Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften
Graduiertenkolleg 1100 "Modellierung, Analyse und Simulation in der Wirtschaftsmathematik"
Institut für Angewandte Informationsverarbeitung

Zur Automatisierung von Softwaretests

– Entwicklung und Bewertung von Orakellösungen –

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat. der Fakultät für Mathematik und Wirtschaftswissenschaften der Universität Ulm

vorgelegt von
Ralph Joachim Guderlei

aus Biberach a. d. Riss

31.10.2008

Amtierender Dekan:	Prof. Dr. Frank Stehling
Erstgutachter:	Prof. Dr. Franz Schweiggert
Zweitgutachter:	Prof. Dr. Volker Schmidt
Tag der Promotion:	31.10.2008

Danksagung

Diese Arbeit entstand am Institut für Angewandte Informationsverarbeitung der Universität Ulm. Ich möchte besonders Prof. Dr. Franz Schweiggert für die Betreuung meiner Promotion danken. Durch seine großzügige Unterstützung wurde diese Arbeit erst möglich. Besonders möchte ich mich für die vielen Einblicke in die Lehre und das damit entgegengebrachte Vertrauen bedanken.

Prof. Dr. Volker Schmidt möchte ich für die wertvollen Kommentare zu den in der Arbeit verwendeten statistischen Verfahren und für die Begutachtung der Arbeit danken.

Ein Großteil der Ergebnisse dieser Arbeit entstanden in enger Zusammenarbeit mit Dr. Johannes Mayer. Die vielen Diskussionen und die Anleitung beim Verfassen von Veröffentlichungen waren mir eine sehr große Hilfe.

Desweiteren möchte ich Christoph Schneckenburger, Dr. Frank Fleischer und Rene Just danken, mit denen ich zusammen an Veröffentlichungen arbeiten durfte. Vielen Dank für Ideen, Kommentare und wertvolle Diskussionen.

Gleichzeitig war ich Stipendiat im DFG Graduiertenkolleg 1100 „Modellierung, Analyse und Simulation in der Wirtschaftsmathematik“. Ich möchte deshalb der DFG und dem Graduiertenkolleg, besonders dessen Sprecher Prof. Dr. Karsten Urban und Prof. Dr. Rüdiger Kiesel für die Unterstützung danken. Durch das Graduiertenkolleg konnte ich meinen Horizont erweitern, indem ich Einblick in Themengebiete erhalten habe, die nicht Gegenstand meiner Promotion waren.

Last but not least möchte ich meinen Mit-Promovierenden bei der IAI und im Graduiertenkolleg für die Zusammenarbeit und die gemeinsamen Unternehmungen der letzen Jahre danken.

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra (source unknown)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Testen

Computer haben in den letzten Jahrzehnten in nahezu alle Bereiche des täglichen Lebens Einzug erhalten. Computer steuern Maschinen, über Computersysteme werden Waren aller Art gehandelt und Informationen verbreitet. Grund für diese weite Verbreitung sind eine deutliche Steigerung der Leistungsfähigkeit der Computer bei gleichzeitig sinkenden Kosten für die Hardware. Mit der zunehmenden Verbreitung und einer immer größeren zur Verfügung stehenden Rechenleistung hat jedoch auch die Komplexität der auf den Computern ausgeführten Programme immer mehr zugenommen.

1.1 Einführung in Softwaretests

Jedes hinreichend große und komplexe Softwaresystem enthält fast notwendigerweise Fehler. Als Fehler wird dabei eine Abweichung des tatsächlichen Verhaltens des Programms vom erwarteten Verhalten bezeichnet. Das kann beispielsweise eine falsche Berechnung oder eine zu langsame Bearbeitung einer Aufgabe sein. Das Finden und Beheben dieser Fehler ist deshalb ein wesentlicher Bestandteil der Softwareentwicklung. In der Literatur (beispielsweise in [6, 91]) wird angegeben, dass etwa 25-50% des Entwicklungsaufwands inzwischen für Tätigkeiten zur Fehlerbeseitigung verwendet werden. Einen Überblick über Anzahl und Art der Fehler kann man sich beispielsweise durch einen Blick in Fehlerdatenbanken von Open Source-Projekten verschaffen. Open Source-Projekte stellen diese Informationen normalerweise vollständig zur Verfügung. So listet beispielsweise die Fehlerdatenbank des HTTP-Servers (Version 2) der Apache Software Foundation im Moment[1] 775 nicht behobene Fehler auf.

[1] 13.03.2008 - `http://issues.apache.org/bugzilla/`

Der Begriff „Fehler“ wird im deutschen Sprachgebrauch für unterschiedliche Dinge verwendet. Deshalb wird in DIN 66271, angelehnt an den englischen Sprachgebrauch, der Fehlerbegriff weiter differenziert. Man unterscheidet dabei nach

- *Fehlhandlung (Error)*: Irrtum des Programmierers
- *Fehlerzustand / Defekt (Fault / Bug)*: innerer Fehler des Programms, beispielsweise fehlerhafter Quelltext
- *Fehlerwirkung (Failure)*: Beobachtbare Auswirkung des inneren Fehlers, beispielsweise der berühmt-berüchtigte „Blue Screen of Death“[2].

Dieser Terminologie liegt eine eindeutige Kausalkette zugrunde: Aus einer Fehlhandlung folgt ein innerer Fehler. Dessen Auswirkungen führen dann zur beobachtbaren Fehlerwirkung. Ziel der meisten analytischen qualitätssichernden Maßnahmen ist es, Fehler im Quelltext, also „Faults“ zu finden und anschließend auch zu beheben.

Um nun diese inneren Fehler zu finden, sind unterschiedliche Vorgehensweisen möglich. Die gebräuchlichsten Vorgehensweisen sind Techniken zur Analyse des Quelltextes, beispielsweise Statische (Code-) Analyse[29, 36], Reviews [37] oder eben Softwaretests. Bei der statischen Analyse wird der Quelltext direkt durch Werkzeuge untersucht. Im Prinzip ist schon ein Compiler ein solches Werkzeug, da dieser oft mögliche Fehlerquellen im Quelltext wie toten Code oder Typprobleme bei Variablenzuweisungen erkennt und meldet. Andere Werkzeuge, beispielsweise LINT [58] (und dessen Abkömmlinge) suchen nach bekannten Mustern im Quelltext, die zu Fehlern führen können. Code-Reviews beziehen sich ebenfalls direkt auf den Quelltext des Programms. Im Unterschied zur statischen Analyse sind Reviews aber die in Augenscheinnahme des Quelltextes durch andere Programmierer, die sich oft auf diese Untersuchungen spezialisiert haben. Der Softwaretest beruht im Gegensatz zu den erst genannten Techniken auf der tatsächlichen Ausführung des zu testenden Systems (SUT – System Under Test).

Für den Begriff „Softwaretest“ gibt es mehrere Beschreibungen oder Definitionen. In Myers „The Art of Software Testing“ [86] ist der Softwaretest beschrieben als „Ausführung eines Programms mit der Absicht, Fehler zu finden“. Im Standard IEEE 610 ist der Softwaretest beschrieben als

[2] `http://en.wikipedia.org/wiki/Blue_Screen_of_Death`

> The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.
>
> IEEE 610

Softwaretests dienen primär dazu, Fehlerwirkungen zu erkennen. Was nun ein fehlerhaftes oder korrektes Verhalten des SUT ist, muss in der Spezifikation des SUT beschrieben sein. Deshalb erfordert das Testen zwingend das Vorhandensein einer solchen Spezifikation. Dadurch kann durch Softwaretests auch nur überprüft werden, ob die Spezifikation richtig umgesetzt wurde. Softwaretests dienen also der Verifikation von Software. Folglich kann nicht festgestellt werden, ob die Spezifikation die Aufgaben des Programms auch richtig beschreibt. Dieser Aspekt ist Gegenstand der Validation von Software. Der Standard IEEE 1012 („Software Validation and Verification Plan“) beschreibt das notwendige Vorgehen zu Validation und Verifikation, also dem Testen, von Software.

Andererseits dient Testen nicht primär der Fehlerbeseitigung, Testen ist nicht *Debugging*. Wird allerdings eine Fehlerwirkung erkannt, können die im Test gewonnenen Informationen dazu verwendet werden, den verursachenden Fehlerzustand zu identifizieren und nach der Identifizierung zu beheben.

Um ein Programm zu testen wird dieses zuerst in einen Startzustand versetzt. Dann werden Aktionen durchgeführt, indem das Programm mit vorher festgelegten (Test-)Eingaben versorgt wird. Um eine Fehlerwirkung feststellen zu können, werden die vom Programm während oder nach der Ausführung gelieferten Ausgaben mit ebenfalls vorher festgelegten erwarteten Ergebnissen verglichen. Bei Programmen, deren Verhalten vom aktuellen Zustand abhängt, muss zudem das Programm in einen definierten Ausgangszustand gebracht werden und der Endzustand des Programms mit dem erwarteten Endzustand verglichen werden. Das ist beispielsweise bei objektorientierten Programmen oft der Fall. Das Tupel aus Startzustand, Testeingaben, erwarteten Ausgaben und erwarteter Endzustand wird als *Testfall* bezeichnet. Abb. 1.1 verdeutlicht den schematischen Ablauf des Testvorgangs.

Von entscheidender Bedeutung beim Testen ist es, die erwarteten Ausgaben für das SUT zu bestimmen und die tatsächlichen Ausgaben des SUT mit den erwarteten Ausgaben zu vergleichen. Die damit verbundenen Probleme werden als *Orakelproblem* be-

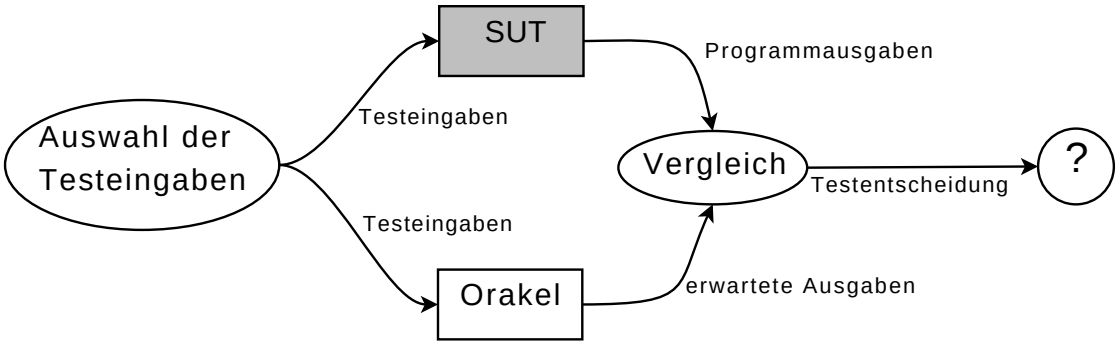


Abbildung 1.1: Schematische Darstellung des Testvorgangs

zeichnet. Dieses wird in Kapitel 1.2.2 diskutiert.

Zur Gewinnung der Testeingaben und der erwarteten Ausgaben können zwei unterschiedliche Ansätze verwendet werden. Beim sog. *Blackbox-Test* wird die Spezifikation des Programms, also Dokumente wie Pflichtenhefte oder Standards dazu verwendet. Beim sog. *Whitebox-Test* wird dagegen zusätzlich zur Spezifikation der Quelltext des Programms verwendet, um Testfälle abzuleiten. Ziel dieses Vorgehens ist es dabei, Testfälle mit bestimmten Eigenschaften zu erzeugen. Eine dieser Eigenschaften wäre es, alle Anweisungen im Quelltext bei der Abarbeitung aller Testfälle mindestens einmal auszuführen (Anweisungsüberdeckung). In der vorliegenden Arbeit werden ausschließlich Blackbox-Tests beschrieben.

1.2 Zentrale Probleme beim Softwaretest

1.2.1 Grenzen des Testens

Laut Definition dienen Softwaretests dem Finden von Fehlern. Daraus folgt, dass durch Tests niemals gezeigt werden kann, dass ein Programm keine Fehler mehr enthält (vgl. E.W. Dijkstra in „The Humble Programmer“, ACM Turing Lecture 1972). Testen darf deshalb nicht mit einem formalen Korrektheitsbeweis des Programms verwechselt werden.

Von theoretischer Seite setzt der Satz von Rice [94, 98] Grenzen des Testens:

Satz 1 *Satz von Rice*

Sei R die Klasse aller Turing-berechenbaren Funktionen und $S \subset R$, $S \neq \emptyset$ eine beliebige nichttriviale Teilmenge davon. Außerdem ist eine Codierung, die einem Codewort w die dadurch codierte Turingmaschine M_w zuordnet, vorausgesetzt. Dann ist die Sprache

$$C(S) = \{w | \textit{die durch } M_w \textit{berechnete Funktion liegt in } S\}$$

unentscheidbar.

Aus diesem Satz folgt, dass die Entscheidung, ob ein Programm gewisse Eigenschaften besitzt, i.A. unentscheidbar ist. Das gilt insbesondere für Softwaretests. Es gibt daher keinen Algorithmus, der für *alle* Programme entscheiden kann, ob diese Fehler enthalten oder nicht. Alle bekannten Ansätze zum Testen von Software sind deshalb nicht allgemein anwendbar. Ebenso sind Entscheidungen, ob ein Fehlverhalten vorliegt oder nicht, eventuell fehlerbehaftet. Insbesondere der Entscheidung, dass kein Fehlverhalten vorliegt, ist grundsätzlich zu misstrauen, da nicht garantiert werden kann, dass eine Testmethode auch alle im Programm enthaltenen Fehler findet. Aber auch im Falle, dass Tests einen Fehler erkennen, muss dieser nicht unbedingt im SUT liegen. Fehler im Testsystem selbst sind der Praxis oft für einen hohen Anteil an den gefundenen „Fehlern“ verantwortlich (siehe beispielsweise [97]).

Aber auch bei der praktischen Durchführung von Tests werden schnell Grenzen sichtbar. Selbst wenn eine korrekte Spezifikation vorliegt und aus dieser für alle Eingaben die notwendigen erwarteten Ausgaben des Programms bestimmt werden können, sind selbst für moderat komplexe Programme zu viele Testfälle nötig, um das Programm vollständig zu testen. Ein vollständiger oder erschöpfender Test umfasst alle möglichen Kombinationen von Eingaben.

Als Beispiel sei eine Funktion $f(a, b)$ gegeben, die zwei (32 bit unsigned) Integer-Werte a und b erwartet. Um nun alle möglichen Kombinationen zweier Integers durchzutesten wären $2^{32} \cdot 2^{32} = 2^{64}$ Testfälle notwendig. Nimmt man nun eine Ausführungszeit von 1 μs pro Testfall an, würde der vollständige Test ca. 580.000 Jahre dauern.

Als Konsequenz muss also von einem vollständigen Test in der Praxis Abstand genommen werden und die Anzahl der Testfälle auf ein praktisch handhabbare Größe re-

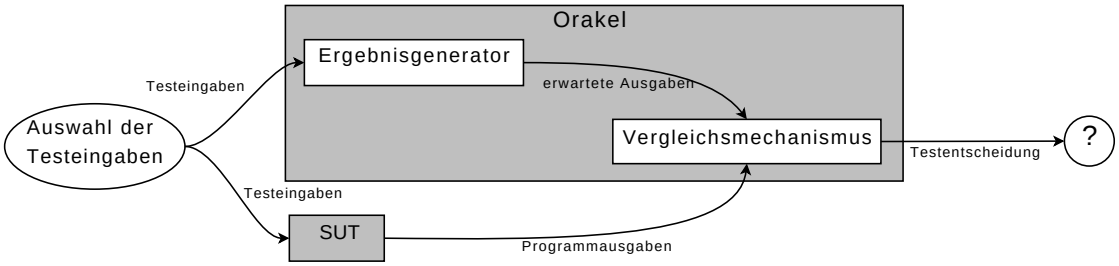


Abbildung 1.2: Schematische Darstellung eines Tests mit Orakel

duziert werden. Das ist natürlich immer mit der Gefahr verbunden, Fehler beim Testen nicht zu finden.

1.2.2 Das Orakel-Problem

Um bei einem Test eine Entscheidung darüber treffen zu können, ob die Programmausgaben der Spezifikation entsprechen, werden zu den Testeingaben ebenfalls die zugehörigen erwarteten Ausgaben benötigt. Als *Orakel*[3] bezeichnet man einen Mechanismus, um diese erwarteten Ausgaben zu bestimmen. Als *Orakel-Problem* werden alle damit verbundenen Probleme bezeichnet. Das Orakelproblem wurde erstmals in [111] beschrieben. In [12] wird als Orakel nicht nur dieser Mechanismus zur Erzeugung der erwarteten Ausgaben bezeichnet. Das Orakel enthält zusätzlich einen Mechanismus zum Vergleich von erwarteten und tatsächlichen Ausgaben. Abb. 1.2 verdeutlicht den Testablauf in Verbindung mit einem Orakel.

Ebenfalls in [12] sind zahlreiche Standard-Ansätze für Orakel beschrieben. Das einfachste Orakel ist das sog. *solved example oracle*: Für einzelne (wenige) Testeingaben werden die erwarteten Ausgaben manuell erstellt. Das *solved example oracle* wird implizit bei jedem manuellen Testvorgang verwendet und dürfte damit das am häufigsten verwendete Orakel sein.

Das sog. *perfect oracle* ist nach [12] das beste Orakel. Dieses ist eine defektfreie Variante des SUT, die auch die selben Anforderungen wie das SUT erfüllt. Dieses Orakel dürfte in der Praxis nicht zu finden sein, denn sonst müsste das SUT nicht entwickelt werden. Im akademischen Bereich kann ein perfektes Orakel jedoch häufiger konstruiert werden, beispielsweise indem in ein korrektes Programm Fehler eingepflanzt werden.

[3] Im Altertum war ein Orakel (von lat. oraculum – Weissagung) eine Stätte, an der Rat bei Problemen gesucht wurde. Das bekannteste Orakel war wohl das Orakel von Delphi. Das Wort Orakel leitet sich von lateinisch orare, sprechen, ab.

Verwendet man diese veränderten Programme als SUT, dann ist das ursprüngliche Programm die defektfreie Version des SUT, und damit ein *perfect oracle*. In der vorliegenden Arbeit wird das *perfect oracle* verwendet, um andere Orakel bewerten zu können.

Das sog. *gold standard oracle* (siehe [12]) ist eine abgeschwächte Form des *perfect oracle*. Die erwarteten Ausgaben werden von einem als „vertrauenswürdig“ eingestuften Programm erzeugt, welches auf derselben Spezifikation beruht, beispielsweise einem abzulösenden Altsystem. Die Alternativ-Implementierung muss aber nicht alle Anforderungen erfüllen, die an das SUT gestellt werden. Beispielsweise kann die Alternativimplementierung einen anderen Algorithmus zu einer Berechnung verwenden oder nichtfunktionelle Anforderungen nicht erfüllen. Will man beispielsweise eine Implementierung des Quicksort-Algorithmus testen, kann die Alternativ-Implementierung beispielsweise den Bubblesort-Algorithmus verwenden, um die Referenzausgaben zu bestimmen. In der Praxis wird dieser Typ jedoch eher selten verwendbar sein, da die benötigte alternative, vertrauenswürdige Implementierung nicht vorhanden ist.

Eine Gruppe von Orakeln wird in der aktuellen Forschung häufig untersucht um Orakellösungen zu konstruieren, die sog. *partiellen Orakel*. In [10] wird vermutet, dass diese partiellen Orakel den Weg hin zu automatischen Orakellösungen und damit zu einer vollständigen Automatisierung des Testvorgangs ebnen könnten.

