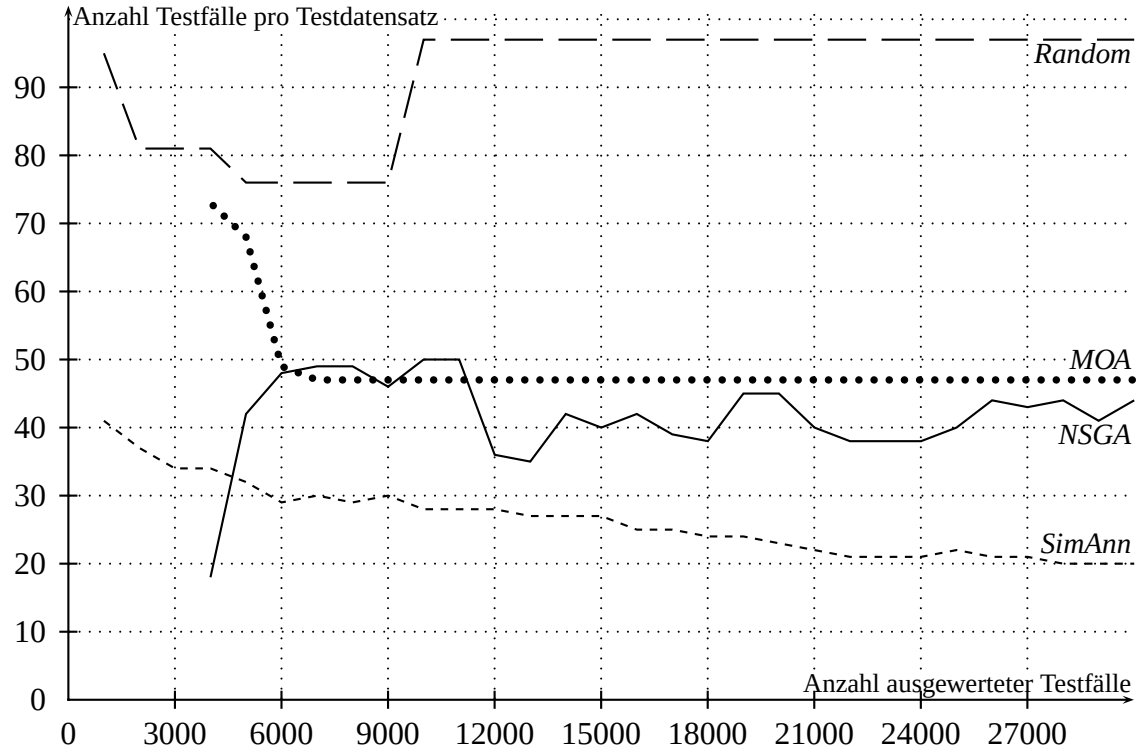


Abbildung 6.4: Vergleich der Optimierungsverfahren (Projekt *BigFloat*): Testumfang

rechnet wurde, welches erwartungsgemäß eine größere Testfallmenge umfaßt als beispielsweise der nach SimAnn ermittelte Testdatensatz.

Dass trotz zwischenzeitlicher Zunahme des Testumfangs eine monotone Verbesserung der Fitness der jeweils besten Testfallmenge über die Zeit erzielt wird, zeigt der Verlauf der aggregierten Fitness in Abbildung 6.5. Der Fitnesswert des durch Abbildung 6.1 und Abbildung 6.2 charakterisierten, jeweils als bis dahin optimal eingestuften Testdatensatzes ermittelt sich entsprechend dem Verfahren der Aggregation mit gewichteten Beiträgen aus Kapitel 5.1.6 zu:

$$f := \left(\frac{c - c_{min}}{c_{max} - c_{min}}\right) + 0{,}05 \cdot \left(1 - \frac{s - s_{min}}{s_{max} - s_{min}}\right),$$

wobei $c_{min} = 1241$ und $c_{max} = 1498$ die im Laufe des betrachteten Ausschnitts der gesamten Ausführung von •gEAr erzielte minimale beziehungsweise maximale Überdeckung darstellen, während $s_{min} = 18$ und $s_{max} = 97$ entsprechend den geringsten beziehungsweise größten Testumfang repräsentieren.

Ein Vergleich der verschiedenen Heuristiken aufgrund einer absoluten Zeitangabe ist aus verschiedenen Gründen nicht aussagekräftig, wie eingangs zu Kapitel 6.1 angedeutet. Dennoch sei hier der Vollständigkeit wegen kurz skizziert, welche diesbezüglichen Informationen zu den experimentellen Einsätzen von •gEAr gesammelt wurden. Zum Zwecke der wissenschaftlichen Untersuchungen wurde •gEAr mit jedem Beispiel-Projekt in einer Langzeitausführung beobachtet, um den Heuristiken ausreichend Zeit zum Erreichen einer möglichst umfassende Optimierung zu gewähren, also die Konvergenz nicht frühzeitig zu verhindern. Die Ergebnisse dieser

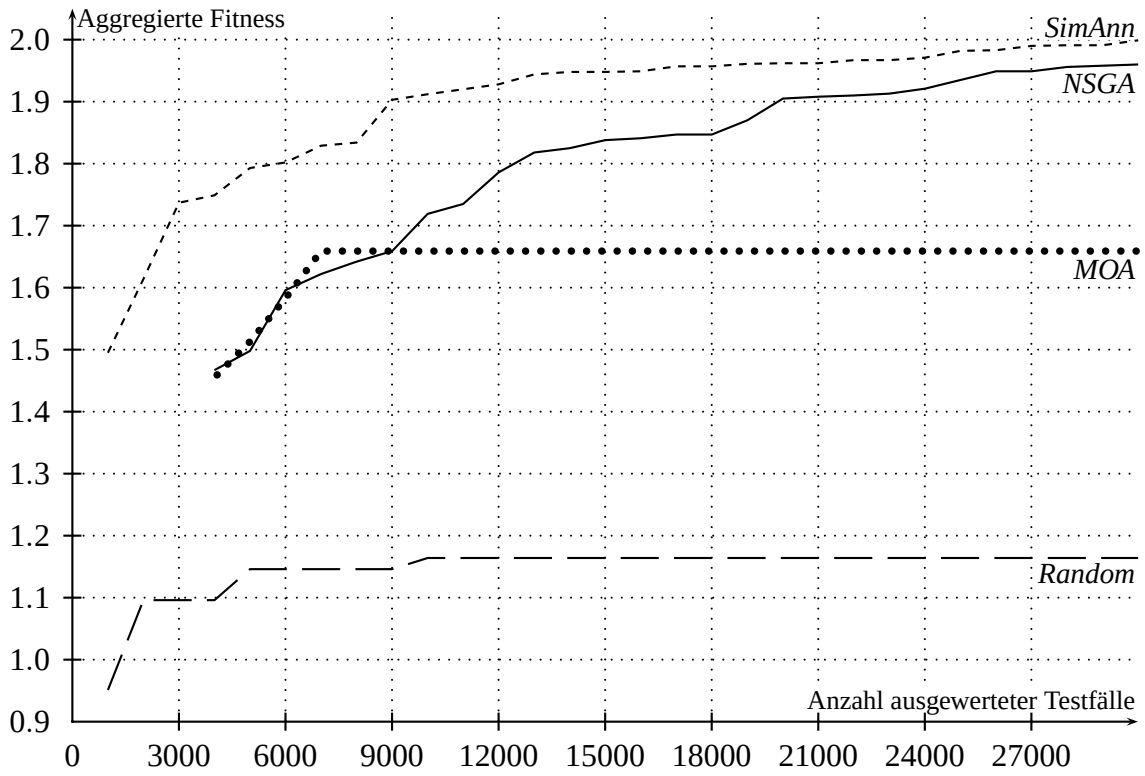


Abbildung 6.5: Vergleich der Optimierungsverfahren (Projekt *BigFloat*): Fitness

Betrachtungen sind in Tabelle 6.2 zusammengefasst. Dabei enthält die Spalte EÜ die Anzahl der *def/use*-Paare, welche von dem am Ende der Langzeitausführung als optimal identifizierten Testdatensatz überdeckt wurden. In der zweiten Spalte GT steht die Größe dieses Testdatensatz im Sinne der Anzahl der umfassenden Testfälle. Die darauf folgende Spalte ATK spiegelt die Anzahl der notwendigerweise ausgeführten Testfälle bis zur Ermittlung des jeweils optimalen Testfalls. Im Gegensatz dazu gibt die Spalte AGT wieder, wie viele Testfälle insgesamt im Laufe des Langzeitexperiments ausgewertet wurden. Dabei ist anzumerken, dass trotz Ausführung der jeweiligen Anzahl der Testfälle, welche über ATK hinausgeht und bis AGT reicht, keine weitere Verbesserung in der Fitness erreicht werden konnte. Die Werte in der Spalte REE stellen die Anzahlen der jeweils maximal vorgegebenen Ausführungsrechner dar, auf denen die Testfälle parallel ausgeführt werden konnten. Dabei ist zu beachten, dass einige dieser Rechner nicht durchgängig oder nicht mit voller Leistung beigetragen haben. Um den Betrieb nicht zu beeinträchtigen, wurden die Rechner außerhalb der üblichen Bürozeiten zum Testen eingesetzt, beziehungsweise tagsüber nur in einem Maße, das eine störungsfreie Bedienung ermöglichte. Die vorletzte Spalte LZM gibt jeweils die Dauer des gesamten Experiments in Minuten an. Schließlich repräsentieren die Angaben in der letzten Spalte DA eine durchschnittliche Auslastung der REEs, ausgedrückt in abgearbeiteten Testfällen pro Rechner und Minute.

Betrachtet man exemplarisch das Projekt *Dijkstra*, so konnte die Metaheuristik *Simulated Annealing* (SimAnn) insgesamt 168 *def/use*-Paare überdecken. Dazu war die Ausführung von 4101 Testfällen notwendig. Im Langzeitexperiment wurde der Heuristik die Auswertung von

Projekt	Metaheuristik	EÜ	GT	ATK	AGT	REE	LZM	DA
BigFloat	NSGA	1510	32	105054	105356	11	2427	3,95
	MOA	1439	39	130261	245391	11	3906	5,71
	SimAnn	1511	17	67300	75333	11	887	7,72
	Random	1333	85	630051	1522955	35	2426	17,94
Dijkstra	NSGA	168	2	1853	17378	58	963	0,31
	MOA	168	2	1921	190120	26	1042	7,02
	SimAnn	168	2	4101	125457	43	1056	2,76
	Random	168	2	112351	581901	58	947	10,59
Hanoi	NSGA	42	2	382	572	43	2	6,65
	MOA	42	2	321	2413	43	14	4,01
	SimAnn	42	2	92	180086	26	1052	6,58
	Random	42	2	6158	374802	43	1031	8,45
Huffman	NSGA	353	3	2929	213854	11	2483	7,83
	MOA	353	3	2322	73621	46	1313	1,22
	SimAnn	353	3	242	44974	28	881	1,82
	Random	353	13	117175	356186	36	936	10,57
JDK sort	NSGA	315	2	5856	11457	26	42	10,49
	MOA	314	2	17115	537112	26	1497	13,80
	SimAnn	315	2	1832	318792	26	2370	5,17
	Random	315	7	594597	3200551	26	4032	30,53
JDK logging	NSGA	1449	56	26222	27158	29	3953	0,24
	MOA	1015	59	119463	129842	29	4030	1,11
	SimAnn	1509	40	9814	11115	47	4010	0,06
	Random	877	75	115135	164760	29	4407	1,29

EÜ: am Ende der Langzeitausführung überdeckte DU-Paare
GT: Anzahl der Testfälle des jeweils optimalen Testdatensatzes
ATK: Anzahl ausgeführter Testfälle bis zum Erreichen von EÜ
AGT: Anzahl insgesamt ausgeführter Testfälle
REE: maximale Anzahl parallel genutzter *Remote Execution Engines*
LZM: Dauer der Laufzeit in Minuten
DA: durchschnittlicher Durchsatz (abgearbeitete Testfälle pro REE und Minute)

Tabelle 6.2: Gesamtübersicht der Langzeitausführung von *.gEAr*

insgesamt 125457 Testfällen erlaubt. Dazu wurden dem Verfahren (maximal) 43 Rechner zur Verfügung gestellt. Zur Auswertung dieser 125457 Testfälle auf bis zu 43 Rechnern erforderte •gEAr eine Gesamtzeit von 1056 Minuten (ca. 17,6 Stunden), während der im Durchschnitt 2,76 Testfälle pro Rechner und Minute ausgeführt wurden. Extrapoliert man die tatsächlich notwendige Laufzeit zum Erreichen des Endergebnisses mit einer Überdeckung von 168 DU-Paaren, waren dazu lediglich ungefähr 35 Minuten notwendig, obwohl der Algorithmus von Dijkstra eine beachtliche Verschachtelungstiefe von Schleifen und Verzweigungen aufweist.

Wendet man die gleiche Betrachtung auf das Projekt *Hanoi* an, so konnte das Kriterium *all-uses* mittels *Simulated Annealing* (SimAnn) bereits nach etwa einer halben Minute erreicht werden, während *Nondominated Sorting Genetic Algorithm* (NSGA) nur wenig mehr als eine Minute beanspruchte. Trotz seiner geringen Größe hat das Projekt *Hanoi* eine Besonderheit: Bevor die rekursive Berechnung der Schritte angestoßen wird, werden sämtliche Eingabeparameter auf Einhaltung einer gültigen Vorbedingung geprüft. Dazu enthält die Startmethode eine Verzweigung, deren Prädikat aus sieben primitiven Teilbedingungen besteht, welche jeweils durch Disjunktionen miteinander verknüpft werden. Der letzte Operand der Disjunktionskette („`sourcePile == destinationPile`“) macht es erforderlich, dass zur Erfüllung des *all-uses*-Kriteriums zwei spezielle Eingabedatenkombinationen getestet werden, unter welchen alle Teilprädikate bis auf die letzte den Wahrheitswert *falsch* annehmen, während letztere jeweils zu *wahr* beziehungsweise *falsch* ausgewertet wird, da nur so die prädikative Verwendung der beiden Variablen (hier `sourcePile` und `destinationPile`) als vollständig getestet gilt.

6.2 Parametrisierung der Evolutionären Verfahren

Jede der vier in •gEAr implementierten Metaheuristiken verfügt über eine Vielzahl von Freiheitsgraden und kann demnach auch im umgesetzten Werkzeug entsprechend frei parametrisiert werden. Allen Verfahren gemeinsam sind freilich die projekt-spezifischen Parameter, wie zum Beispiel minimale beziehungsweise maximale Anzahl von Testfällen pro Testfallmenge und die Schnittstellenbeschreibung zum Testobjekt, welche im Falle der Verwendung von Testtreibern auch die Komplexität eines Testfalls im Sinne der Anzahl von Ereignissen im Testszenario beeinflusst. Ebenfalls unabhängig von der Heuristik sind die für die Teststrategie spezifischen Einstellungen, allen voran das Testkriterium sowie Optionen im Zusammenhang mit dem Kriterium, darunter beispielsweise die Interpretation der prädikativen Verwendungen (siehe „*deep branch tracing*“ in Kapitel 5.1.4) im Falle von datenflussbasiertem Testen.

Den pseudo-multi-objektiven Heuristiken *Random*, *SimAnn* und *MOA* gemeinsam sind die notwendigerweise vorzugebenden Gewichtungen für die jeweiligen Teilaspekte der Bewertungsfunktion, sowie die Option, ob ein *Windowing* eingesetzt werden soll – eine Option, die für *NSGA* nicht von Belang ist. Speziell *SimAnn* hat die beiden zusätzlichen Freiheitsgrade „initiale Temperatur“ und „Temperaturabnahmefaktor“ (siehe Kapitel 4.4). Über die mit Abstand meisten Parameter verfügen die Evolutionären Verfahren MOA und NSGA. Dazu gehört insbesondere die Populationsgröße. Falls die Algorithmen mit Selbst-Adaption gestartet werden, so sind die jeweils eingestellte Selektionsstrategie (und je nach Strategie evtl. Tournament-Größe), Kreuzungsstrategie (und evtl. Crossoverwahrscheinlichkeit), Mutationswahrscheinlichkeit und

Mutationsvarianz lediglich in der Startphase der Optimierung von Belang und werden später ohnehin dynamisch angepasst. Unabhängig davon ist die Größe des Elitismus-Pools und im Falle des NSGA die Art des Elitismus (siehe „*Pareto-Elitismus*“ in Kapitel 5.1.6). Ebenfalls ein zusätzlicher Freiheitsgrad des NSGA ist die Wahl des *Niching-Radius* (siehe Kapitel 4.5.5) zur Verhinderung einer Konvergenz der Population zu einem einzigen Punkt der Pareto-Front.

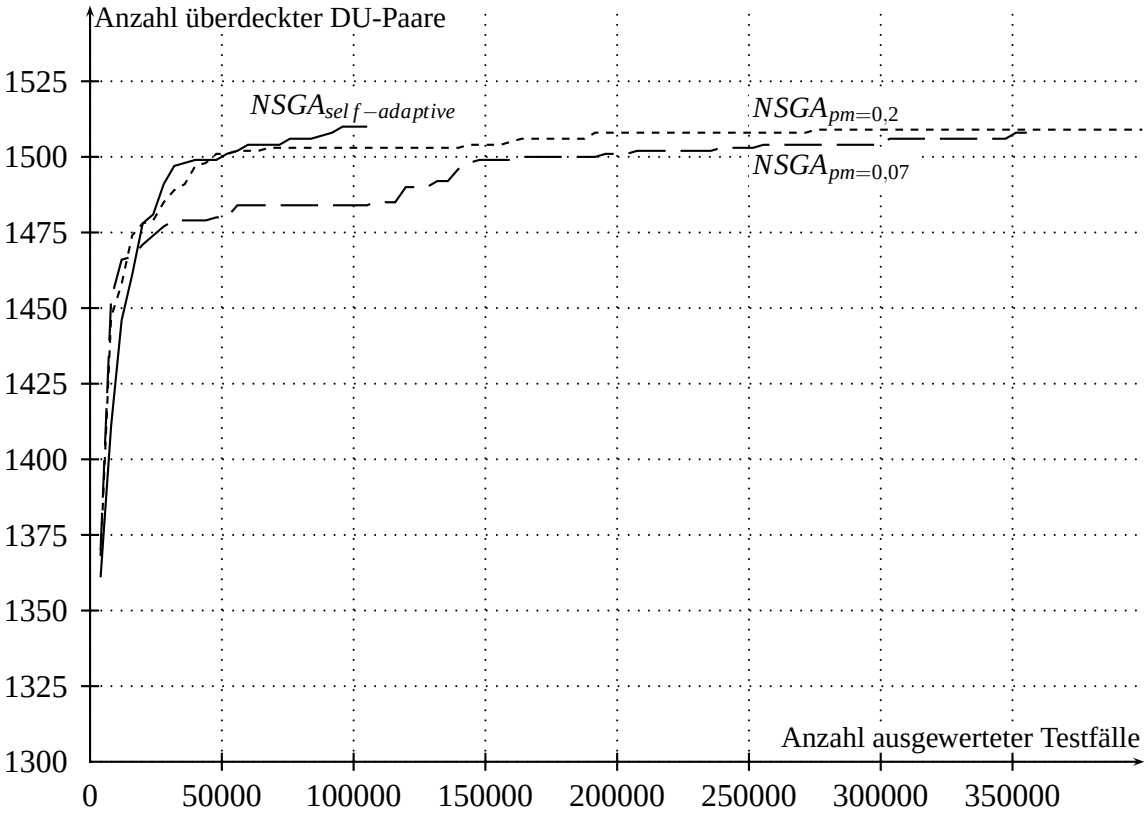


Abbildung 6.6: Vergleich der Parametrisierungen (NSGA, Projekt *BigFloat*)

Aufgrund obiger Aufzählung ist ersichtlich, dass eine vollständige und statistisch aussagekräftige experimentelle Untersuchung der verschiedenen Verfahren mit allen „repräsentativen“ oder gar allen möglichen Parametrisierungen selbst für ein einziges Beispiel-Projekt einen realistischen Zeitrahmen übersteigt. Stellvertretend wurde im Rahmen dieses Forschungsprojektes jedoch eine interessante Fragestellung im Zusammenhang mit der voll-automatischen Selbst-Rekonfiguration des *Nondominated Sorting Genetic Algorithm* untersucht. Dazu wurde dieser NSGA mit der automatischen Generierung optimaler Testdatenmengen für das Projekt *BigFloat* nach dem *all-uses*-Kriterium betraut. Unter den drei unterschiedlichen Parametrisierungen war eine mit aktivierter Selbstadaption. Die anderen beiden Experimente wurden mit abgeschalteter Selbst-Rekonfiguration durchgeführt, wobei alle Parameter, mit Ausnahme der Mutationswahrscheinlichkeit, identisch mit denen der initialen Konfiguration der selbst-adaptiven Variante waren. Lediglich die Mutationswahrscheinlichkeit wurde variiert, wobei sie in einem Experiment mit 0,07 der üblichen Empfehlung für Genetische Algorithmen entsprach, während sie in einem weiteren Versuch mit 0,2 deutlich höher lag.

Das Ergebnis des Vergleichs zeigt Abbildung 6.6. Erwartungsgemäß erzielte die selbst-adaptive Variante des NSGA wesentlich schneller eine höhere Überdeckung. Bereits nach 49164 Generationen und damit nach 93221 insgesamt ausgeführten und evaluierten Testfällen wurde eine Testfallmenge identifiziert, welche 1510 *def/use*-Paare überdecken konnte, jedoch mit 32 Testfällen in Bezug auf den Umfang suboptimal war. Mit 87812 Generationen und 619316 ausgewerteten Testfällen erforderte die Variante mit einer konstanten Mutationsrate von 0,2 wesentlich mehr Aufwand. Im Gegenzug war die dann ermittelte Testfallmenge mit 22 Testfällen verhältnismäßig besser. Mit Abstand am Ungünstigsten schnitt in diesem Vergleich die Variante mit der konstanten Mutationsrate von 0,07 ab, welche selbst nach 91001 Generationen und 356068 betrachteten Testfällen insgesamt lediglich eine Testfallmenge mit einer Überdeckung von 1508 DU-Paaren (bei 26 Testfällen) erreichen konnte.

6.3 Fehlererkennung verschiedener Überdeckungskriterien

Mittels dynamischer Analyse nach dem Verfahren aus Kapitel 5.1 wird die absolute Anzahl der Entitäten bestimmt, die von einem Testfall überdeckt wurden. Bei zusätzlichem Einsatz einer statischen Analyse entsprechend Kapitel 5.2.1 kann auch ein relatives Überdeckungsmaß für jeden Testfall beziehungsweise jede Testfallmenge berechnet werden. Insbesondere dieser Erfüllungsgrad des vorgegebenen strukturellen Überdeckungskriteriums reflektiert, wie vollständig das zu testende System hinsichtlich seiner Struktur, hier zum Beispiel der Kontroll- oder Datenflussstruktur, überprüft wurde. Daher ist im Allgemeinen eine gewisse Korrelation zwischen erreichter Testüberdeckung und Fehlererkennungspotential eines Testdatensatzes zu erwarten, was in einer Reihe experimenteller Untersuchungen empirisch bestätigt werden konnte [FW93].

Einen wesentlich objektiveren Indikator für das Fehleraufdeckungsvermögen einer Testfallmenge stellt der in Kapitel 3.6 vorgestellte *mutation score* dar. Während das Überdeckungsmaß lediglich den Anteil der überdeckten strukturellen Entitäten eines Programms berücksichtigt, wird der *mutation score* als Anteil der aufgedeckten „Fehler“ von den bekannten und gezielt injizierten „Fehlern“ bestimmt. Um dieses Maß möglichst allgemein gültig und unabhängig von einem speziellen Testobjekt zu gestalten, bedient man sich bei der Injektion eines Katalogs repräsentativer Fehler, wie sie von Entwicklern erwartungsgemäß gemacht werden. In •gEAr wird daher eine *Mutationsanalyse* nach Kapitel 3.6 zur Bestimmung der Fehleraufdeckungsquote einer automatisch generierten Testfallmenge verwendet. Die Anzahl der zu diesem Zweck von *µJava* ([OMK04, MOK05]) generierten und lauffähigen „fehlerhaften“ Varianten eines jeden Projekts sind in Tabelle 6.1 aufgeführt.

Anmerkung: Vorweg seien noch zwei wesentliche Details der Mutationsanalyse von •gEAr erwähnt. Die für JAVA™ implementierte Variante entspricht dem aus der Literatur unter der Bezeichnung *strong mutation testing* bekannten Verfahren. Damit dabei ein Mutant als „getötet“ gilt, muss er eine andere Ausgabe erzeugen als die ursprüngliche Programmversion. Im Gegensatz zum *weak mutation testing* wird demnach der jeweilige, interne Zustand während der Ausführung nicht explizit berücksichtigt, sondern lediglich das beobachtbare Verhalten des Testobjekts.