Definition 1 *Partielles Orakel*
Ein partielles Orakel ist durch zwei Eigenschaften charakterisiert:

1. *Die Entscheidung, dass ein Programm fehlerhaft ist, ist immer korrekt.*
2. *Wird ein Programm als nicht fehlerhaft klassifiziert, kann es trotzdem Fehler enthalten.*

Für die Automatisierung der Testentscheidung ist es besonders interessant, dass das SUT nicht fälschlicherweise als fehlerhaft klassifiziert wird. In der Praxis sind solche Fehlentscheidungen fatal, da sie eine aufwändige und erfolglose Fehlersuche nach sich ziehen. Treten diese Fehlentscheidungen oft auf, führt das zu einer sinkenden Akzeptanz des Testverfahrens. Alle in der vorliegenden Arbeit vorgestellten und untersuchten Orakellösungen gehören zu der Gruppe der partiellen Orakel. Inwieweit eine Fehlentscheidung, die keinen Fehler im SUT anzeigt beeinflusst werden kann, ist ein Hauptaspekt der vorliegenden Arbeit.

In [111] werden drei Klassen von Programmen beschrieben, bei denen das Orakel-Problem auftritt:

- Programme, für die der Tester keine korrekte Spezifikation besitzt
- Programme, bei denen der Aufwand für die Gewinnung der erwarteten Ausgaben zu groß ist
- "programs which were written to determine the answer" ([111])

Bei der ersten Klasse an Programmen handelt es sich sicherlich um die in der Praxis relevanteste Form des Orakelproblems. In der vorliegenden Arbeit wird darauf allerdings kein Bezug genommen.

In die zweite Klasse fallen beispielsweise Programme mit komplexen Berechnungen. Bei diesen Programmen ist es oft so, dass für ganz bestimmte Eingaben das Ergebnis exakt bekannt ist (z.B. e^0, $\sin(\pi)$, usw.), im Allgemeinen ist das exakte erwartete Ergebnis der Berechnung jedoch unbekannt. Bei Testen dieser Programme wird nun normalerweise so vorgegangen, dass die Berechnung für diese bekannten Ein-/Ausgabepaare (sog. *Spezialfälle*) überprüft wird. Zusätzlich wird dann oft noch für wenige beliebig gewählte andere Eingaben von Hand das erwartete Resultat bestimmt. Dieses Vorgehen ist offensichtlich suboptimal, es ist nicht nur zeitaufwändig sondern auch hochgradig fehleranfällig. In der vorliegenden Arbeit werden Techniken vorgestellt, die den Aufwand zur Bestimmung dieser erwarteten Ausgaben deutlich reduzieren, teilweise sogar unnötig machen. Das sog. *Metamorphische Testen* (siehe Kapitel 3.1) wird als Ergänzung zu bekannten Testmethoden vorgeschlagen und untersucht werden.

Die dritte Klasse von Programmen ist besonders im wissenschaftlichen Umfeld oft anzutreffen. Oft werden Programme zur Ermittlung von bislang unbekannten Lösungen für unterschiedlichste Probleme verwendet. Besonders bei mathematischen Problemen ist die Situation gegeben, dass zwar eine Lösung durch Formeln beschrieben werden kann, diese aber nicht analytisch gelöst werden können. Zur tatsächlichen Berechnung müssen dann numerische Verfahren oder Simulationen eingesetzt werden. Ein Beispiel für eine solche Problemstellung ist das Bewerten von Finanzinstrumenten, etwa Optionen. Mehrere Stipendiaten des DFG-Graduiertenkollegs 1100[4] arbeiten an Modellen solcher Finanzinstrumente. Die Modelle werden dazu verwendet, um Verfahren zum Berechnen

[4]DFG-Graduiertenkollegs 1100 – „Modellierung, Analyse und Simulation in der Wirtschaftsmathematik“: `http://www.uni-ulm.de/einrichtungen/gradkoll1100`

von Preisen abzuleiten. Die Implementierung der Verfahren wird dann verwendet, die Preise der Finanzinstrumente konkret auszurechnen. Das Testen der Implementierungen ist überaus problematisch, da alternative Bewertungsverfahren entweder nicht zur Verfügung stehen oder aber nur als Annäherung der erwarteten Ergebnisse verwendet werden können.

In diesem Fall besteht ganz offensichtlich keine Möglichkeit, von Hand erwartete Ergebnisse zu bestimmen. Wenn genügend Zeit und Programmierer zur Verfügung stehen, können mehrere Implementierungen erstellt werden. Dieses Vorgehen wird als Diversität bzw. *n-version programming* (siehe [20]) bezeichnet. Diese werden dann verwendet, um auf unterschiedliche Weise das Ergebnis zu berechnen. Anschließend kann dann per Mehrheitsentscheid (*judging oracle*) oder auf andere Weise das korrekte Ergebnis bestimmt werden.

Bei der dritten Klasse von Programmen stellt sich auch ein fast schon philosophisches Problem in den Weg. Da diese Programme geschrieben werden, um die Antwort auf eine bestimmte Frage zu bekommen, kann keine erwartete (korrekte) Antwort bekannt sein, sonst würde das Programm nicht geschrieben werden müssen. Allerdings kann in diesem Fall immer noch durch Tests geprüft werden, ob die Resultate plausibel sind.

1.2.3 Vollständige Automatisierung des Testens

Aus Zeit- wie aus Kostengründen ist eine Automatisierung des Testablaufs zu empfehlen. Die Automatisierung der Testdurchführung kann mit einer Vielzahl an Werkzeugen umgesetzt werden. Im Java-Umfeld werden oft Werkzeuge wie JUnit oder FIT zur Testautomatisierung verwendet. Im Prinzip werden kurze Programme geschrieben, die die SUT mit den spezifizierten Testeingaben aufrufen und anschließend die erhaltenen Ausgaben der SUT mit den erwarteten Ausgaben vergleichen. Verglichen mit dem manuellen Testen werden durch die Automatisierung Fehler bei der Testdurchführung vermieden und der zeitliche Aufwand für die Testdurchführung wird deutlich reduziert.

Sollen jedoch weitere Aufgaben im Testablauf automatisiert werden, treten zwei Grundprobleme hervor: Die automatische Erzeugung von Testeingaben und die automatische Testentscheidung. Der Schwerpunkt der vorliegenden Arbeit ist die automatische Testentscheidung, auf Strategien zur automatischen Erzeugung von Testeingaben wird in den folgenden Kapiteln kurz eingegangen.

Bei der automatischen Testentscheidung wird das Orakelproblem weiter verschärft. In

der Literatur, v.a. in [12], werden einige Orakel beschrieben, die sich zum automatischen Treffen einer Testentscheidung eignen. Diese Orakel müssen nicht nur aus irgendeinem Mechanismus zur Erzeugung von erwarteten Ausgaben bestehen, dieser Mechanismus muss dazu in der Lage sein, bei nahezu beliebigen Eingaben automatisch eine Testentscheidung zu treffen. Alle beschriebenen Orakel haben aber den entscheidenden Nachteil, nur sehr selten wirklich eingesetzt werden zu können. Als Beispiel hierfür seien das schon erwähnte *gold standard oracle* oder das sog. *reversing oracle* genannt. Letzteres geht davon aus, dass eine errechnete Lösung durch Einsetzen überprüft werden kann. Soll beispielsweise die Lösung eines linearen Gleichungssystems bestimmt werden, kann die z.B. durch ein Gauß-Algorithmus bestimmte Lösung in das ursprüngliche Gleichungssystem eingesetzt und überprüft werden. Eine andere Möglichkeit wäre das Vorhandensein einer Umkehrfunktion. Soll etwa e^x berechnet werden, kann die Lösung über $x = \ln(e^x)$ überprüft werden.

Die Automatisierung des Tests erfordert also sowohl eine Methode zur automatischen Erzeugung von Testeingaben, als auch ein Orakel. In den Fallstudien der vorliegenden Arbeit werden beide Aspekte gemeinsam untersucht. Die Kombination aus einer Methode zur Erzeugung von Testeingaben und einem Orakel wird im Folgenden als *Teststrategie* bezeichnet.

1.3 Testen im Software-Entwicklungsprozess

Zur Entwicklung von Software existieren zahlreiche Vorgehensmodelle. Bei allen zeitgemäßen Vorgehensmodellen ist der Softwaretest ein wesentlicher Bestandteil. In älteren Vorgehensmodellen wird das Testen als eigener Teil der Softwareentwicklung angesehen und wird im Ablauf nach der Implementierung des Programms angesiedelt.

Während der Entwickung eines Programms kann man unterschiedliche Testarten unterscheiden (vgl. z.B. [86]). Der *Modultest* überprüft kleine, isolierte Teile des Programms. Dabei werden einzelne Funktionen oder bei objektorientierten Programmen einzelne Klassen getestet. Beim *Integrationstest* wird untersucht, ob die einzelnen Module des Programms auch zusammen wie erwartet funktionieren. Der *Systemtest* überprüft das Programm als Ganzes. Häufig werden Systemtests im Gegensatz zu Modul- und Integrationstest von Teams durchgeführt, die nicht an der Entwicklung des Programms beteiligt waren. Diese werden vom Auftraggeber des Programms extra beauftragt um zu überprüfen, ob die funktionalen und nicht-funktionalen Anforderungen erfüllt sind.

Als Beispiel sei das wohl bekannteste Vorgehensmodell, das sog. *Wasserfall-Modell* [95], genannt. Dieses Modell unterscheidet fünf Phasen der Softwareentwicklung, die nacheinander durchlaufen werden: Anforderungsanalyse und Spezifikation, Systemdesign, Programmierung, Test sowie Auslieferung und Wartung. Abb. 1.3 zeigt eine schematische Darstellung des Wasserfall-Modells. Das Wasserfall-Modell ist sehr starr und

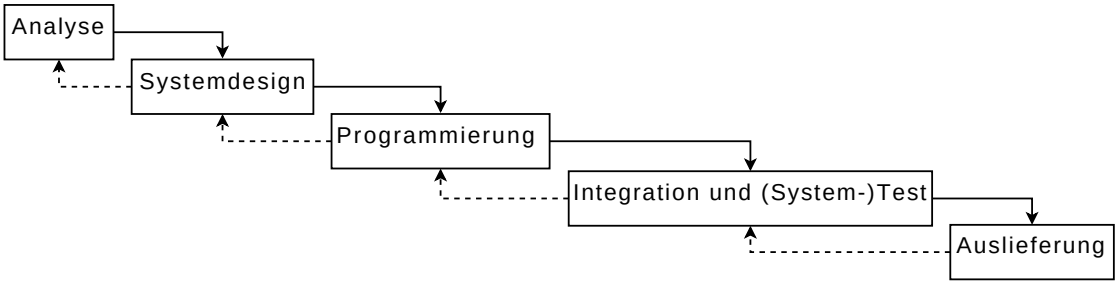


Abbildung 1.3: Schematische Darstellung des Wasserfallmodells

unflexibel. Ein besonderes Problem ist, dass der Test des Systems erst in einer späten Phase der Entwicklung vorgesehen ist. Die Testphase im Wasserfallmodell enthält nicht nur Systemtests, sondern auch Modul- und Integrationstests. Die Modultests finden also erst nach der Implementierung statt. Dadurch werden Fehler erst spät im Entwicklungsprozess entdeckt. Zum Beheben der Fehler muss ein Rücksprung in die Implementierungsphase und einem erneuten Durchlauf der Integrations- und Testphase erfolgen. Das bringt einen hohen Aufwand mit sich und erhöht die Kosten. Modernere Interpretationen des Wasserfallmodells versuchen deshalb, Modultests schon während der Entwicklungsphase durchzuführen, und dadurch Fehler schneller zu entdecken.

Heutige Vorgehensmodelle sehen generell vor, Tests schon während der Implementierungs-Phase, parallel zur Programmierung durchzuführen. Bekannte Vertreter dieser Prozesse sind Test-driven Development / eXtreme Programming [9, 8], Crystal[28] oder Scrum [99]. Diese Prozesse gehören zur Gruppe der iterativen Entwicklungsprozesse. Stark vereinfacht gesehen haben diese iterativen Prozesse die Eigenschaft, Projektphasen wiederholt in Zyklen zu durchlaufen. Üblicherweise werden die Phasen Systemdesign, Programmierung und (Modul-/Integrations-)Test zu einem Zyklus zusammengesetzt. In jeder Iteration wird die Implementierung erweitert und verbessert. Der zentrale Punkt ist, dass in jeder Iteration der gegenwärtige Entwicklungsstand getestet wird. Damit wird versucht, Fehler schon kurz nach deren Entstehung zu entdecken und zu beheben, so dass damit nur relativ geringe Kosten bei der Fehlerbehebung anfallen.

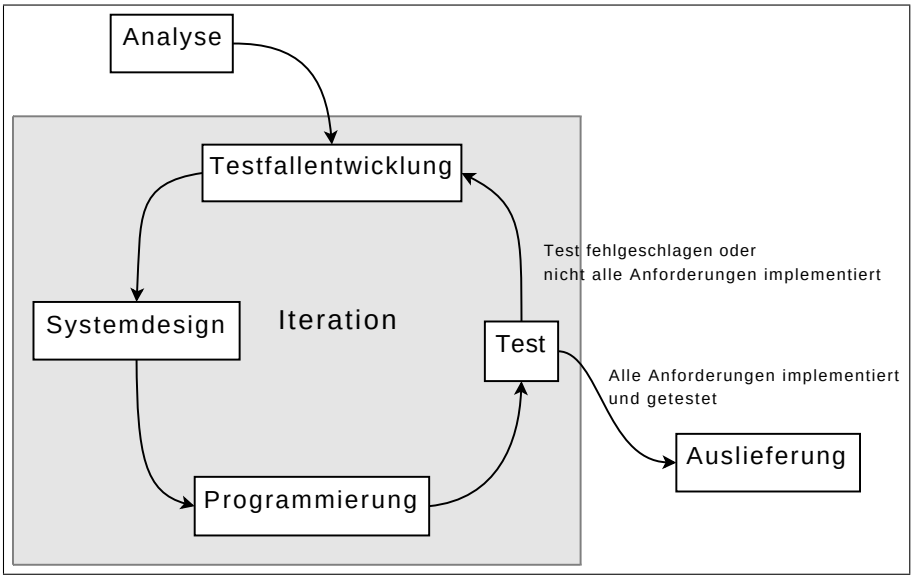


Abbildung 1.4: Schematische Darstellung eines iterativen (testgetriebenen) Prozesses

Test-driven Development geht noch einen Schritt weiter, indem Testfälle schon vor der Implementierung entwickelt werden. Das hat den Effekt, dass Modultests schon quasi parallel zur Implementierung durchgeführt werden können, also die Zeit zwischen Implementierung eines Programmteils und der Behebung von evtl. enthaltenen Fehlern weiter reduziert wird. Dementsprechend werden Integrationstests durchgeführt, sobald alle notwendigen Programmteile implementiert sind. Schwierigkeiten bei der Entwicklung der Testfälle deuten auf Mängel in der Spezifikation hin, damit kann diese ebenfalls schon vor der eigentlichen Implementierung verbessert werden. Abb. 1.4 illustriert das Vorgehen. Nach der Analyse beginnt die iterative Entwicklung des Programms. Jede Iteration besteht aus den Phasen Testfallentwicklung, Systemdesign, Programmierung und Test. In jeder Iteration wird ein Teilaspekt des Programms realisiert. Zuerst werden dazu Testfälle abgeleitet und in Form von Unit-Tests implementiert. Diese ergänzen schon vorhandene Unit-Test aus vergangenen Iterationen. Dann folgt das Design und die eigentliche Implementierung des Teilaspekts. Die Implementierung des neuen Teilaspekts und die bereits fertiggestellten Teile des Programms werden in der letzten Phase der Iteration mit Hilfe der Unit-Tests getestet. Es werden so lange neue Iterationen begonnen, bis alle Unit-Tests ohne Fehler abgearbeitet werden und das Programm alle Anforderungen erfüllt.

Kapitel 2

Zielsetzung und Aufbau der Arbeit

Die vorliegende Arbeit beschäftigt sich mit der Automatisierung des Softwaretests. In der Einleitung wurden die dabei auftretenden Probleme, die Erzeugung von Testeingaben und das Treffen einer Testentscheidung bereits beschrieben. In den folgenden Kapiteln der vorliegenden Arbeit werden nun Lösungsansätze für diese beiden Probleme vorgestellt.

Der Schwerpunkt der Arbeit liegt in der Automatisierung der Testentscheidung. Deshalb werden mehrere Ansätze zur Gewinnung von automatisierbaren Orakeln vorgestellt und an ausgesuchten Beispielen demonstriert. Diese Beispiele stammen überwiegend aus dem mathematisch-naturwissenschaftlichen Umfeld, u.a. wird Software zur Bildverarbeitung untersucht.

Die Vorstellung und Demonstration von Orakellösungen ist alleine jedoch wenig aussagekräftig. Es ist weitaus interessanter, die vorgestellten Lösungsansätze auch bezüglich ihrer Fähigkeit zu bewerten, fehlerhafte Programme als solche zu erkennen.

Die Bewertung der Orakellösungen erfolgt in mehreren Fallstudien. Zur Untersuchung werden dabei „echte“ Programme, hauptsächlich aus Open Source-Projekten verwendet. Damit soll demonstriert werden, dass die vorgestellten Lösungen in der Lage sind, Programme in praktisch relevanter Größe zu testen und nicht nur „akademische Spielzeuge“ sind.

Zur Durchführung der empirischen Untersuchungen werden zudem Methoden zur automatischen Erzeugung von Testeingaben benötigt. Diese sind in der vorliegenden Ar-

beit zwar ein wichtiger, aber nicht ein zentraler Bestandteil. Daher werden nur spezielle Erzeugungsmethoden vorgestellt, die auf die jeweiligen Testprobleme abgestimmt sind und wahrscheinlich auch nur im Zusammenhang mit diesen von Interesse sind. Da die Art der Eingabeerzeugung jedoch einen Einfluss auf die Bewertung der Orakellösungen haben kann, muss auch diese in die Untersuchungen miteinbezogen werden.

Obwohl sich die empirischen Studien immer nur auf sehr spezielle Testprobleme stützen, können die Resultate der Studien zusammengefasst und dazu verwendet werden, um allgemeine Hinweise für die Benutzung der untersuchten Testverfahren abzuleiten. Diese sollen helfen, diese Testverfahren auf andere Anwendungsgebiete zu übertragen und dort effektiv einzusetzen. Diese Hinweise reichen von einfachen Vorgehensbeschreibungen bis zu Kriterien, um a priori auf die Effektivität des Testverfahrens zu schließen.

Die Intention der Arbeit kondensiert sich in einem Verfahren zum vollständig automatisierten Testen von Bildanalysesoftware. Ein Teil der untersuchten Implementierungen stammt aus der GeoStoch-Bibliothek [109], die vom Institut für Stochastik in Zusammenarbeit mit dem Institut für Angewandte Informationsverarbeitung und dem DFG-Graduiertenkolleg 1100 entwickelt wird. In dem vorgestellten Testverfahren werden unterschiedliche Testtechniken demonstriert und zu einem effektiven Test kombiniert. Das vorgestellte Verfahren kann zum Testen einer Vielzahl von Bildanalyse/-verarbeitungs-Algorithmen verwendet werden und demonstriert die Möglichkeiten eines vollständig automatisierten Tests komplexer Anwendungen.

Die Arbeit ist wie folgt aufgebaut:

In Kapitel 3 werden zunächst unterschiedliche Ansätze für automatisierbare Orakellösungen vorgestellt. Zuerst wird in Kapitel 3.1 das sog. *Metamorphische Testen* präsentiert. Diese Technik wurde 1998 in [22] vorgestellt und ist seitdem Gegenstand wissenschaftlicher Untersuchungen. In Kapitel 3.2 wird erläutert, wie statistische Hypothesentests verwendet werden können, um eine Testentscheidung zu treffen. Der Schwerpunkt in der vorliegenden Arbeit ist es, Fehlentscheidungen dieser statistischen Tests zu reduzieren. Als dritte Methode werden dann in Kapitel 3.3 diese beiden Ansätze zu einem neuen Ansatz, dem sog. *statistisch-metamorphischen Testen* kombiniert.

Im folgenden Kapitel 4 werden dann die Techniken eingeführt, die zur Untersuchung und Bewertung der vorgestellten Orakellösungen benötigt werden. Zur Erzeugung der Testeingaben werden Methoden des Zufallstests verwendet, die in Kapitel 4.1 vorgestellt werden. Die Bewertung der Orakellösungen ist auf Ergebnissen der sog. *Mutationsanalyse*

aufgebaut. Diese wird in Kapitel 4.2 beschrieben. Anschließend wird die Bewertungsmethode für Teststrategien bzw. Orakellösungen präsentiert.

Nach der Vorstellung der Orakellösungen und der Untersuchungsmethoden folgen in Kapitel 5 zwei Fallstudien zur Untersuchung des Metamorphischen Testens. Die erste Fallstudie untersucht Implementierungen zur Berechnung von Matrixdeterminanten. Die zweite Fallstudie betrachtet unterschiedliche Implementierungen von Bildverarbeitungsalgorithmen.

Die Ergebnisse der Fallstudien aus Kapitel 5 werden im darauffolgenden Kapitel 6 zusammengefasst. Ein wesentliches Ergebnis der vorliegenden Arbeit ist dabei die Herleitung von Kriterien zur Anwendung Metamorphischer Tests und Maßnahmen zur Steigerung der Effektivität der Testmethode.

In Kapitel 7 werden Fallstudien vorgestellt, in denen die Anwendung der in den Kapiteln 3.2 und 3.3 präsentierten Verfahren analysiert wird. Nach der Präsentation der Fallstudien werden in Kapitel 7.3 allgemeine Hinweise für die Verwendung von statistischen Methoden im Softwaretest diskutiert.

In Kapitel 8 folgt dann die Zusammenfassung der Arbeit.

Kapitel 3

Untersuchte Methoden

3.1 Metamorphisches Testen

3.1.1 Einführung

Bei der praktischen Durchführung von Tests werden zunächst entweder Testfälle aus der Spezifikation abgeleitet oder der Tester wählt diese intuitiv oder nach seiner bisherigen Erfahrung aus. Besonders bei komplexen Berechnungen ist es jedoch oft nur für wenige spezielle Eingaben einfach möglich, die benötigten erwarteten Ausgaben zu bestimmen. Deswegen ist die manuelle Berechnung von erwarteten Ausgaben nicht dazu geeignet, eine große Anzahl an Testfällen zu erzeugen. Alternative Orakellösungen wie beispielsweise das *gold standard oracle* sind oft auch nicht anwendbar, da z.B. eine alternative, vertrauenswürdige Implementierung fehlt.

Es wäre also überaus hilfreich, eine andere Orakellösung zu haben, die es erlaubt, ohne Kenntnis der erwarteten Ausgaben eine Testentscheidung treffen zu können. *Metamorphisches*[1] *Testen* [22] ist eine solche Technik. Die Idee des Metamorphischen Testens ist es, die Testentscheidung nicht durch den Vergleich von tatsächlicher Ausgabe der SUT und einer erwarteten Ausgabe zu treffen. Stattdessen wird überprüft, ob mehrere Ausgaben der SUT gewisse Bedingungen erfüllen.

Als Beispiel sei eine Funktion `ggT(int a, int b)` gegeben, welche den größten gemeinsamen Teiler (ggT) zweier (positiver) Integerwerte bestimmt. Der ggT besitzt bekanntlich die Symmetrieeigenschaft `ggT(a,b)=ggT(b,a)`. Diese Symmetrieeigenschaft kann nun zum Testen einer Implementierung des ggT verwendet werden. Wählt man

[1] griechisch $\mu\varepsilon\tau\alpha\mu\rho\rho\varphi\omega\sigma\eta$ – Umwandlung, Verformung

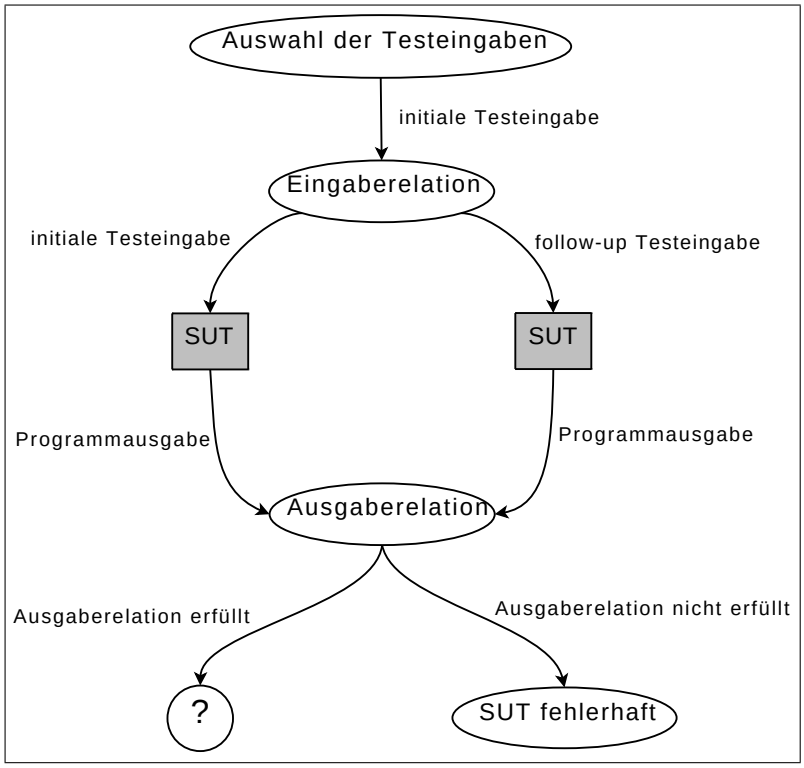


Abbildung 3.1: Schematische Darstellung von Metamorphischem Testen

die Eingaben so, dass das zweite Eingabepaar dem ersten Eingabepaar mit vertauschter Reihenfolge entspricht, dann müssen die Ausgaben zu diesen Eingaben gleich sein. Ist das nicht der Fall, so muss ein Fehler in der Implementierung vorliegen.

Beim Metamorphischen Testen werden die üblichen Testfälle bestehend aus Eingabe und zugehöriger erwarteter Ausgabe durch zwei Bedingungen ersetzt. Eine Bedingung muss für ein Tupel an Eingaben gelten und eine weitere muss dann durch die daraus erhaltenen Ausgaben erfüllt werden. Beide Bedingungen werden, wie sonst die Testfälle, entweder aus der Spezifikation des Programms abgeleitet oder stammen aus dem Domänenwissen des Testers.

Abb. 3.1 zeigt eine schematische Darstellung des Metamorphischen Testens. Zuerst werden ein oder mehrere initiale Testeingaben gewählt, üblicherweise zufällig. Zu diesen initialen Testeingaben werden weitere sog. *follow-up*-Testeingaben konstruiert, so dass die Eingaben die verlangte Bedingung erfüllen. Metamorphisches Testen ist damit nicht nur eine Methode zum Treffen der Testentscheidung, metamorphisches Testen ist zudem

eine Technik zur Erzeugung von Testeingaben. Anschließend wird das SUT mehrmals mit den erzeugten Eingaben ausgeführt. Dadurch erhält man mehrere Ausgaben. Als letztes wird nun überprüft, ob die erhaltenen Ausgaben der SUT die geforderte Bedingung erfüllen. Ist dies nicht der Fall, ist die SUT fehlerhaft.

Im obigen ggT-Beispiel würde also das erste Eingabe-Paar $i_1 = (i_{1,1}, i_{1,2})$ zufällig gewählt werden. Das zweite Eingabepaar i_2 würde dann aus dem ersten konstruiert, in dem die beiden Zahlen vertauscht würden. Anschließend wird der ggT beider Paare berechnet. Sind beide ggTs nicht gleich, ist die Implementierung fehlerhaft. Sind aber beide ggTs gleich, kann keine Aussage über die Korrektheit des SUT getroffen werden. Gibt das SUT beispielsweise für beliebige Eingaben immer die selbe Zahl zurück, sind die Programmausgaben offensichtlich gleich, das SUT ist aber fehlerhaft.

Formal kann die Methode wie folgt beschrieben werden:
Sei $S : \mathcal{I} \to \mathcal{O}$ die (funktionale) Spezifikation eines Programms wobei $\mathcal{I}$ die Menge der Eingaben des Programms und $\mathcal{O}$ die Menge der Soll-Ausgaben ist. Sind nun zwei Relationen $R_\mathcal{I} \subset \mathcal{I}^n$ und $R_\mathcal{O} \subset \mathcal{O}^n$ gegeben mit

$$\forall(i_1, \ldots, i_n) \in R_\mathcal{I} \Rightarrow (S(i_1), \ldots, S(i_n)) \in R_\mathcal{O},$$

dann muss für jede korrekte Implementierung P von S ebenfalls gelten, dass

$$\forall(i_1, \ldots, i_n) \in R_\mathcal{I} \Rightarrow (P(i_1), \ldots, P(i_n)) \in R_\mathcal{O}. \tag{3.1}$$

Die Implikation in (3.1) wird in Anlehnung an die Fachliteratur als *Metamorphische Relation* bezeichnet. Es handelt sich hierbei aber nicht um eine Relation im eigentlichen (mathematischen) Sinn. In der formalen Beschreibung des metamorphischen Testens werden die Bedingungen an Ein- und Ausgaben implizit über (mathematische) Relationen, also Teilmengen kartesischer Produkte, beschrieben.

Das Eingangs beschriebene Beispiel der Symmetrie des ggT kann formal wie folgt beschrieben werden. Die Menge der Eingabe ist die Menge $\mathcal{I} = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$ aller Paare ganzer Zahlen, von denen eine nicht gleich 0 ist. Die Menge der Soll-Ausgaben entspricht der Menge der ganzen Zahlen ohne 0, also $\mathcal{O} = \mathbb{Z} \setminus \{0\}$. Die Eingaberelation $R_\mathcal{I}$ setzt zwei Eingaben $i_1 = (i_{1,1}, i_{1,2})$ und $i_2 = (i_{2,1}, i_{2,2})$ zueinander in Beziehung und läßt sich durch

$$R_\mathcal{I} = \{(i_1, i_2) \in \mathcal{I}^2 | i_{2,1} = i_{1,2} \text{ und } i_{2,2} = i_{1,1}\}$$

beschreiben. Für zwei Ausgaben muss dann die Relation

$$R_{\mathcal{O}} = \{(g_1, g_2) \in \mathcal{O}^2 | g_1 = g_2\}$$

gelten.

In den folgenden empirischen Untersuchungen werden jedoch aus Gründen der besseren Verständlichkeit die Bedingungen angegeben, da die Mengendefinitionen für $R_{\mathcal{I}}$ und $R_{\mathcal{O}}$ die zugrunde liegenden Bedingungen oft nicht klar erkennen lassen. Am Beispiel des ggT könnten Relationen durch

- $ggT(a, b) = ggT(b, a)$
- $ggT(ka, kb) = k\, ggT(a, b),\ k > 0$
- $ggT(a, ka) = a,\ k > 0$

beschrieben werden.

Das Metamorphische Testen gehört zur Klasse der partiellen Orakel (siehe Definition 1). Da die Metamorphische Relation aus der Spezifikation des SUT abgeleitet ist, muss ein korrektes SUT die Metamorphische Relation erfüllen. Ist das nicht der Fall, ist das SUT nicht korrekt. Andererseits kann aus der Erfüllung der Metamorphischen Relation nicht auf die Korrektheit des SUT geschlossen werden, was am Beispiel des ggT bereits erläutert wurde.

3.1.2 Offene Fragen

Es ist an sich schon verwunderlich, dass man mit einem so einfachen Ansatz wie dem Metamorphischen Testen in der Lage ist, Fehlerwirkungen in Programmen zu erkennen. Deshalb ist die erste Frage, ob die Methode überhaupt effektiv einsetzbar ist. Bisherige Forschungsarbeiten haben versucht, Antworten auf diese Frage zu finden (beispielsweise in [24, 21]), die Ergebnisse beruhen aber auf sehr dünnem Datenmaterial, können und müssen deshalb noch verbessert werden. Unter Effektivität wird im Folgenden immer die Fähigkeit einer Testmethode bezeichnet, möglichst viele Fehler in Programmen zu finden. Im folgenden Kapitel 4 wird eine Methode vorgestellt, um die Effektivität einer Testmethode zu bewerten.

In den empirischen Untersuchungen zur Effektivität des Metamorphischen Testens soll zudem ein besonderes Augenmerk auf die Faktoren gelegt werden, die die Effekti-

vität beeinflussen. Eine Fragestellung ist, ob ausgehend von Struktureigenschaften der Relationen auf deren Effektivität geschlossen werden kann. Struktureigenschaften haben den Vorteil, bereits vor der Durchführung von Tests bekannt zu sein und können es daher ermöglichen, ohne empirische Versuche die Effektivität einer Relation einzuschätzen.

Basierend auf Struktureigenschaften und anderen Informationen sollen deshalb in der vorliegenden Arbeit Regeln für die Auswahl von Metamorphischen Relationen erarbeitet werden. Diese Regeln sollten eine a priori Einschätzung der Effektivität von Metamorphischem Testen ermöglichen. Dazu werden unterschiedliche Implementierungen mit einer großen Anzahl an Metamorphischen Relationen untersucht. Aus den Ergebnissen dieser empirischen Untersuchungen werden dann die Auswahlregeln abgeleitet werden.

Des weiteren wird die Effektivität sicherlich auch von der Art und Weise beeinflusst, wie die Eingabedaten erzeugt werden. Die zufällige Generierung ist i. A. nicht unproblematisch, da Fehler in Programmteilen, die nur mit einer sehr kleinen Wahrscheinlichkeit ausgeführt werden, selten oder im Extremfall nie entdeckt werden. Deshalb wird auch der Einfluss der Art der Eingabedatenerzeugung in dieser Arbeit untersucht werden.

3.1.3 Bisherige Forschung (related work)

Metamorphisches Testen wurde erstmals in [16] und [22] beschrieben und nochmals in [25] überarbeitet dargestellt.

Allerdings existieren frühere Ansätze, die durchaus in Zusammenhang mit Metamorphischem Testen stehen und die sicherlich zur Entstehung von Metamorphischem Testen beigetragen haben. Im wesentlichen können zwei Arbeiten, [2] und [34], als sehr verwandt mit dem Ansatz des Metamorphischen Testen bezeichnet werden. In der ersten Arbeit, [2], geht es allerdings nicht um Softwaretests, sondern um die Konstruktion fehlertoleranter Systeme. Diese sollen in der Lage sein, zur Laufzeit auftretende Fehler, beispielsweise in Berechnungen, zu korrigieren um davon unbeeinflusst weiterarbeiten zu können. In der aktuellen Forschung wird diese Eigenschaft auch als *self-healing* bezeichnet. Im Prinzip handelt es sich hier um eine dem Testen sehr ähnliche Aufgabenstellung. Beim Testen wird versucht, eine Fehlerwirkung als solche zu erkennen. Beim self-healing wird versucht, nicht nur Fehler bei der Programmausführung zu erkennen, sondern diese zur Laufzeit zu korrigieren, damit das Programm weiterarbeiten kann.

In [2] wird die Idee verwendet, dass oft unterschiedliche Programmabläufe zu denselben Ergebnissen führen. Ein Programm besteht üblicherweise aus einer Sequenz von

Funktionsaufrufen mit bestimmten Parametern. Will man beispielsweise eine einfache Berechnung wie $(a + b)c$ mit Aufrufen von `mult`(für eine Multiplikation) und `add`(für eine Addition) umsetzen, wäre eine mögliche Sequenz von Funktionsaufrufen durch

```
tmp=add(a,b);
result=mult(tmp,c);
return result;
```

gegeben. Das selbe Ergebnis könnte man aber auch mit der Sequenz

```
tmp=mult(a,c);
tmp2=mult(b,c);
result=add(tmp, tmp2);
return result;
```

berechnen. In [2] wird nun davon ausgegangen, dass nur bestimmte Parameter bzw. bestimmte Sequenzen von Funktionsaufrufen zu Fehlern führen, jedoch nicht alle. Werden nun Ergebnisse durch mehrere unterschiedliche Programmabläufe bestimmt, werden auch nur einige, aber nicht alle, Ergebnisse fehlerbehaftet sein. Aus allen Ergebnissen können dann die korrekten ausgewählt werden, beispielsweise durch einen Mehrheitsentscheid. Die Idee, dass unterschiedliche Programmabläufe zu denselben Ergebnissen führen, wird auch beim Metamorphischen Testen verwendet. Dort wird jedoch aus einer Abweichung der Ergebnisse auf einen Fehler im Programmablauf geschlossen, und nicht wir im hier vorgestellten Ansatz aus der Übereinstimmung der Ergebnisse auf ein funktionierendes Programm.

Diese Idee, unterschiedliche Programmabläufe mit gleichem Ergebnis als Testkriterium zu verwenden, wurde in [34] auf das Testen von (objektorientierter) Software übertragen. Auf Objekten einer zu testenden Klasse werden unterschiedliche Sequenzen von Methodenaufrufen abgearbeitet, die laut Spezifikation zu dem gleichen Zustand der untersuchten Objekte führen müssen. Befinden sich die Objekte nach den Methodenaufrufen nicht im gleichen Zustand, muss sich ein Fehler in einer der aufgerufenen Methoden befinden. Das SUT wird deshalb als fehlerhaft klassifiziert.

Der Unterschied von Metamorphischem Testen und dem in [34] beschriebenen Ansatz ist, dass nicht objektorientierte SUTs sondern prozedurale SUTs getestet werden. In der vorliegenden Literatur zu Metamorphischem Testen wird immer nur eine einzige Funktion bzw. Prozedur untersucht, deren Verhalten nur von den Parametern der

Funktion abhängen. Unterschiedliche Funktionsaufrufe mit dem selben Ergebnis können also nur durch eine geeignete Wahl der Parameter ausgelöst werden. Zudem verlangt Metamorphisches Testen nicht, dass die Ausgaben des Programms für unterschiedliche Eingaben gleich sind, sondern nur dass sie gewisse Bedingungen erfüllen (die durch die Ausgaberelation $R_{\mathcal{O}}$ beschrieben werden).

Unabhängig von diesen Arbeiten wurde in [45] ein mit Metamorphischem Testen vergleichbarer Ansatz entwickelt, der sich jedoch auf symmetrische Relationen beschränkt. Dazu wird versucht, über Methoden der statischen Analyse Paare von Eingaben (i_1, i_2) zu bestimmen, für die die Eigenschaft $P(i_1) = \eta P(i_2)$ gelten muss.

Im weiteren Sinne ist Metamorphisches Testen auch mit dem in [13] beschriebenen Ansatz des *program checkers* eines Programms verwandt. Dort werden Algorithmen vorgestellt, die ähnlich zu metamorphischen Relationen basierend auf speziell gewählten Eingaben über die Korrektheit der Programmausgaben für diese Eingaben entscheiden. In den Beispielen der Arbeit werden u.a. *program checkers* für Graphisomorphie-Probleme vorgestellt.

Ein verwandter Ansatz, der auf den Ansatz der *program checker* zurückzuführen ist, ist in [26] beschrieben. Die Arbeit beschreibt QuickCheck, ein Werkzeug zum automatisierten Testen von Haskell-Programmen. Zur Testentscheidung werden Eigenschaften des zu testenden Programms verwendet, die vergleichbar mit Metamorphischen Relationen sind. Als Beispiel wird eine Funktion `reverse` getestet, die eine Liste invertiert. Der Operator `++` konkateniert zwei Listen. Eine der zum Testen verwendeten Eigenschaften ist `reverse(x++y)= reverse(y)++reverse(x)`: Wird eine Liste, die aus zwei Listen `x` und `y` zusammengesetzt ist, invertiert, so erhält man das Ergebnis auch, indem man die invertierten Listen `reverse(y)` und `reverse(x)` zusammensetzt.