Die im Folgenden genannten „*mutation score*“-Kennzahlen sind insofern pessimistisch, als dass die zur ursprünglichen Programmvariante äquivalenten Mutanten nicht explizit herausgerechnet wurden. Im Gegensatz zur Definition des *mutation score MS* in Kapitel 3.6, gilt hier also die pessimistische Variante *MSP*. Sei $\mathcal{P}$ ein Programm, $\mathcal{T}$ eine Testfallmenge, $M_t(\mathcal{P})$ die Menge aller aus $\mathcal{P}$ entstandenen Mutanten und $M_k(\mathcal{P},\mathcal{T}) \subseteq M_t(\mathcal{P})$ die Menge aller von $\mathcal{T}$ getöteten Mutanten, dann ist der sogenannte *mutation score (pessimistisch)* gegeben durch:

$$MSP(\mathcal{P},\mathcal{T}) = \frac{|M_k(\mathcal{P},\mathcal{T})|}{|M_t(\mathcal{P})|}$$

Für die sechs Beispiel-Projekte aus Tabelle 6.1 wurden mittels •gEAr zunächst optimale Testfallmengen hinsichtlich des Kriteriums der *Verzweigungsüberdeckung (branch coverage)* generiert. In einem zweiten Durchlauf wurden zusätzlich auch Testdatensätze erstellt und optimiert, die eine möglichst hohe Überdeckung nach dem Maß der datenflussorientierten *all-uses*-Teststrategie erzielen. Anschließend wurden die jeweils optimalen Testfallmengen einer Mutationsanalyse unterzogen und so deren jeweiliger (pessimistischer) *mutation score MSP* ermittelt.

ID	Projekt	AGT	AÜ-V	V-Ü	*V-Ü feas.*	AGM	ATM	MSP	GOA
1	BigFloat	5	146	83,72	*98,63*	1528	1057	69,18	*NSGA*
2	Dijkstra	1	26	92,86	*100*	220	155	70,45	*NSGA*
3	Hanoi	2	4	100	*100*	227	170	74,89	*NSGA*
4	Huffman	3	61	78,21	*100*	623	463	74,32	*NSGA*
5	JDK sort	1	37	92,50	*100*	852	524	61,50	*NSGA*
6	JDK logging	35	317	65,23	*k.A.*	1970	485	24,62	*NSGA*

AGT: Anzahl generierter Testfälle
AÜ-V: Anzahl überdeckter Verzweigungen
V-Ü: Verzweigungsüberdeckung (in %; basierend auf statischer Analyse)
AGM: Anzahl generierter Mutanten (einschließlich evtl. äquivalenter)
ATM: Anzahl der von den jeweiligen Testfällen getöteten Mutanten
MSP: mutation score (pessimistisch): MSP=ATM/AGM (in %)
GOA: Generierungs- und Optimierungsalgorithmus
feas.: korrigierte Angaben unter Berücksichtigung der *feasibility*

Tabelle 6.3: Kontrollflussorientiert generierte Testfälle (*branch*)

Die Ergebnisse dieser experimentellen Untersuchungen sind in Tabelle 6.3 (bezüglich Verzweigungsüberdeckung) und Tabelle 6.4 (*all-uses*-Überdeckung) dargestellt[5]. Für beide Überdeckungskriterien enthält die mit AGT beschriftete Spalte den Testumfang, also die Anzahl der jeweils generierten Testfälle in der optimalen Testfallmenge. In den Spalten AÜ-V beziehungsweise AÜ-AU sind die Anzahlen der jeweils von diesen Testfällen überdeckten Verzweigungen respektive *def/use*-Paare aufgeführt. Die in beiden Übersichten jeweils darauf folgenden Spalten V-Ü beziehungsweise AU-Ü enthalten das Überdeckungsmaß in % basierend auf der

[5]Tabelle 6.3, Tabelle 6.4 und Tabelle 6.5 wurden in [Ost05, OS06] auszugsweise vorgestellt, allerdings noch ohne Berücksichtigung der Feasibility, was in diesem Kapitel genauer behandelt wird.

ID	Project	AGT	AÜ-AU	AU-Ü	AGM	ATM	MSP	GOA
1	BigFloat	17	1511	82,83*	1528	1159	75,85	*SimAnn*
2	Dijkstra	2	168	93,75*	220	158	71,82	*NSGA*
3	Hanoi	2	42	96,67	227	174	76,65	*MOA*
4	Huffman	3	353	92,86*	623	470	75,44	*SimAnn*
5	JDK sort	2	315	83,44	852	557	65,38	*NSGA*
6	JDK logging	61	1599	82,82*	1970	497	25,23	*NSGA*

AGT: Anzahl generierter Testfälle
AÜ-AU: Anzahl überdeckter def/use-Paare
AU-Ü: all-uses-Überdeckung (in %; basierend auf statischer Analyse)
AGM: Anzahl generierter Mutanten (einschließlich evtl. äquivalenter)
ATM: Anzahl der von den jeweiligen Testfällen getöteten Mutanten
MSP: mutation score (pessimistisch): MSP=ATM/AGM (in %)
GOA: Generierungs- und Optimierungsalgorithmus
* Durchschnittswerte analysierbarer Einstiegsmethoden

Tabelle 6.4: Datenflussorientiert generierte Testfälle (*all-uses*)

statischen Analyse des Testobjekts. Da die statische Analyse nicht zwischen ausführbaren und nicht-ausführbaren Teilpfaden unterscheidet, werden hier zunächst *alle* graphentheoretisch ermittelbaren Entitäten unabhängig von ihrer Überdeckbarkeit (Feasibility) in die Berechnung eingeschlossen, weshalb die prozentualen Angaben in den Spalten V-Ü und AU-Ü konservativ sind. Zur Demonstration der zum Teil gravierenden Unterschiede in den berechneten Überdeckungsmaßen enthält Tabelle 6.3 zusätzlich die Spalte „*V-Ü feas.*“, deren Datengrundlage mittels einer manuellen Untersuchung der Feasibility jeder Verzweigung korrigiert wurde – was aufgrund der Komplexität der Fragestellung lediglich bei den Projekten 1 bis 5 und nur für die Verzweigungsüberdeckung realisiert wurde. Bei den in der Spalte AU-Ü der Tabelle 6.4 mit „*“ gekennzeichneten Werten wurde als Einstiegsmethode für die Analyse (siehe Kapitel 5.2.1 ab Seite 184) nicht die `main`-Methode des Testtreibers gewählt, sondern die Überdeckung für jede aufrufbare Methode der Komponente getrennt analysiert und diese anschließend arithmetisch gemittelt. Im Weiteren folgen in den Ergebnisübersichten jeweils die Anzahl der von der jeweils optimalen Testfallmenge getöteten Mutanten (ATM) und der pessimistische *mutation score* (MSP). Zur Information benennt die letzte Spalte (GOA) der beiden Tabellen jeweils das Optimierungsverfahren, mit welchem die optimale Testfallmenge ermittelt wurde.

Für das Projekt *JDK sort* konnte •gEAr einen hochgradig optimierten Testdatensatz mit nur einem Testfall finden, welcher 37 und damit 92,5% der 40 statisch identifizierten Zweige (infolge von 20 binären Verzweigungen) zu überdecken vermochte. Effektiv lag die Verzweigungsüberdeckung dieses Testfalls jedoch bei 100%, da aufgrund einer manuellen Untersuchung lediglich 37 der 40 Zweige als dynamisch überdeckbar (feasible) identifiziert wurden. Trotz der Komplexität des Testobjektes konnte dieser Testfall von den 852 insgesamt erstellten Mutanten 524 Variationen töten, was einem *mutation score* von etwa 61,5% entspricht. Eine nach dem Kriterium *all-uses* generierte Testfallmenge umfasst doppelt so viele Testfälle und erzielt 315 *def/use*-Paare, was einer Überdeckung von ca. 83,4% der statisch ermittelten DU-Paare ausmacht. Von den 852

Mutanten wurden durch diesen datenflussorientierten Testdatensatz nunmehr 557 getötet. Damit konnten 33 „Fehler“ mehr aufgedeckt werden als mit der verzweigungsüberdeckungsorientierten Testfallmenge. Um einen Eindruck von der Komplexität und dem Umfang einer datenflussorientierten Teststrategie zu geben, sie hier erwähnt, dass die statische Analyse 50 verschiedene Definitionen (*defs*) identifiziert hat, von denen die zugehörigen Verwendungen (*uses*) über insgesamt 501166 DU-Teilpfade erreicht werden können.

Ähnliche Ergebnisse wies das Experiment auch beim wesentlich größeren und komplexeren Projekt *BigFloat* auf. Die minimale Testfallmenge mit einer Überdeckung von 146 Verzweigungen umfasste fünf Testfälle, welche zusammen ein Überdeckungsmaß von nominell 83,7% beziehungsweise effektiv 98,6% erzielten. Von den 1528 Mutanten konnten mit diesem Testdatensatz insgesamt 1057 (69,2%) getötet werden. Für den aufwändigeren datenflussorientierten Test waren zwölf zusätzliche Testfälle notwendig, so dass die Testfallmenge mit nun 17 Testelementen insgesamt 1511 *def/use*-Paare überdecken konnte. Da es sich bei *BigFloat* um eine Komponente und nicht um eine selbständig ablauffähige Applikation handelt, wurde die statische Datenflussanalyse für jede Methode getrennt durchgeführt und die auf dieser Grundlage berechnete Überdeckung arithmetisch gemittelt, was entsprechend Tabelle 6.4 ein Maß von 82,83* ergibt. Dieser größere Testfall tötete von 1528 nunmehr 1159 Mutanten und erreichte einen pessimistischen *mutation score* von etwa 75,9%.

Besonders hervorzuheben ist an dieser Stelle, dass das hier präsentierte Verfahren für das Projekt-Beispiel *BigFloat* mit lediglich 12 weiteren Testfällen bereits 102 zusätzliche „Fehler“ aufdecken konnte. Erreicht wurde diese Verbesserung der Fehleraufdeckung dadurch, dass der Testdatengenerierung und -optimierung nicht die Strategie der Verzweigungsüberdeckung sondern das datenflussorientierte *all-uses*-Kriterium zugrunde gelegt wurde. Abgesehen von dem zusätzlichen Aufwand zur Überprüfung der 12 zusätzlichen Testfälle, erfordert das Testen mit •gEAr nach dem *all-uses*-Kriterium keinen höheren Aufwand als anhand der Verzweigungsüberdeckungsstrategie. Darüber hinaus wird gegenüber einer manuellen Testdatenermittlung auch der Zusatzaufwand für die Validierung der Testergebnisse minimiert, da die Anzahl der Testfälle pro optimaler Testfallmenge ebenfalls von •gEAr minimiert wird.

6.3.1 Vergleich der Fehlerarten

In Kapitel 3.4 wird mit dem Beispiel aus Abbildung 3.5 exemplarisch demonstriert, dass mit datenflussorientiertem Testen, aufgrund der Anreicherung des Kontrollflusses um Datenflussinformationen, nicht nur weitere Aspekte der Programmstruktur untersucht werden – vielmehr subsumiert ein Großteil der Kriterien aus der Datenflussfamilie die verbreiteten Kontrollflusskriterien Anweisungs- und Verzweigungsüberdeckung (Abbildung 3.9), weshalb im Allgemeinen eine höhere Fehleraufdeckungsquote beim Testen nach Datenflussstrategien zu erwarten ist als bei einfacheren kontrollflussorientierten Kriterien. Obwohl datenflussbasiertes Testen dem rein kontrollflussorientierten nachweisbar überlegen ist, behinderte bislang vorwiegend der höhere Aufwand bei der Erstellung und Überprüfung entsprechender Testfälle den verbreiteten praktischen Einsatz. Doch mit dem hier vorgestellten Ansatz ist dieses Argument entkräftet, da entsprechende Testfallmengen nicht nur vollautomatisch generiert werden, sondern auch hinsichtlich des Umfangs und damit des Validierungsaufwands minimiert werden.

Dass die obige Annahme gerechtfertigt ist, zeigen insbesondere die experimentellen Ergebnisse bezüglich der beiden Projekte *Hanoi* und *Huffman*. Obwohl die Größe der generierten und optimierten Testfallmengen sowohl zur Verzweigungsüberdeckung (Tabelle 6.3) als auch zur *all-uses*-Überdeckung (Tabelle 6.4) gleich ist, deckten die datenflussorientierten Testfälle vier (*Hanoi*) beziehungsweise sieben (*Huffman*) zusätzliche Fehler auf. Auch bei den anderen Beispiel-Anwendungen stieg die Anzahl der Testfälle im Vergleich zur gewonnen Verlässlichkeit im Vergleich der Verzweigungs- und der *all-uses*-Überdeckung nicht überproportional an.

Zu den typischen Produktfehlern oder potentiellen Programmanomalien, die eventuell einer genaueren manuellen Betrachtung bedürfen, welche im Zuge eines datenflussorientierten Testens nach einem höheren Kriterium, wie zum Beispiel *all-uses*, aufgedeckt werden können, gehören unter anderem auch die im Folgenden beschriebenen.

Die statische Analyse entsprechend Kapitel 5.2.1 erfordert die Konstruktion eines interprozeduralen Kontrollflussgraphen (ICFG) und eine Anreicherung desselben um datenflussrelevante Informationen. Bereits eine Untersuchung dieses Graphen kann unerreichbaren Code (sogenannten *dead code*) aufdecken, welcher sich durch Knoten offenbart, die von keinem der Einstiegsknoten erreichbar sind. Im Falle von Komponenten ist jede von außen aufrufbare Methode eine potentielle Einstiegsstelle, so dass deren Kopfknoten jeweils Einstiegsknoten des ICFG der Komponente darstellen. Bei vollständigen Applikationen ist jede der aufrufbaren `main`-Methoden einzeln zu untersuchen. Ebenfalls vermag eine statische Analyse des ICFG sogenannte *syntaktische Endlosschleifen* zu offenbaren. Diese werden durch stark zusammenhängende Teilgraphen[6] repräsentiert, welche zwar über eingehende jedoch über keine ausgehenden Kanten verfügen, das heißt, von jedem Knoten des Teilgraphen ist zwar jeder andere Knoten des gleichen Teilgraphen erreichbar, jedoch kein weiterer Knoten des ICFG.

Nebst kontrollflussorientierten „Gefahrenstellen“ im Code kann eine statische Analyse des datenflussannotierten Kontrollflussgraphen auch eine ganze Reihe informationsflussrelevanter Anomalien oder Fehler identifizieren [CL95]. Dazu gehören insbesondere Verwendungen (*uses*) von Variablen, welche entlang von Teilpfaden von einem Einstiegsknoten erreicht werden, entlang derer jedoch keine Zuweisungen (*defs*) dieser Variablen erfolgen. Diese „use-ohne-vorangehendes-def“ stellen demnach Verwendungen nicht-initialisierter Variablen dar und sind somit eindeutig Produktfehler. Eine ähnliche Anomalie muss nicht zu einem Versagen des Testobjekts führen, weist jedoch auf einen grundsätzlichen Architekturfehler hin. Es handelt sich dabei um Definitionen von Variablen, von denen graphentheoretisch keine erreichbaren Verwendungen der gleichen Variablen identifiziert werden können. Diese „def-ohne-erreichbares-use“ können als falsch platzierte Anweisungen oder fehlerhafte Anweisungsfolgen betrachtet werden.

Über die statische Analyse hinaus erlaubt auch die tatsächliche Ausführung des Programms zur dynamischen Analyse datenflussorientierter Testfälle die Aufdeckung diverser Anomalien und Fehler. Dazu gehören beispielsweise *def/use*-Paare, deren Assoziation zwar graphentheoretisch ermittelt werden kann, deren Überdeckung aber mit keiner Eingabe gelingt. Kann für eine beliebige Definition kein einziges dieser DU-Paare überdeckt werden, so ist dies statisch nicht als Anomalie zu erkennen, wohl jedoch während der Testfallermittlung. Weil eine datenflussba-

[6] Ein (Teil-)Graph heißt *stark zusammenhängend*, wenn von jedem seiner Knoten jeder andere Knoten des Graphen über eine endliche Kantenfolge erreichbar ist.

sierte Teststrategie wie *all-uses* oder *all-DU-paths* die Überdeckung aller (möglichen) *def/use*-Paare fordert, ist die Wahrscheinlichkeit gegenüber der Anweisungsüberdeckung ungleich höher, dass fehlerhafte Typ-Konvertierungen (*type-cast*) oder typ-inkonsistente Verwendungen entdeckt werden, die auch durch die statische Analyse eines Übersetzers nicht identifizierbar sind.

Im Gegensatz zur Verzweigungsüberdeckung werden bei *all-p-uses* (oder dieses Kriterium subsumierende Strategien) die einzelnen Prädikate wesentlich genauer getestet. Zwar subsumiert *all-p-uses* kein Kriterium aus der Familie der Bedingungsüberdeckung (siehe Kapitel 3.3 beziehungsweise Abbildung 3.9), jedoch ergänzt es diese durch Betrachtung der in die Auswertung von Teilbedingungen einfließenden Informationen. Beinhaltet ein Prädikat die Verwendung einer Variablen, so kann dieser *p-use* Teil unterschiedlicher *def/use*-Paare sein, nämlich dann, wenn dieser Verwendung verschiedene Definitionen der betrachteten Variablen graphentheoretisch vorausgehen, von denen der *p-use* jeweils erreichbar ist. Somit ist das Ergebnis des Prädikats von unterschiedlichen, vorangehenden Berechnungen und Zuweisungen abhängig. Diese Zusammenhänge zu testen fordern weder die Verzweigungsüberdeckung noch eine der Bedingungsüberdeckungskriterien.

Umfassendere datenflussorientierte Teststrategien, darunter diejenigen aus der Familie der *required k-tuples* von Ntafos (Kapitel 3.4.3) oder der *(ordered) context coverage* von Laski/Korel (Kapitel 3.4.4) sowie insbesondere deren Erweiterungen nach [FB01] unterstützen in besonderer Weise die Erkennung von Versagen während der Testphase. Die dabei betrachteten und zu überdeckenden Entitäten, sogenannte *„basic program functions“* [FB01], sind def/use-Ketten ähnlich denen obiger Strategien, deren letzte Verwendung jedoch mit einer Programmausgabe gekoppelt ist. Im Gegensatz zu den meisten Kontrollflusskriterien wird so erzwungen, dass eventuell fehlerhafte Programmzustände sich auch wirklich in einem beobachtbaren Versagen, also in einer Abweichung des tatsächlichen vom spezifizierten Ein-/Ausgabe-Verhalten, manifestieren.

Im Falle objekt-orientierter Software besteht der „Zustand“ eines Objektes in der jeweils aktuellen Belegung der Felder der betrachteten Instanz. Das Verhalten einer Methode hängt im Allgemeinen vom Zustand dieses Objektes oder sogar weiterer Objekte ab. Da jede Änderung einer Variablenbelegung eine Zustandsänderung bedeuten kann, beeinflusst jede Definition unter Umständen das nachfolgende Verhalten der Applikation. Entsprechend wird das Verhalten einer Methode dadurch von der Belegung einer Variablen gesteuert, dass dieser Wert ausgelesen wird, was einem *(c/p-)use* entspricht. Demnach impliziert das *all-uses*-Kriterium eine effektivere Teststrategie für objekt-orientierte Systeme, als dies klassische kontrollflussorientierte Verfahren darstellen. Auf diese Weise können mittels datenflussorientiert generierter Testfälle zum Beispiel fehlerhafte Methodenaufrufsequenzen identifiziert werden, welche sich dadurch äußern, dass die an ein Objekt gerichteten Nachrichten selbiges in einem unzulässigen Zustand erreichen.

Trotz der evidenten Vorteile und der gegenüber vergleichbaren Teststrategien erhöhten Aufdeckungsquote, insbesondere im Bezug auf Fehler in der Datenverarbeitung, darf jedoch nicht unerwähnt bleiben, dass datenflussorientierte Testverfahren zwar den Fluss der Daten im Sinne der Kontrollflussstruktur betrachten, nicht jedoch die tatsächlich gespeicherte und transformierte Information. Aus diesem Grund ist selbst ein Kriterium mit hohen Anforderungen aus dieser Familie, zum Beispiel *all-uses* oder *all-DU-paths*, nur in Kombination mit alternativen, orthogonalen Testverfahren erfolgreich. Ein typisches Beispiel für diese Notwendigkeit wird in Kapitel 5.1.7 erläutert. Der dort beschriebene Äquivalenzklassengrenzwerttest ist ein besonders

gut geeigneter Kandidat für eine Zusammenführung mit einem klassischen Datenflusskriterium – was mit dem hier vorgestellten Verfahren der multi-objektiven Generierung und Optimierung auf einfache Weise erreicht werden kann.

Unabhängig vom zugrunde liegenden Testkriterium hat das hier vorgestellte Verfahren zur automatischen Testdatengenerierung und -optimierung einen weiteren entscheidenden Vorteil: Es ist unabhängig davon, ob die statische oder dynamische Analyse auf Ebene des Quellcodes oder des Bytecodes umgesetzt wird. Insbesondere wenn auf den Bytecode angewandt, generiert •gEAr geeignete Testfälle, mit denen auch die Korrektheit des Übersetzers validiert werden kann - eine Anforderung, die insbesondere für Software mit hoher Sicherheitsrelevanz von Bedeutung ist und gerade für komplexe Compiler, welche mit Optimierungsstrategien arbeiten [Lho02].

Zusammenfassend kann aus Tabelle 6.3 und Tabelle 6.4 geschlossen werden, dass in allen Projekt-Beispielen mehr Fehler mit Testfällen aufgedeckt werden konnten, welche nach dem *all-uses*-Kriterium generiert und optimiert wurden, als mit denjenigen, die entsprechend der Verzweigungsüberdeckung ermittelt wurden. Dabei konnten bis zu 7% der injizierten Produktfehler (genauer: 102 „Fehler“ im Projekt *BigFloat*) mit der datenflussorientierten Testfallmenge aufgedeckt werden – Fehler, welche der kontrollflussorientierten Strategie verborgen geblieben sind.

6.3.2 Erweiterte Testfallmengen

In vorangehenden Kapiteln (insbesondere Kapitel 5.1.7) wurde erläutert, wie mit dem im Rahmen dieses Forschungsprojektes entwickelten und im Werkzeug •gEAr umgesetzten Verfahren qualitativ hochwertige Testfälle zusammen mit den zu ihrer Ausführung notwendigen Testdaten automatisch generiert und optimiert werden. Dabei kann die vorgestellte Methode dahingehend konfiguriert werden, die Testdatenmengen nicht nur hinsichtlich eines einzigen, vorgegebenen Testkriteriums zu identifizieren, sondern mittels multi-objektiver Metaheuristiken gleichzeitig mehrere Teststrategien zugleich zu verfolgen.

Alternativ kann •gEAr jedoch auch dazu verwendet werden, optimale Testfallmengen iterativ und damit inkrementell zu ermitteln. Dies kann insbesondere dann von Interesse sein, wenn bereits in einer vorangehenden Testphase eine gewisse Anzahl unterschiedlicher Testfälle erstellt und eventuell sogar manuell überprüft wurde. In diesem Fall ist es sinnvoll, diese bestehenden Tests wiederzuverwenden und die dadurch erreichte Überdeckung zu nutzen. In der Praxis ist es üblich, ja sogar von Zertifizierungsstandards (zum Beispiel DO-178B/ED-12B für Software in Luftverkehrssystemen) empfohlen, zunächst funktional zu testen und die so ermittelten Testfälle um zusätzliche anzureichern, um damit eine vorgegebene strukturelle Überdeckung zu erreichen. Im Rahmen der Wartung und der sich anschließenden Regressionstestphase können viele Testfälle ebenfalls ohne umfangreiche Anpassungen wiederverwendet werden [Pol04], wobei meist noch zusätzliche Tests hinzuzufügen sind.

In Softwareentwicklungsprojekten mit eng gestecktem Zeithorizont, dicht aufeinander folgenden Meilensteinen oder inkrementeller Entwicklung (typisch für den *„eXtreme Programming“*-Ansatz) kann es sinnvoll sein, zunächst einen Grundstock an Testfällen nach einem einfacheren, kontrollflussorientierten Kriterium zu generieren, zum Beispiel zur Verzweigungsüberdeckung. Sobald jedoch die endgültige Freigabe des Produktes oder einzelner Komponenten ansteht, können die bestehenden Testfälle, in Abhängigkeit vom aktuellen Zeit- und Kostenrah-

men, so erweitert werden, dass umfassendere Überdeckungskriterien erfüllt werden, eben beispielsweise datenflussorientierte Kriterien wie *all-uses*, und das Produkt daher genauer testen.

Ein solches Vorgehen wurde im Rahmen der experimentellen Anwendung des Werkzeugs •gEAr ebenfalls genutzt, jedoch um eine ganz besondere Fragestellung zu untersuchen. Betrachtet man die Ergebnisse aus Tabelle 6.3 oder Tabelle 6.4, so fällt auf, dass die erzielte Fehleraufdeckung (*mutation score*, MSP) mit durchschnittlich 63% beziehungsweise 73% trotz hochgradig optimierter und damit minimierter Testfallmengen beachtlich ist, jedoch anscheinend noch genügend Spielraum für Verbesserungen lässt. Dabei ist jedoch vorab zu klären, ob die nicht getöteten Mutanten lediglich aufgrund einer Schwäche des jeweils den Testfallmengen zugrunde gelegten Testkriteriums überlebt haben oder ob sie lediglich funktional äquivalent zur ursprünglichen Fassung des Programms sind und daher gar nicht getötet werden können. Da eine deterministische Untersuchung aufgrund der Anzahl der generierten Mutanten prohibitiv ist, wurde hier die in Kapitel 3.6.1 vorgestellte Übertragung des Konzeptes des statistischen Testens angewandt.