In den vergangenen Jahren wurden zahlreiche Arbeiten zum Thema Metamorphisches Testen veröffentlicht. Dabei sind zwei Hauptrichtungen der Forschung erkennbar. Zum einen wurde versucht, Metamorphisches Testen in unterschiedlichen Anwendungsgebieten zu verwenden, zum anderen wurde versucht, a priori Aussagen über die Qualität von Metamorphischen Relationen zu treffen.

In den bisherigen Arbeiten wurde Metamorphisches Testen hauptsächlich dazu verwendet Programme zu testen, die auf einer mathematisch formulierten Spezifikation beruhen, beispielsweise Programmen zum Lösen partieller Differentialgleichungen ([23]) oder Programmen zur Suche kürzester Wege in Graphen ([24]). Allerdings gibt es eben-

falls Arbeiten zur Anwendung von Metamorphischem Testen in anderen Bereichen , beispielsweise beim Testen von Middleware-basierten Anwendungen ([108, 19]) oder Web Services ([18]). Weitere potentielle Anwendungsbereiche sind in [113] beschrieben.

Zum Thema „Qualität von Metamorphischen Relationen“ und dem „Einfluss der Testdatengenerierung auf die Fehlererkennungsfähigkeit der Testmethode“ sind nur zwei Arbeiten veröffentlicht worden. In [24] wird versucht, Auswahlkriterien für Metamorphische Relationen zu formulieren, in [21] wird Metamorphisches Testen in Verbindung mit Testen mit Spezialfällen untersucht. In beiden Arbeiten werden zu wenige empirische Daten verwendet um allgemeinere Aussagen abzuleiten. Diese Aussagen konnten in der vorliegenden Arbeit teilweise widerlegt werden. Diese Resultate wurden bereits in [70] veröffentlicht.

3.2 Testen von randomisierter Software

Bei Programmen besteht immer ein deterministischer Zusammenhang zwischen Ein- und Ausgaben. D. h., mehrere Ausführungen eines Programms führen bei gleichen Eingaben immer zu den selben Ausgaben. Oft können jedoch nicht alle Eingaben kontrolliert werden. Kontrollierbar bedeutet in diesem Fall, dass der Wert der Eingabe vor der Programmausführung festgelegt werden kann. Wenn die nicht kontrollierbaren Eingaben bei unterschiedlichen Programmausführungen unterschiedliche Werte annehmen, können auch die Ausgaben voneinander abweichen. Diesen Einfluss der nicht kontrollierbaren Eingaben kann man nun als zufällig ansehen.

Das einfachste Beispiel für ein solches Programm ist ein Pseudo-Zufallszahlengenerator. Dieser wird durch einen Seed initialisiert. Der Seed wird dabei aus Informationen gewonnen, die als zufällig angesehen werden können, beispielsweise die Systemzeit oder -auslastung. Wird der Seed nicht kontrolliert, wird bei jedem Aufruf eine andere „zufällige“ Folge von Zahlen erzeugt.

Im wissenschaftlichen und technischen Umfeld ist die Simulation von komplexen Systemen ein häufig verwendetes Hilfsmittel, um Aussagen über diese Systeme zu treffen. Im Graduiertenkolleg werden beispielsweise Zeitreihen von Aktienkursen oder Wechselkursen simuliert, um synthetisches Datenmaterial für weitere finanzmathematische Verfahren zu erhalten.

Der „klassische“ Testansatz, zu (kontrollierbaren) Eingaben die erwarteten Ausgaben zu bestimmen und diese dann mit den tatsächlichen Ausgaben zu vergleichen, kann hier jedoch nicht angewendet werden, da ein deterministischer Zusammenhang zwischen den kontrollierbaren Eingaben und den Ausgaben des Programms erforderlich ist.

Im folgenden wird nun davon ausgegangen, dass nicht alle Eingaben kontrollierbar sind, und dass die nicht kontrollierbaren Eingaben einen Einfluss auf die Programmausgaben haben. Die daraus resultierenden unterschiedlichen Ausgaben zu gleichen (kontrollierbaren) Eingaben führen zu der Idee, die Ausgaben als zufällig zu betrachten. Die Klasse von Programmen mit zufälligen Ausgaben wird im Weiteren als *randomisierte Software* bezeichnet.

Es spielt nun offensichtlich keine Rolle, warum keine Kontrolle der Eingaben möglich ist. Mögliche Ursachen sind entweder in der Konstruktion des Programms oder in der Wahl der Eingaben begründet. Enthält ein Programm beispielsweise einen Zufallszahlengenerator dessen Seed nicht beeinflusst werden kann, liegt eine konstruktionsbeding-

te Unkontrollierbarkeit der Eingaben vor. Werden dagegen Zufallszahlen als Eingaben verwendet, so ist die Wahl der Eingaben der Grund für die Unkontrollierbarkeit der Eingaben. Wenn im folgenden von „Eingaben“ die Rede ist, werden damit nur die kontrollierbaren Eingaben gemeint.

3.2.1 Offene Fragen

Um randomisierte Software zu testen wird oft die Verwendung von statistischen Methoden vorgeschlagen. In der Literatur finden sich immer wieder Fallstudien ([17, 39, 68, 69, 100]), die die Verwendung von statistischen Hypothesentests zum Treffen von Testentscheidungen empfehlen. In allen Arbeiten fehlen jedoch sowohl eine theoretische Begründung der Verwendung von statistischen Tests als auch ein probabilistisches Modell dieser Programme und eine fundierte Analyse der Methode. In der vorliegenden Arbeit werden deshalb zuerst die Voraussetzungen für die Anwendung von statistischen Methoden im Softwaretest überprüft.

Das größte Problem bei der praktischen Anwendung statistischer Verfahren ist jedoch die Möglichkeit von Fehlentscheidungen. Ein wesentlicher Bestandteil der vorliegenden Arbeit ist deshalb ein Verfahren, die Wahrscheinlichkeit von Fehlentscheidungen zu reduzieren. In Kapitel 3.2.5 wird dieses Verfahren hergeleitet. Abschließend wird in Kapitel 3.2.6 der Ablauf eines Softwaretests mit statistischen Methoden beschrieben.

Die Resultate aus diesem Bereich wurden teilweise bereits in [49] veröffentlicht.

3.2.2 Statistische Grundlagen

Der Schwerpunkt der vorliegenden Arbeit ist das Testen von Software. Beim Testen von randomisierter Software sind statistische Verfahren ein wichtiges Werkzeug für die Durchführung der Softwaretests. In der vorliegenden Arbeit werden ausschließlich bekannte statistische Verfahren verwendet. Die zugrundeliegende Mathematik wird deswegen nur sehr selektiv dargelegt. Ausführliche Informationen zu statistischen Tests und deren Anwendung findet man z.B. in [15].

Die Statistik dient der Beschreibung und der Interpretation beobachteter Daten. Im folgenden werden die notwendigen Begriffe erläutert.

Die Grundlage der statistischen Methoden ist eine Beobachtung $x = (x_1, \ldots, x_n)$, *Stichprobe* genannt. Die Größe der Stichprobe wird als Stichprobenumfang n bezeichnet. Diese Stichprobe wird als Realisierung eines Zufallsvektors $X = (X_1, \ldots, X_n)$ interpre-

tiert. Zudem wird in der vorliegenden Arbeit angenommen, dass die Zufallsvariablen $X_1, \ldots, X_n$ unabhängig sind und dieselbe Verteilung besitzen. Die Annahme der Unabhängigkeit der Stichprobenvariablen ist nicht zwingend erforderlich, ermöglicht aber die Verwendung einfacherer Verfahren. In Kapitel 3.2.4 wird begründet, warum die Annahme der Unabhängigkeit für die Untersuchungen der vorliegenden Arbeit keine wesentliche Einschränkung darstellt.

Da angenommen wird, dass die Zufallsvariablen $X_1, \ldots, X_n$ identisch verteilt sind, gilt $\mathbb{E}(X_1) = \mathbb{E}(X_2) = \ldots \mathbb{E}(X_n)$. Damit besteht der Erwartungswert(vektor) $\mathbb{E}(X)$ von X aus identischen Einträgen. Diese können aus der Zufallsstichprobe durch das sog. Stichprobenmittel (empirischer Erwartungswert)

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i$$

näherungsweise bestimmt werden. Sind die Stichprobenvariablen zudem unabhängig, ist die Kovarianzmatrix von X eine Diagonalmatrix. Die Werte der Diagonalelemente ist ebenfalls identisch und können mit der Stichprobenvarianz

$$S_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X}_n)^2$$

näherungsweise ermittelt werden. Beziehen sich diese Funktionen nicht auf die Zufallsstichprobe X sondern auf die konkrete Stichprobe x, werden sie mit $\bar{x}_n$ bzw. s_n^2 bezeichnet und werden oft zur Beschreibung der konkreten Stichprobe verwendet.

Neben der Beschreibung von Daten (deskriptive Statistik) können mit statistischen Verfahren auch Rückschlüsse auf Eigenschaften der Daten gezogen werden. Dazu werden sog. *Hypothesentests* verwendet. Diese bestimmen grob gesagt die Wahrscheinlichkeit, mit der eine gewisse Annahmen, die sog. *Nullhypothese*, aufgrund rein zufälliger Effekte zutreffen. Ist die Wahrscheinlichkeit dafür groß, muss die Nullhypothese verworfen werden. Ist diese Wahrscheinlichkeit dagegen gering, kann die Nullhypothese nicht verworfen werden.

Die Ergebnisse statistischer Hypothesentests sind generell als fehlerbehaftet anzusehen. Zwei Arten von Fehlern können dabei auftreten. Der Fehler erster Art (α-Fehler) tritt auf, wenn die Nullhypothese verworfen wird, obwohl diese zutrifft. Die Wahrscheinlichkeit für das Auftreten dieses Fehlers (*Konfidenzniveau*) kann konstruktionsbedingt

bei statistischen Hypothesentests a priori festgelegt werden. Der Fehler zweiter Art (β-Fehler) tritt dagegen dann auf, wenn die Nullhypothese nicht verworfen wird, obwohl diese zu verwerfen wäre. Im Gegensatz zum α-Fehler kann die Wahrscheinlichkeit für diesen Fehler nicht festgelegt werden.

3.2.3 Testentscheidung durch statistische Hypothesentests

Zum Testen von randomisierter Software wird ebenfalls eine funktionale Spezifikation benötigt. In der Spezifikation werden unter anderem die erwarteten Ausgaben des Programms beschrieben. Diese werden als Zufallsvariable interpretiert und im folgenden mit S bezeichnet. Die Spezifikation ist damit also eine messbare Abbildung $S: \Omega \to \mathcal{O}$ von einem nicht näher spezifizierten Wahrscheinlichkeitsraum $(\Omega, \mathcal{F}, \mathbb{P})$ in die Menge der Ausgaben $\mathcal{O}$. Die Ausgaben des SUT werden ebenfalls als Zufallsvariable interpretiert. Diese Zufallsvariable wird mit P bezeichnet.

Das Problem der Testentscheidung, also ob die Programm-Ausgaben P den erwarteten Ausgaben S entsprechen, kann formal durch die Gleichheit $S = P$ beschrieben werden. Da es sich bei S und P um Zufallsvariablen handelt, muss natürlich ein anderer als der herkömmliche Gleichheitsbegriff verwendet werden. Für diese Gleichheit von Zufallsvariablen existieren unterschiedliche Definitionen. Die Wahl der Definition der Gleichheit legt fest, welche statistischen Verfahren verwendet werden müssen, um Aussagen über die Gleichheit der Ausgaben der SUT und den in der Spezifikation beschriebenen erwarteten Ausgaben machen zu können.

Die Definition der „Gleichheit in Verteilung“ $S \stackrel{d}{=} P$ führt zu einem sog. *nicht-parametrischen* Testproblem. Zwei Zufallsvariablen sind gleich in Verteilung, wenn ihre Verteilungsfunktionen F_S und F_P gleich sind, also wenn $F_S(x) = F_P(x) \forall x \in \mathbb{R}$ gilt. Diese Art der Gleichheit ist direkt durch statistische Tests nachgeprüft werden. Es eignen sich beispielsweise der Kolmogorov-Smirnov-Test oder der χ^2-Anpassungstest (vgl. [15]).

In der vorliegenden Arbeit wird jedoch ein abgeschwächter Gleichheitsbegriff verwendet. Die Programmausgaben und die erwarteten Ausgaben werden dann als gleich angesehen, wenn sie in bestimmten Verteilungs-Charakteristiken übereinstimmen. Ein Beispiel für eine solche Charakakteristik ist der Erwartungswert. Dieser Ansatz führt zu einem sog. *parametrischen* Testproblem. Dabei wird angenommen, dass die Verteilungen von P und S zu einer parametrischen Verteilungsfamilie gehören. Beide Zufallsvariablen werden als gleich bezeichnet, wenn die Parameter ihrer Verteilungen übereinstimmen.

Als Tests eignen sich hier beispielsweise der t-Test bzw. bei großen Stichprobenumfängen der (asymptotische) Gauß-Test. Dieser wird in Kapitel 3.2.6 beschrieben.

Unabhängig von der Wahl des Gleichheitsbegriffs sind grundsätzlich zwei Vorgehensweisen bei der Durchführung der Tests möglich. Ist die (theoretische) Verteilungsfunktion F_S von S (oder im parametrischen Fall die Charakteristiken bzw. Parameter der Verteilung) bekannt, kann der Test mit einem geeigneten statistischen (ein-Stichproben-) Test durchgeführt werden. Sind die theoretischen Größen nicht bekannt, aber es existiert eine alternative Implementierung, so kann diese verwendet werden um Tests durchzuführen. Dazu wird dann ein geeigneter statistischer (zwei-Stichproben-) Test auf Gleichheit der Verteilungen (bzw. Gleicheit der Charakteristiken/Parameter) zweier Stichproben durchgeführt. Damit erhält man die statistische Version eines *gold standard oracles* (vgl. [12]).

Das in der vorliegenden Arbeit angewendete Vorgehen wird in Kapitel 3.2.6 beschrieben.

3.2.4 Vorüberlegungen

Um statistische Hypothesentests überhaupt anwenden zu können, muss zuerst sichergestellt sein, dass deren Voraussetzungen überhaupt zutreffen. Grundvoraussetzung ist, dass die Ausgaben P von randomisierter Software als Zufallsvariable interpretiert werden können. Diese Voraussetzung wird oft als gegeben vorausgesetzt, kann aber auch formal begründet werden. In [61] finden sich dazu zwei Beweise. Die einfache Modellierung der Programmausgaben als Zufallsvariable erlaubt es nun, auf die bekannten Methoden der Statistik zurückgreifen zu können, ohne neue Techniken entwickeln zu müssen.

In Kapitel 3.2.2 wird angenommen, dass die Stichprobenvariablen unabhängig sind. Diese Annahme ist nicht zwingend erforderlich. Durch die Annahme der Unabhängigkeit können jedoch einfachere Standardverfahren verwendet werden. Diese sind bereits in einer Vielzahl von Softwarepaketen implementiert. Die Verwendung von bereits implementierten Standardverfahren senkt den Aufwand für die eigentlichen Softwaretests.

Die Unabhängigkeit der Stichprobenvariablen muss bei der Durchführung der Tests durch den Tester sichergestellt sein. Im Allgemeinen sind diese unabhängig, wenn die Programmausgabe nicht durch vorherige Ausführungen des SUT beeinflusst werden. Im einfachsten Fall kann dieses durch die Zustandslosigkeit des SUT gewährleistet werden. Problematisch sind hier objektorientierte SUTs, da diese sehr oft ein zustandsabhängiges Verhalten aufweisen. In diesem Fall kann die Unabhängigkeit von Programmausgaben

dadurch erreicht werden, dass bei jeder Ausführung der SUT auf neu erzeugten Objekten gearbeitet wird. Da es im Allgemeinen erwünscht ist, dass sich unterschiedliche Testläufe nicht gegenseitig beeinflussen, stellt die Annahme der Unabhängigkeit keine wesentliche Einschränkung dar.

Statistische Standardverfahren operieren häufig auf reellen Zahlen oder Mengen mit endlich vielen Elementen. Deshalb können statistische Hypothesentests oft nicht direkt auf die Programmausgaben angewendet werden. Das ist beispielsweise der Fall, wenn Funktionen getestet werden sollen, die komplexe Objekte als Rückgaben liefern. Die Modellierung der Programmausgaben als Zufallsvariable hat hier den entscheidenden Vorteil, dass die Ausgaben durch (messbare) Abbildungen transformiert werden können (siehe Abb. 3.2.4). Die transformierten Ausgaben sind dann ebenfalls wieder Zufallsvariablen. Diese transformierten Ausgaben können dann wieder mit Standardverfahren untersucht werden. Durch die Verwendung von Standardverfahren kann wiederum der Aufwand für die Testdurchführung reduziert werden, weil aufwändige Implementierungen von speziellen Testverfahren nicht unbedingt notwendig sind.

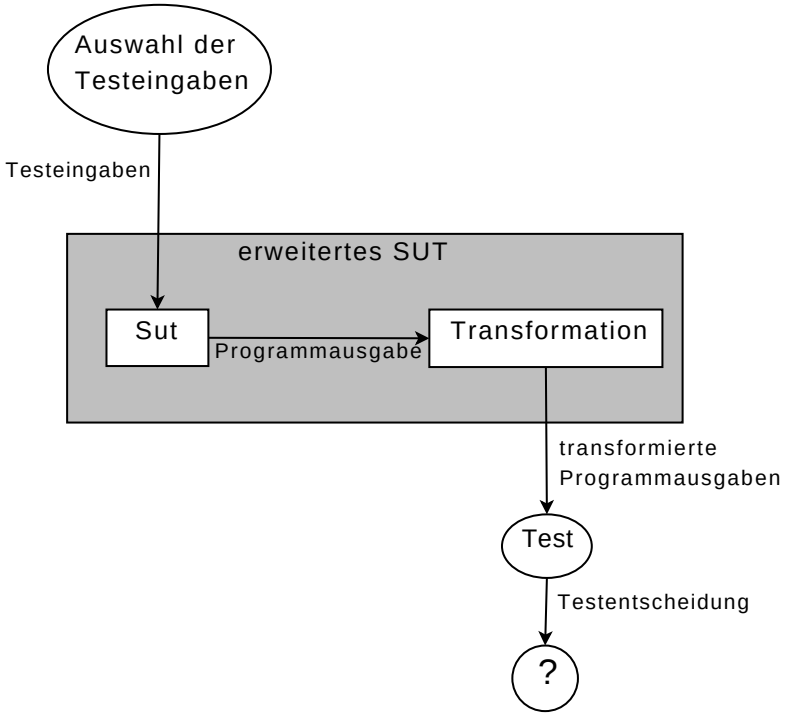


Abbildung 3.2: Testaufbau beim statistischen Testen

Da bei den statistischen Tests nur die (ggf. transformierten) Programmausgaben betrachtet werden, kann das SUT und die Transformation der Ausgaben zu einer Art erweiterten SUT zusammengefasst werden. Im folgenden ist deshalb oft statt des eigentlichen SUT das „erweiterte SUT“ gemeint und die Transformation der Ausgaben wird nicht explizit erwähnt.

Durch die Transformation der Ausgaben wird die Anwendung von statistischen Hypothesentests oft erst ermöglicht. In Kapitel 7 werden beispielsweise geometrische Objekte auf Kennzahlen dieser Objekte abgebildet. Die Objekte selbst könnten nicht direkt durch statistische Tests geprüft werden, nur die Kennzahlen sind dazu geeignet.

3.2.5 Reduktion von Fehlentscheidungen

In Kapitel 3.2.2 wurde bereits erwähnt, dass die Entscheidung durch einen statistischen Hypothesentest evtl. fehlerbehaftet ist und kann deshalb nicht ohne weiteres verwendet werden, um Fehler in Programmen festzustellen.