ID	Project	AAT	ART	AGM	ATM	MSP	$p^*_{\beta=0.05}$	$p^*_{\beta=0.01}$
1	BigFloat	3000	232	1528	1463	95,75	$1,0 \cdot 10^{-3}$	$1,5 \cdot 10^{-3}$
2	Dijkstra	10000	8	220	167	75,91	$3,0 \cdot 10^{-4}$	$4,6 \cdot 10^{-4}$
3	Hanoi	1000	11	227	195	85,90	$3,0 \cdot 10^{-3}$	$4,6 \cdot 10^{-3}$
4	Huffman	*k.A.*	6	623	623	100	*entf.*	*entf.*
5	JDK sort	4000	108	852	694	81,46	$7,5 \cdot 10^{-4}$	$1,2 \cdot 10^{-3}$

AAT: Anzahl insgesamt ausgeführter Testfälle
ART: Anzahl relevanter (Mutanten tötender) Testfälle
AGM: Anzahl generierter Mutanten (einschließlich evtl. äquivalenter)
ATM: Anzahl der von den relevanten Testfällen (ART) getöteten Mutanten
MSP: mutation score (pessimistisch): MSP=ATM/AGM (in %)
p^*_β $p^*_\beta = 1 - \sqrt[AAT]{\beta}$ (siehe Kapitel 3.6.1)

Tabelle 6.5: Mutationstestorientiert generierte Testfälle

Die Ergebnisse des Experiments sind in Tabelle 6.5 aufgeführt. Dabei wurde für jedes betrachtete Projekt die jeweils optimale und nach dem Kriterium *all-uses* generierte Testfallmenge als Basis wiederverwendet. Im Rahmen des klassischen Mutationstesten wurden darüber hinaus zusätzliche Testfälle zufällig und nach einer uniformen Verteilung der Eingaben generiert und mittels des *back-to-back*-Ansatzes dahingehend untersucht, ob diese jeweils mindestens einen verbliebenen Mutanten töten konnten. Die Spalte AAT der Tabelle 6.5 enthält die Gesamtzahl der zufällig generierten und bewerteten Testfälle, während die darauf folgende Spalte ART die Anzahl derjenigen Testfälle wiedergibt, welche mindestens einen lebendigen Mutanten zu töten vermochten. Auf diese Weise konnte beispielsweise die Fehleraufdeckungsquote der *all-uses*-Testfälle für das Projekt *BigFloat* (siehe Tabelle 6.4) von circa 75,9% (1159 von 1528 getöteten Mutanten) auf beinahe 95,8% (1463 von 1528) verbessert werden. Dazu mussten jedoch dem ursprünglichen Testdatensatz, mit einem Umfang von lediglich 17 unterschiedlichen Testfällen, zusätzliche 232 Tests hinzugefügt werden, zu deren automatisierter Identifikation insgesamt 3000 Testfälle zufällig ausgewählt und untersucht wurden.

Die Tabelle 6.5 enthält darüber hinaus in den letzten Spalten die Wahrscheinlichkeit p^*_β dafür, dass ein beliebiger aber fester Mutant aus der Menge der schließlich noch immer nicht getöteten Mutanten des jeweiligen Projektes, bei Ausführung mit einer zufälligen, nach einem uniformen Operationsprofil ausgewählten Eingabe, ein von der ursprünglichen Variante des Programms abweichendes Verhalten aufweist (siehe Kapitel 3.6.1). Für alle Beispiel-Projekte liegt diese Wahrscheinlichkeit in der Größenordnung von 10^{-4} bis 10^{-3}, was die Annahme rechtfertigt, dass die verbliebenen Mutanten weitgehend funktional äquivalent zur Originalversion sind.

Schließlich kann auf Basis der Tabelle 6.5 gegenüber Tabelle 6.4, insbesondere aufgrund der experimentellen Ergebnisse für das Projekt *Dijkstra*, eine Bestätigung der herausragenden Effizienz des im Rahmen dieser Arbeit entwickelten und hier vorgestellten Verfahrens zur automatischen Testdatengenerierung und vor allem -optimierung aufgezeigt werden. Dabei zeigen die Daten, dass sich eine Optimierung der Testfallmengen, trotz drastisch reduziertem Aufwand zur Überprüfung der Testergebnisse, kaum nennenswert auf die Fehleraufdeckungsquote der automatisch ermittelten Testdaten auswirkt. Nach Tabelle 6.4 konnte •gEAr einen optimalen Testdatensatz mit lediglich zwei Testfällen identifizieren, welcher bereits 71,8% der injizierten Fehler aufzudecken vermochte. Diese datenflussorientiert generierte Testfallmenge erfordert lediglich die Validierung von zwei einzelnen Testfällen. Ohne die Anwendung einer automatischen Testfallgenerierung entsprechend •gEAr hätte ein Tester laut Tabelle 6.5 insgesamt 10.000 Testfälle einzeln überprüfen müssen, um eine vergleichbar hohe Anzahl von Fehlern (*mutation score*: 75,9%) aufzuspüren.

Kapitel 7

Ausblick: Erweiterungs- und Hybridisierungsansätze

„You cannot reduce the complexity of a given task beyond a certain point. Once you've reached that point, you can only shift the burden around."
Tesler's Law of Conservation of Complexity, Lawrence G. (Larry) Tesler

Im Rahmen dieser Arbeit (schwerpunktmäßig in Kapitel 5) wurde ein Verfahren zur automatisierten Testdatengenerierung vorgestellt, welches die Testfälle während der Generierung derart optimiert, dass mit einer minimalen Anzahl von Testfällen eine maximale strukturelle Code-Überdeckung erreicht wird. Dazu werden multi-objektive Metaheuristiken eingesetzt, welche die Verfolgung dieser beiden an sich gegenläufigen Ziele ermöglichen. Das Verfahren ist insofern generisch, als dass es sowohl auf strukturelle als auch auf zeitgemäße objekt-orientierte Programmiersprachen anwendbar ist. Eine weitere Besonderheit des hier beschriebenen Verfahrens erlaubt es, der Testgenerierung mehrere, teils orthogonale Überdeckungsmaße gleichzeitig zugrunde zu legen, wobei diese je nach Bedarf und in Abhängigkeit von der Art des Testobjektes beliebig gewichtet werden können. Das Verfahren zur Testdatengenerierung wird in einem integrierten Prozess von einer ebenfalls automatisierten Bewertung der ermittelten Testfälle im Sinne ihrer Effizienz hinsichtlich der Fehleraufdeckung ergänzt, so dass die jeweils identifizierten Testfallmengen bei Bedarf ebenfalls automatisch um zusätzliche Testfälle erweitert werden können. Das vorgestellte Verfahren erlaubt es somit, den Aufwand zur Verifikation und Validierung komplexer, sicherheitskritischer Software dadurch deutlich zu verringern, dass die automatisch generierten Testfallmengen sowohl eine maximale Überdeckung als auch einen minimalen Umfang aufweisen.

Zwar erlauben deutsche Zertifizierungsgremien derzeit nicht den Einsatz Künstlicher Intelligenz, zu denen die hier verwendeten Metaheuristiken zweifellos gehören, in hochsicherheitskritischen Anwendungen, beispielsweise in der Steuerung von öffentlichen Personentransportmitteln oder chemischer sowie nuklearer Reaktoren, doch sei darauf hingewiesen, dass das hier präsentierte Verfahren nicht selbst Teil der zukünftig eingesetzten Anwendung ist. Vielmehr dient es zur Unterstützung der Entwicklungs- und Verifikations-/Validierungsphasen. Die schließlich automatisch generierten Testdaten können und sollen unabhängig vom Verfahren sowie selbst-

redend auf einem nicht-instrumentierten Testobjekt ausgeführt werden. Ebenso beeinflusst die entwickelte Technik nicht den Überprüfungsprozess der Testergebnisse selbst.

Der vorangehend vorgestellte Ansatz ist bereits aufgrund der Kombination aus globaler (Kapitel 5.1) und lokaler (Kapitel 5.2) Optimierung der Klasse der sogenannten hybriden Heuristiken zuzuordnen, da hierbei nicht nur eine Instanz einer Metaheuristik zum Einsatz kommt, sondern mehrere dedizierte Varianten, welche jeweils verschiedene Aspekte der Gesamtaufgabe lösen sollen. Aufgrund dieser Aufteilung der Zuständigkeiten kann die Performanz dieses Verfahrens gegenüber einer einfachen globalen Optimierung deutlich gesteigert werden – andererseits ermöglicht nur die globale Optimierung eine effektive Minimierung des Testumfangs und damit des Testaufwands, was ein entscheidender Vorteil gegenüber bestehenden Ansätzen ist, welche lediglich die hier vorgestellte „lokale Optimierung“ wiederholt für jedes Testziel anwenden. Dennoch sind auch auf diesem Gebiet durchaus noch Erweiterungen des präsentierten Verfahrens denkbar. Insbesondere beruhen beide „Optimierungsebenen“ auf (evolutionären) Metaheuristiken, was keine Garantie zur Identifikation der optimalen Lösung zulässt. Daher wäre beispielsweise eine Hybridisierung Evolutionärer Verfahren, wie sie hier zur lokalen Optimierung eingesetzt werden, mit einer lokalen Suche oder insbesondere mit fachspezifischem Wissen denkbar. Dabei kann in besonderen Fällen auch der Einsatz einer statischen Analyse des Testobjekts und einer darauf aufbauenden, deterministischen Testdatenermittlung aufgrund einer symbolischen Auswertung (wie in [Grz04]) erwogen werden. In diesem Kontext sollte auch der Einsatz einer verhältnismäßig neuen Variante hybrider Optimierungsverfahren namens *Memetic Algorithms* [AMA01] in Betracht gezogen werden, welche die Vorteile Evolutionärer Verfahren und klassischer lokaler Suche vereinen.

Ebenfalls anzustreben ist eine Erweiterung der prototypischen Implementierung des Werkzeugs •gEAr um zusätzliche strukturelle oder funktionale Teststrategien, um die Eignung des hier vorgestellten Verfahrens zur Ermittlung von Testdaten auf Basis anderer Überdeckungskriterien zu untersuchen. Ein erster Schritt in diese Richtung wurde bereits mit der Verallgemeinerung des Ansatzes vom ursprünglich nur datenflussorientierten Testen über die Verzweigungsüberdeckung bis hin zum sogenannten *strukturellen Äquivalenzklassentesten* (Kapitel 5.1.7) und der Unterstützung der Bedingungsüberdeckung gemacht. Eine interessante Fragestellung ergibt sich bezüglich der Kombinierbarkeit dieses Ansatzes zur Testfallgenerierung mit dem statistischen Testen. Letzteres dient zur Bewertung der Zuverlässigkeit der Software. Die dabei zugrunde liegende statistische Stichprobentheorie fordert typischerweise, dass die ausgeführten Testfälle statistisch unabhängig, jedoch gemäß einem zukünftigen Operationsprofil gewählt werden, um eine verlässliche Aussage über die Zuverlässigkeit in der geplanten Anwendungsumgebung treffen zu können. Diesbezüglich wäre es denkbar, die zunächst uniforme und schließlich durch die strukturellen Anforderungen des Testkriteriums induzierte Verteilung der mit dem hier vorgestellten Verfahren generierten Testfälle gezielt mit einem zukünftig zu erwartenden Operationsprofil zu überlagern.

In eine vergleichbare Richtung stößt auch der Aspekt der Testpriorisierung. In der industriellen Praxis setzt sich zunehmend der Trend zu einem Kompromiss zwischen Testintensität (zum Beispiel in Sinne der Überdeckung) und dem dafür notwendigen Aufwand (Personal, Zeit, Geld) durch. Die Lösung für dieses Problem bietet die Priorisierung der Testaktivitäten, wobei besonders kritische Softwarekomponenten oder besonders häufig durchlaufene Kontrollflusspfade mit

mehr Testfällen bedacht und damit intensiver getestet werden. Das hier vorgestellte Verfahren betrachtet zunächst jede zu überdeckende Entität als gleichwertig, doch kann auch hier eine Priorisierung, ähnlich einem Operationsprofil auf struktureller Ebene, berücksichtigt werden. Ansätze zur Unterstützung einer solchen Priorisierung für datenflussorientierte Testkriterien wurden zum Beispiel in [AL98] und [MMB03] vorgestellt.

In Kapitel 6 wurde eine Reihe experimenteller Ergebnisse vorgestellt, welche verschiedene Aspekte der prototypischen Implementierung des im Rahmen dieser Arbeit vorgestellten Verfahrens beleuchten. Darüber hinaus kann das Werkzeug .gEAr für eine Vielzahl weiterer Untersuchungen genutzt werden, um weitergehende Erkenntnisse sowohl über die aktuelle Implementierung als auch über potentielle Verbesserungen und Erweiterungen des Ansatzes zu identifizieren. Da das Verfahren eine Minimierung des Testaufwandes bei gleichzeitiger Maximierung der Überdeckung anstrebt, haben die Experimente (allen voran Tabelle 6.3 und Tabelle 6.3) gezeigt, dass die vorgegebenen Testziele selbst dann mit verhältnismäßig wenigen, wenn auch hochgradig optimierten Testfällen erreicht werden können, wenn die Testobjekte hinsichtlich Kontroll- und Datenfluss einige Komplexität aufweisen. Dies führt zur berechtigten Frage, inwiefern mit einem so geringen Testumfang eine akzeptable Fehleraufdeckung erreicht werden kann beziehungsweise ob größere Testfallmengen nicht vorzuziehen wären. Dieser Fragestellung kann dadurch nachgegangen werden, dass für das gleiche Testobjekt und nach dem gleichen Testkriterium sowohl optimale (also minimale) Testfallmengen als auch gezielt umfangreichere Testdatensätze generiert werden. Vergleicht man anschließend jeweils die Fehleraufdeckungsquoten beider Ergebnisse, kann daraus womöglich eine Empfehlung hinsichtlich des Einsatzes einer solchen Testdatenoptimierung im Bereich sicherheitskritischer Software ausgesprochen werden. Dazu kann .gEAr bereits ohne weitere Änderungen beispielsweise so konfiguriert werden, dass die schließlich generierte Testfallmenge bei größeren Projekten zum Beispiel einen 2-3fach und bei kleineren einen 10fach größeren Umfang hat, als die erwartungsgemäß optimale (minimale).

Besonders im Hinblick auf einen breiten industriellen Einsatz des hier vorgestellten Verfahrens ist eine pragmatische Frage von großer Bedeutung: Welche Einsparungen können bei einer automatischen Generierung und Optimierung struktureller Testdaten gegenüber einer weitgehend manuellen Ermittlung gleichwertiger Testfälle erwirtschaftet werden? Während zur Ausführung manuell erstellter Testfälle für den Modultest einzelner Komponenten geeignete Testtreiber meist ebenfalls manuell programmiert werden müssen, erlaubt das Verfahren aus Kapitel 5.3 die automatisierte Generierung geeigneter Testtreiber, zusammen mit der notwendigen Schnittstellenbeschreibung, zur Ansteuerung der Testobjekte mit entsprechenden Testdaten, welche das hier entwickelte Verfahren bereitstellt – wodurch bereits im Vorfeld ein entsprechender Einspareffekt zu erwarten ist. Das größte Sparpotential bietet natürlich die eigentliche Ermittlung der Testdaten selbst, welche aufgrund des vorliegenden Verfahrens vollständig automatisiert erfolgen kann. Dabei können bereits vorhandene Testfälle wiederverwendet werden, was sich insbesondere auf die nächste Phase positiv auswirkt: Nach der Ermittlung und Ausführung entsprechender Testfälle müssen diese dahingehend untersucht werden, ob das Testobjekt bei ihrer Abarbeitung ein Fehlverhalten offenbart. Da dieser Schritt sowohl bei der manuellen als auch automatischen Testermittlung erforderlich ist und meist manuell erfolgt, müssen hier zwei gegensätzliche Effekte abgewogen werden: Einerseits kann der Tester bei manuell abgeleiteten Testfällen bereits Vorkenntnisse des Testobjekts einfließen lassen, wodurch die Überprüfung aufgrund der Erwar-

tungshaltung oder der Spezifikation leichter fällt als wenn die Testdaten automatisiert „scheinbar beliebig“ aus dem Eingaberaum gewählt werden. Andererseits werden die automatisch generierten Testfälle ja gerade hinsichtlich des Testumfangs minimiert, was erwartungsgemäß auch den Validierungsaufwand reduzieren soll. Eventuell jedoch vermögen optimierte Testfallmengen die Komplexität noch geringfügig zu steigern, da die einzelnen Testfälle mehrere Entitäten zugleich überdecken müssen.

Wenngleich sich aufgrund des in dieser Arbeit vorgestellten Verfahrens und der praktisch anwendbaren Implementierung gewiss noch eine Vielzahl weiterer Betrachtungsfelder und praktische wie theoretische Untersuchungsaspekte eröffnen, sei abschließend noch kurz auf einen aktuellen Ansatz verwiesen, welcher aus Sicht des Software Engineering von besonderem Interesse ist. Aufbauend auf den Erkenntnissen aus dem Projekt und den vielversprechenden Ergebnissen der experimentellen Untersuchungen, die jeweils in die vorliegende Arbeit eingeflossen sind, wurde ein öffentlich gefördertes und in Kooperation mit der lokalen Industrie (Medizintechnik) durchzuführendes Forschungsprojekt namens *UnITeD*[1] lanciert, welches das entwickelte Verfahren von der Betrachtung struktureller Teststrategien auf Code-Ebene abstrahieren und auf frühere Phasen der Softwareentwicklung übertragen soll. Dabei werden die Testszenarien im Sinne des modellbasierten Testansatzes aus UML-Modellen abgeleitet, während die zur Ausführung des Testobjektes notwendigen Testdaten mit einem zum vorliegenden Verfahren vergleichbaren Ansatz ermittelt werden. Im Rahmen dieses Folgeprojektes werden sowohl die modellbasierte Generierung optimaler Testdaten für einzelne Komponenten angestrebt als auch ein Ansatz verfolgt, der die Integrationstestphase (vergleiche auch [AO00]) unterstützen soll. Für beide Aspekte wurde jeweils eine Kriterienhierarchie definiert, die es erlaubt, ähnlich dem strukturellen Testen, ein modellbasiertes Überdeckungsmaß und damit eine Bewertungsfunktion für eine entsprechende Metaheuristik festzulegen, mit deren Hilfe die notwendigen Testdaten ohne Betrachtung des Programmcodes automatisiert generiert und optimiert werden können.

[1] UnITeD: Unterstützung Inkrementeller TestDaten

Anhang A

Zusätzliche Beispiele

„Why program by hand in five days
what you can spend five years of your life automating.“
Terence Parr, University of San Francisco / `http://www.antlr.org/`

In den Abbildungen dieses Kapitels werden folgende Abkürzungen verwendet:

- *CDG*: Control Dependence Graph (Kontrollabhängigkeitsgraph)
- *CFG*: Control Flow Graph (Kontrollflussgraph)
- *dCFG*: data flow annotated Control Flow Graph (datenflussannotierter Kontrollflussgraph)
- $dCFG_{cp}$: Kennzeichnet einen datenflussannotierten Kontrollflussgraphen mit der Art der Annotierung nach Rapps und Weyuker, bei der zwischen berechnenden (*c-use*) und prädikativen (*p-use*) Verwendungen unterschieden wird. Dabei ordnet man einen c-use einer Variablen demjenigen Knoten zu, welcher die verwendende Anweisung repräsentiert. Stellt ein Knoten eine Bedingungsauswertung dar, dann ordnet man allen Kanten, die diesen Knoten verlassen, jeweils ein p-use jeder in der Bedingung verwendeten Variablen zu.
- $dCFG_u$: Kennzeichnet einen datenflussannotierten Kontrollflussgraphen mit einer Annotierung, bei der nicht zwischen berechnenden und prädikativen Verwendungen unterschieden wird. Alle Verwendungen (*use*) werden lediglich demjenigen Knoten zugeordnet, welcher die verwendende Anweisung repräsentiert.
- *partial evaluation*: Ist ein Prädikat aus mehreren Teilbedingungen zusammengesetzt, so werden bei einer Auswertung nach *partial evaluation* (auch: Kurzschlussauswertung) lediglich diejenigen ausgewertet, die zur Bestimmung der gesamten Bedingung notwendig sind.

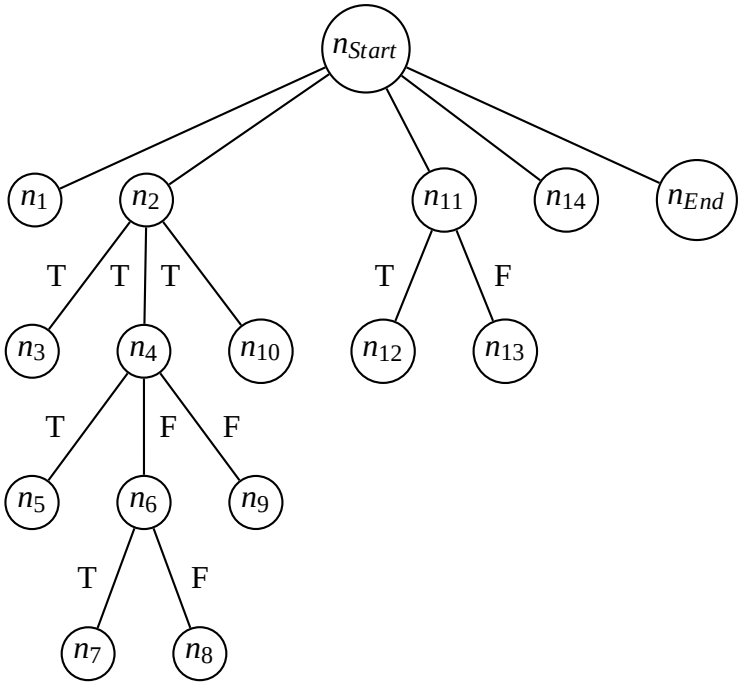


Abbildung A.1: Kontrollabhängigskeitsgraph (*CDG*) für *BinarySearch*

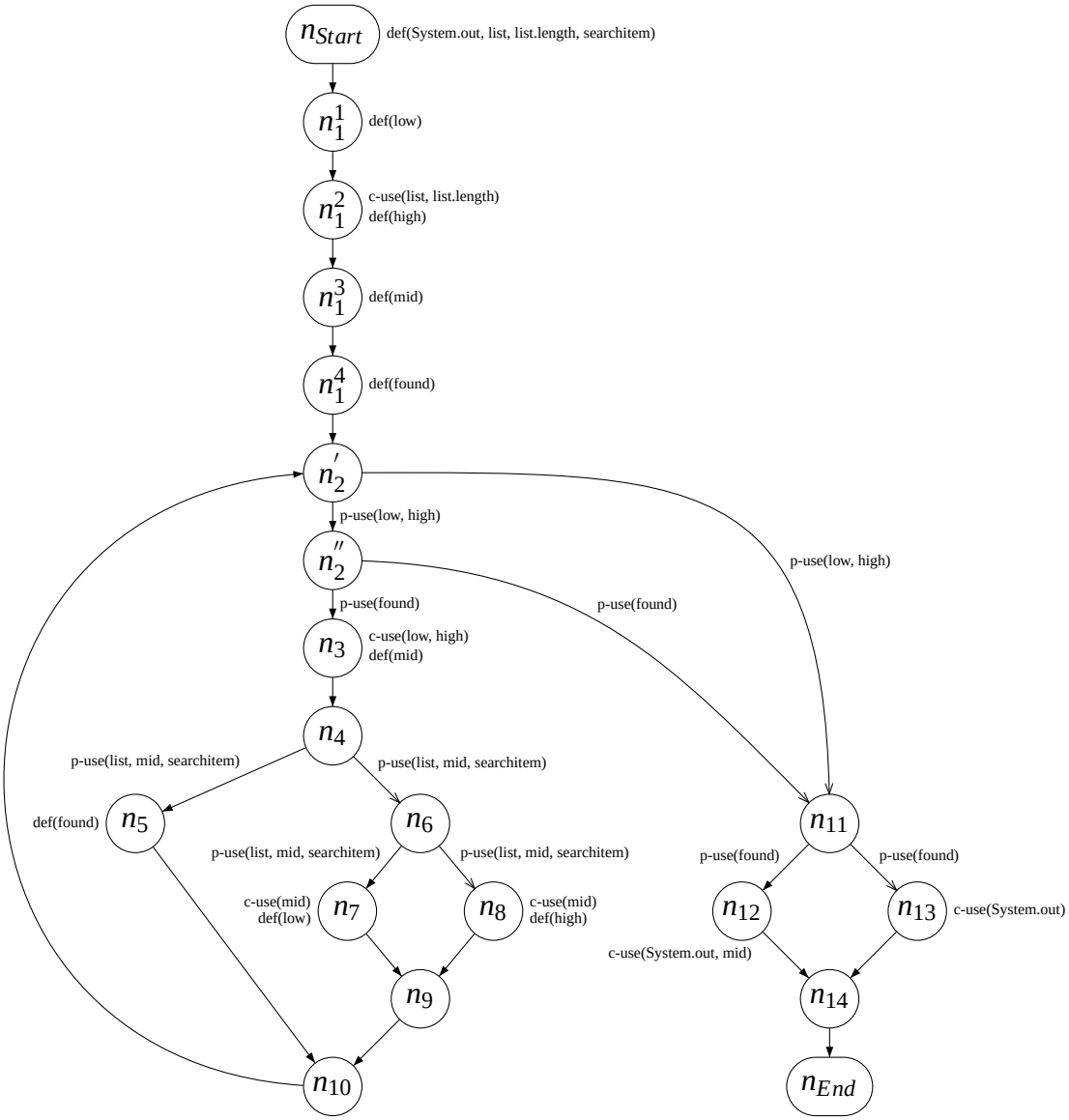


Abbildung A.2: Detaillierter $dCFG_{cp}$ für *BinarySearch* mit partial evaluation

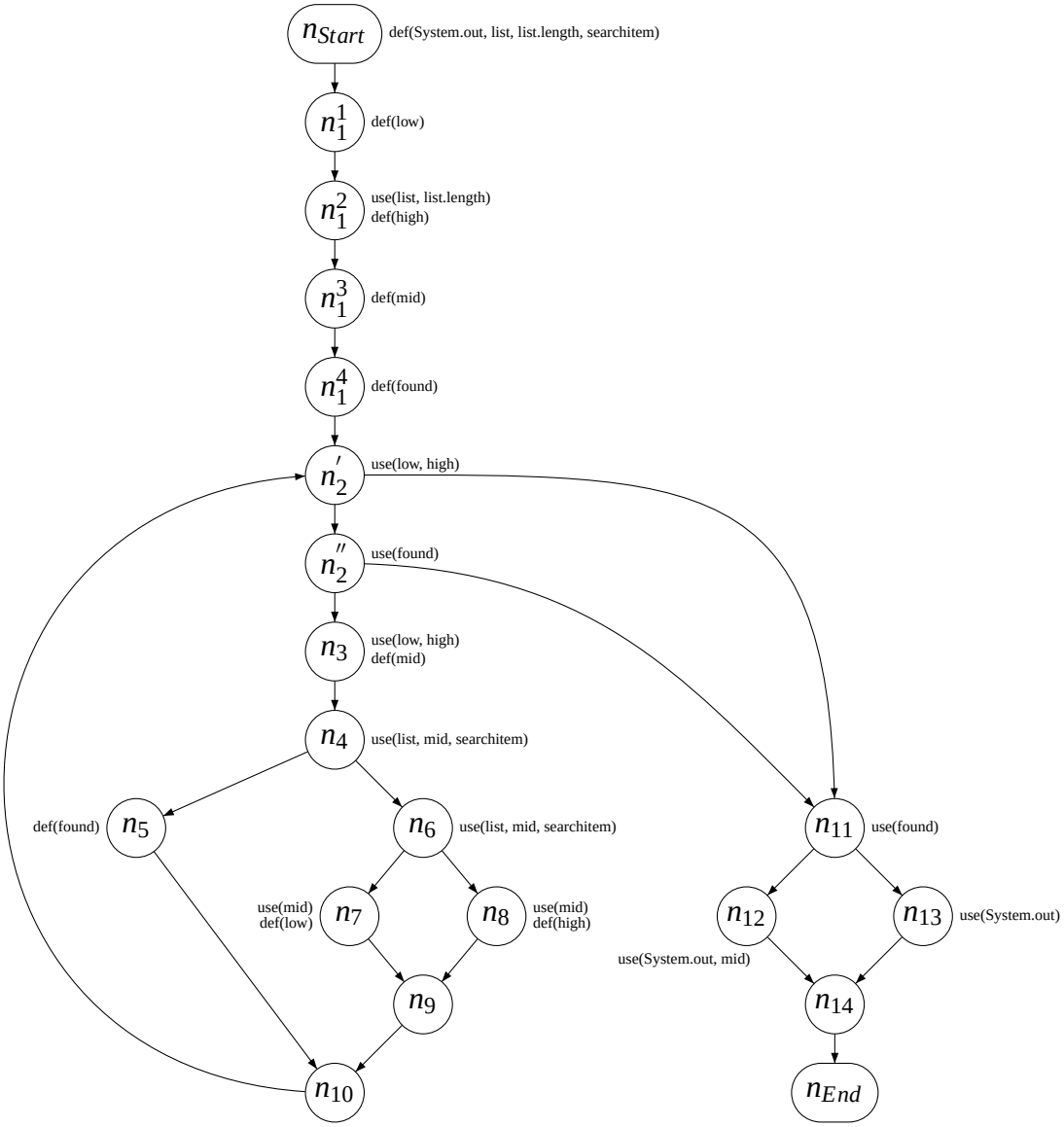


Abbildung A.3: Detaillierter $dCFG_u$ für *BinarySearch* mit partial evaluation

Abbildung A.4: Detaillierter $dCFG_u$ für *BinarySearch* ohne partial evaluation

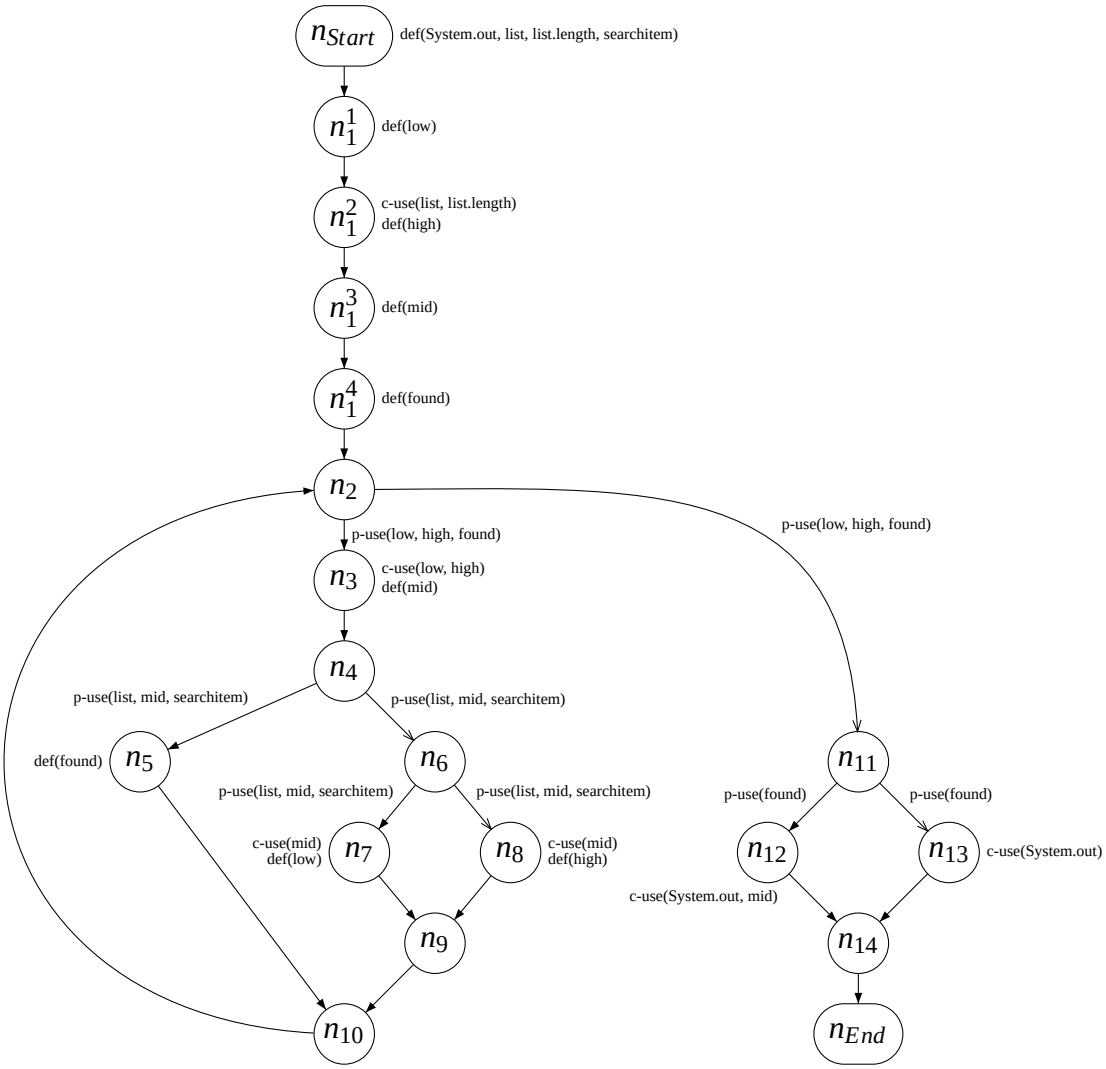


Abbildung A.5: Detaillierter $dCFG_{cp}$ für *BinarySearch* ohne partial evaluation

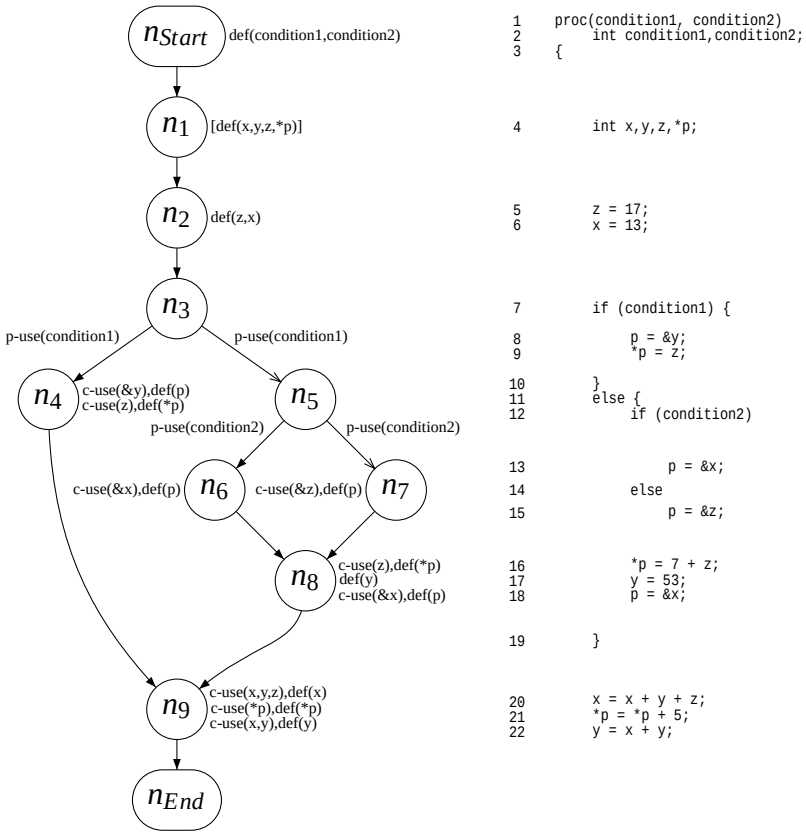


Abbildung A.6: Beispiel zu Pointer Aliasing in der Sprache C

Hillclimbing — first ascent

Wähle zufällig eine initiale Lösung $s \in S$
Bestimme die Bewertung f_s von s: $f_s \leftarrow f(s)$
Wiederholen, bis Abbruchkriterium erfüllt (z.B. s hinreichend gut oder Rechenzeit erschöpft)
Wähle zufällig eine Lösung $\bar{s} \in \mathcal{N}(s)$
Bestimme die Bewertung $f_{\bar{s}}$ von $\bar{s}$: $f_{\bar{s}} \leftarrow f(\bar{s})$
Ist die neue Lösung besser als die bisherige ($f_{\bar{s}} > f_s$)? (Ja / Nein)
Ersetze s durch $\bar{s}$: $s \leftarrow \bar{s}$ | Behalte s bei
Ausgabe der besten gefundenen Lösung s

Abbildung A.7: Hillclimbing-Variante: „first ascent“

Hillclimbing — steepest ascent

Wähle zufällig eine initiale Lösung $s \in S$
Bestimme die Bewertung f_s von s: $f_s \leftarrow f(s)$
Wiederholen, bis lokales Optimum gefunden ($\forall \hat{s} \in \mathcal{N}(s) : f(s) \geq f(\hat{s})$) oder Abbruchkriterium erfüllt (z.B. s hinreichend gut oder Rechenzeit erschöpft)
Wähle eine Lösung $\bar{s} \in \mathcal{N}(s)$ so, dass $\forall \hat{s} \in \mathcal{N}(s) : f(\bar{s}) \geq f(\hat{s})$
Ersetze s durch $\bar{s}$: $s \leftarrow \bar{s}$
Ausgabe der besten gefundenen Lösung s

Abbildung A.8: Hillclimbing-Variante: „steepest ascent“

```
1  public class DataflowExample {
2    public static int CONSTANT = 2006;
3
4    public DataflowExample fieldOne = null;
5    public int fieldTwo;
6
7    public DataflowExample() {
8      this(3);
9      try {
10       fieldTwo = 1 / 0;
11     } catch (Exception exception) {
12     }
13     fieldOne = new DataflowExample(4);
14     fieldOne.fieldTwo = fieldOne.fieldTwo + DataflowExample.CONSTANT;
15   }
16
17   public DataflowExample(int aValue) {
18     int anotherValue = 1;
19     for (int index = 2; index <= aValue; index++) {
20       anotherValue *= index;
21     }
22     fieldTwo = anotherValue;
23   }
24
25   public int getFieldTwo(int selection) {
26     switch(selection) {
27       case 0:
28         fieldTwo++;
29       case 1:
30         CONSTANT++;
31         break;
32     }
33     return fieldTwo;
34   }
35
36   public static void main(String[] args) {
37     DataflowExample de1 = new DataflowExample();
38     DataflowExample.CONSTANT = 4711;
39     DataflowExample de2 = new DataflowExample(5);
40     if (args.length > 0) {
41       de2 = de1;
42     }
43     int temp = de2.fieldTwo + de1.getFieldTwo(1);
44   }
45 }
```

Listing A.1: Nicht-instrumentierter Quellcode der Klasse *DataflowExample*

```
1  public class DataflowExample implements InstanceId{
2    public final int ___instanceId=DULog.getNewInstanceId(0);
3    public final int ___getInstanceId(){return ___instanceId;}
4
5    {DULog.enterInit(1);
6      try{___doZeroInit();}
7      finally{DULog.leaveInit(2);}
8    }
9    static{DULog.enterSInit(3);
10     try{___doStaticZeroInit();}
11     finally{DULog.leaveSInit(4);}
12   }
13
14   public static int CONSTANT;
15   static{DULog.reenterSInit(4);
16     try{CONSTANT=(int)DULog.defStatic(5,2006);}
17     finally{DULog.leaveSInit(4);}
18   }
19
20   public DataflowExample fieldOne;
21   {DULog.reenterInit(2);
22     try{fieldOne=(DataflowExample)DULog.defField(7,this,null);}
23     finally{DULog.leaveInit(2);}
24   }
25
26   public int fieldTwo;
27
28   public DataflowExample(){
29     this((int)DULog.bin(DULog.earlyCtorEnter(10),3));DULog.enter(22);
30   try{
31     try{
32       DULog.defField(11,DataflowExample.this,fieldTwo=1/0);
33     } catch(Exception exception){
34       DULog.exceptHandlerCall(13);DULog.defLocal(12);
35     }
36     DULog.defField(16,DataflowExample.this,fieldOne
37       =(DataflowExample)DULog.newCallCompleted(15,DULog.newCall(14),new DataflowExample(4)));
38     DataflowExample ___t1;DULog.defField(21,___t1
39       =(DataflowExample)DULog.useField(17,DataflowExample.this,fieldOne),___t1.fieldTwo
40       =((DataflowExample)DULog.useField(19
41         ,(DataflowExample)DULog.useField(18,DataflowExample.this,fieldOne))).fieldTwo
42           +(int)DULog.useStatic(20,DataflowExample.CONSTANT));
43   }finally{DULog.leave(23);}
44   }
45
46   public DataflowExample(int aValue){DULog.enter(36);
47   try{int anotherValue=(int)DULog.defLocal(24,1);
48     for(int index=(int)DULog.defLocal(25,2);
49       DULog.predResult(29,DULog.newPredicate(28),(int)DULog.useLocal(26,index)
50         <=(int)DULog.useLocal(27,aValue));
51       DULog.useDefLocal(30,index++)){
52
53       anotherValue*=(int)DULog.bin(DULog.useLocal(32)
54         ,DULog.defLocal(33,(int)DULog.useLocal(31,index)));
55     }
56     DULog.defField(35,DataflowExample.this,fieldTwo=(int)DULog.useLocal(34,anotherValue));

57   }finally{DULog.leave(37);}
58   }
59
60
61   public int getFieldTwo(int selection){DULog.enter(47);
62   try{switch(DULog.switchPredResult(40,DULog.newSwitchPredicate(39),
63       DULog.useLocal(38,selection))){
64     case 0:DULog.switchPredEquivalent(42,1);
```

```
65          DULog.useDefField(41,DataflowExample.this,fieldTwo++);
66        case 1:DULog.switchPredEquivalent(44,2);
67          DULog.useDefStatic(43,CONSTANT++);break;
68        default:DULog.switchPredEquivalent(45,0);}
69        return(int)DULog.useField(46,DataflowExample.this,fieldTwo);
70      }finally{DULog.leave(48);}
71      }
72
73      public static void main(String[] args){DULog.enter(67);
74      try{
75        DataflowExample de1=(DataflowExample)DULog.defLocal(51
76          ,(DataflowExample)DULog.newCallCompleted(50,DULog.newCall(49),new DataflowExample()));
77        DULog.defStatic(52,DataflowExample.CONSTANT=4711);
78        DataflowExample de2=(DataflowExample)DULog.defLocal(55
79          ,(DataflowExample)DULog.newCallCompleted(54,DULog.newCall(53),new DataflowExample(5)));
80        if(DULog.predResult(59,DULog.newPredicate(58),
81            DULog.useArrayLength(57,(java.lang.String[])DULog.useLocal(56,args))>0)){
82          DULog.defLocal(61,de2=(DataflowExample)DULog.useLocal(60,de1));}
83        int temp=(int)DULog.defLocal(66,
84          ((DataflowExample)DULog.useField(63,(DataflowExample)DULog.useLocal(62,de2))).fieldTwo
85          +((DataflowExample)DULog.useLocal(64,de1)).getFieldTwo((int)DULog.cp(65,1)));
86      }finally{DULog.leave(68);}
87      }
88
89      protected static void ___doStaticZeroInit(){DULog.defStatic(6);}
90      protected void ___doZeroInit(){DULog.defField(8,this);DULog.defField(9,this);}
91      {DULog.initCompleted(2);}
92      static{DULog.sInitCompleted(4);}
93  }
```

Listing A.2: Instrumentierter Quellcode der Klasse *DataflowExample*

ID	Ereignis	Betroffenes Element	Zusätzlich betroffene Elemente	Datei	Zeile	Spalte
1	initEnter	DataflowExample		DataflowExample.java	2	0
2	initLeave	DataflowExample		DataflowExample.java	2	0
3	clinitEnter	DataflowExample		DataflowExample.java	2	0
4	clinitLeave	DataflowExample		DataflowExample.java	2	0
5	defStatic	public static int DataflowExample.CONSTANT		DataflowExample.java	2	0
6	defStatic	public static int DataflowExample.CONSTANT		DataflowExample.java	2	0
7	defField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java	4	0
8	defField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java	4	0
9	defField	public int DataflowExample.fieldTwo		DataflowExample.java	5	0
10	earlyCtorEnter	public DataflowExample()		DataflowExample.java	7	0
11	defField	public int DataflowExample.fieldTwo		DataflowExample.java	10	25
12	defLocal	java.lang.Exception exception		DataflowExample.java	11	0
13	exceptHandlerCall	-		DataflowExample.java	11	19
14	newCall	DataflowExample		DataflowExample.java	13	28
15	newCallCompleted	DataflowExample		DataflowExample.java	13	28
16	defField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java	13	17
17	useField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java	14	17
18	useField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java	14	37
19	useField	public int DataflowExample.fieldTwo		DataflowExample.java	14	45
20	useStatic	public static int DataflowExample.CONSTANT		DataflowExample.java	14	72
21	defField	public int DataflowExample.fieldTwo		DataflowExample.java	14	25
22	ctorEnter	public DataflowExample()		DataflowExample.java	7	0
23	ctorLeave	public DataflowExample()		DataflowExample.java	7	0
24	defLocal	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java	18	0
25	defLocal	int DataflowExample.DataflowExample(int).index		DataflowExample.java	19	0
26	useLocal	int DataflowExample.DataflowExample(int).index		DataflowExample.java	19	37
27	useLocal	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java	19	46
28	newPredicate	-		DataflowExample.java	19	17
29	predResult	-		DataflowExample.java	19	17
30	useDefLocal	int DataflowExample.DataflowExample(int).index		DataflowExample.java	19	54
31	useLocal	int DataflowExample.DataflowExample(int).index		DataflowExample.java	20	41
32	useLocal	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java	20	25
33	defLocal	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java	20	25
34	useLocal	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java	22	28
35	defField	public int DataflowExample.fieldTwo		DataflowExample.java	22	17
36	ctorEnter	public DataflowExample(int)	int DataflowExample.DataflowExample(int).aValue	DataflowExample.java	17	0
37	ctorLeave	public DataflowExample(int)		DataflowExample.java	17	0
38	useLocal	int DataflowExample.getFieldTwo(int).selection		DataflowExample.java	26	24
39	newSwitchPredicate	-		DataflowExample.java	26	17
40	switchPredResult	-		DataflowExample.java	26	17
41	useDefField	public int DataflowExample.fieldTwo		DataflowExample.java	28	33
42	switchPredEquivalent	-	1	DataflowExample.java	26	17
43	useDefStatic	public static int DataflowExample.CONSTANT		DataflowExample.java	30	33
44	switchPredEquivalent	-	2	DataflowExample.java	26	17
45	switchPredEquivalent	-	0	DataflowExample.java	26	17
46	useField	public int DataflowExample.fieldTwo		DataflowExample.java	33	24
47	enter	public int DataflowExample.getFieldTwo(int)	int DataflowExample.getFieldTwo(int).selection	DataflowExample.java	25	0
48	leave	public int DataflowExample.getFieldTwo(int)		DataflowExample.java	25	0
49	newCall	DataflowExample		DataflowExample.java	37	39
50	newCallCompleted	DataflowExample		DataflowExample.java	37	39
51	defLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java	37	0
52	defStatic	public static int DataflowExample.CONSTANT		DataflowExample.java	38	32
53	newCall	DataflowExample		DataflowExample.java	39	39
54	newCallCompleted	DataflowExample		DataflowExample.java	39	39
55	defLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		DataflowExample.java	39	0
56	useLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java	40	28
57	useArrayLength	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args		DataflowExample.java	40	21
58	newPredicate	-		DataflowExample.java	40	17
59	predResult	-		DataflowExample.java	40	17
60	useLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java	41	31
61	defLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		DataflowExample.java	41	25
62	useLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		DataflowExample.java	43	28
63	useField	public int DataflowExample.fieldTwo		DataflowExample.java	43	31
64	useLocal	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java	43	43
65	cp	public int DataflowExample.getFieldTwo(int)		DataflowExample.java	43	58
66	defLocal	int DataflowExample.main([Ljava.lang.String;).temp		DataflowExample.java	43	0
67	enter	public static void DataflowExample.main(java.lang.String[])	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args	DataflowExample.java	36	0
68	leave	public static void DataflowExample.main(java.lang.String[])		DataflowExample.java	36	0

Tabelle A.1: Instrumentierungsprotokoll der Klasse *DataflowExample*

```
1 public class DefaultInitExample {
2   public static int sField = initStaticField();
3   public int field = initField(this);
4
5   public static int initStaticField() {
6     System.out.print("␣sInit1:" + DefaultInitExample.sField);
7     return 91058;
8   }
9
10  public int initField(DefaultInitExample defaultInitExample) {
11    System.out.print("␣fInit1:" + defaultInitExample.field);
12    return 91126;
13  }
14
15  public static void main(String[] args) {
16    DefaultInitExample defaultInitExample = new DefaultInitExample();
17    System.out.print("␣sInit2:" + DefaultInitExample.sField);
18    System.out.print("␣fInit2:" + defaultInitExample.field);
19  }
20 }
```

Listing A.3: Beispiel zum Nachweis der doppelten Initialisierung von Feldern

```
1 public class PredicateExample {
2   public static void main(String[] args) {
3     if (MainProgramExample.mainFunction()) {
4       System.out.println("Application␣finished␣successfully.");
5     } else {
6       System.err.println("Application␣aborted␣abnormally!");
7     }
8   }
9 }
```

Listing A.4: Codebeispiel zur Beleuchtung des Problems prädikativer Verwendungen

```
1  public class ConditionCoverageExample {
2    public ConditionCoverageExample(boolean b) {
3      this(g() ? f() : 2);
4    }
5
6    public static void main(String[] args) {
7      int a = Integer.parseInt(args[0]);
8      int b = Integer.parseInt(args[1]);
9      int c = Integer.parseInt(args[2]);
10     boolean d = (Boolean.valueOf(args[3])).booleanValue();
11     if (a < 10 && b > 0 || c > 10 && d) {
12       System.out.println("Non-atomic: TRUE");
13     } else {
14       System.out.println("Non-atomic: FALSE");
15     }
16     switch(a) {
17       case 0:
18         System.out.println("Switch: 0");
19       case 2:
20         System.out.println("Switch: 2");
21         break;
22       case 3:
23         System.out.println("Switch: 3");
24     }
25     for (int i = 1; i < 7; i++) {
26       b = (i == 1 ? 1 : d ? 0 : i == 3 ? 3 : 7);
27       System.out.println("Ternary: " + b);
28     }
29     A x = new A();
30     A y = new B();
31     B z = new B();
32     overloaded(x);
33     overloaded(z);
34     x.polymorphic();
35     y.polymorphic();
36   }
37
38   public static void overloaded(A a) {
39     System.out.println("Overloaded: A");
40   }
41   public static void overloaded(B b) {
42     System.out.println("Overloaded: B");
43   }
44   public ConditionCoverageExample(int i) {}
45   public static int f() {return 1;}
46   public static boolean g() {return true;}
47 }
48
49 class A {
50   public void polymorphic() {
51     System.out.println("Polymorphic: A");
52   }
53 }
54
55 class B extends A {
56   public void polymorphic() {
57     System.out.println("Polymorphic: B");
58   }
59 }
```