Schon in [69] wurde versucht, durch eine geschickte Konstruktion des statistischen Hypothesentests den eigentlich interessanten β-Fehler zu kontrollieren. In dieser Arbeit wurde statt eines zweiseitigen t-Tests ein Intersection-Union-Test verwendet, um statt der eigentlichen Nullhypothese $H_0: \mu = \mu_0$ die Nullhypothese $H_0: \mu \neq \mu_0$ zu testen. Dadurch werden die Bedeutung von α-Fehler und β-Fehler vertauscht. Dieses Vorgehen erfordert allerdings eine sehr spezielle Wahl der verwendeten Hypothesentests und ist deshalb nicht allgemein anwendbar.

In diesem Kapitel wird deshalb eine allgemein anwendbare Methode vorgestellt, um die Wahrscheinlichkeit einer Fehlentscheidung zu reduzieren. Die Methode ist auf bekannten Resultaten aus der Komplexitätstheorie aufgebaut. Die folgende Beschreibung orientiert sich an den Ausführungen in [92]. Im Zusammenhang mit randomisierten Algorithmen wird diese Methode auch als *Wahrscheinlichkeitsverstärkung* bezeichnet.

Zur Herleitung der Methode wird die Testentscheidung durch einen statistischen Hypothesentest als Binärexperiment interpretiert. Bei diesem Experiment wird mit Wahrscheinlichkeit $0 < p < 1$ die falsche Entscheidung getroffen, und entsprechend mit Wahrscheinlichkeit $1 - p$ die richtige. Die Wahrscheinlichkeit für eine Fehlentscheidung ist durch die Wahrscheinlichkeiten für einen α-Fehler und einen β-Fehler vorgegeben. Sind beide Wahrscheinlichkeiten kleiner als $\frac{1}{2}$, so kann die im folgenden beschriebene Methode angewandt werden.

Gilt für die Wahrscheinlichkeit einer Fehlentscheidung p das $p < \frac{1}{2}$, gehört der beschriebene Ansatz zur Komplexitätsklasse **BPP**(„bounded probability of error“) (nach [92]):

Definition 2 *Komplexitätsklasse **BPP***

*Die Komplexitätsklasse **BPP** enthält alle Sprachen L, für die eine nicht-deterministische und polynomiell beschränkte Turing-Maschine N existiert, so dass für alle Eingaben $x \in L$ ein $p < \frac{1}{2}$ existiert, so dass mindestens $(1-p)\%$ der Berechnungen von N die Eingabe x akzeptieren. Wenn $x \notin L$, wird x in mindestens $(1-p)\%$ der Berechnungen nicht akzeptiert.*

Für Probleme aus **BPP** lässt sich durch R-fache Wiederholung der Berechnung und anschließendem Mehrheitsentscheid die Wahrscheinlichkeit für ein falsches Ergebnis deutlich reduzieren. Diese Technik wird auch als *Wahrscheinlichkeitsverstärkung* bezeichnet. Die *Chernoff-Hoeffding-Schranke* [92] gibt eine Abschätzung für die Wahrscheinlichkeit einer Fehlentscheidung nach R Wiederholungen an. Damit kann die Chernoff-Hoeffding-Schranke dazu verwendet werden, die notwendige Anzahl der Wiederholungen R zu bestimmen, so dass die Wahrscheinlichkeit für eine falsche Mehrheitsentscheidung kleiner als $\psi > 0$ ist.

Satz 2 *Chernoff-Hoeffding-Schranke*

Seien $X_1, \ldots, X_R$ unabhängige Zufallsvariablen, die die Werte 1 und 0 mit Wahrscheinlichkeit p bzw. $1-p$ annehmen. Des weiteren sei $X = \sum_{i=1}^{R} X_i$. Dann gilt für $0 \leq \theta \leq 1$

$$\mathbb{P}(X \leq (1+\theta)pR) \leq e^{-\frac{\theta^2}{3}pR}$$

Setzt man $p = \frac{1}{2} + \epsilon$ und $\theta = \frac{\epsilon}{\frac{1}{2}+\epsilon}$ für ein $\epsilon > 0$ (vgl. [92]), so erhält man aus der Chernoff-Hoeffding-Schranke

$$\psi \leq e^{-\frac{\epsilon^2 R}{6}}$$

In [92] wird zudem empfohlen, R ungerade zu wählen, also $R = 2k+1$, um für ϵ nahe bei 0 ein Unentschieden bei der Mehrheitsentscheidung auszuschließen. In diesem Fall

vereinfacht sich die obige Ungleichung zu

$$\psi \leq e^{-2\epsilon^2 k}. \tag{3.2}$$

Gibt man nun eine Wahrscheinlichkeit ψ für eine Fehlentscheidung vor, kann man die notwendige Anzahl an Wiederholungen einfach bestimmen:

$$k = \left\lceil -\frac{\ln \psi}{2\epsilon^2} \right\rceil, \tag{3.3}$$

wobei $\lceil x \rceil$ die kleinste Integer-Zahl k mit $k \geq x$ bezeichnet.

Die Methode der Wahrscheinlichkeitsverstärkung lässt sich direkt auf Testentscheidungen durch statistische Hypothesentests übertragen. Sind Abschätzungen für α- und β-Fehler bekannt, kann die notwendige Anzahl an Wiederholungen R nach (3.3) für eine gegebene Gesamt-Fehlerwahrscheinlichkeit ψ bestimmt werden. Der eigentliche Test wird dann R-mal durchgeführt. Wird die Nullhypothese des Tests verworfen, wird ein Zähler inkrementiert. Ist der Zähler nach R Testdurchführungen größer als $R/2$, so ist die SUT als fehlerhaft anzusehen.

3.2.6 Zusammenfassung des Testansatzes

Der bisher beschriebene Ansatz ist sehr allgemein gehalten, da prinzipiell jeder statistische Hypothesentest zur Testentscheidung herangezogen werden kann. Praktisch kann jedoch für die meisten bekannten statistischen Tests die Wahrscheinlichkeit für den β-Fehler nicht abgeschätzt werden. Aber nur wenn $\beta < \frac{1}{2}$ gilt, ist die Methode der Wahrscheinlichkeitsverstärkung anwendbar. Für einen (zweiseitigen) asymptotischen Gauß-Test kann diese Wahrscheinlichkeit abgeschätzt werden. Deshalb wurde bei der praktischen Durchführung der Experimente ein (zweiseitiger) asymptotischer Gauß-Test für den Erwartungswert verwendet.

Die Nullhypothese dieses Tests lautet $H_0 : \mu = \mu_0$, also dass der tatsächliche Erwartungswert der Stichprobenvariablen $X_1, \ldots, X_n$ einem vorgegebenen Wert μ_0 entspricht. Die Alternativ-Hypothese H_1 lautet entsprechend $H_1 : \mu \neq \mu_0$. Zur Konstruktion des Tests wird wie folgt vorgegangen:

Seien $X_1, \ldots, X_n$ unabhängig und identisch verteilt und $\mu_0 \in \mathbb{R}$ gegeben. Dann gilt

$$T_n(X_1, \ldots, X_n) = \frac{\bar{X}_n - \mu_0}{\sqrt{\frac{S_n^2}{n}}} \xrightarrow{d} \mathcal{N}(0,1)(n \to \infty). \tag{3.4}$$

Diese Behauptung folgt aus dem Zentralen Grenzwertsatz und dem Satz von Slutzky.

Aus (3.4) folgt also, dass die Größe T_n asymptotisch normalverteilt ist. Es gilt, dass

$$\lim_{n\to\infty} \mathbb{P}\left(|T_n(X_1, \ldots, X_n)| \leq z_{1-\frac{\alpha}{2}}\right) \leq \alpha,$$

wobei $z_{1-\frac{\alpha}{2}}$ das $1-\frac{\alpha}{2}$-Quantil der Standardnormalverteilung ist und $0 < \alpha < \frac{1}{2}$. Die Größe T_n kann deshalb als Testfunktion für die oben formulierte Hypothese zum Konfidenzniveau α verwendet werden. Für eine konkrete Stichprobe ist die Nullhypothese abzulehnen, wenn

$$|T(x_1, \ldots, x_n)| > z_{1-\frac{\alpha}{2}}.$$

Der Gauß-Test hat den Vorteil, dass selbst bei nicht normalverteilter Grundgesamtheit der Test durchgeführt werden kann, dann aber nicht als exakter Test sondern als asymptotischer Test. In diesem Fall muss allerdings ein großer Stichprobenumfang gewählt werden um ein gültiges Resultat zu erhalten. In der Literatur werden Stichprobenumfänge von mindestens $n = 30$ vorgeschlagen.

Unter der Annahme, dass $T(\cdot)$ normalverteilt ist, lässt sich für den ein-Stichproben-Gauß-Test die Wahrscheinlichkeit β für eine falsch-negative Entscheidung wie folgt abschätzen. Sind der Stichprobenumfang n und eine Toleranz/Genauigkeit $\delta > 0$ gegeben, so gilt:

$$\begin{aligned}
\beta &= \mathbb{P}\left(\left|\frac{\bar{X}_n - \mu_0}{S_n/\sqrt{n}}\right| \leq z_{1-\alpha/2}\right) \\
&= \mathbb{P}\left(z_{\alpha/2} \leq \frac{\bar{X}_n - \mu}{S_n/\sqrt{n}} + \frac{\delta\sqrt{n}}{S_n} \leq z_{1-\alpha/2}\right) \\
&= \mathbb{P}\left(z_{\alpha/2} - \frac{\delta\sqrt{n}}{S_n} \leq \frac{\bar{X}_n - \mu}{S_n/\sqrt{n}} \leq z_{1-\alpha/2} - \frac{\delta\sqrt{n}}{S_n}\right) \\
&= \Phi\left(z_{1-\alpha/2} - \frac{\delta\sqrt{n}}{S_n}\right) - \Phi\left(z_{\alpha/2} - \frac{\delta\sqrt{n}}{S_n}\right) \\
&\leq \Phi\left(z_{1-\alpha/2} - \frac{\delta\sqrt{n}}{S_n}\right),
\end{aligned}$$

wobei $\Phi(\cdot)$ die Verteilungsfunktion der Standardnormalverteilung bezeichnet. Die Abschätzung für den zwei-Stichproben-Test erfolgt analog.

Sind der β-Fehler und der Stichprobenumfang gegeben, so folgt aus der obigen Abschätzung

$$z_\beta \geq z_{1-\alpha/2} - \frac{\delta\sqrt{n}}{S_n} \Leftrightarrow \delta \geq \frac{S_n}{\sqrt{n}}(z_{1-\alpha/2} - z_\beta) \tag{3.5}$$

Bei der praktischen Anwendung kann es sinnvoller sein, die Fehlerwahrscheinlichkeit und den Stichprobenumfang vorzugeben, und dann a posteriori die Genauigkeit δ zu berechnen. Ist diese „zu groß“, kann der Test mit anderen Parametern wiederholt werden.

Ein Test auf Gleichheit der Erwartungswerte ist allein genommen ein schwaches Kriterium für die Gleichheit zweier Zufallsvariablen. Zusätzlich zum Test auf gleiche Erwartungswerte wäre ein allgemein anwendbarer Test auf Gleichheit höherer Momente sinnvoll. Allerdings hängen die Teststatistiken für diese Tests von der Verteilung der Grundgesamtheit ab, was eine Anwendung desselben Tests zum Testen von beliebigen Programmen (mit beliebiger Verteilung der Ausgaben) verhindert. Stattdessen wird in der vorliegenden Arbeit vorgeschlagen, dass im Zweistichprobenfall, also dem Test gegen eine Referenzimplementierung, die Ausgaben zu transformieren. Seien $(P_1, \ldots, P_n)$ und $(S_1, \ldots, S_n)$ Folgen von Ausgaben des SUT und der Referenzimplementierung. Zudem seien

$$\begin{aligned}
(P'_1, \ldots, P'_n) &= (|P_1 - \bar{P}_n|, \ldots, |P_n - \bar{P}_n|) \\
(S'_1, \ldots, S'_n) &= (|S_1 - \bar{S}_n|, \ldots, |S_n - \bar{S}_n|) \text{ mit} \\
\bar{P}_n = \frac{1}{n}\sum_{i=1}^{n} P_i \quad &\text{bzw.} \quad \bar{S}_n = \frac{1}{n}\sum_{i=1}^{n} S_i
\end{aligned}$$

die absolut zentrierten Ausgaben der beiden Programme. Deren empirischer Erwartungswert ist ein Indikator für die Variabilität der Stichprobe. Dieser ist allerdings wieder nur eine Größe erster Ordnung, und darf deshalb nicht mit der Varianz identifiziert werden. Für diese Stichproben kann wieder ein Zweistichproben-Gauß-Test auf Gleichheit dieser Erwartungswerte angewendet werden.

Der vorgestellten (asymptotische) Gauß-Test wurde in der vorliegenden Arbeit hauptsächlich gewählt, weil der β-Fehler abschätzbar ist. Generell kann der vorgestellte Testansatz mit beliebigen Hypothesentests durchgeführt werden, solange diese die Voraus-

setzung für die Wahrscheinlichkeitsverstärkung erfüllen. Das bietet sich besonders an, wenn viele Informationen über die Verteilung der Programmausgaben verfügbar sind. Durch die Wahl von speziellen Tests kann in diesem Fall die Effektivität des Testansatzes sicherlich gegenüber der Verwendung von allgemein anwendbaren Hypothesentests verbessert werden.

Zusammenfassend besteht der vorgeschlagene Testansatz aus folgenden Schritten:

1. Wähle statistischen Test, Konfidenzniveau α, Stichprobenumfang n so, dass α und β kleiner als $\frac{1}{2}$
2. Bestimme die Anzahl der Wiederholungen R aus (3.3) für eine gewünschte Wahrscheinlichkeit einer Fehlentscheidung ψ
3. Wiederhole die folgenden Schritte R mal
   (a) Führe das SUT (und ggf. die Referenzimplementierung) n Mal aus
   (b) Führe den gewählten statistischen Test durch
   (c) Wenn die Nullhypothese des Tests verworfen wurde, erhöhe einen Zähler um eins
4. Wenn es mehr als $\frac{R}{2}$ Verwerfungen der Nullhypothese gab, ist das SUT als fehlerhaft anzusehen.

3.2.7 Bisherige Forschung (related work)

Randomisierte Software wird gerade im wissenschaftlichen Umfeld oft zur Untersuchung von Fragestellungen eingesetzt. Deshalb sind in der Literatur einige Ansätze zum Testen dieser Art von Software beschrieben. Teilweise verwenden diese statistische Hypothesentests, teilweise werden andere Methoden vorgeschlagen.

Wie schon erwähnt findet man z.B. in [69, 68, 100] den Ansatz, Testentscheidungen mit Hilfe von statistischen Hypothesentests zu treffen. In allen Arbeiten fehlt jedoch eine theoretische Begründung des Ansatzes und eine fundierte empirische Untersuchung des Ansatzes bzgl. der Effektivität der vorgeschlagenen Methoden.

In [100] wird das Vorgehen beim Testen eines Systems zur Simulation von städtischen Großräumen namens UrbanSim beschrieben. Die Simulationen sollen Planungen für Stadtentwicklung und Infrastrukturmaßnahmen unterstützen. In der Arbeit werden

einige statistische Hypothesentests vorgestellt, die in Softwaretests Verwendung finden sollen. Die vorgeschlagenen Tests erfordern jedoch eine normal- bzw. Poisson-verteilte Grundgesamtheit. Diese Einschränkung auf bestimmte Verteilungen verhindern jedoch eine Anwendung der vorgeschlagenen Vorgehensweise auf beliebige Programme, da bei diesen die Verteilung der Ausgaben entweder unbekannt ist oder nicht den geforderten entspricht. Sind diese Voraussetzungen erfüllt, wird der Erwartungswert der Programmausgaben mit einem erwarteten, theoretischen Erwartungswert verglichen. Für die vorgeschlagenen Tests wird eine obere Schranke für den α-Fehler angegeben, aber der eigentlich interessante β-Fehler wird nicht betrachtet, insbesondere wird keine theoretische Schranke angegeben. Stattdessen wird eine kurze empirische Studie präsentiert, bei der mehrere Fehler (von Hand) in das Programm eingepflanzt wurden. Anschließend wurden diese fehlerhaften Programmversionen mit dem beschriebenen Ansatz getestet und die fehlgeschlagenen Tests analysiert. Es bleibt jedoch unklar, ob der Grund für den Fehlschlag des Test tatsächlich ein Fehler im Programm oder ein Fehler bei der Anwendung des Tests (z.B. verletzte Voraussetzungen oder versehentliche Ablehnung (α-Fehler)) ist. In dieser Arbeit wird zwar auch die Möglichkeit, Tests zu wiederholen um Fehlentscheidungen zu reduzieren vorgeschlagen, allerdings fehlt die theoretische Unterlegung und ein objektives Entscheidungskriterium. Stattdessen wird vorgeschlagen, die SUT als fehlerhaft zu klassifizieren, wenn die SUT „signifikant häufiger als durch die Wahl von α zu erwarten“ abgelehnt würde, ohne jedoch näher zu beschreiben, was mit „signifikant“ hier gemeint ist.

In [69, 68] wurden ebenfalls statistische Hypothesentests verwendet um eine Testentscheidung zu treffen. In der Anwendung ging es um das Testen von Bildanalyse-Software. Zur Kontrolle des β-Fehlers wurde ein t-Test modifiziert, was jedoch dazu führt, dass der α-Fehler nicht mehr kontrollierbar ist.

Aufbauend auf den Ergebnissen in [69, 68] wird in [39] die praktische Anwendung statistischer Verfahren im Softwaretest demonstriert. Die vorgestellten Testverfahren wurden zur Qualitätssicherung der GeoStoch-Bibliothek eingesetzt. Dabei wurden hauptsächlich Implementierungen zur Erzeugung von zufälligen Mosaiken, sog. Tessellationen, getestet. Die empirischen Ergebnisse bescheinigen den propagierten Methoden eine gute Anwendbarkeit und hohe Effektivität. Die Beurteilung der Effektivität beruht jedoch nur auf der Anwendung der Methode während der Implementierung. Angaben über Art und Anzahl der gefundenen Fehler werden nicht gemacht. Die gewonnenen Erkenntnisse

flossen auch in die in [49] publizierten Ergebnisse der vorliegenden Arbeit ein.

Chan et. al. beschreiben in [17] das automatische Testen einer Rendering-Software. Die Ausgaben der SUT werden ebenfalls als zufällig betrachtet, zur Testentscheidung werden allerdings keine statistischen Hypothesentests sondern Methoden aus dem Data Mining verwendet. Die verwendeten Klassifikatoren werden mit Ausgaben einer Referenzimplementierung trainiert. Es handelt sich also ebenfalls um eine Art *gold standard*-Ansatz für das Orakel.

Das sog. *Monte-Carlo-Testing* [33, 82, 83] verwendet dieselben theoretischen Grundlagen wie der in der vorliegenden Arbeit vorgestellte Ansatz. Monte-Carlo-Testing kann als Testen mit Monte Carlo-Simulationen beschrieben werden. In den referenzierten Arbeiten werden jedoch nur theoretische Aspekte des Verfahrens untersucht, keines beschäftigt sich mit dem Testen konkreter Anwendungen bzw. mit konkreten Anwendungen des Verfahrens.

Ein komplett anderer Ansatz wird in [11] beschrieben. Dort wird vorgeschlagen, aus einer randomisierten Anwendung eine deterministische zu machen, indem der Seed des verwendeten Zufallszahlengenerators festgesetzt wird. Wird der Zufallszahlengenerator bei jeder Programmausführung mit der gleichen Zahl (dem Seed) initialisiert, liefert er immer dieselbe Folge von „Zufalls“-Zahlen. Damit verhält sich das SUT wie ein „normales“ deterministisches Programm. Allerdings kann der Seed oft nur geändert werden, wenn man auch Änderungen am zu testenden Programm vornimmt. Da Änderungen am Programm zu Testzwecken aber oft nicht erwünscht sind, kann dieser Ansatz nicht immer eingesetzt werden.