Listing A.5: Nicht-instrumentierter Quellcode des Beispiels *ConditionCoverageExample*

```
1  public class ConditionCoverageExample {
2    public ConditionCoverageExample( boolean b ) {
3      this((Logger._catchException(Logger._startExpression(2),
4        Logger._logB(Logger._logB(g(), 1), 2),
5        Logger._endExpression(2))) ? (f()) : (2));
6      Logger._enterMethod(3);
7      try {
8      } finally {
9        Logger._handleException(10); Logger._leftMethod(3);
10     }
11   }
12
13   public static void main(String [] args) {
14     Logger._enterMethod(24);
15     try {
16       int a = Integer.parseInt(args[0]);
17       int b = Integer.parseInt(args[1]);
18       int c = Integer.parseInt(args[2]);
19       boolean d = Boolean.valueOf(args[3]).booleanValue();
20       if (Logger._catchException(
21         Logger._startExpression(11),
22           Logger._logB(
23             Logger._logB(
24               (
25                 Logger._logB(
26                   (
27                     Logger._logB(a < 10, 4)
28                     &&
29                     Logger._logB(b > 0, 5)
30                   )
31                 , 6)
32                 ||
33                 Logger._logB(
34                   (
35                     Logger._logB(c > 10, 7)
36                     &&
37                     Logger._logB(d, 8)
38                   )
39                 , 9)
40               )
41             , 10)
42           , 11)
43         , Logger._endExpression(11))) {
44         System.out.println( "Non-atomic: TRUE" );
45       } else {
46         System.out.println( "Non-atomic: FALSE" );
47       }
48       switch (Logger._enterSwitch(a, 12)) {
49         case 0:
50           Logger._logC(13);
51           System.out.println("Switch: 0");
52         case 2:
53           Logger._logC(14);
54           System.out.println("Switch: 2");
55           break;
56         case 3:
57           Logger._logC(15);
58           System.out.println("Switch: 3");
59       }
60       Logger._leftSwitch(12);
61       for (int i = 1; Logger._catchException(Logger._startExpression(17),
62           Logger._logB(Logger._logB(i < 7, 16), 17),
63           Logger._endExpression(17)); i++ ) {
64         b =  (Logger._catchException(Logger._startExpression(23),
65           Logger._logB(Logger._logB(i == 1, 18), 23),
```

```
66            Logger._endExpression(23))) ? (1)
67            : ((Logger._catchException(Logger._startExpression( 22 ),
68            Logger._logB(Logger._logB(d, 19), 22),
69            Logger._endExpression(22))) ? (0)
70            : ((Logger._catchException(Logger._startExpression( 21 ),
71            Logger._logB(Logger._logB(i == 3, 20), 21),
72            Logger._endExpression(21))) ? (3) : (7))) ;
73          System.out.println("Ternary: " + b);
74        }
75        A x = new A ();
76        A y = new B ();
77        B z = new B ();
78        overloaded(x);
79        overloaded(z);
80        x.polymorphic();
81        y.polymorphic();
82      } finally {
83        Logger._handleException(63);
84        Logger._leftMethod(24);
85      }
86    }
87
88    public static void overloaded(A a) {
89      Logger._enterMethod(25);
90      try {System.out.println("Overloaded: A");
91      } finally {Logger._handleException(74); Logger._leftMethod(25);
92    } }
93    public static void overloaded(B b) {
94      Logger._enterMethod(26);
95      try {System.out.println("Overloaded: B");
96      } finally {Logger._handleException(85); Logger._leftMethod(26);
97    } }
98    public ConditionCoverageExample(int i) {
99      Logger._enterMethod(27);
100     try {
101     } finally {Logger._handleException(95); Logger._leftMethod(27);
102   } }
103   public static int f() {
104     Logger._enterMethod(28);
105     try {return 1;
106     } finally {Logger._handleException(105); Logger._leftMethod(28);
107   } }
108   public static boolean g() {
109     Logger._enterMethod(29);
110     try {return true;
111     } finally {Logger._handleException(115); Logger._leftMethod(29);
112 } } }
113
114 class A {
115   public void polymorphic() {
116     Logger._enterMethod(31);
117     try {System.out.println("Polymorphic: A");
118     } finally {Logger._handleException(129); Logger._leftMethod(31);
119 } } }
120
121 class B extends A {
122   public void polymorphic() {
123     Logger._enterMethod(33);
124     try {System.out.println("Polymorphic: B");
125     } finally {Logger._handleException(143); Logger._leftMethod(33);
126 } } }
```

Listing A.6: Instrumentierter Quellcode des Beispiels *ConditionCoverageExample*

ID	*Zeile*	*Typ*	*Ausdruck*
30	1	ke	ConditionCoverageExample
1	5	p	g()
2	5	b	g() ? f() : 2
24	17	m	main (String[] args)
4	22	a	(a < 10)
5	22	a	(b > 0)
6	22	c	(a < 10) && (b > 0)
7	22	a	(c > 10)
8	22	p	d
9	22	c	(c > 10) && d
10	22	c	(a < 10) && (b > 0) || (c > 10) && d
11	22	b	(a < 10) && (b > 0) || (c > 10) && d
12	30	w	a
13	31	s	case: 0
14	36	s	case: 2
15	41	s	case: 3
16	49	a	(i < 7)
17	49	b	(i < 7)
18	50	a	(i == 1)
23	50	b	(i == 1) ? 1 : d ? 0 : (i == 3) ? 3 : 7
19	50	p	d
22	50	b	d ? 0 : (i == 3) ? 3 : 7
20	50	a	(i == 3)
21	50	b	(i == 3) ? 3 : 7
25	70	m	overloaded (A a)
26	81	m	overloaded (B b)
27	93	o	ConditionCoverageExample (int i)
28	102	m	f ( )
29	112	m	g ( )
30	120	kl	ConditionCoverageExample
32	120	ke	A
31	125	m	polymorphic ( )
32	134	kl	A
34	134	ke	B
33	139	m	polymorphic ( )
34	148	kl	B

Tabelle A.2: Instrumentierungsprotokoll des Beispiels *ConditionCoverageExample*

Tabelle A.3: Ausführungsprotokoll des Beispiels *ConditionCoverageExample*

	Ausführungsprotokoll				Aufbereitung mit Daten aus dem Instrumentierungsprotokoll			
Nr.	*ID*	*T*	*I*	*D*	*Ereignis*	*Betroffenes Element*	*Zusätzlich betroffene Elemente*	*Datei, Zeile, Spalte*
1	-1	1	0	0	NewThread			
2	-1	2	0	1	NewThread			
3	3	2	0	0	EnterClassInitialisation	DataflowExample		DataflowExample.java, 2, 0
4	6	2	0	0	DefineStaticVariable	public static int DataflowExample.CONSTANT		DataflowExample.java, 2, 0
5	5	2	0	0	DefineStaticVariable	public static int DataflowExample.CONSTANT		DataflowExample.java, 2, 0
6	4	2	0	0	LeaveClassInitialisation	DataflowExample		DataflowExample.java, 2, 0
7	67	1	0	0	EnterMethod	public static void DataflowExample.main(java.lang.String[])	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args	DataflowExample.java, 36, 0
8	49	1	0	0	NewCall	DataflowExample		DataflowExample.java, 37, 39
9	10	1	0	0	EarlyConstructorEnter	public DataflowExample()		DataflowExample.java, 7, 0
10	-2	1	1	0	NewInstance			
11	1	1	0	0	EnterInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
12	8	1	1	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
13	9	1	1	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 5, 0
14	7	1	1	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
15	2	1	0	0	LeaveInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
16	36	1	0	0	EnterConstructor	public DataflowExample(int)	int DataflowExample.DataflowExample(int).aValue	DataflowExample.java, 17, 0
17	24	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 18, 0
18	25	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 0
19	28	1	1	0	NewPredicate			DataflowExample.java, 19, 17
20	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
21	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
22	29	1	1	1	PredicateResult			DataflowExample.java, 19, 17
23	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
24	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
25	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
26	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
27	28	1	2	0	NewPredicate			DataflowExample.java, 19, 17
28	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
29	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
30	29	1	2	1	PredicateResult			DataflowExample.java, 19, 17
31	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
32	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
33	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
34	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
35	28	1	3	0	NewPredicate			DataflowExample.java, 19, 17
36	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
37	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
38	29	1	3	0	PredicateResult			DataflowExample.java, 19, 17
39	34	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 22, 28
40	35	1	1	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 22, 17
41	37	1	0	0	LeaveConstructor	public DataflowExample(int)		DataflowExample.java, 17, 0
42	22	1	0	0	EnterConstructor	public DataflowExample()		DataflowExample.java, 7, 0
43	13	1	0	0	ExceptionHandlerCall			-, 11, 19
44	12	1	0	0	DefineLocalVariable	java.lang.Exception exception		DataflowExample.java, 11, 0
45	14	1	0	0	NewCall	DataflowExample		DataflowExample.java, 13, 28
46	-2	1	2	0	NewInstance			
47	1	1	0	0	EnterInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
48	8	1	2	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
49	9	1	2	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 5, 0
50	7	1	2	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
51	2	1	0	0	LeaveInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
52	36	1	0	0	EnterConstructor	public DataflowExample(int)	int DataflowExample.DataflowExample(int).aValue	DataflowExample.java, 17, 0
53	24	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 18, 0
54	25	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 0
55	28	1	4	0	NewPredicate			DataflowExample.java, 19, 17
56	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
57	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
58	29	1	4	1	PredicateResult			DataflowExample.java, 19, 17
59	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
60	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
61	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25

Tabelle A.3 – Fortsetzung

	Ausführungsprotokoll				Aufbereitung mit Daten aus dem Instrumentierungsprotokoll			
Nr.	*ID*	*T*	*I*	*D*	*Ereignis*	*Betroffenes Element*	*Zusätzlich betroffene Elemente*	*Datei, Zeile, Spalte*
62	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
63	28	1	5	0	NewPredicate			DataflowExample.java, 19, 17
64	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
65	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
66	29	1	5	1	PredicateResult			DataflowExample.java, 19, 17
67	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
68	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
69	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
70	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
71	28	1	6	0	NewPredicate			DataflowExample.java, 19, 17
72	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
73	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
74	29	1	6	1	PredicateResult			DataflowExample.java, 19, 17
75	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
76	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
77	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
78	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
79	28	1	7	0	NewPredicate			DataflowExample.java, 19, 17
80	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
81	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
82	29	1	7	0	PredicateResult			DataflowExample.java, 19, 17
83	34	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 22, 28
84	35	1	2	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 22, 17
85	37	1	0	0	LeaveConstructor	public DataflowExample(int)		DataflowExample.java, 17, 0
86	15	1	2	0	NewCallCompleted	DataflowExample		DataflowExample.java, 13, 28
87	16	1	1	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 13, 17
88	17	1	1	0	UseField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 14, 17
89	18	1	1	0	UseField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 14, 37
90	19	1	2	0	UseField	public int DataflowExample.fieldTwo		DataflowExample.java, 14, 45
91	20	1	0	0	UseStaticVariable	public static int DataflowExample.CONSTANT		DataflowExample.java, 14, 72
92	21	1	2	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 14, 25
93	23	1	0	0	LeaveConstructor	public DataflowExample()		DataflowExample.java, 7, 0
94	50	1	1	0	NewCallCompleted	DataflowExample		DataflowExample.java, 37, 39
95	51	1	0	0	DefineLocalVariable	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java, 37, 0
96	52	1	0	0	DefineStaticVariable	public static int DataflowExample.CONSTANT		DataflowExample.java, 38, 32
97	53	1	0	0	NewCall	DataflowExample		DataflowExample.java, 39, 39
98	-2	1	3	0	NewInstance			
99	1	1	0	0	EnterInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
100	8	1	3	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
101	9	1	3	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 5, 0
102	7	1	3	0	DefineField	public DataflowExample DataflowExample.fieldOne		DataflowExample.java, 4, 0
103	2	1	0	0	LeaveInstanceInitialisation	DataflowExample		DataflowExample.java, 2, 0
104	36	1	0	0	EnterConstructor	public DataflowExample(int)	int DataflowExample.DataflowExample(int).aValue	DataflowExample.java, 17, 0
105	24	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 18, 0
106	25	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 0
107	28	1	8	0	NewPredicate			DataflowExample.java, 19, 17
108	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
109	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
110	29	1	8	1	PredicateResult			DataflowExample.java, 19, 17
111	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
112	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
113	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
114	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
115	28	1	9	0	NewPredicate			DataflowExample.java, 19, 17
116	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
117	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
118	29	1	9	1	PredicateResult			DataflowExample.java, 19, 17
119	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
120	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
121	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
122	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
123	28	1	10	0	NewPredicate			DataflowExample.java, 19, 17

Tabelle A.3 – Fortsetzung

	Ausführungsprotokoll				Aufbereitung mit Daten aus dem Instrumentierungsprotokoll			
Nr.	*ID*	*T*	*I*	*D*	*Ereignis*	*Betroffenes Element*	*Zusätzlich betroffene Elemente*	*Datei, Zeile, Spalte*
124	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
125	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
126	29	1	10	1	PredicateResult			DataflowExample.java, 19, 17
127	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
128	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
129	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
130	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
131	28	1	11	0	NewPredicate			DataflowExample.java, 19, 17
132	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
133	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
134	29	1	11	1	PredicateResult			DataflowExample.java, 19, 17
135	32	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
136	31	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 20, 41
137	33	1	0	0	DefineLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 20, 25
138	30	1	0	0	UseDefineLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 54
139	28	1	12	0	NewPredicate			DataflowExample.java, 19, 17
140	26	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).index		DataflowExample.java, 19, 37
141	27	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).aValue		DataflowExample.java, 19, 46
142	29	1	12	0	PredicateResult			DataflowExample.java, 19, 17
143	34	1	0	0	UseLocalVariable	int DataflowExample.DataflowExample(int).anotherValue		DataflowExample.java, 22, 28
144	35	1	3	0	DefineField	public int DataflowExample.fieldTwo		DataflowExample.java, 22, 17
145	37	1	0	0	LeaveConstructor	public DataflowExample(int)		DataflowExample.java, 17, 0
146	54	1	3	0	NewCallCompleted	DataflowExample		DataflowExample.java, 39, 39
147	55	1	0	0	DefineLocalVariable	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		DataflowExample.java, 39, 0
148	58	1	13	0	NewPredicate			DataflowExample.java, 40, 17
149	56	1	0	0	UseLocalVariable	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args		DataflowExample.java, 40, 21
150	-2	1	4	0	NewInstance			
151	57	1	4	-1	UseArrayLength	[Ljava.lang.String; DataflowExample.main([Ljava.lang.String;).args		DataflowExample.java, 40, 21
152	59	1	13	0	PredicateResult			DataflowExample.java, 40, 17
153	62	1	0	0	UseLocalVariable	DataflowExample DataflowExample.main([Ljava.lang.String;).de2		DataflowExample.java, 43, 28
154	63	1	3	0	UseField	public int DataflowExample.fieldTwo		DataflowExample.java, 43, 31
155	64	1	0	0	UseLocalVariable	DataflowExample DataflowExample.main([Ljava.lang.String;).de1		DataflowExample.java, 43, 43
156	65	1	0	0	CallPoint	public int DataflowExample.getFieldTwo(int)		DataflowExample.java, 43, 58
157	47	1	0	0	EnterMethod	public int DataflowExample.getFieldTwo(int)	int DataflowExample.getFieldTwo(int).selection	DataflowExample.java, 25, 0
158	39	1	14	0	NewSwitchPredicate			DataflowExample.java, 26, 17
159	38	1	0	0	UseLocalVariable	int DataflowExample.getFieldTwo(int).selection		DataflowExample.java, 26, 24
160	40	1	14	1	SwitchPredicateResult			DataflowExample.java, 26, 17
161	44	1	0	2	SwitchPredicateEquivalent		2	DataflowExample.java, 26, 17
162	43	1	0	0	UseDefineStaticVariable	public static int DataflowExample.CONSTANT		DataflowExample.java, 30, 33
163	46	1	1	0	UseField	public int DataflowExample.fieldTwo		DataflowExample.java, 33, 24
164	48	1	0	0	LeaveMethod	public int DataflowExample.getFieldTwo(int)		DataflowExample.java, 25, 0
165	66	1	0	0	DefineLocalVariable	int DataflowExample.main([Ljava.lang.String;).temp		DataflowExample.java, 43, 0
166	68	1	0	0	LeaveMethod	public static void DataflowExample.main(java.lang.String[])		DataflowExample.java, 36, 0
167	0	0	0	0	EndOfLog			

ID: Ereigniskennung
T: Threadkennung
I: Instanzkennung
D: Zusätzliche Informationen

Nr.	Variablen-ID Symbolischer Wert Def-Site	Use-Site
1	this(#0) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.<init>()V#-1	PointerAliasingExample.<init>()V#0
2	this(#0) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14 PointerAliasingExample.<init>()V#-1	PointerAliasingExample.<init>()V#0
3	<noname>(#2) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.main([Ljava/lang/String;)V#30	PointerAliasingExample.main([Ljava/lang/String;)V#31
4	<noname>(#1) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.main([Ljava/lang/String;)V#7	PointerAliasingExample.main([Ljava/lang/String;)V#8
5	<noname>(#2) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14 PointerAliasingExample.main([Ljava/lang/String;)V#21	PointerAliasingExample.main([Ljava/lang/String;)V#37
6	[PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14,PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0].i int PointerAliasingExample.<init>()V#6	PointerAliasingExample.main([Ljava/lang/String;)V#39
7	<noname>(#1) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.main([Ljava/lang/String;)V#7	PointerAliasingExample.main([Ljava/lang/String;)V#22
8	<noname>(#1) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.main([Ljava/lang/String;)V#7	PointerAliasingExample.main([Ljava/lang/String;)V#29
9	<noname>(#2) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.main([Ljava/lang/String;)V#30	PointerAliasingExample.main([Ljava/lang/String;)V#37
10	[PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14,PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0].i int PointerAliasingExample.main([Ljava/lang/String;)V#11	PointerAliasingExample.main([Ljava/lang/String;)V#39
11	this(#0) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0 PointerAliasingExample.<init>()V#-1	PointerAliasingExample.<init>()V#4
12	this(#0) PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14 PointerAliasingExample.<init>()V#-1	PointerAliasingExample.<init>()V#4
13	PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0.i int PointerAliasingExample.main([Ljava/lang/String;)V#11	PointerAliasingExample.main([Ljava/lang/String;)V#23
14	[PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#14,PointerAliasingExample:PointerAliasingExample.main([Ljava/lang/String;)V#0].i int PointerAliasingExample.main([Ljava/lang/String;)V#34	PointerAliasingExample.main([Ljava/lang/String;)V#39

Tabelle A.4: DU-Paare aufgrund statischer Analyse des Beispiels *PointerAliasingExample*

```
1  public class DataflowExampleTESTDRIVER {
2    private static String[] testDataStream = new String[0];
3    private static int testDataStreamPosition = 0;
4
5    public static void main(String[] args) {
6      try {
7        testDataStream = args;
8        int nextCommand, arrayIndex;
9
10       int[] arraySizes = new int[1];
11       for (int i = 0; i < 1; i++) {
12         arraySizes[i] = (int)(Long.parseLong(getNextTestData()));
13       }
14
15       String[] stringData = new String[6];
16       for (int i = 0; i < 6; i++) {
17         stringData[i] = getNextTestData();
18       }
19
20       int[] intData = new int[2];
21       for (int i = 0; i < 2; i++) {
22         intData[i] = (int)(Long.parseLong(getNextTestData()));
23       }
24
25       java.lang.String[][] var_1 = new java.lang.String[1][];
26       DataflowExample[] var_2 = new DataflowExample[1];
27       int[] var_3 = new int[2];
28
29       while(true) {
30         nextCommand = Integer.parseInt(getNextTestData());
31         try {
32           switch(nextCommand) {
33             case 0:
34               var_1 = new java.lang.String[1][arraySizes[0]];
35               break;
36             case 1:
37               arrayIndex = 0;
38               for (int i0 = 0; i0 < 1; i0++) {
39                 var_3[i0] = intData[0 + arrayIndex];
40                 arrayIndex++;
41               }
42               break;
43             case 2:
44               arrayIndex = 0;
45               for (int i0 = 0; i0 < 1; i0++) {
46                 for (int i1 = 0; i1 < arraySizes[0]; i1++) {
47                   var_1[i0][i1] = stringData[1 + arrayIndex];
48                   arrayIndex++;
49                 }
50               }
51               break;
52             case 3:
53               var_2[0] = new DataflowExample();
54               break;
55             case 4:
56               var_2[0] = new DataflowExample(var_3[0]);
57               break;
58             case 5:
59               DataflowExample.main(var_1[0]);
60               break;
61             case 6:
62               var_3[0] = var_2[0].getFieldTwo(var_3[1]);
63               break;
64             case 7:
65               rotateObjects(var_3);
```

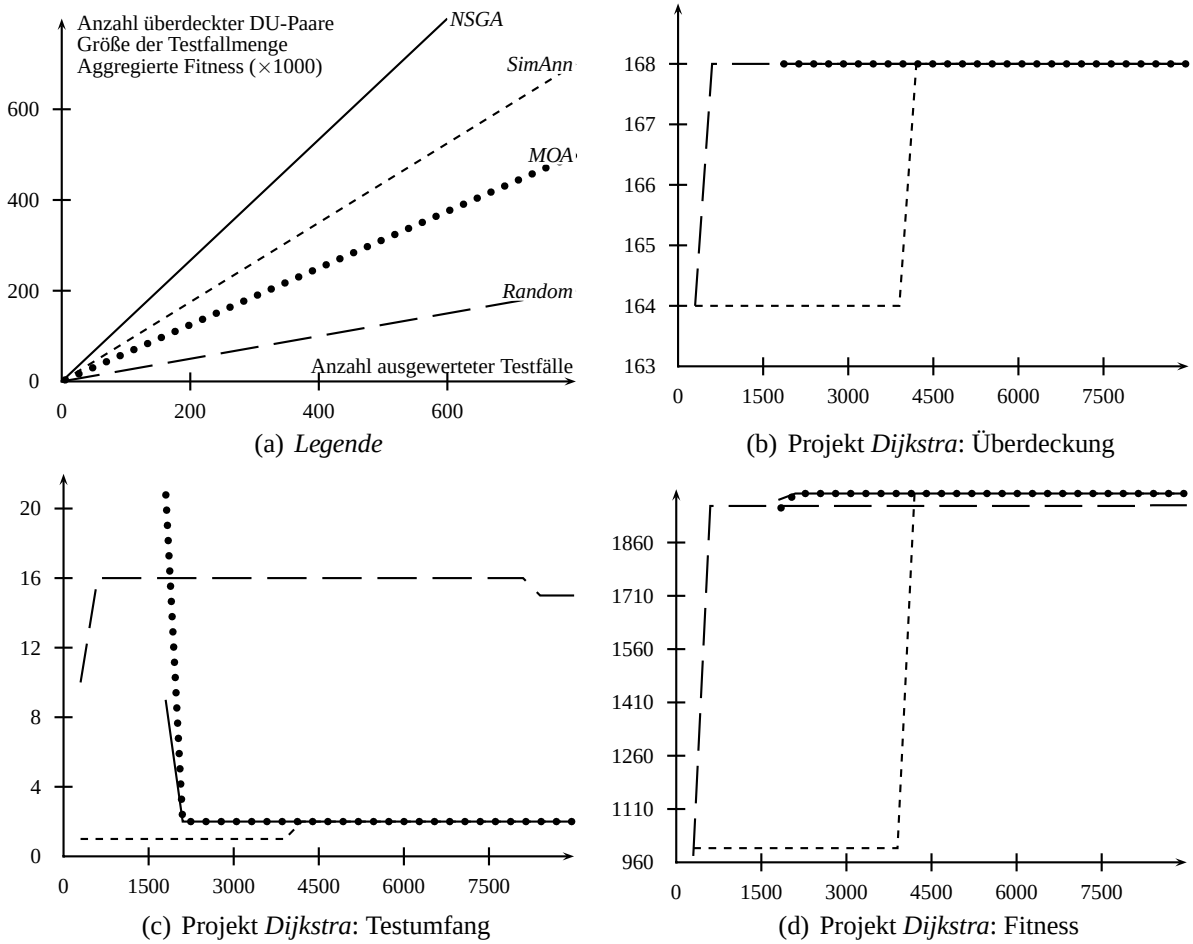
```
66                  break;
67                default:
68              }
69            } catch (Throwable t) {
70            }
71          }
72        } catch (Throwable t) {
73          System.exit(0);
74        }
75      }
76
77      private static String getNextTestData() {
78        if (testDataStreamPosition < testDataStream.length) {
79          return testDataStream[testDataStreamPosition++];
80        } else {
81          System.exit(0);
82          return null;
83        }
84      }
85
86      private static void rotateObjects(Object[] objects) {
87        Object temp = objects[0];
88        for (int objectIndex = 0; objectIndex < objects.length - 1; objectIndex++) {
89          objects[objectIndex] = objects[objectIndex + 1];
90        }
91        objects[objects.length - 1] = temp;
92      }
93
94      private static void rotateObjects(int[] objects) {
95        int temp = objects[0];
96        for (int objectIndex = 0; objectIndex < objects.length - 1; objectIndex++) {
97          objects[objectIndex] = objects[objectIndex + 1];
98        }
99        objects[objects.length - 1] = temp;
100     }
101 }
```

Listing A.7: Automatisch generierter Testtreiber für die Klasse *DataflowExample*

Anhang B

Weitere experimentelle Ergebnisse

„The real problem is not whether machines think but whether men do.“
Burrhus Frederic Skinner



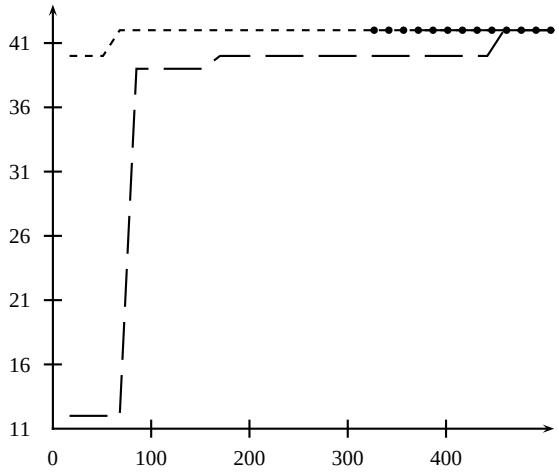
(a) *Legende*

(b) Projekt *Dijkstra*: Überdeckung

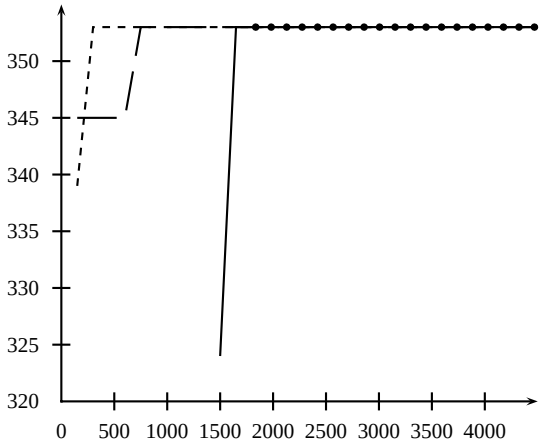
(c) Projekt *Dijkstra*: Testumfang

(d) Projekt *Dijkstra*: Fitness

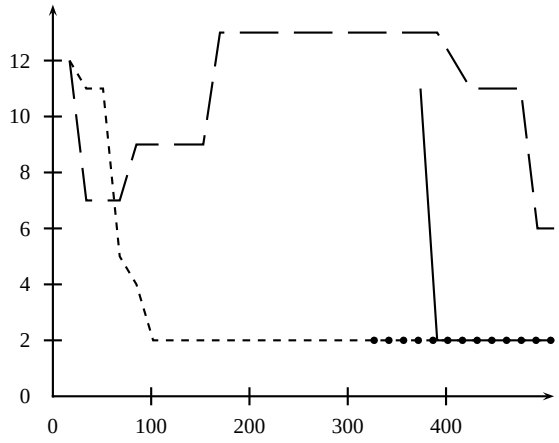
Abbildung B.1: Vergleich der Optimierungsverfahren



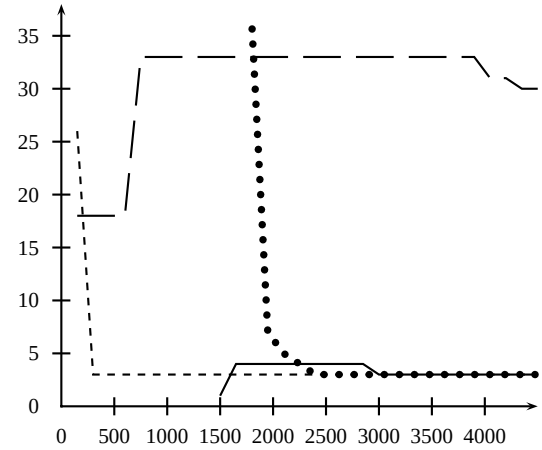
(e) Projekt *Hanoi*: Überdeckung



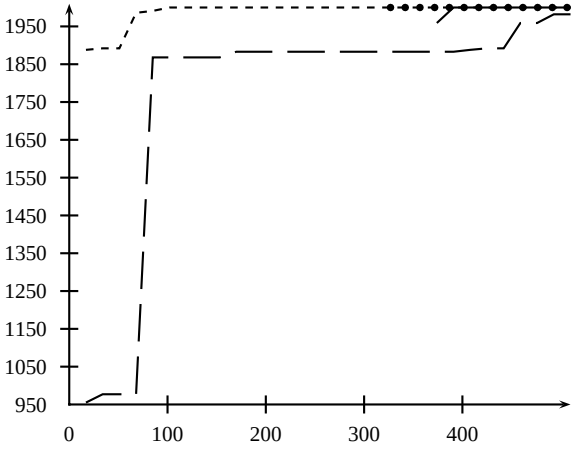
(f) Projekt *Huffman*: Überdeckung



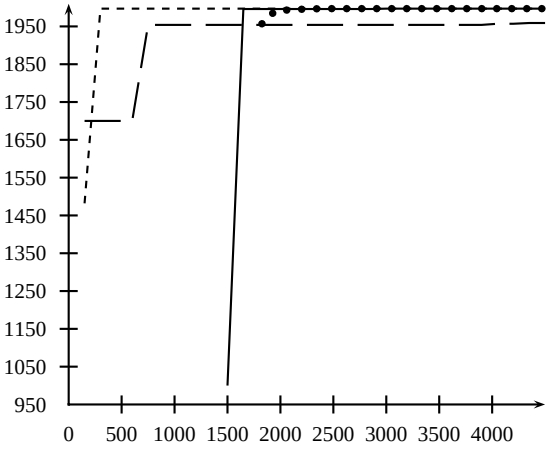
(g) Projekt *Hanoi*: Testumfang



(h) Projekt *Huffman*: Testumfang

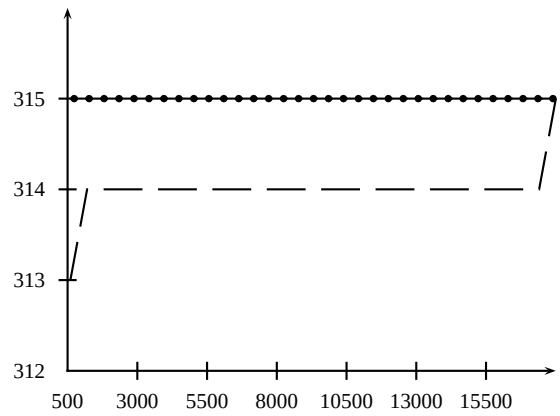


(i) Projekt *Hanoi*: Fitness

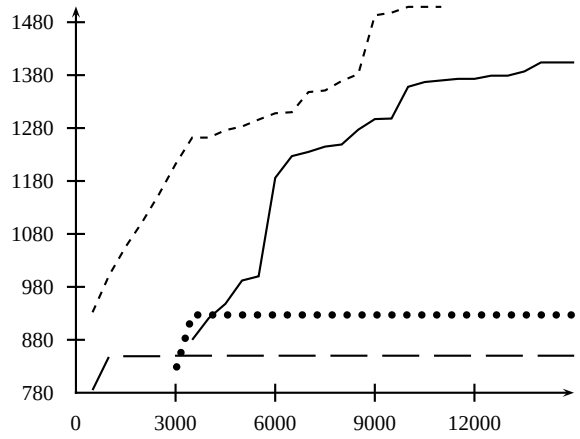


(j) Projekt *Huffman*: Fitness

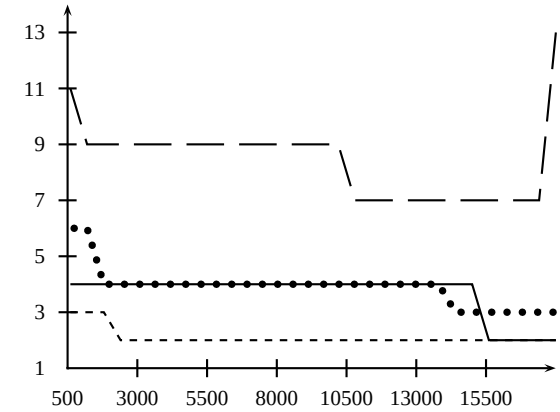
Abbildung B.1: Vergleich der Optimierungsverfahren (Fortsetzung)



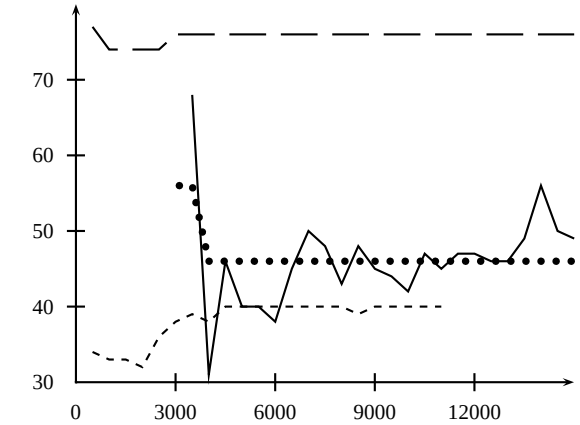
(k) Projekt *JDK sort*: Überdeckung



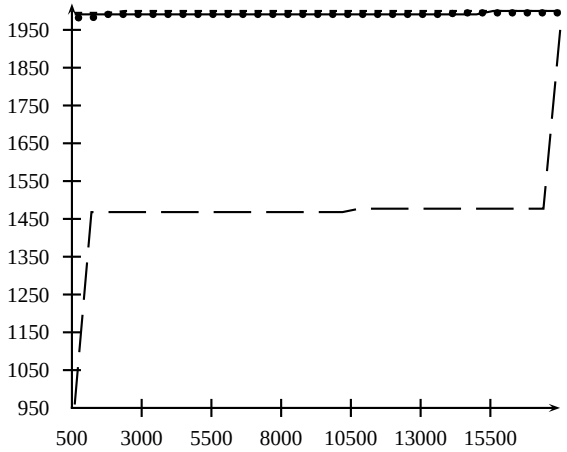
(l) Projekt *JDK logging*: Überdeckung



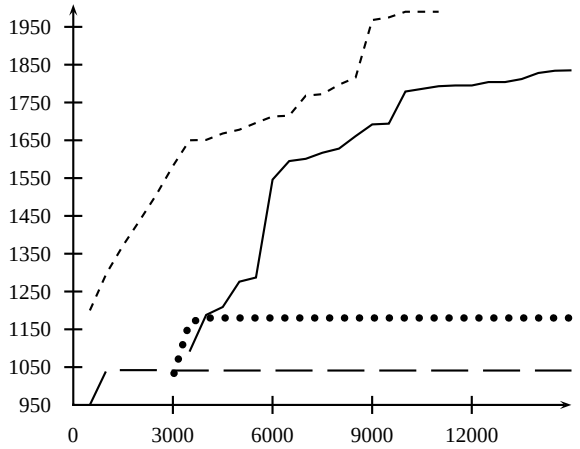
(m) Projekt *JDK sort*: Testumfang



(n) Projekt *JDK logging*: Testumfang



(o) Projekt *JDK sort*: Fitness



(p) Projekt *JDK logging*: Fitness

Abbildung B.1: Vergleich der Optimierungsverfahren (Fortsetzung)

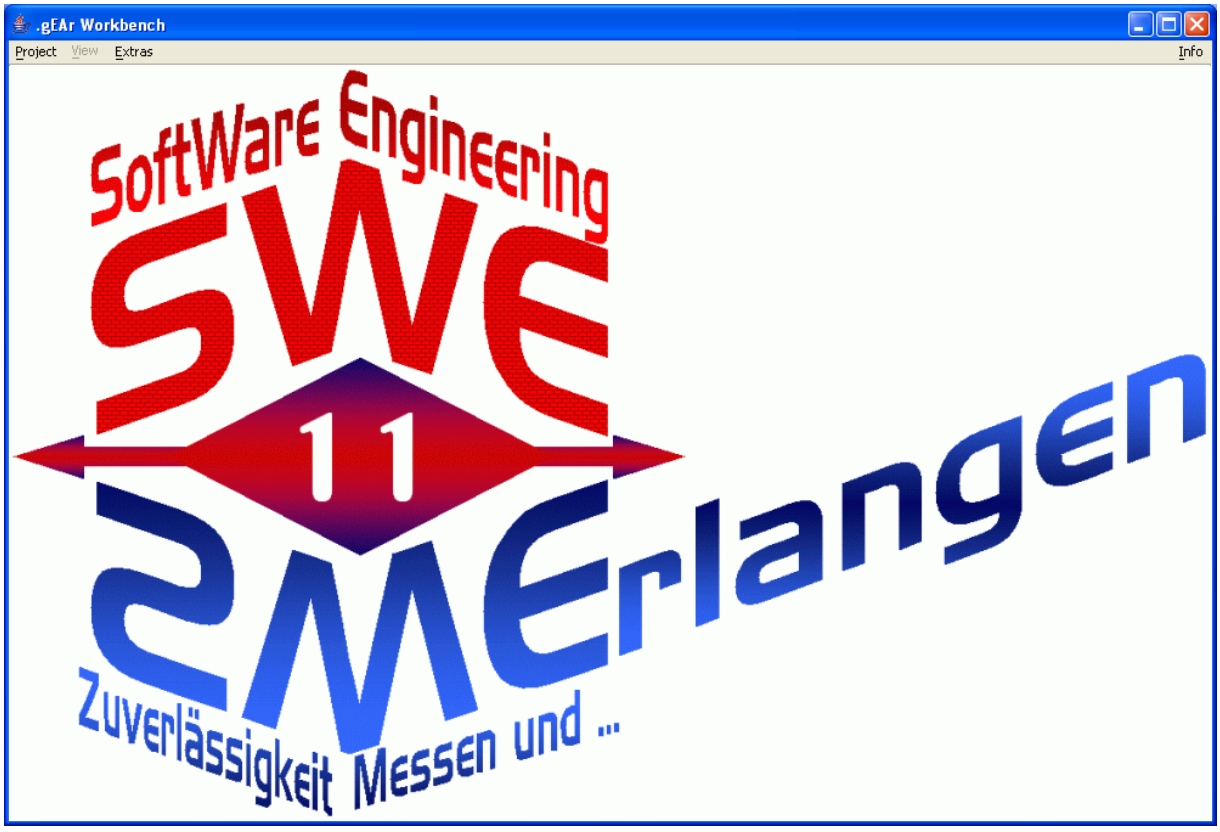
Projekt	BigFloat				JDK logging			
Heuristik / Parameter	*Random*	*MOA*	*SimAnn*	*NSGA*	*Random*	*MOA*	*SimAnn*	*NSGA*
TC/TS	1 – 100	1 – 100	1 – 100	1 – 100	1 – 80	1 – 70	1 – 40	1 – 80
NrArgs	1 – 50	1 – 50	1 – 50	1 – 50	1 – 5	1 – 5	1 – 5	1 – 5
Argumentspezifikation	Long [0;16]	Long [0;16]	Long [0;16]	Long [0;16]	Long [0;135]	Long [0;135]	Long [0;135]	Long [0;135]
Gewicht (Überdeckung)	1,0	1,0	1,0	–	1,0	1,0	1,0	–
Gewicht (Anzahl Testfälle)	0,05	0,05	0,05	–	0,05	0,05	0,05	–
Windowing	ja	ja	ja	–	ja	ja	ja	–
Self-Adaptive	–	ja	–	ja	–	ja	–	ja
Populationsgröße	–	70	–	70	–	80	–	80
Selektionsverfahren*	–	Roulette	–	Roulette	–	Roulette	–	Roulette
Kreuzungsstrategie*	–	Uniform	–	Uniform	–	Uniform	–	Uniform
Crossover-Wahrscheinlichkeit*	–	0,875	–	0,875	–	0,875	–	0,875
Elitismus-Größe	–	0,07	–	1	–	0,07	–	1
Pareto-Elitismus	–	–	–	ja	–	–	–	ja
Tournament-Größe	–	0,5	–	0,5	–	0,5	–	0,5
Mutationswahrscheinlichkeit*	–	0,07	–	0,07	–	0,07	–	0,07
Mutationsvarianz*	–	1	1	1	–	1	1	1
Niching-Radius	–	–	–	0,07	–	–	–	0,07
Initiale Temperatur	–	–	10	–	–	–	10	–
Temperaturabnahmefaktor	–	–	0,1	–	–	–	0,1	–

TC/TS:	Minimale/maximale Anzahl der Testfälle pro Testdatensatz
NrArgs	Minimale/maximale Anzahl der Argumente pro Testfall
Elitismus-Größe	Anteil (in %) der geretteten Individuen an der gesamten Population
„*“	nur initiale Konfiguration, falls Heuristik selbst-adaptiv
„–“	trifft bei dieser Metaheuristik nicht zu