Ein weiterer Ansatz zum Testen randomisierter Programme basiert auf probabilistischen endlichen Automaten [53, 87]. Dazu wird die Spezifikation der SUT als nichtdeterministischer endlicher Automat modelliert. Das zufällige Verhalten des Systems wird oft durch die Möglichkeit unterschiedlicher Rückgabewerte bei gleichen Eingaben verursacht, beispielsweise bei Netzwerk-Protokollen. Beim Test wird dann die SUT mit dem Automaten verglichen, etwa ob die SUT für eine bestimmte Eingabesequenz den gleichen Endzustand wie der Automat erreicht. Es existieren zahlreiche Verfahren, um aus endlichen Automaten Eingaben für Tests zu gewinnen. Die Anwendbarkeit dieses Test-Verfahrens wird jedoch durch Probleme bei der Modellierung komplexer Systeme als endlichem Automat und mögliche Explosion des Zustandsraumes des Automaten in der Praxis oft eingeschränkt.

3.3 Statistisch-Metamorphische Tests

Die im vorherigen Kapitel vorgestellte Möglichkeit einer Testentscheidung durch statistische Hypothesentests erfordert zwingend (theoretische) Referenzwerte oder eine Referenzimplementierung zur Durchführung der statistischen Hypothesentests. Referenzwerte oder Referenzimplementierung stehen oft nicht zur Verfügung oder können nur mit hohem Aufwand gewonnen werden. Deshalb sind andere Techniken notwendig, um auch ohne diese Referenzen statistische Hypothesentests zur Testentscheidung einsetzen zu können. In der vorliegenden Arbeit wurde dazu ein neues Testverfahren, das sog. *statistisch-metamorphische Testen* entwickelt und in [47] publiziert. Die nun im Folgenden beschriebene Technik kombiniert dazu Metamorphisches Testen mit Testentscheidungen durch statistische Hypothesentests.

Da beim Testen mit statistischen Hypothesentests das SUT mehrmals mit den selben Eingaben ausgeführt wird, bietet es sich an, nicht mehr von Eingaben, sondern von Parametern zu sprechen. Dies entspricht dem mathematischen Parameterbegriff[2], der für Variablen verwendet wird, die Eigenschaften von Funktionen beschreiben. Die Eingaberelation $R_{\mathcal{I}}$ beschreibt den Zusammenhang zwischen zwei oder mehr Parametrisierungen des SUT. Die Ausgaberelation $R_{\mathcal{O}}$ bezieht sich nun nicht mehr auf die (deterministischen) Beziehungen zwischen den Ausgaben, sondern beschreibt, wie die Zufallsvariablen, welche die Ausgaben modellieren, (statistisch) zueinander in Beziehung stehen.

3.3.1 Ein einführendes Beispiel

Zur weiteren Motivation und Erläuterung des Verfahrens der Statistisch-Metamorphischen Tests wird nun ein Beispiel aus der räumlichen Statistik vorgestellt. Als SUT dient dabei ein Simulator für zufällige Mosaike, sog. Tessellationen [39, 66]. In Abb. 3.3 sind drei Beispiele für unterschiedliche Tessellationen abgebildet. Tessellationen können dazu verwendet werden, netzwerkähnliche Strukturen wie beispielsweise Straßennetze oder Telekommunikationsinfrastruktur zu simulieren. Basierend auf den Simulationen können dann die Strukturen diese Netze untersucht werden. In [42, 43, 44] werden die Ergebnisse eines Projekts des Instituts für Stochastik der Universität Ulm und France Télécom R&D, Paris beschrieben, in dem die Netzwerkinfrastruktur von France Télécom untersucht wurde. Da sich die Komponenten des Netzwerks typischerweise an Straßen befin-

[2] griech. $\pi\alpha\rho\alpha\mu\varepsilon\tau\rho o\varsigma$, Formvariable

den, wurden in diesem Projekt Tessellationen benutzt, um Straßensysteme zu simulieren. Dabei wurden Simulationen in unterschiedlichen Maßstäben verwendet. Es wurden sowohl innerstädtische als auch überregionale Straßensysteme simuliert. Basierend auf den simulierten Strukturen wurden dann Untersuchungen zu Kosten und Risiken der Netzwerke durchgeführt. In [44] wurden Untersuchungen zur mittleren kürzesten Weglänge zwischen zwei Knoten durchgeführt. Diese Weglänge bestimmt beispielsweise die Kosten für die Verbindung der Knoten durch Kabel. Die Implementierung der Simulation der Tessellationen ist Bestandteil der Geostoch-Bibliothek ([72, 109]).

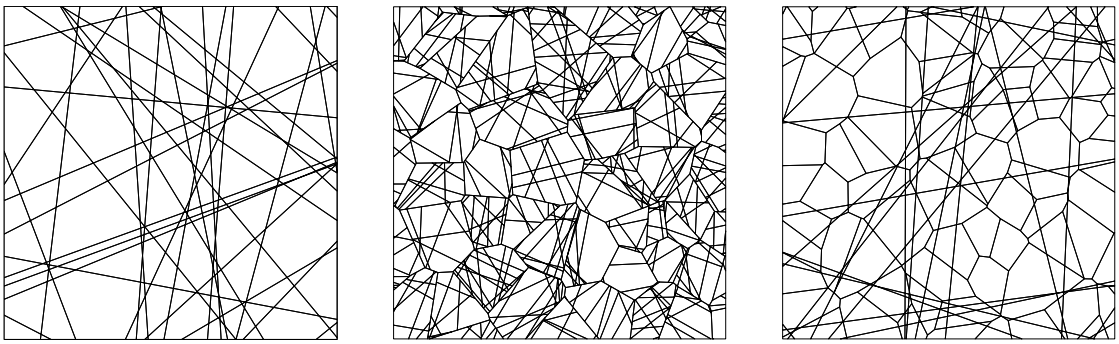


Abbildung 3.3: Beispiele für Tessellationen

Das SUT ist in Java implementiert, die Ausgabe des SUT ist ein Objekt, das eine Realisierung dieses Mosaiks beschreibt. Dieses Objekt kann nicht direkt in einem statistischen Test verwendet werden. Stattdessen können bestimmte Charakteristiken des Mosaiks (z.B. die Anzahl der Zellen im Beobachtungsfenster) für Tests herangezogen werden. Diese Charakteristiken sind gut untersucht (z.B. in [73]), daher ist der Einfluss der Parameter auf diese Charakteristiken bekannt. Es existieren sogar Formeln für die theoretischen Werte der Charakteristiken, siehe [110].

Die SUT kann durch zwei Parameter gesteuert werden. Der Parameter W beschreibt das Beobachtungsfenster in dem die Tessellation realisiert wird, der zweite Parameter γ steuert einen zugrunde liegenden stochastischen Prozess. Aus γ lässt sich die mittlere Anzahl der Zellen pro Flächeneinheit durch die Formel $\lambda_3 = \frac{1}{\pi}\gamma^2$ (nach [110]) berechnen. Der Erwartungswert der Anzahl an Zellen ζ im Beobachtungsfenster ergibt sich somit durch $\mathbb{E}\zeta = \frac{1}{\pi}\gamma^2 A(W)$, wobei $A(W)$ die Fläche des Beobachtungsfensters ist.

Aus dieser Formel kann nun auf einfache Art eine Metamorphische Relation konstruiert werden. Wählt man zwei Parametrisierungen (W_1, γ_1) und (W_2, γ_2) nun so, dass W_1 die vierfache Fläche von W_2 hat und $\gamma_2 = 2\gamma_1$, ist der Erwartungswert der Anzahl

der Zellen im Beobachtungsfenster für beide Parametrisierungen gleich. Da aber diese Anzahl eine zufällige Größe ist, werden statistische Hypothesentests benötigt, um Aussagen über deren Gleichheit zu treffen. Der Testablauf mit dieser Relation ist in Abb. 3.4 schematisch dargestellt.

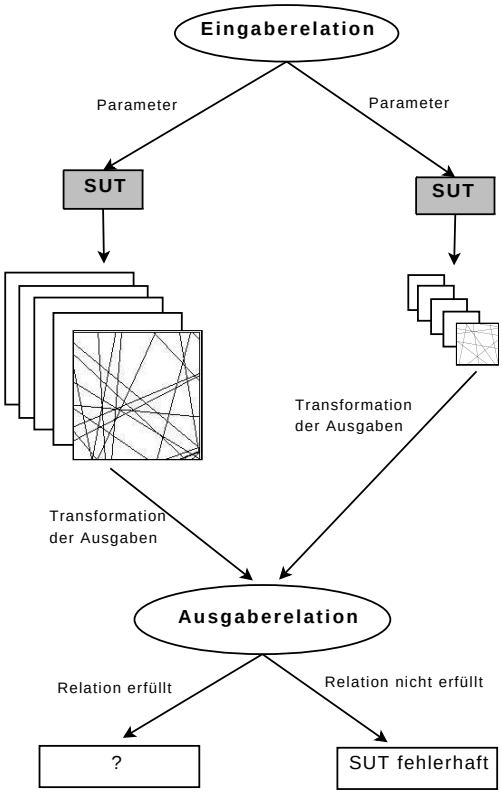


Abbildung 3.4: Schematische Darstellung des statistisch-metamorphischen Testens

Ausgehend von zwei Parametrisierungen des SUT wird für jede Parametrisierung eine hinreichend große Stichprobe an Ausgaben des SUT erzeugt. In Abbildung 3.4 ist dies durch die abgebildeten Tessellationen dargestellt. Anschließend wird die Anzahl der Zellen in jeder Tessellation berechnet. Diese Werte werden dann mit einem statistischen Hypothesentest verglichen. Wird die Nullhypothese verworfen, ist das SUT als fehlerhaft anzusehen. Im Falle das die Nullhypothese nicht verworfen wird, kann wie beim Metamorphischen Testen keine Aussage über die Korrektheit des SUT getroffen werden.

Der Ablauf entspricht im Wesentlichen dem zuvor vorgestellten Testen mit statistischen Hypothesentests. Bei dem in Kapitel 3.2 vorgestellten Verfahren wird nur ei-

ne Stichprobe durch das SUT erzeugt. Im Unterschied dazu werden beim statistisch-metamorphischen Testen mit demselben SUT mehrere Stichproben mit jeweils unterschiedlichen Parametrisierungen erzeugt. Die Stichproben werden dann mit statistischen Hypothesentests verglichen. Das Verfahren orientiert sich also im Prinzip am Test gegen eine Referenzimplementierung.

3.3.2 Eine formalere Beschreibung

Die formale Beschreibung des statistisch-metamorphischen Tests orientiert sich an den Begrifflichkeiten des Metamorphischen Testens, wie in Kapitel 3.1 beschrieben.

Wie bei der Verwendung von statistischen Hypothesentests wird angenommen, dass die Programmausgaben die Realisierungen einer Zufallsvariablen P sind, welche Werte in $\mathcal{O}$ annimmt. Um nun statistisch-metamorphische Tests durchführen zu können wird zusätzlich angenommen, dass die Verteilung von P durch die Wahl von Parametern beeinflusst werden kann. Die Verteilung von P gehört also zu einer parametrischen Familie von Verteilungsfunktionen. Die Menge der zulässigen Parameter ist eine Teilmenge der „Eingabemenge“. Es kann durchaus der Fall sein, dass nicht alle Eingaben einen Einfluss auf die Verteilung der Programmausgaben haben. Die „Eingaberelation“ $R_{\mathcal{I}}$ beschreibt, wie zwei oder mehr Parametrisierungen des SUT gewählt werden müssen.

Für jede dieser Parametrisierungen wird dann eine hinreichend große Stichprobe an Programmausgaben erzeugt. Dabei wird immer das SUT ausgeführt, und nicht etwa eine Referenzimplementierung.

Die „Ausgaberelation“ $R_{\mathcal{O}}$ beschreibt dann, welche statistischen Zusammenhänge zwischen den Stichproben gelten müssen. Diese Zusammenhänge werden dann durch einen statistischen Hypothesentest überprüft. Analog zum obigen Beispiel wird angenommen, dass die Programmausgaben nicht geeignet sind, um direkt durch statistische Hypothesentests überprüft zu werden. Deshalb besteht die „Ausgaberelation“ $R_{\mathcal{O}}$ aus zwei (messbaren) Funktionen f und g, so dass

$$\forall(i_1, \ldots, i_n) \in R_{\mathcal{I}} \Rightarrow g(P(i_1)) \stackrel{d}{=} f\left(P(i_2), \ldots, P(i_n)\right) \tag{3.6}$$

gilt. Die Implikation (3.6) wird als *stochastisch-metamorphische Relation* bezeichnet.

Im obigen Beispiel sind f und g identisch. Sie sind gleich der Funktion ζ zur Bestimmung der Anzahl der Zellen im Beobachtungsfenster. Die stochastisch-metamorphische

Relation lässt sich für das präsentierte Beispiel so formulieren:

$$\forall((W_1, \gamma_1), (W_2, \gamma_2)) \in R_{\mathcal{I}} \Rightarrow \zeta(P(W_1, \gamma_1)) \stackrel{d}{=} \zeta(P(W_2, \gamma_2))$$

mit

$$R_{\mathcal{I}} = \{(i_1 = (W_1, \gamma_1), i_2 = (W_2, \gamma_2)) : \ \gamma_2 = 2\gamma_1 \text{ und } A(W1) = 4A(W_2)\}$$

Im Gegensatz zum Ansatz in Kapitel 3.2 wird nun nicht mehr die theoretische Verteilungsfunktion oder eine Referenzimplementierung benötigt, um statistische Tests durchführen zu können. Stattdessen reicht die Kenntnis, wie sich die Verteilung der Programmausgaben in Abhängigkeit von den Parametern verhält.

3.3.3 Verwandte Arbeiten

In [39] findet sich ein vergleichbares Vorgehen. Das dort beschriebene Verfahren verwendet ebenfalls statistische Beziehungen zwischen Stichproben zum Testen von Implementierungen. Allerdings wird eine wiederholte Durchführung der statistischen Tests zur Reduktion von Fehlentscheidungen verzichtet. Auch eine systematische Untersuchung der Effektivität des propagierten Verfahrens wird in [39] nicht durchgeführt.

Kapitel 4

Techniken zur Untersuchung von Orakellösungen

4.1 Zufallstest

Um vollständig automatisiert testen zu können, werden nicht nur automatische Entscheidungsmechanismen, also Orakel, benötigt, sondern auch Methoden, um automatisiert Eingaben erzeugen zu können. Eine der Möglichkeiten ist die (pseudo-) zufällige Erzeugung der Eingaben. Im einfachsten Fall kann dies durch (Pseudo[1]-) Zufallszahlengeneratoren geschehen. Komplexere Eingaben können beispielsweise durch nichtdeterministische endliche Automaten erzeugt werden (siehe beispielsweise [102]). Diese Methode des Testens wird als *Zufallstest (random testing)* [1, 52, 63] bezeichnet.

4.1.1 Generelles Vorgehen

Allgemein gesprochen werden beim Zufallstest Realisierungen von Zufallsvariablen erzeugt und diese dann als Testeingaben verwendet. Die Zufallsvariable wird durch das sog. *operational profile* beschrieben. Das *operational profile* entspricht dem „typischen“ Verhalten eines Benutzers des SUT. Es wird üblicherweise aus der Spezifikation oder aus Informationsquellen wie Logfiles abgeleitet. Das *operational profile* besteht im Falle einer diskreten Zufallsvariable aus einer Häufigkeitstabelle. Im „stetigen“ Fall wird entweder die Dichte oder die Verteilungsfunktion der Zufallsvariable angegeben. Ist dies

[1] Die erzeugten Zahlen erscheinen zufällig, werden aber deterministisch, abhängig von einer Initialisierung des Generators, berechnet.

nicht möglich, so wird als *operational profile* die Gleichverteilung über einem Intervall oder einer Menge verwendet.

Erwartet das SUT nur numerische Eingaben, so reichen oft die von Betriebssystemen oder Standardbibliotheken zur Verfügung gestellten Zufallszahlengeneratoren zur Erzeugung der Eingaben aus. In der Untersuchungen der vorliegenden Arbeit war es jedoch notwendig, komplexere Methoden zur zufälligen Erzeugung der Eingaben zu verwenden. Diese werden in den folgenden Kapiteln zusammen mit den Untersuchungen beschrieben. Es ist anzunehmen, dass die Art der Eingabeerzeugung auch die Effektivität der eigentlich untersuchten Orakellösungen beeinflusst. Deshalb wurde der Einfluss der Eingabeerzeugung auf die Effektivität in die Untersuchungen miteinbezogen.

Bei der Testentscheidung stellt der Zufallstest hohe Anforderungen an das Orakel, da eine Testentscheidung für (nahezu) beliebige Eingaben getroffen werden muss. Aus diesem Grund muss der Zufallstest mit einem automatischen Orakel durchgeführt werden. Das einfachste anwendbare Orakel ist der sog. *smoke test*. Die SUT besteht diesen Test, wenn während der Berechnung kein Laufzeitfehler, beispielsweise eine Speicherschutzverletzung (*segmentation fault*[2]), auftritt. Der *smoke test* überprüft also die Robustheit einer Anwendung. Er ist deshalb immer anwendbar, weil für die Testentscheidung keine Referenzwerte benötigt werden. *Smoke tests* haben sich als einfaches aber wirkungsvolles Orakel herausgestellt, beispielsweise zum Testen von GUIs[3] (siehe [75]).

4.1.2 Vorteile des Zufallstests

Die zufällige Erzeugung von Eingaben hat mehrere Vorteile. Es ist möglich, einfach eine große Anzahl an Eingaben (automatisch) zu erzeugen, was in der vorliegenden Arbeit besonders bei der Untersuchung von statistischen Methoden im Softwaretest benötigt wird. Für die Erzeugung der Testeingaben wird üblicherweise zudem nur wenig Rechenzeit benötigt. Dadurch können selbst Tests, die viele Eingaben benötigen, in kurzer Zeit durchgeführt werden. In den Fallstudien der vorliegenden Arbeit konnten 5000 Testläufe in weniger als 10 Sekunden durchgeführt werden. Damit sollte die Testdauer bei der Verwendung der vorgeschlagenen Methoden auch bei der praktischen Anwendung akzeptabel sein.

Die schnelle Erzeugung von Eingaben kann zudem für Lasttests verwendet werden.

[2] siehe beispielsweise `http://en.wikipedia.org/wiki/Segmentation_fault`
[3] Graphical User Interface

Bei Lasttests wird neben dem funktional korrekten Verhalten auch untersucht, welche Zeit das SUT bei der Ausführung benötigt bzw. welche Anzahl von Aufgaben die SUT innerhalb einer vorgegebenen Zeitspanne abarbeiten kann. Bei Webservern kann beispielsweise untersucht werden, wie viele Seitenabrufe der Server innerhalb einer Minute abarbeiten kann (siehe beispielsweise [5]). Das *operational profile* könnte in diesem Fall das Verhalten eines einzelnen Benutzers beschreiben, also wieviele Seiten ein Benutzer pro Besuch abruft, welche Seite mit welcher Wahrscheinlichkeit aufgerufen wird, bzw. welche Zeit zwischen zwei Seitenaufrufen vergeht. In der Praxis können beispielsweise Logfiles des Webservers benutzt werden, um das *operational profile* zu erstellen.