Tabelle B.1: Parametrisierung von •gEAr beim Vergleich der Heuristiken

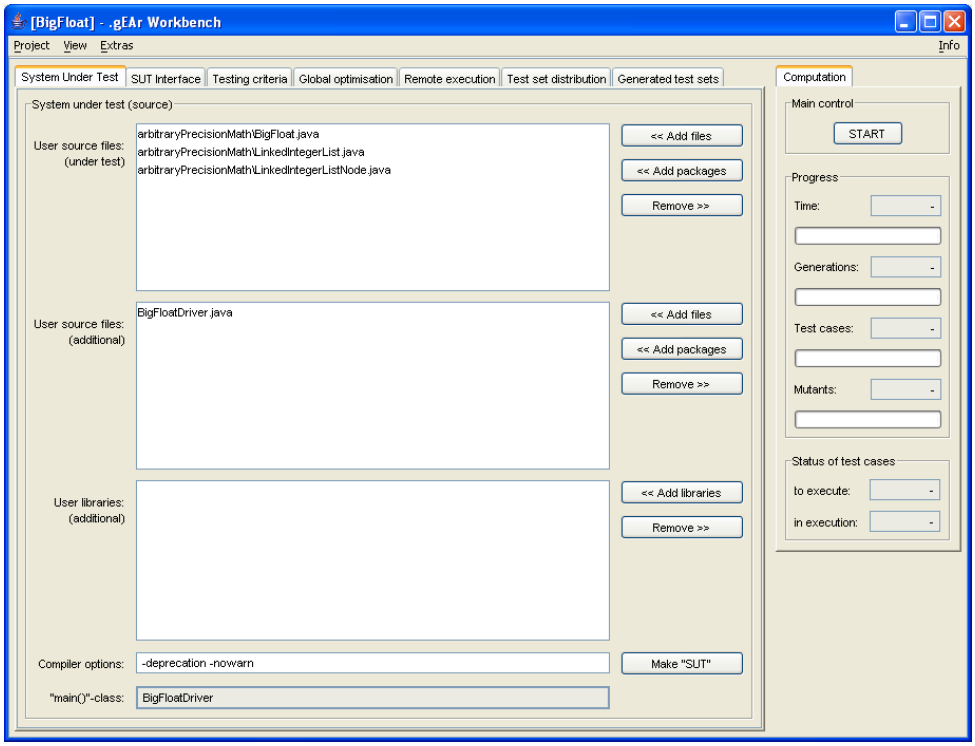
Anhang C

Das Werkzeug .gEAr

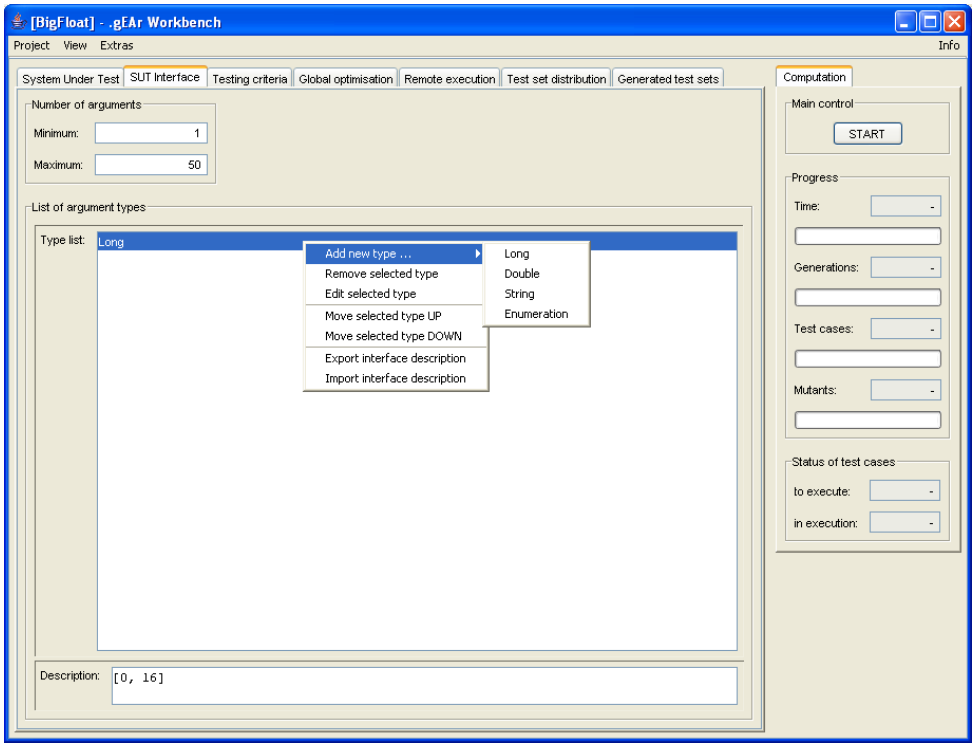


(a) Startbildschirm

Abbildung C.1: Screenshots des Werkzeugs .gEAr

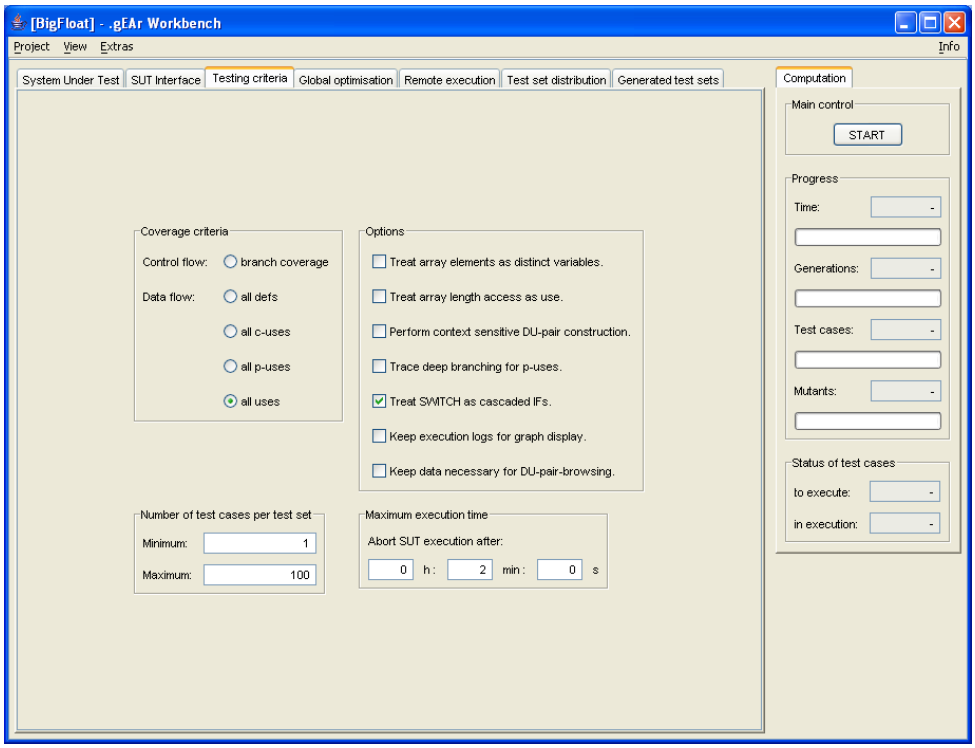


(b) Zusammenstellung des *System Under Test*

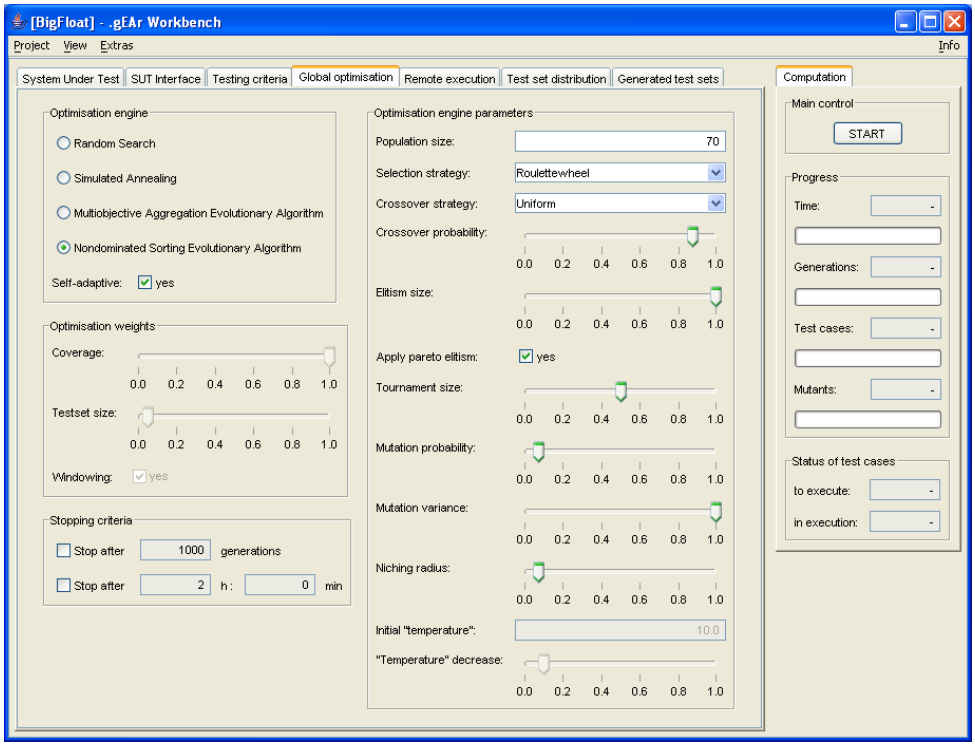


(c) Deklaration der Schnittstelle zum *SUT*

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

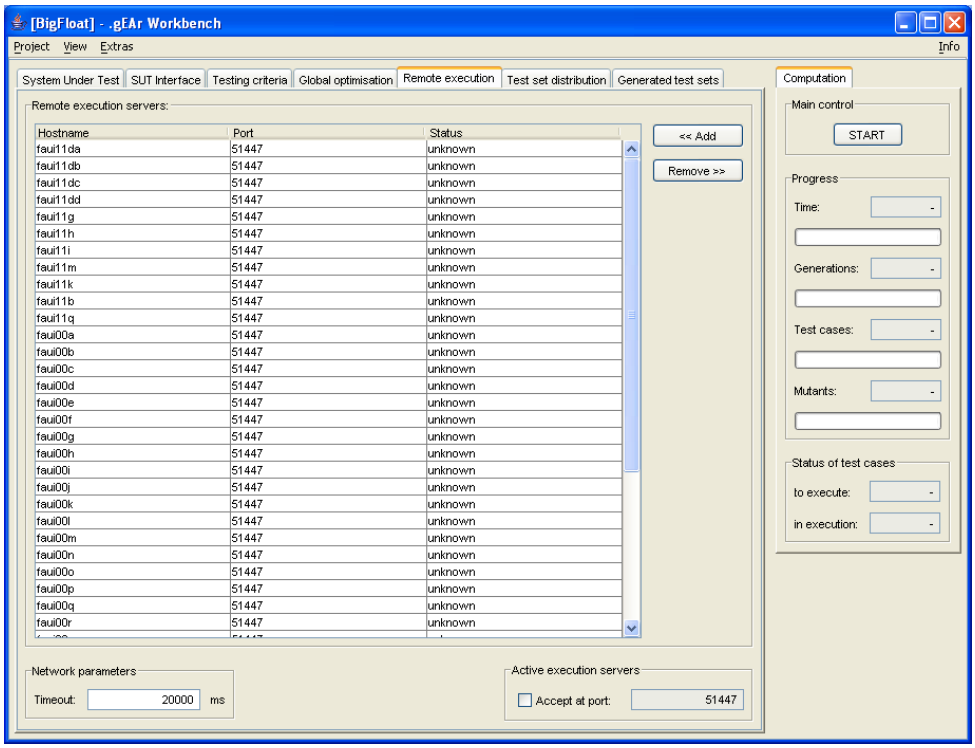


(d) Angabe und Tuning der Testkriterien

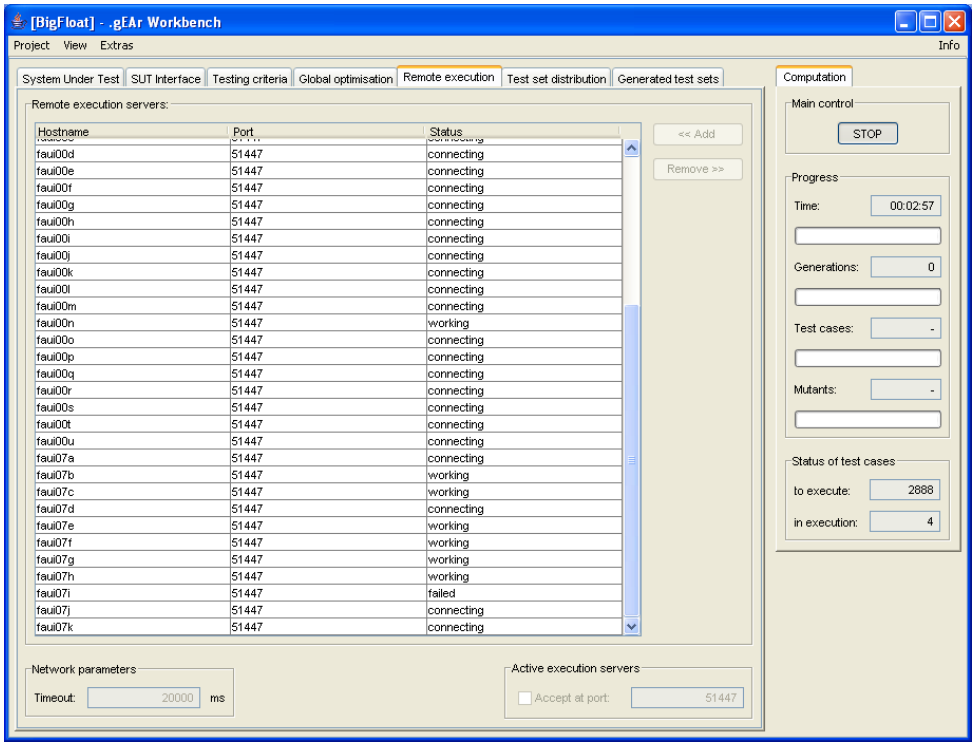


(e) Einstellung der Optimierungsparameter

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

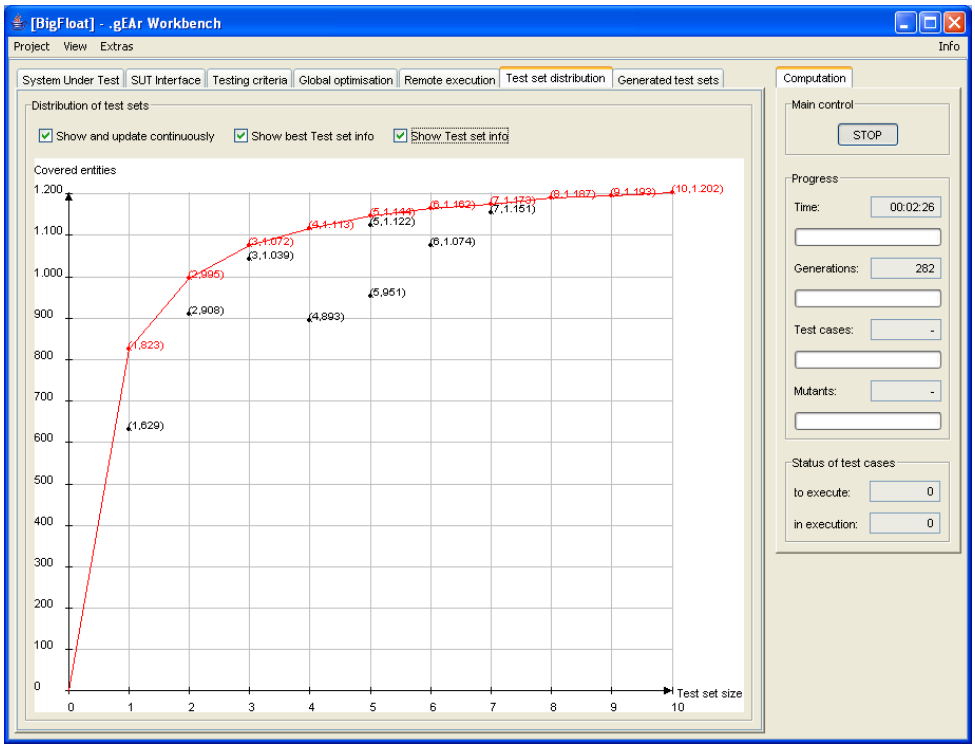


(f) Angabe der *Remote Execution Engines*

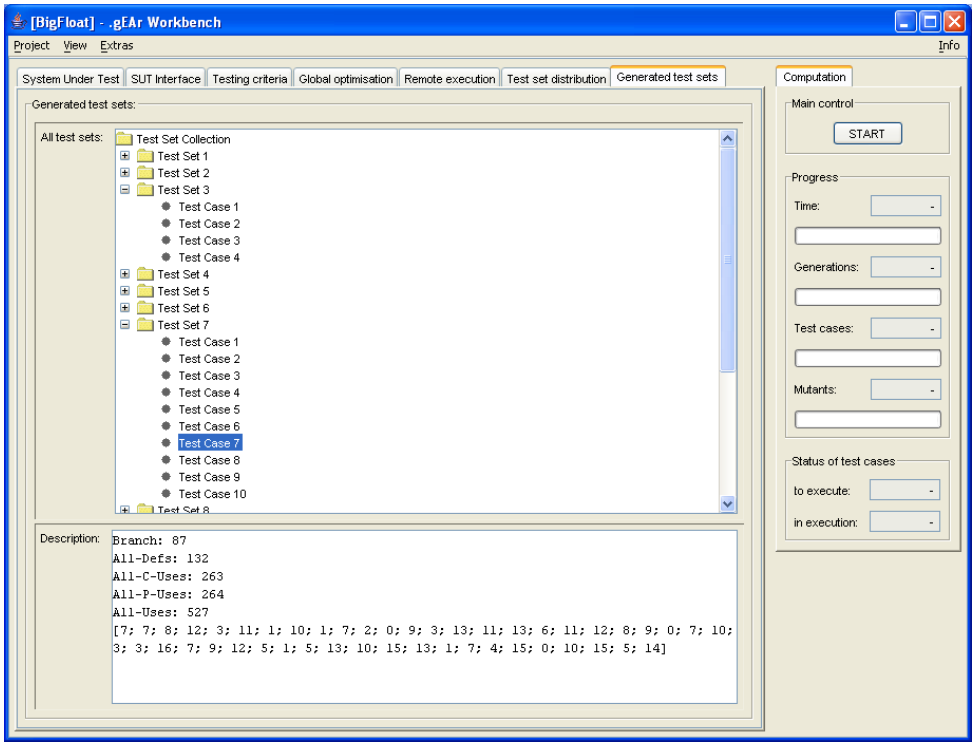


(g) Status der *REE* während der Ausführung

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

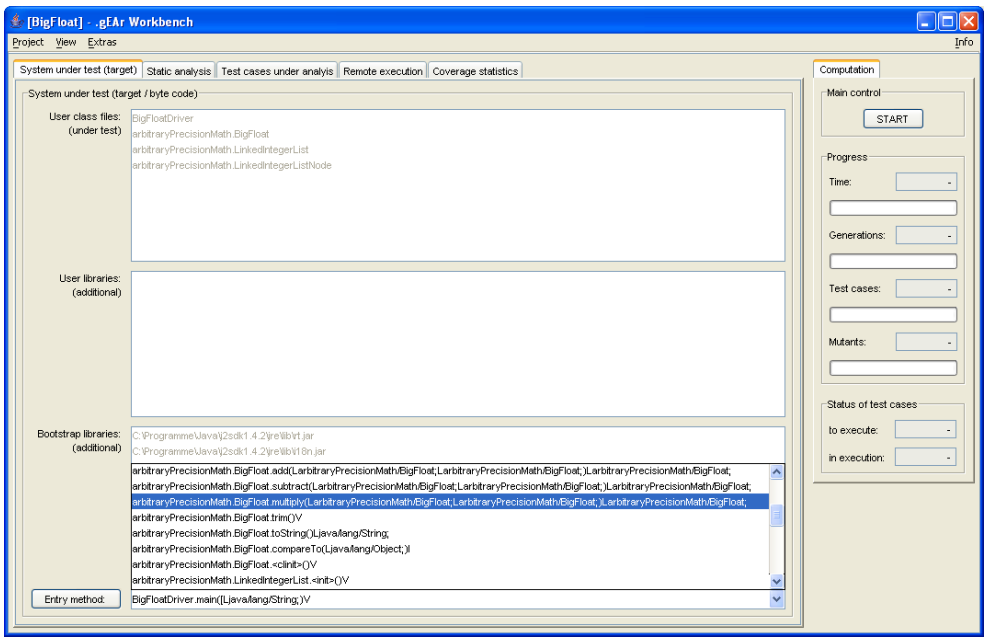


(h) Laufende Verteilung der Testdatenmengen

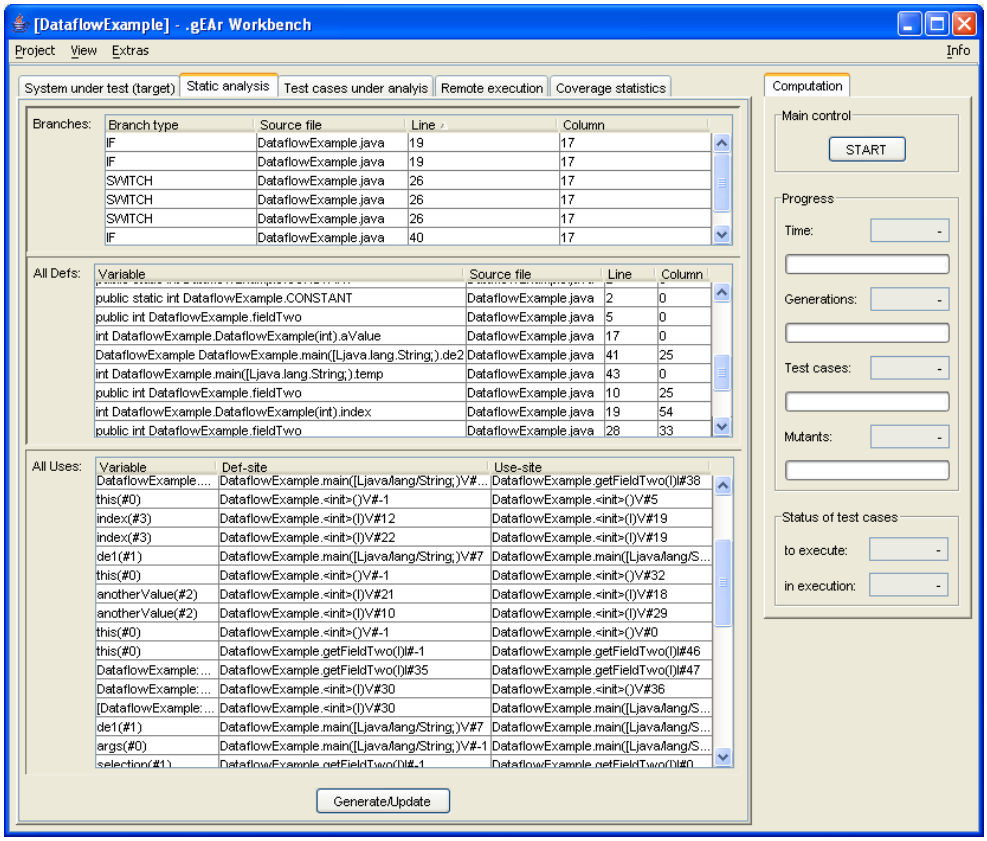


(i) Darstellung der generierten Testdatenmengen

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

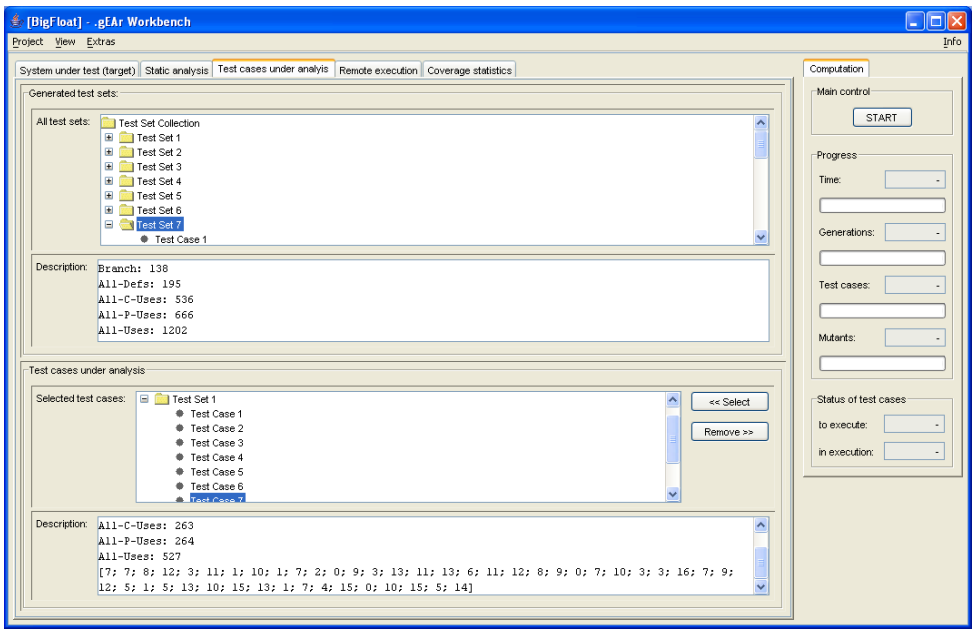


(j) Auswahl der Einstiegsmethode (statische Analyse)