Beim manuellen Testen von Software werden Testfälle oft so gewählt, dass diese bestimmte Kriterien erfüllen, beispielsweise eine geforderte Mindest-Überdeckung der Anweisungen eines Programms. Die Einhaltung dieser Kriterien sorgt jedoch nicht automatisch dafür, dass auch viele Fehler gefunden werden. Die Auswahl der Testfälle wird zudem durch die Kenntnisse und die Erfahrung des Testers beeinflusst. Dabei können unter Umständen auch weniger gute Testfälle gewählt werden. Bei der zufälligen Erzeugung von Testeingaben spielt die Kenntnis und Erfahrung des Testers dagegen keine Rolle. Unter der Voraussetzung, dass das *operational profile* richtig gewählt ist werden so auch ungewöhnliche Testeingaben gewählt und das SUT mit diesen getestet. Der Zufallstest eignet sich damit besonders dafür, die Robustheit des SUT zu überprüfen (siehe beispielsweise [40, 77]).

Die zufällige Erzeugung erlaubt Rückschlüsse auf die Zuverlässigkeit der untersuchten Software. Dieser Aspekt wird in Kapitel 4.1.5 diskutiert.

4.1.3 Nachteile des Zufallstests

Die Verwendung von Methoden des Zufallstests ist aber auch nicht unproblematisch. Dabei kann zwischen praktischen und theoretischen Problemen unterschieden werden.

Das größte Problem in der praktischen Anwendung ist, dass Orakel benötigt werden, die in der Lage sind, für alle erzeugten Eingaben auch Testentscheidungen zu treffen. Diese Orakel sind jedoch Gegenstand dieser Arbeit, stehen also in allen Untersuchungen dieser Arbeit zur Verfügung.

Besonders bei komplexen Eingaben ist es aber oft nicht klar, wie diese zu erzeugen sind. Objekte wie ein „zufälliges Programm“ oder ein „zufälliges Bild“ sind nicht klar definiert und können deshalb auch nicht ohne zusätzliche Annahmen erzeugt werden.

Diese Annahmen können aber einen Einfluss auf die Ergebnisse des Zufallstests haben. Bei interaktiven Systemen ist das Problem, dass es „den“ Benutzer, dessen Verhalten ja durch das *operational profile* beschrieben wird, so gar nicht gibt. Oft gibt es unterschiedliche Gruppen von Benutzern, beispielsweise „Normalbenutzer“ und „Experten“, deren Benutzerverhalten sich sehr stark unterscheiden.

Die Probleme werden oft noch verschärft, wenn das *operational profile* nicht Bestandteil der Spezifikation ist. Es muss dann aus dem Domänenwissen des Testers gewonnen werden.

4.1.4 Zufallstest in der Literatur

Die Verwendung von zufällig erzeugten Eingaben wurde in der Literatur schon häufiger beschrieben, beispielsweise wurden SQL-basierte Datenbanksysteme [102], die Robustheit von WindowsNT-Anwendungen [40] und OSX-Anwendungen ([77]) oder Java Just-In-Time Compiler (JIT) [112] getestet.

4.1.5 Zufallstest und Zuverlässigkeit von Software

Der Zufallstest kann nicht nur verwendet werden, um Fehler im SUT festzustellen, es können zusätzlich auch Informationen über die Zuverlässigkeit des SUT abgeleitet werden (vgl. [52]). In den Arbeiten über die Zuverlässigkeit von Software (beispielsweise in [85]) wird angenommen, dass ein Programm P eine konstante Fehlerrate θ $(0 \leq \theta \leq 1)$aufweist. D. h., dass bei einem einzelnen Test des SUT mit Wahrscheinlichkeit θ ein Fehler auftritt. Entsprechend wird mit Wahrscheinlichkeit $1 - \theta$ kein Fehler festgestellt.

Werden nun N Testeingaben nach dem *operational profile* des SUT (iid) erzeugt, ergibt sich, dass die Wahrscheinlichkeit, keinen Fehler festgestellt zu haben $(1 - \theta)^N$ beträgt bzw. dass mit Wahrscheinlichkeit $e = 1-(1-\theta)^N$ mindestens ein Fehler festgestellt wird. Die Wahrscheinlichkeit $1 - e$ ist dabei so zu interpretieren, dass nicht öfter als ein Mal in $\frac{1}{\theta}$ Programmausführungen ein Fehler auftritt. Die Wahrscheilichkeit $1 - e$ kann durch die Durchführung des Zufallstests empirisch bestimmt werden. Daraus kann dann durch die Gleichung

$$\theta = 1 - (1 - e)^{\frac{1}{N}}$$

die tatsächliche Fehlerrate θ berechnet werden.

Allerdings ist anzumerken, dass wenn statt eines korrekten *operational profiles* die Gleichverteilung über dem Eingabebereich zur Erzeugung der Eingaben verwendet wird, die Zuverlässigkeitsaussage beliebig ungenau werden kann. In [52] wird festgestellt, dass die Schätzung der Zuverlässigkeit dabei immer zu optimistisch ausfällt. Das kann dadurch erklärt werden, dass durch die Gleichverteilung alle Eingaben mit derselben Wahrscheinlichkeit gewählt werden. Eingaben, die im tatsächlichen Betrieb (also nach dem *operational profile*) oft verwendet werden, werden im Test dadurch seltener verwendet. Verursachen diese Fehler, werden die Fehler im Test auch seltener festgestellt.

4.2 Bewertung von Testmethoden und Orakellösungen

In diesem Kapitel wird eine Methode zur Messung und Bewertung der Effektivität von Testmethoden vorgestellt. Die Methode beruht auf bekannten Ergebnissen zur Bewertung von Testmengen, der sog. *Mutationsanalyse*. Die Ergebnisse zur Mutationsanalyse werden für das in der vorliegenden Arbeit vorgeschlagene Verfahren leicht abgewandelt, um auch Testverfahren und nicht nur konkrete Testmengen zu bewerten.

4.2.1 Mutationsanalyse

Die Mutationsanalyse ist ein Verfahren zur Überprüfung der Güte von Testmengen. Die Methode wurde ab 1977 entwickelt, siehe [32, 51]. Der aktuelle Stand der Forschung in diesem Bereich ist in [90] zusammengefasst. Verwandte Verfahren werden in Kapitel 4.2.4 diskutiert.

Um die Effektivität einer Testmenge zu messen, werden systematisch einzelne Veränderungen am Quellcode des SUT durchgeführt. Diese orientieren sich an typischen Programmierfehlern. Die Mutationsanalyse beruht also auf Techniken zur Fehlereinpflanzung. Die Art und Weise der Einpflanzung ist genau definiert und wird durch einen sog. *Mutationsoperator* beschrieben. Ein Beispiel für einen Mutationsoperator ist der Austausch von arithmetischen Operatoren (AOR für arithmetic operator replacement). Als Beispiel sei folgendes Programm gegeben:

```
public int foo(int a, int b){
    int c = 5;
    a = b + c;
    return a;
}
```

Durch den Mutationsoperator wird die Anweisung `a=b+c;` durch die Abweisung `a=b-c;` ersetzt, indem der (binäre) Operator `+` durch den Operator `+` ausgetauscht wird:

```
public int foo(int a, int b){
    int c = 5;
    a = b - c; //MUTATION: b + c -> b - c (AORB)
    return a;
}
```

Dabei einsteht ein weiteres Programm, welches bis auf diese veränderte Anweisung mit dem SUT identisch ist. Dieses Programm wird als *Mutant* bezeichnet. Im Quelltext des Mutanten wird die Stelle der Mutation oft durch einen Kommentar gekennzeichnet, der die Art der Mutation beschreibt. Dadurch wird die Analyse der Ergebnisse der Mutationsanalyse vereinfacht.

Andere Mutationsoperatoren sind beispielsweise der Austausch von Vergleichsoperatoren, das Vertauschen von Variablen oder Ersetzung von Variablen durch Konstanten. In Tabelle 4.1 ist eine Übersicht über die Mutationsoperatoren gegeben, die in den Fallstudien der vorliegenden Arbeit verwendet wurden.

Die ersten Mutationsoperatoren wurden ursprünglich für Fortran entwickelt [59], später wurden auch Mutationsoperatoren für objektorientierte Programme vorgeschlagen [64]. Letztere werden in dieser Arbeit allerdings nicht berücksichtigt, da Mechanismen der Objektorientierung in den untersuchten Programmen keine Rolle spielen.

Bei der Erzeugung der Mutanten wird jeweils nur eine Mutation an einer Stelle im Programm durchgeführt. Da aber jeder Mutationsoperator an jeder möglichen Stelle angewendet wird, führt dieses Vorgehen schon bei moderat großen Programmen von wenigen hundert Codezeilen zu einer großen Anzahl an Mutanten.

Bei der Durchführung der eigentlichen Mutationsanalyse wird zuerst die Testmenge auf das Originalprogramm angewendet und eventuell entdeckte Fehler werden behoben. Danach werden die Mutanten mit der Testmenge untersucht und jede Ausgabe eines Mutanten mit der des Original-Programms verglichen. Weichen diese voneinander ab,

Operator		Beschreibung
AORB	Arithmetic Operator Replacement (Binary)	Austausch eines binären arithmetischen Operators (+,−,⋆,/,%)
AORS	Arithmetic Operator Replacement (short-cut)	Austausch eines Inkrement-/Dekrement-Operators (++,−−)
AOIU	Arithmetic Operator Insertion (Unary)	Einfügen eines unären arithmetischen Operators (+,−)
AOIS	Arithmetic Operator Insertion (short-cut)	Einfügen eines Inkrement-/Dekrement-Operators
AODU	Arithmetic Operator Deletion (Unary)	Löschen eines unären arithmetischen Operators
AODS	Arithmetic Operator Deletion (short-cut)	Einfügen eines Inkrement-/Dekrement-Operators
ROR	Relational Operator Replacement	Austausch eines Vergleichsoperators (>, <, >=, <=, ! =, ==)
COR	Conditional Operator Replacement	Austausch eines binären logischen Operators (||,&&)
COI	Conditional Operator Insertion	Einfügen eines unären logischen Operators (!)
SOR	Shift Operator Replacement	Austausch eines Shift-Operators (<<, >>)
LOR	Logical Operator Replacement	Austausch eines binären Bitoperators (&, |, ˆ)
LOI	Logical Operator Insertion	Einfügen eines unären Bitoperators ( )
LOD	Logical Operator Deletion	Löschen eines unären Bitoperators
ASRS	Assignment Operator Replacement (short-cut)	Austausch einer vereinfachten Zuweisung (+ =,− =,∗ =,/ =, ...)

Tabelle 4.1: Mutationsoperatoren in MuJava (nach [65]).

wurde der Mutant von der Testmenge als fehlerhaft identifiziert. Man sagt, der Mutant wurde „getötet“. Allerdings führt nicht jede Mutation tatsächlich zu einem Fehlverhalten des Mutanten. Der folgende Ausschnitt aus einem Programm ist ein Beispiel für einen äquivalenten Mutanten.

```
public int foo(int a, int b){
    int c = 5;
    a = b + c;
    return a++; //MUTATION: a -> a++ (AOIS)
}
```

Der eingefügte Operator `++` wird erst nach der Rückgabe des Wertes von `a` ausgeführt. Das hat dann aber keine Auswirkung mehr auf das Programm, da die Variable `a` später nicht mehr verwendet wird.

Die Mutanten können daher in zwei Gruppen aufgeteilt werden, Mutanten äquivalent und nicht-äquivalent zum Originalprogramm.

Definition 3 *Programmäquivalenz*

Zwei Programme S, $P : \mathcal{I} \rightarrow \mathcal{O}$ *sind äquivalent* $\Leftrightarrow P(x) = S(x) \; \forall \, x \in \mathcal{I}$

Zur Bewertung der Testmenge ist es also zuerst erforderlich, alle Mutanten zu identifizieren, die äquivalent zum Originalprogramm sind. Da Programmäquivalenz nach dem Satz von Rice (vgl. Kapitel 1.2) i.A. nicht entscheidbar ist, kann die Äquivalenz von Mutant und Originalprogramm oft nur durch manuelle Inspektion bestimmt werden. Bei einer Fallstudie ist es aber gewünscht, eine möglichst große Anzahl an Mutanten zu erzeugen, wodurch die Bestimmung der äquivalenten Mutanten ein sehr zeitaufwändiger Teil der Fallstudie ist.

Die Bewertung einer Testmenge erfolgt durch den sog. *mutation score*. Dieser ist das Verhältnis von nicht getöteten Mutanten zu den nicht-äquivalenten, also tötbaren Mutanten:

$$m_s = \frac{\#\{\text{getötete Mutanten}\}}{\#\{\text{nicht äquivalente Mutanten}\}}$$

Der *mutation score* kann dann dazu verwendet werden, die Qualität der Testmenge zu verbessern. Wurde ein Mutant weder getötet noch als äquivalent identifiziert, werden zusätzliche Testfälle entwickelt, um auch diesen Mutanten zu töten. Die Testmenge wird

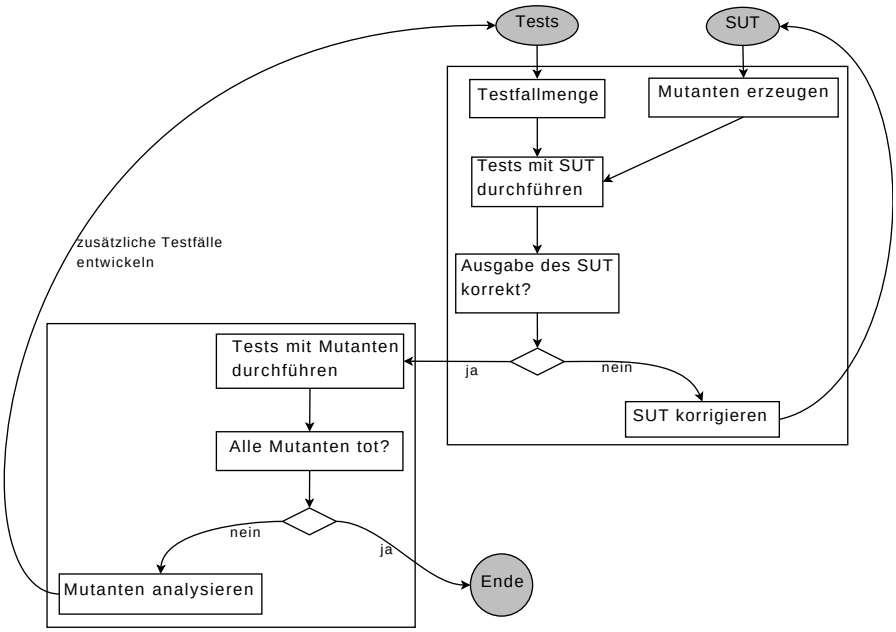


Abbildung 4.1: Schematische Darstellung der Mutationsanalyse

also so lange erweitert, bis alle nicht-äquivalenten Mutanten getötet werden können. Das Verfahren der Mutationsanalyse ist schematisch in Abb. 4.1 dargestellt.

Die Mutationsanalyse beruht auf zwei Annahmen, der sog. *competent programmer hypothesis* und dem sog. *coupling effect* (siehe beispielsweise [90]). Die *competent programmer hypothesis* unterstellt, dass Programmierer nur kleine, einfache Fehler machen, die zudem nicht besonders oft vorkommen. Diese Fehler entstehen der Annahme nach durch Tippfehler, beispielsweise weil Variablennamen vertauscht werden. Nach der *competent programmer hypothesis* ist es damit ausreichend, kleine Veränderungen am Quelltext vorzunehmen um diese Fehler nachzubilden.

Der *coupling effect* unterstellt, dass komplexe Fehler aus mehreren einfachen Fehlern im Quelltext zusammengesetzt sind. Daraus wird abgeleitet, dass eine Testmenge, die kleine Fehler entdeckt auch komplexe Fehler entdeckt. Der *coupling effect* ist damit der Grund, nicht mehrfach Mutationsoperatoren auf ein und dasselbe Programm anzuwenden, da diese komplexen Fehler auch von Testmengen entdeckt werden, die einfache Fehler finden. Ein weiterer Grund ist, dass die mehrfache Anwendung von Mutationsoperatoren die Anzahl an erzeugten Mutanten deutlich erhöht. Dadurch erhöht sich auch der Aufwand für die Mutationsanalyse. Laut Annahme verbessert sich aber trotz des

höheren Aufwands nicht das Ergebnis der Mutationsanalyse.

4.2.2 Bewertung von Orakellösungen mittels Mutationen

Die Bewertung der Güte von Testeingaben ist ein gut untersuchtes Gebiet. Neben dem im vorherigen Kapitel vorgestellten *mutation score* sind beispielsweise Überdeckungsmaße ein oft vorgeschlagenes Gütekriterium für Testeingaben.

Die Bewertung von Orakellösungen dagegen ist weit weniger gut untersucht. In der vorliegenden Arbeit wird deshalb ein auf Ergebnissen der Mutationsanalyse beruhender Ansatz vorgeschlagen, um die Güte von Orakellösungen zu bewerten. Die Güte eines Orakels soll ausdrücken, in wie weit das Orakel in der Lage ist, fehlerhafte Programme als solche zu erkennen. Um diese fehlerhaften Programme zu erzeugen, werden, wie bei der klassischen Mutationsanalyse auch, Mutationsoperatoren auf ein Programm angewendet. Die Mutationsanalyse hat hier den entscheidenden Vorteil, dass die Methode zum Erstellen von modifizierten Versionen eines Programms genau definiert ist. Die Erzeugung der modifizierten Versionen wird dadurch reproduzierbar, was bei manueller Erzeugung nicht so einfach möglich ist. Die Idee, Techniken der Mutationsanalyse zum Bewerten von Testmethoden zu verwenden wurde u.a. in [41, 55, 107] verwendet.

Im folgenden wird beschrieben, wie zur Bewertung von Orakellösungen vorgegangen werden kann. Dieser Ansatz wurde in den Fallstudien der vorliegenden Arbeit entwickelt und basierend auf den Erfahrungen aus der praktischen Anwendung verfeinert.

1. **Wahl des Untersuchungsgegenstands**:
   Die Untersuchung einer Testmethode beginnt mit der Wahl eines geeigneten Untersuchungsgegenstandes. Dieser Untersuchungsgegenstand ist ein Programm, welches sich zuerst natürlich dazu eignen muss, durch das zu untersuchende Verfahren getestet zu werden. Zudem sollte dieses Programm eine hinreichend komplexe Programmstruktur besitzen, damit auch eine große Anzahl an Mutanten erzeugt werden kann. In den Fallstudien der vorliegenden Arbeit wurden Implementierungen mit mehr als 200 LOC[4] verwendet. Der Untersuchungsgegenstand wird im Folgenden als *Originalprogramm* bezeichnet.

2. **Überprüfung des Testverfahrens und des Originalprogramms**:
   Es muss sichergestellt sein, dass das untersuchte Testverfahren korrekt implemen-

[4]Lines Of Code: Anzahl der Zeilen des Quellcodes

tiert ist. Dazu wird das Testverfahren auf das Originalprogramm angewendet. Findet man einen Fehler, so muss untersucht werden, ob das Originalprogramm oder die Testmethode fehlerhaft ist und die Fehler müssen behoben werden. In den durchgeführten Untersuchungen wurden ausschließlich Fehler in der Implementierung der Testverfahren gefunden, die Originalimplementierungen waren fehlerfrei. Sollte jedoch das Originalprogramm Fehler enthalten, müssen auch diese behoben werden.

3. **Erzeugung von Mutanten**:
   Um Orakellösungen bzw. Testmethoden zu bewerten wird ähnlich zur Mutationsanalyse vorgegangen. Aus dem gewählten Originalprogramm werden dazu Mutanten erzeugt. Für in Java geschriebene Programme bietet sich das Werkzeug MuJava [65] an. Der sonst zeitaufwändige Prozess der Mutantenerzeugung ist damit automatisiert. Der Aufwand für die Erzeugung der Mutanten kann so drastisch reduziert werden.