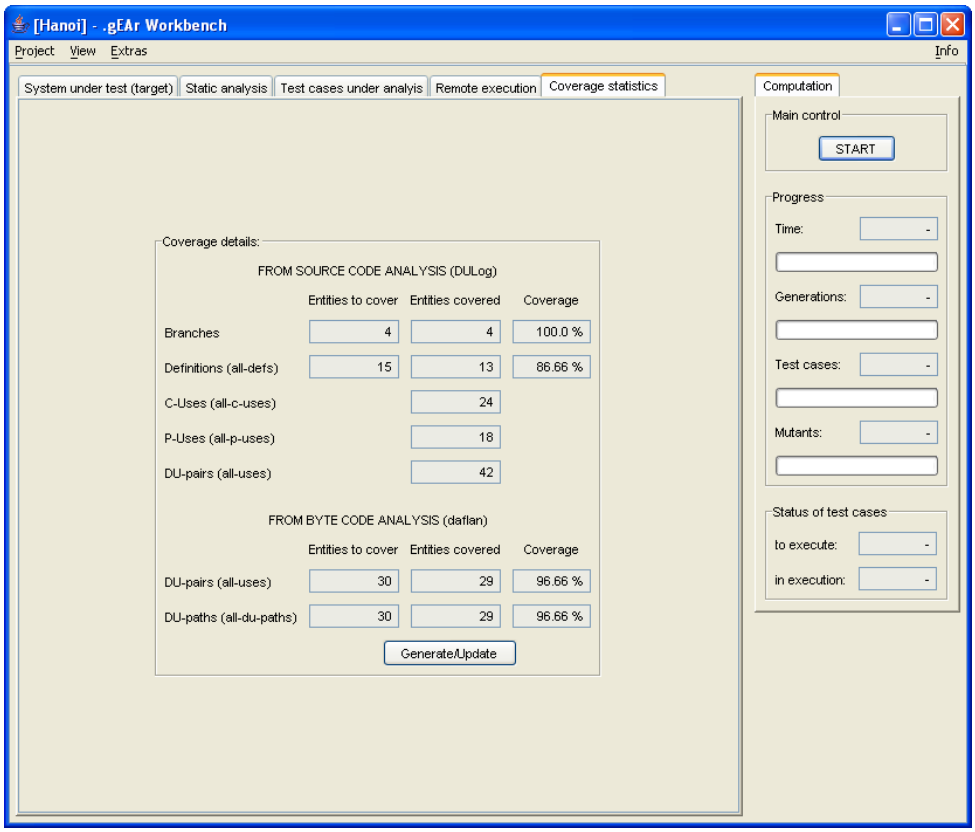


(k) Ergebnisse der statischen Analyse

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

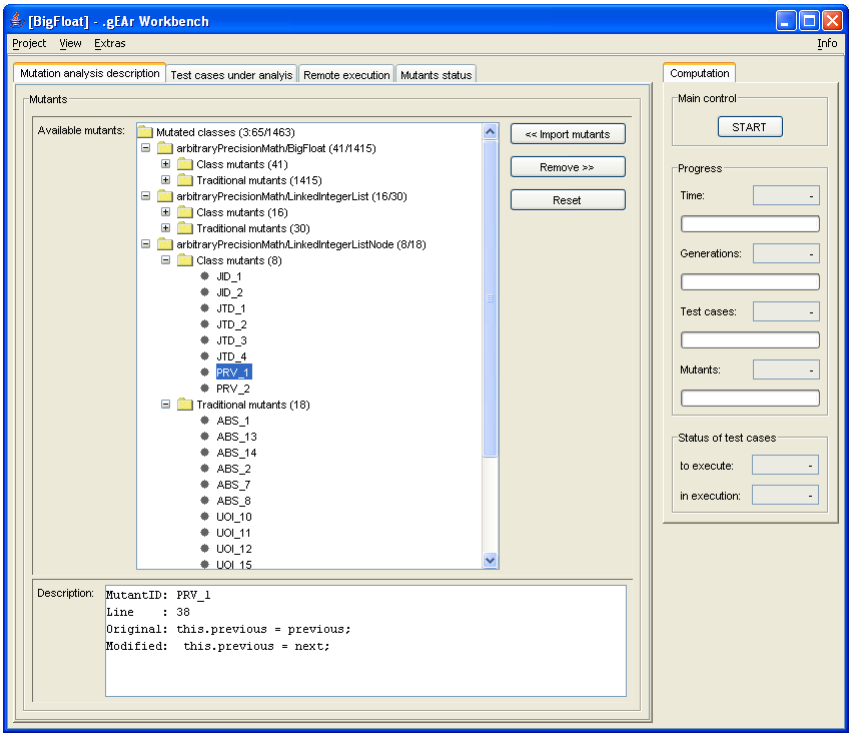


(l) Auswahl der zu analysierenden Testdatenmenge

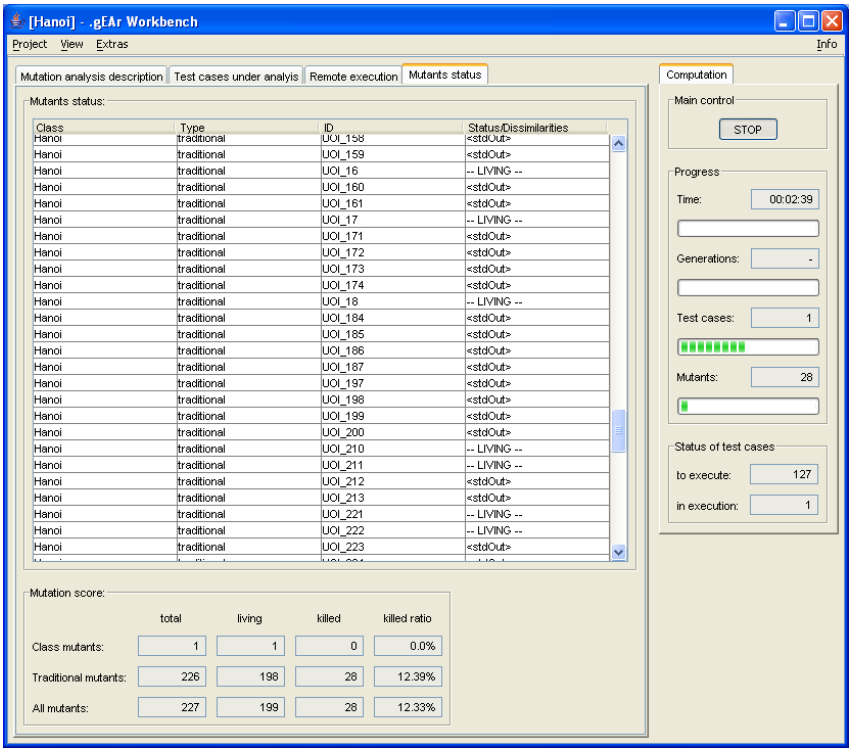


(m) Ergebnisse der Überdeckungsmessung

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

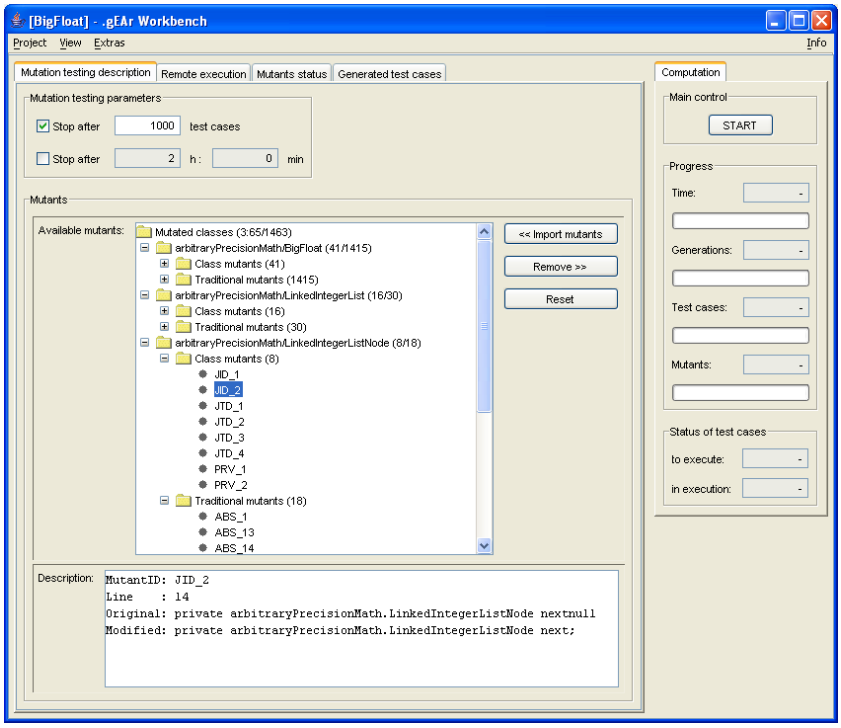


(n) Angabe der Mutanten des *SUT*

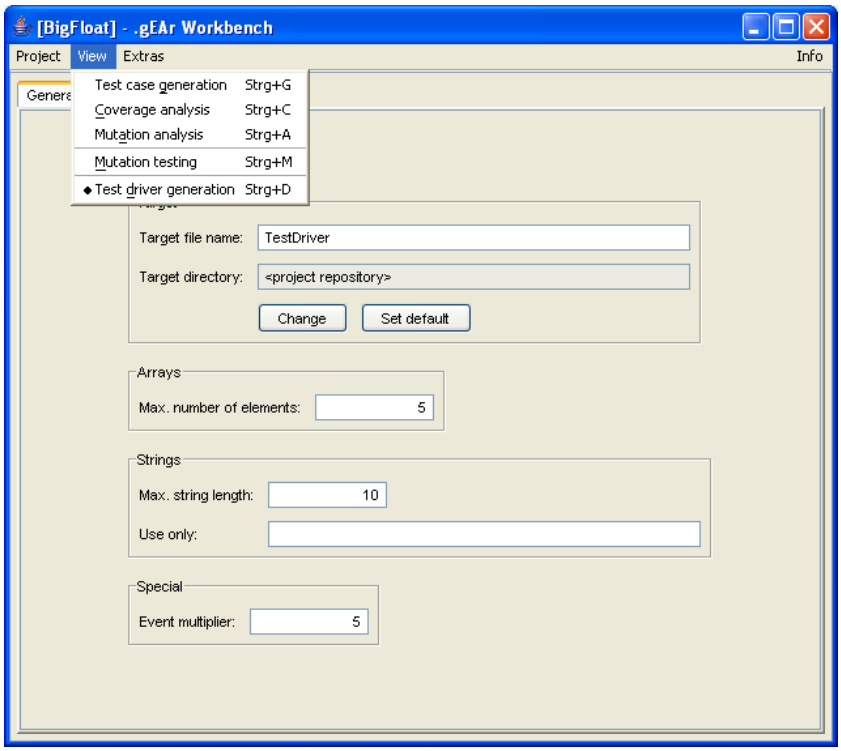


(o) Laufender Status der Mutanten

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

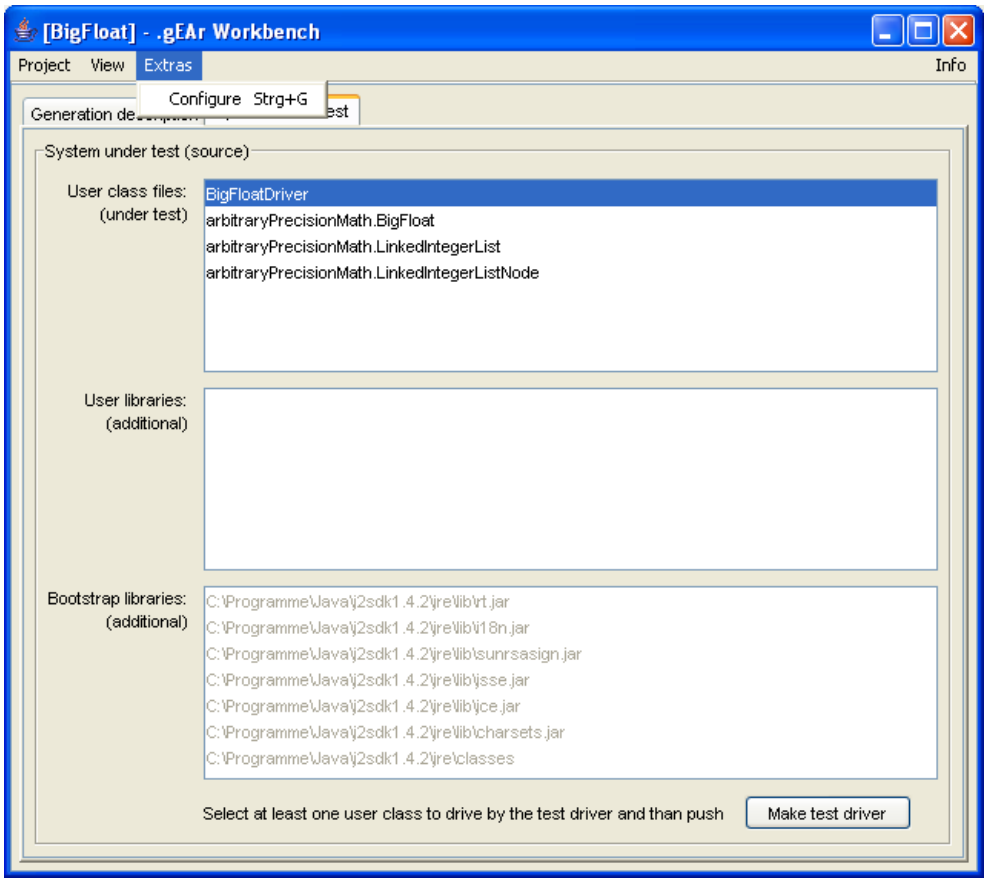


(p) Angabe der Parameter für das Mutationstesten



(q) Angabe der Parameter für den Testtreibergenerator

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)



(r) Auswahl der zu treibenden Klassen

Abbildung C.1: Screenshots des Werkzeugs .gEAr (Fortsetzung)

Komponente		Hauptprogramm „Kernel“	Dynamische Analyse (DÜ)	Statische Analyse (DÜ)	Integrierte Analyse (BÜ)	Erweiterter Testtreiber	$\sum$ $\emptyset^{+}$ $\max^{*}$
`Version`		`21.03.06`	`31.08.05`	`20.07.05`	`12.07.06`	`28.08.06`	`2.2RC65`
Metrik	AD	325	25	131	41	98	620
	AK	343	61	187	49	166	806
	LOC	24.195	6.146	17.369	35.386	15.902	98.998
	LK	1,8%	8,1%	15,5%	3,7%	27,1%	9,3%$^{+}$
	AA	15.552	3.467	6.334	9.781	5.459	40.593
	VA	14,1%	16,7%	13,6%	32,7%	15,4%	18,9%$^{+}$
	MA	12.043	3.032	3.874	12.800	3.775	35.524
	CC	177	399	102	211	83	399
	DOIH	7	5	6	6	6	7*
	NOCC	49	14	7	1	11	49*
	HPLen	63.155	14.405	11.460	15.229	21.618	125.877
	HPVoc	531	508	435	428	332	531*
	HPVol	419.537	105.687	75.645	112.010	147.500	860.402

Messung mit „Source Monitor 1.3.1.6“

AD: Anzahl der Quelltext-Dateien
AK: Anzahl der Klassen
LOC: Anzahl der Quelltextzeilen (*Lines Of Code*)
LK: Anteil der Kommentare bezogen auf LOC
AA: Anzahl der Anweisungen
VA: Anteil der Verzweigungsanweisungen bezogen auf AA
MA: Anzahl der Methodenaufrufe

Messung mit „Borland Together 2006“

CC: Maximale Zyklomatische Komplexität
DOIH: Maximale Vererbungstiefe
NOCC: Maximale Anzahl Unterklassen
HPLen: Programmlänge nach Halstead
HPVoc: Maximales Programmvokabular nach Halstead
HPVol: Programmvolumen nach Halstead

Weitere Abkürzungen

DÜ: Datenflussüberdeckung
BÜ: Bedingungsüberdeckung

Art der Aggregation

$\sum$: Summe
$\emptyset^{+}$: Durchschnitt
$\max^{*}$: Maximum

Tabelle C.1: Das Werkzeug •gEAr in Zahlen

Literaturverzeichnis

[AL98] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 33(5):72–84, May 1998.

[AMA01] Shawki Areibi, Medhat Moussa, and Hussein Abdullah. A comparison of genetic/-memetic algorithms and other heuristic search techniques. In *Proceedings of the International Conference on Artificial Intelligence IC-AI 2001*, pages 660–666, Las Vegas, Nevada, June 2001.

[Ang95] Peter J. Angeline. Adaptive and self-adaptive evolutionary computations. In Marimuthu Palaniswami and Yianni Attikiouzel, editors, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press, Loral Federal Systems - Owego, New York, 1995.

[AO00] Roger T. Alexander and A. Jefferson Offutt. Criteria for testing polymorphic relationships. In *ISSRE 2000: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, pages 15–23, Washington, DC, USA, October 2000. IEEE Computer Society.

[Bäc91] Thomas Bäck. Optimization by means of genetic algorithms. In *36th International Scientific Colloquium*, pages 163–169, Technical University of Ilmenau, 1991.

[Bäc95] Thomas Bäck. Evolution strategies: An alternative evolutionary algorithm. In J.-M. Alliot, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume V of *LNCS*, pages 3–20. Springer Verlag, 1995.

[Bal97] Helmut Balzert. *Lehrbuch der Software-Technik – Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung*, volume 2. Spektrum Akademischer Verlag, Heidelberg Berlin, November 1997.

[Bar00] André Baresel. Automatisierung von Strukturtests mit evolutionären Algorithmen. Diplomarbeit, Lehr- und Forschungsgebiet Softwaretechnik, Humboldt-Universität Berlin, Berlin, July 2000.

[BB00] Carlos C. H. Borges and Helio J. C. Barbosa. A non-generational genetic algorithm for multiobjective optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pages 172–179, La Jolla Marriott Hotel La Jolla, California, USA, 2000. IEEE Press.

[BBM93a] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[BBM93b] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

[Bea93] David Beasley and David R. Bull et al. A sequential niche technique for multimodal function optimization. UWCC COMMA Technical Report 93001, Department of Computing Mathematics, University of Wales, College of Cardiff, Cardiff, CF2 4YN, UK, February 1993.

[Bez93] J.C. Bezdek. Editorial: Fuzzy models - what are they, and why? *IEEE Transactions on Fuzzy Systems*, 1(1), February 1993.

[BFJT02] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the .NET environment. In *ISSRE 2002: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 195–206. IEEE Computer Society, 2002.

[BH91] Thomas Bäck and Frank Hoffmeister. Extended selection mechanisms in genetic algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms and their Applications*, 1991.

[BHJT00] Benoit Baudry, Vu Le Hanh, Jean-Marc Jézéquel, and Yves Le Traon. Building trust into OO components using a genetic analogy. In *ISSRE 2000: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, pages 4–14, Washington, DC, USA, October 2000. IEEE Computer Society.

[BK95] Thomas Bäck and Frank Kursawe. Evolutionary algorithms for fuzzy logic: A brief overview. *Fifth International Conference IPMU: Information Processing and Management of Uncertainty in Knowledge-Based Systems*, 1995.

[Boe79] Barry W. Boehm. Guidelines for verifying and validating software requirements and design specifications. In Paul A. Samet, editor, *Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing (EuroIFIP 79)*, pages 711–719, New York, September 1979. IFIP, North-Holland Pub. Co.

[Boo85] Lashon B. Booker. Improving the performance of genetic algorithms in classifier systems. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 80–92, Mahwah, NJ, USA, July 1985. Lawrence Erlbaum Associates, Inc.

[BSW04] André Baresel, Harmen Sthamer, and Joachim Wegener. Applying evolutionary testing to search for critical defects. In Kalyanmoy Deb and Riccardo Poli et al, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume

3103 of *Lecture Notes in Computer Science*, pages 1427–1428, Heidelberg, June 2004. ISGEC, Springer-Verlag.

[BT95] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report 11, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, 8092 Zürich, Switzerland, June 1995.

[BW03] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of an automated parking system. In *International Conference on Computer, Communication and Control Technologies (CCCT'03) and The 9th. International Conference on Information Systems Analysis and Synthesis (ISAS'03)*, Orlando, Florida, August 2003.

[CCCL96] Kai H. Chang, James H. Cross, W. Homer Carlisle, and Shih-Sung Liao. A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 6(4):585–608, December 1996.

[CDM+96] Alberto Colorni, Marco Dorigo, F. Maffioli, Vittorio Maniezzo, G. Righini, and M. Trubian. Heuristics from nature for hard combinatorial optimization problems. *International Transactions in Operational Research*, 3(1):1–21, 1996.

[Cha00] Ramkrishna Chatterjee. *Modular Data-Flow Analysis of Statically Typed Object-Oriented Programming Languages*. Phd thesis, State University of New Jersey, New Brunswick, January 2000.

[CL95] T. Y. Chen and C. K. Low. Dynamic data flow analysis for C++. In *Proceedings of the 1995 Asia Pacific Software Engineering Conference*, pages 22–29. IEEE, December, 6th 1995.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[CR01] Ramkrishna Chatterjee and Barbara G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Department of Computer Science, Rutgers University, New Jersey, March 2001.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–146, New York, 1999. ACM, ACM Press.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[DY96] Paul Darwen and Xin Yao. Every niching method has its niche: Fitness sharing and implicit sharing compared. Technical report, School of Computer Science, University College UNSW, Australian Defence Force Academy, Canberra ACT 2600 Australia, 1996.

[Ehr02] Wolfgang Ehrenberger. *Software-Verifikation - Verfahren für den Zuverlässigkeitsnachweis von Software*. Carl Hanser Verlag (Fachbuch), München, Januar 2002.

[FB01] István Forgács and Antonia Bertolino. Preventing untestedness in data-flow based testing. *Journal of Software Testing, Verification and Reliability*, pages 29–61, November, 19th 2001.

[FW93] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[GG02] Neelam Gupta and Rajiv Gupta. *The Compiler Design Handbook: Optimizations & Machine Code Generation*, chapter Data Flow Testing. CRC Press, September, 25th 2002.

[GN97] Matthew J. Gallagher and V. Lakshmi Narasimhan. ADTEST: A test data generation suite for Ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, August 1997.

[Gol89] David Edward Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[GR87] David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms and their application*, pages 41–49, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc.

[Grz04] Thomas Grzenkowski. Testfallerzeugung nach der Äquivalenzklassenmethode. Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Software-Engineering, August 2004.

[GW98] David E. Goldberg and Liwei Wang. Adaptive niching via coevolutionary sharing. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 21–38. John Wiley and Sons, Chichester, 1998.

[Han94] Peter J.B. Hancock. An empirical comparison of selection methods in evolutionary algorithms. *Proceedings of the AISB workshop on Evolutionary Computation*, 1994.

[Hea94] Jeffrey Horn and Nicholas Nafpliotis et al. A niched pareto genetic algorithm for multiobjective optimization. *Proceedings of the First IEEE Conference on Evolutionary Computation*, 1(0-7803-1899-4/94):82–87, 1994.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, number 16 in ICSE, pages 191—-200. IEEE, May 1994.

[HHH+02] Mark Harman, Lin Hu, Robert Hierons, Andre Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*, 2002.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 312–326, New York, November 2001. ACM, ACM Press.

[HL91] Joseph Robert Horgan and Saul London. Data flow coverage and the C language. In *Proceedings of the symposium on Testing, Analysis and Verification*, pages 87–97. ACM, ACM Press, October, 8th 1991.

[HL96] Francisco Herrera and Manuel Lozano. Fuzzy genetic algorithms: Issues and models. CICYT TIC96-0778, Dept. of Computer Science and A.I., University of Granada, 1996.

[HLV93] Francisco Herrera, Manuel Lozano, and José L. Verdegay. Genetic algorithms applications to fuzzy logik based systems. DECSAI Technical Report 93116, Department of Computer Science and Artificial Intelligence, ETS de Ingenieria Informática, Universidad de Granada, 18071 Granada, Spain, October 1993.

[HLV94] Francisco Herrera, Manuel Lozano, and José L. Verdegay. Applying genetic algorithms in fuzzy optimization problems. *Fuzzy Systems & A.I. Reports and Letters*, 3(1):39–52, January 1994.

[HM96] Francisco Herrera and Luis Magdalena. Genetic fuzzy systems: A tutorial. CICYT TIC96-0778/TAP94-0115, Dept. of Computer Science and A.I., University of Granada, 1996.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, Ann Arbor, 1975. (republished 1992, The MIT Press, ISBN: 0262581116).

[Hor97] Jeffrey Horn. The nature of niching: Genetic algorithms and the evolution of optimal, cooperative populations. IlliGAL Report 97008, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, Illinois 61801, August 1997.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163, New York, 1994. ACM, ACM Press.

[Jan98] Jan Jantzen. Tutorial on fuzzy logic. Technical Report 98-E 868, Technical University of Denmark, Department of Automation, DK-2800 Lyngby, Denmark, August 1998.

[JSE96] Bryan F. Jones, Harmen-Hinrich Sthamer, and David E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[KS93] Peter Klement and Wolfgang Slany. Fuzzy logic in artificial intelligence. CD-Technical Report 94/67, Christian Doppler Laboratory for Expert Systems, TU Wien, Institut für Informationssysteme, A-1040 Vienna, Austria, 1993.

[Lho02] Ondřej Lhoták. Side-effect analysis in Java classfile attributes. Course project, 308-621 Optimizing Compilers, McGill University, 2002.

[Lig02] Peter Liggesmeyer. *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. Spektrum - Akademischer Verlag, Heidelberg; Berlin, 2002.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.

[LPH01] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. Technical Report GIT-CC-01-03, College of Computing, Georgia Institute of Technology, March 2001.

[Mah95] Samir W. Mahfoud. Niching methods for genetic algorithms. IlliGAL Report 95001, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, Illinois 61801, May 1995.

[McM04] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[MG99] Ole J. Mengshoel and David E. Goldberg. Probabilistic crowding: Deterministic crowding with probabilistic replacement. IlliGAL Report 99004, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, Illinois 61801, January 1999.

[Mic94] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin Heidelberg, 2 edition, 1994.

[MKO02] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 352–366. IEEE Computer Society, November 2002.

[ML04] Leszek A. Maciaszek and Bruc Lee Liong. *Practical Software Engineering*. Pearson Education, Edinburgh Gate, September 2004.

[MLK03] Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. GlassTT - a symbolic Java virtual machine using constraint solving techniques for glass-box test case generation. Technical Report 102, University of Münster, Department of Information Systems, November 2003.

[MM63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, February 1963.

[MMB03] M. Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, November 2003.

[MMS98] Gary McGraw, Christoph Michael, and Michael Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, Ridgetop Circle, Sterling, VA 20166, February, 9th 1998.

[MOK05] Yu-Seung Ma, Jefferson A. Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2), 2005.

[MS96] Brad L. Miller and Michael J. Shaw. Genetic algorithms with dynamic niche sharing for multimodal function optimization. In *International Conference on Evolutionary Computation*, pages 786–791, New York, May 1996. IEEE Press.

[Nat88] I. A. Natour. On the control dependence in the program dependence graph. In *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 510–519, New York, USA, February 1988. ACM Press.

[Nta81] Simeon C. Ntafos. On testing with required elements. In *Proceedings of COMPSAC '81*, pages 132–139, Chicago, November 1981. IEEE Computer Society.

[Nta84] Simeon C. Ntafos. An evaluation of required element testing strategies. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 250–256, Piscataway, NJ, USA, March 1984. IEEE Press.

[OD04] Norbert Oster and Robert-Daniel Dorn. A data flow approach to testing object-oriented Java-programs. In Cornelia Spitzer, Ulrich Schmocker, and Vinh N. Dang, editors, *Probabilistic Safety Assessment and Management (PSAM7/ESREL'04)*, volume 2, pages 1114–1119, Berlin, June 2004. Springer-Verlag London.

[Oea91] Christopher K. Oei and David E. Goldberg et al. Tournament selection, niching, and the preservation of diversity. IlliGAL Report 91011, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, Illinois 61801, December 1991.

[OL91] Jefferson A. Offutt and Stephen D. Lee. How strong is weak mutation? In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 200–213, New York, NY, USA, October 1991. ACM Press.

[OMK04] Jefferson A. Offutt, YuSeung Ma, and YongRae Kwon. An experimental mutation system for Java. *Proceedings of the Workshop on Empirical Research in Software Testing / ACM SIGSOFT Software Engineering Notes*, 29(5), September 2004.

[OPTZ96] Jefferson A. Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software - Practice and Experience*, 26(2):165–176, February 1996.

[OS06] Norbert Oster and Francesca Saglietti. Automatic test data generation by multi-objective optimisation. In Janusz Gorski, editor, *Computer Safety, Reliability and Security*, volume 4166 of *Lecture Notes in Computer Science (LNCS)*, pages 426–438, Berlin Heidelberg, September, 13th 2006. Springer-Verlag.

[Ost01a] Norbert Oster. Implementierung eines Evolutionären Verfahrens zur Risikoabschätzung. Diplomarbeit, Dezember 2001. Lehrstuhl für Informatik 2, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[Ost01b] Norbert Oster. Stundenplanerstellung für Schulen mit Evolutionären Verfahren. Studienarbeit, Mai 2001. Lehrstuhl für Informatik 2, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[Ost04] Norbert Oster. Automatische Generierung optimaler datenflussorientierter Testdaten mittels Evolutionärer Verfahren. In Ernst-Erich Doberkat and Udo Kelter, editors, *Softwaretechnik-Trends*, volume 24 of *3*. GI/FG-TAV (21), August 2004.

[Ost05] Norbert Oster. Automated generation and evaluation of dataflow-based test data for object-oriented software. In R. Reussner et al, editor, *Proceedings of the Second International Workshop on Software Quality (QoSA-SOQUA 2005)*, volume 3712 of *Lecture Notes in Computer Science (LNCS)*, pages 212–226, Berlin Heidelberg, September, 22nd 2005. Springer-Verlag.

[Ost06] Johannes Ostler. Entwicklung eines Testtreibergenerators für objekt-orientierte Java-Komponenten. Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Software-Engineering, Juli 2006.

[OVW98] Michael O'Sullivan, Siegfried Vössner, and Joachim Wegener. Testing temporal correctness of real-time systems - a new approach using genetic algorithms and

cluster analysis. In *EuroSTAR98 Software Testing Analysis & Review*, number 6 in EuroSTAR, pages 397–418, Munich Park Hilton, December, 1st 1998.

[OW91] Thomas J. Ostrand and Elaine J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 74–86, New York, October, 8th 1991. ACM, ACM Press.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability*, 9(4):263–282, December 1999.

[PLR94] Hemant D. Pande, William Landi, and Barbara G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transaction on Software Engineering*, 20(5):385–403, May 1994.

[Pol04] Benjamin Polak. Regressionstest für Java-Software. Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Software-Engineering, April 2004.

[Pol05] Benjamin Polak. Statische Analyse des Datenflusses objekt-orientierter Java-Programme. Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Software-Engineering, Juni 2005.

[PR96] Witold Pedrycz and Marek Reformat. Genetic optimization with fuzzy coding. In Francisco Herrera and José Luis Verdegay, editors, *Genetic Algorithms and Soft Computing*, pages 51–67, Heidelberg, 1996. Physica / Springer Verlag.

[Rec73] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart, 1973.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, September 1982. IEEE Computer Society Press.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Sch85] J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, Mahwah, NJ, USA, July 1985. Lawrence Erlbaum Associates, Inc.

[SD94] N. Srinivas and Kalyanmoy Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[Sea93] William M. Spears and Kenneth A. DeJong et al. An overview of evolutionary computation. *Proceedings of the 1993 European Conference on Machine Learning*, 1993.

[Sha97] Yi Shang. *Global Search Methods for Solving Nonlinear Optimization Problems*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1997.

[Sok04] Dehla Sokenou. Testfallgenerierung aus Statecharts und Interaktionsdiagrammen. *Softwaretechnik-Trends*, Band 24(Heft 3), August 2004.

[SP03] Amie L. Souter and Lori L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018, November 2003.

[Spi02] Andreas Spillner. The W-modell - strengthening the bond between development and test. *STAReast 2002*, 2002.

[TB03] Jan Tretmans and Ed Brinksma. Torx: Automated model based testing. In Alan Hartman and Klaudia Dussa-Zieger, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 31–43, Nürnberg, Dezember 2003.

[TG05] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In Michael Ernst and Thomas Jensen, editors, *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42, New York, NY, USA, September 2005. ACM Press.

[Tob04] Meyer Tobias. Testfallbewertung zur strukturellen Testüberdeckung objektorientierter Java-Programme. Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Software-Engineering, September 2004.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, July 2004. ACM Press.

[Vig05] Uwe Vigenschow. *Objektorientiertes Testen und Testautomatisierung in der Praxis – Konzepte, Techniken und Verfahren*. dpunkt.verlag, Heidelberg, 2005.

[Voi92] Hans-Michael Voigt. Fuzzy evolutionary algorithms. Technical Report TR-92-038, International Computer Science Institute (ICSI), Berkeley, California, June 1992.

[WBP02] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing.

In William B. Langdon and Erick Cantú-Paz et al, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1233–1240, New York, July 2002. Morgan Kaufmann Publishers.

[WGO02] Peter Wilke, Matthias Gröbner, and Norbert Oster. A hybrid genetic algorithm for school timetabling. In B. McKay and J. Slaney, editors, *AI 2002: Advances in Artificial Intelligence*, volume 2557 / 2002 of *Lecture Notes in Computer Science*, pages 455–464, Heidelberg, January 2002. Springer-Verlag.

[Yur99] Jyh-Shiarn Yur. *Incremental Analysis For Flow- And Context-Sensitive Data-Flow Problems*. Phd thesis, State University of New Jersey, New Brunswick, October 1999.

[ZHM94] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software test coverage and adequacy. Technical Report 94/15, Department of Computing, The Open University, Milton Keynes, UK, August 1994.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[ZT98] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms – a comparative case study. Technical report, Swiss Federal Institute of Technology Zurich, Computer Engineering and Communication Networks Laboratory (TIK), Gloriastrasse 35, CH-8092 Zurich, Switzerland, 1998.

Index