4. **Anwendung des Testverfahrens auf die Mutanten**:
   Die erzeugten Mutanten werden durch die zu untersuchende Testmethode getestet. Es werden also dem Testverfahren entsprechend Testeingaben erzeugt und anschließend die Ausgabe des Mutanten durch das Orakel untersucht. Die dabei getöteten Mutanten werden gezählt.

5. **Vergleich von Mutant und Originalprogramm**:
   Jede im vorherigen Schritt erzeugte Eingabe wird zusätzlich durch das Originalprogramm verarbeitet und anschließend die Ausgabe des Mutanten und des Originalprogramms bei derselben Eingabe miteinander verglichen. Die Anzahl der durch Vergleich mit der Originalimplementierung getöteten Mutanten wird ebenfalls gezählt.

   Durch dieses Vorgehen erhält man mit dem Vergleich gegen die Originalimplementierung bei gleichen Eingaben eine echte Obergrenze für die Anzahl der getöteten Mutanten, da die Verwendung einer fehlerfreien strukturgleichen Implementierung als Orakel das bestmögliche Orakel darstellt (siehe [12]).

6. **Berechnung des Effektivitätsmaßes**:
   Setzt man nun die Anzahl der durch ein anderes Testverfahren getöteten Mutanten

zu dieser in Relation, so erhält man ein Maß für die Effektivität des Testverfahrens:

$$m_{\text{eff}}(T) = \frac{\#\{\text{durch Testverfahren } T \text{ getötete Mutanten}\}}{\#\{\text{durch Vergleich mit Original getötete Mutanten}\}}$$

Will man zwei Testverfahren T_1, T_2 miteinander vergleichen, so gilt:
T_1 ist effektiver als $T_2 \Leftrightarrow m_{\text{eff}}(T_1) > m_{\text{eff}}(T_2)$.

4.2.3 Bewertung von Teststrategien

In der Literatur wurde bisher die Güte von Testeingaben und Orakeln nur getrennt betrachtet. Da aber die Effektivität von Orakeln wesentlich von der Güte der Testeingaben bestimmt wird, sollten beide gemeinsam betrachtet werden. Man kann daher bei der gemeinsamen Bewertung von Testeingaben und Orakeln auch von der Bewertung von Teststrategien sprechen.

Die Grundidee ist, die Bewertung der Teststrategie aus den Bewertungen der Testeingaben und des Orakels zusammenzusetzen. Formal kann das durch

$$E = E_{\text{in}} \cdot E_{\text{out}}$$

ausgedrückt werden. Die Gütemaße E_{in} und E_{out} bewerten die Güte bzw. die Effektivität der Testeingaben und des Orakels. Es wird angenommen, dass für die Gütemaße $0 \leq E_{\text{in}}, E_{\text{out}} \leq 1$ gilt. Damit gilt offensichtlich auch $0 \leq E \leq 1$. Diese Annahmen sind durchaus üblich, da beispielsweise Überdeckungsmaße[5] ebenfalls in Prozent angegeben werden. Durch den einfachen Zusammenhang zwischen den beiden Teilen der Bewertung sollte es einfach möglich sein, Ergebnisse aus Fallstudien zu interpretieren.

In der vorliegenden Arbeit werden die Gütemaße $E_{\text{in}} = m_s$ und $E_{\text{out}} = m_{\text{eff}}$ verwendet. Prinzipiell ist es natürlich auch möglich, andere Gütemaße zu verwenden. Überdeckungsmaße könnten mit Tools automatisch bestimmt werden, was beim *mutation score* nicht möglich ist. Das würde den Aufwand für die Bestimmung des Effektivitätsmaßes einer Teststrategie verringern.

Der vorgeschlagene Ansatz wurde in [46] publiziert.

[5]Mit Überdeckungsmaßen wird festgestellt, welcher Anteil der Strukturen eines Programms (durch Tests) ausgeführt wird. Das einfachste Überdeckungsmaß ist die sog. Anweisungsüberdeckung. Diese bestimmt den Anteil der durch Tests ausgeführten Anweisungen eines Programms (siehe [78]). Weitere Überdeckungsmaße betrachten Verzweigungen oder Ausführungspfade eines Programms.

4.2.4 Verwandte Verfahren

Die Einpflanzung von Fehlern (engl. *error seeding*) wird im Zusammenhang mit Zuverlässigkeitsuntersuchungen schon lange verwendet. Fehlereinpflanzung wurde u.a. in [79] vorgeschlagen und beispielsweise in [60, 74] systematisch untersucht.

In [35] wird beschrieben, wie die Fehlereinpflanzung zur Restfehlerprognose verwendet werden kann. Es wird angenommen, dass das zu untersuchende Programm eine unbekannte Anzahl E_n an Fehlern enthält. In das Programm werden zusätzlich eine bestimmte Anzahl E_s an Fehlern eingepflanzt. Anschließend wird das System getestet. Bei der Auswertung der Tests muss dann festgestellt werden, ob der gefundene Fehler eingepflanzt wurde oder nicht. Die Anzahl der gefundenen eingepflanzten Fehler wird mit F_s, die der gefundenen nicht-eingepflanzten Fehler wird mit F_n bezeichnet. Unter der Annahme, dass $\frac{F_s}{F_n}$ sich proportional zu $\frac{E_s}{E_n}$ verhält, kann die Anzahl der noch im Programm vorhandenen Fehler dann durch

$$\text{Anzahl der Restfehler} = F_n \cdot \frac{F_s}{E_s}$$

abgeschätzt werden.

In der Literatur werden unterschiedliche Verfahren zur Fehlereinpflanzung verwendet, um die Effektivität von Testmethoden zu messen. Diese Verfahren können durch die Art und Weise der Fehlereinpflanzung unterschieden werden. Die häufigste Methode der Fehlereinpflanzung ist das manuelle Ändern des Quellcodes durch einen erfahrenen Programmierer. Die Art der Fehler und der Ort der Einpflanzung wird durch die Erfahrung und die Intuition des Durchführenden bestimmt. Teilweise werden aber auch Fehler, die während der Entwicklung eines Programms entdeckt und beseitigt wurden, bei Untersuchungen wieder eingepflanzt. In [7] werden bei Untersuchungen zur Effektivität unterschiedlicher Testmethoden Programme verwendet, für die Informationen zu behobenen Fehlern zur Verfügung stehen. Diese wieder eingepflanzten Fehler wurden dann zur Messung der Effektivität verwendet.

In [60] werden Fehler automatisch eingepflanzt. Die Art der Fehler wird formal beschrieben, ähnlich wie Mutationsoperatoren. Allerdings werden in dieser Arbeit nur 6 unterschiedliche Fehlertypen verwendet, die auch nicht empirisch begründet werden. Der Ort der Einpflanzung wird manuell bestimmt. In [74] werden nicht näher spezifizierte Fehler an einer zufällig gewählten Stelle im Quelltext eingepflanzt. Beide Ansätze pflan-

zen auch mehrere Fehler in ein Programm ein.

4.2.5 Probleme der Methode

Die Probleme des Bewertungsverfahrens für Teststrategien entsprechen den Problemen der Mutationsanalyse. Beide Grundannahmen der Mutationsanalyse sind schon in Zweifel gezogen worden. Gegen die *competent programmer hypothesis* sprechen einige empirische Untersuchungen zu Fehlertypen in Software. Dort wurde festgestellt, dass ein großer Teil der Fehler (22%-54%) durch Auslassungen, also vergessenem Code, verursacht werden (siehe z.B. [6, 91]).

Der *coupling effect* ist sicherlich auch nicht uneingeschränkt gültig. Der *coupling effect* impliziert ja ein Stück weit, dass komplexe Fehler einfacher zu finden sind als einfache Fehler. Das muss aber nicht unbedingt der Fall sein, beispielsweise wenn ein Programm zwei Fehler enthält, die sich für bestimmte Eingaben gegenseitig wieder aufheben.

In einer anderen Untersuchung von Techniken zur Fehlereinpflanzung (siehe [60]) wurde dieser Aspekt besonders berücksichtigt. Dabei werden im Gegensatz zum in Kapitel 4.2.1 beschriebenen Vorgehen auch mehrere Fehler gleichzeitig in ein Programm eingepflanzt. Der Fall, dass sich Fehler gegenseitig aufheben tritt dort aber nur sehr selten auf. Man muss also prinzipiell davon ausgehen, dass dieser Effekt in der Realität vorkommt, allerdings ist ungewiss, wie oft.

In [89] wurde der *coupling effect* ebenfalls empirisch untersucht. Dort konnte der *coupling effect* belegt werden. Die beiden Untersuchungen in [60, 89] liefern nicht unbedingt widersprüchliche Ergebnisse, aber der *coupling effect* wird nicht uneingeschränkt gültig sein. Andererseits ist es aber auch unklar, wie abwegig dieser Effekt ist, oder ob er nicht eine zulässige Vereinfachung im Rahmen einer Modellierung von Fehlern ist.

Die zweite Frage ist, ob die künstlich durch Mutation erzeugten Fehler mit realen Fehlern vergleichbar sind. Eine ausführliche Untersuchung dieser Frage findet sich in [3]. Dort wird festgestellt, dass die Ergebnisse einer auf Mutation basierten Untersuchung durchaus auf reale Fehler übertragen werden kann, allerdings nur nachdem äquivalente Mutanten entfernt wurden.

Die Identifizierung von äquivalenten Mutanten ist überaus problematisch. Da Programmäquivalenz nach dem Satz von Rice [98] unentscheidbar ist, muss entweder von Hand durch Inspektion der Quellcodes über die Äquivalenz von Mutant und Original entschieden werden. Alternativ kann man Mutant und Original als äquivalent bezeichnen,

wenn beide für eine große Anzahl an Eingaben dasselbe Ergebnis liefern. Diese „simulationsbasierte“ Äquivalenzprüfung ist aber nicht unproblematisch. Wenn der Ausführungspfad, der den Fehler enthält, nur mit sehr geringer Wahrscheinlichkeit durch die zufällig erzeugten Eingaben ausgeführt wird, können Fehler übersehen werden. Zur Erläuterung sei folgendes Beispiel gegeben:

```
int foo(double x){
    if(x<0.5) return 1; //MUTATION: x<=0.5 -> x<0.5 (ROR)
    else return 0;
}
```

Der Rückgabewert des Mutanten unterscheidet sich nur für `x==0.5` von der des Originalprogramms. Die Wahl genau dieser Eingabe ist jedoch bei einer zufälligen Erzeugung sehr unwahrscheinlich.

Da in dieser Arbeit mit zufällig erzeugten Eingaben gearbeitet worden ist, ist damit zu rechnen, dass einige wenige der als äquivalent betrachteten Mutanten doch fehlerbehaftet sind.

In der Literatur zur Mutationsanalyse wird davon ausgegangen, dass ca. 10% der erzeugten Mutanten äquivalent zum Original sind. Werden nun mehr als diese 10% beim Vergleich mit der Originalimplementierung nicht als fehlerhaft identifiziert, kann das zwei Ursachen haben. Entweder wurden mehr als 10% äquivalente Mutanten erzeugt oder aufgrund der Auswahl der Eingaben wurden fehlerhafte Mutanten übersehen. Beides kann leider nur manuell überprüft werden. Eine große Anzahl äquivalenter Mutanten ist aus praktischer Sicht nicht wünschenswert, da diese die Dauer der Untersuchungen unnötig verlängern. Bestimmte Struktureigenschaften des Programms (z.B. häufige Methodenaufrufe mit lokalen Variablen am Ende deren Lebenszeit als Parameter) und häufige Anwendung bestimmter Mutationsoperatoren (z.B. Einpflanzung von Post-Increment/-Decrement) führen aber zu einer großen Anzahl äquivalenter Mutanten. Bei der Wahl des Untersuchungsgegenstands sollte also auf diese Struktureigenschaften geachtet werden, um weniger Zeit bei den folgenden Untersuchungen zu benötigen.

Kapitel 5

Empirische Untersuchungen von Metamorphischem Testen

Im Rahmen der vorliegenden Arbeit wurden mehrere Fallstudien durchgeführt, um die in Kapitel 3 beschriebenen Testtechniken zu untersuchen und mit anderen Testtechniken zu vergleichen. In diesem Kapitel werden zunächst zwei Fallstudien vorgestellt, die sich mit der Untersuchung des Metamorphischen Testens befassen. Die erste Fallstudie untersucht Implementierungen zur Berechnung von Determinanten, die zweite Fallstudie beschäftigt sich mit dem automatisierten Testen von Bildverarbeitungsalgorithmen. Die Ergebnisse der Studien wurden in [70] und [48, 71] veröffentlicht.

5.1 Untersuchung Metamorphischer Relationen

In den bis dato veröffentlichten Arbeiten zum Thema Metamorphisches Testen fehlten bislang größere systematisch durchgeführte Studien über die Effektivität der Testmethode. In der vorliegenden Arbeit sollen deswegen möglichst viele unterschiedliche Metamorphische Relationen untersucht werden. In einer Fallstudie soll zunächst die Effektivität dieser Relationen bestimmt werden. Neben den Relationen spielt das SUT eine große Rolle bei der Durchführung der Fallstudie. Das SUT sollte ein echtes Testproblem darstellen, es soll also nicht möglich sein, auf triviale Art und Weise Referenzwerte für die Programmausgaben zu erhalten. Das Testproblem soll durch einen vollständig automatisierten Testansatz gelöst werden. Dazu muss also nicht nur die Testentscheidung

(durch Metamorphische Relationen) automatisiert getroffen werden, auch die Testeingaben müssen automatisch erzeugt werden. Der Quellcode des SUT sollte hinreichend komplex sein, um eine große Anzahl an Mutanten erzeugen zu können.

Als Untersuchungsobjekt wurden unterschiedliche Implementierungen zur Berechnung von Matrix-Determinanten gewählt. Um einen eventuellen Einfluss der Art und Weise der Implementierung bestimmen zu können, wurden insgesamt sechs unterschiedliche Implementierungen gewählt:

I1 Math-Bibliothek aus dem Apache Commons Project (Version 1.0) [4]

I2 JScience (Version 2.0.1) [31]

I3 JAMA (Version 1.0.2) [57]

I4 eine Implementierung von Michael Thomas Flanagan (01.05.2005) [38]

I5 eine Implementierung von Jon Squire (20.10.2005) [104]

I6 GeoStoch-Bibliothek der Universität Ulm [109].

Alle Implementierungen verwenden allerdings zur Berechnung den gleichen Algorithmus, eine LU-Zerlegung durch einen Gaußschen Algorithmus mit Pivotsuche. Mit Hilfe von MuJava wurden aus fünf der sechs Implementierungen Mutanten erzeugt. Bei JScience (I2) mussten die Mutanten von Hand erzeugt werden, da MuJava nicht mit den Sprachelementen von Java umgehen kann, die ab Version 1.5 neu hinzugekommen sind. Tabelle 5.1 gibt einen Überblick über die erzeugten Mutanten.

Implementierung	term.	Exception	nicht-term.	insg.[1]	äquiv.
Commons.Math	175	339	20	611	27
Jama	102	300	12	468	27
Geostoch	82	223	6	357	24
Flanagan	221	432	25	782	39
Squire	83	225	18	400	24
JScience	10	0	0	10	0
insgesamt	673	1519	81	2628	141

Tabelle 5.1: Übersicht über die erzeugten Mutanten für Implementierungen zur Determinantenberechnung

[1] enthält auch Mutanten, die nicht syntaktisch korrekt sind und deshalb nicht compiliert werden können.

In der fünften Spalte ist die Gesamtzahl der erzeugten Mutanten angegeben, in drei Spalten davor die Anzahl der normal terminierenden, durch Werfen einer Exception terminierenden und der nicht innerhalb von fünf Minuten terminierenden Mutanten angegeben. Dieser Schwellenwert wurde aus zwei Gründen gewählt. Zum einen hat sich nach ersten Versuchen herausgestellt, dass einige Mutanten eine deutlich längere Laufzeit als die Originalimplementierung aufweisen. Zum anderen soll der Schwellenwert auch bei der Verwendung von unterschiedlichen Matrixgrößen bei den Eingaben ohne Anpassung verwendet werden können. Wird der Schellenwert zu niedrig angesetzt, werden einige Mutanten bei großen Eingabematrizen fälschlicherweise als nicht-terminierend eingestuft. Da nicht alle der erzeugten Mutanten vom Compiler akzeptiert werden, ist die Summe dieser drei Spalten nicht unbedingt gleich der Gesamtanzahl in Spalte 5.

Die äquivalenten Mutanten wurden durch Simulation bestimmt. Dazu wurden für jeden Mutanten 10.000.000 Matrizen zufällig erzeugt und das Ergebnis der Berechnung durch den Mutanten mit dem Ergebnis der Originalimplementierung verglichen. Wenn keine Abweichung festgestellt wurde, wurde der Mutant als äquivalent bezeichnet.

5.1.1 Matrizen und Metamorphische Relationen

In den Fallstudien wird vereinfachend angenommen, dass die Matrixelemente reelwertig sind. Eine Matrix $A \in \mathbb{R}^{n \times m}$ mit

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{pmatrix}$$

kann auch als Vektor ihrer Spalten(vektoren) $A = (a^1, \ldots, a^m)$ bzw. ihrer Zeilen $A = (a_1, \ldots, a_n)^T$ dargestellt werden. Die *Transposition* einer Matrix ist definiert durch $A^T = (a_{i,j})^T_{1 \leq i \leq n, 1 \leq j \leq m} = (a_{j,i})_{1 \leq i \leq n, 1 \leq j \leq m}$. Die *Untermatrix* $A_{i,j}$ einer Matrix $A \in \mathbb{R}_{n,m}$ erhält man durch Streichen der i-ten Zeile und der j-ten Spalte der Matrix A.

Für quadratische Matrizen $A \in \mathbb{R}^{n \times n}$ kann die Determinante $\det(A)$ bzw. $|A|$ berechnet werden. Diese ist durch

$$det(A) = \begin{vmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{vmatrix} := \sum_{\pi \in \mathcal{S}_n} sgn(\pi) \prod_{j=1}^{n} a_{j,\pi(j)}$$

definiert, wobei $\mathcal{S}_n$ die Permutationsgruppe (also die Menge aller Permutationen) von $(1,\ldots,n)$ und $sgn(\pi)$ das Vorzeichen der Permutation π bezeichnet.

Aus der mathematischen Definition und den Eigenschaften von Determinanten (siehe z.B. [76]) wurden insgesamt 12 unterschiedliche Relationen abgeleitet.

R1 *Transposition*
$det(A) = det(A^T)$

R2 *Vertauschung von Zeilen*
$-\,det((a_1,\ldots,a_n)^T)$
$= det((a_1,\ldots,a_{j-1},a_i,a_{j+1},\ldots,a_{i-1},a_j,a_{i+1},\ldots,a_n)^T)$
für $i,j \in \{1,\ldots,n\}, j < i$

R3 *Vertauschung von Spalten*
$-\,det((a^1,\ldots,a^n))$
$= det((a^1,\ldots,a^{j-1},a^i,a^{j+1},\ldots,a^{i-1},a^j,a^{i+1},\ldots,a^n))$
für $i,j \in \{1,\ldots,n\}, j < i$

R4 *Multiplikation einer Zeile mit einem Skalar*
$\beta\, det((a_1,\ldots,a_{k-1},a_k,a_{k+1},\ldots,a_n)^T)$
$= det((a_1,\ldots,a_{k-1},\beta a_k,a_{k+1},\ldots,a_n)^T)$
für $k \in \{1,\ldots,n\}$

R5 *Multiplikation einer Spalte mit einem Skalar*
$\beta\, det((a^1,\ldots,a^{k-1},a^k,a^{k+1},\ldots,a^n))$
$= det((a^1,\ldots,a^{k-1},\beta a^k,a^{k+1},\ldots,a^n))$
für $k \in \{1,\ldots,n\}$

R6 *Addition von zwei Zeilen zweier Matrizen*
$det((a_1,\ldots,a_{k-1},a_k+b_k,a_{k+1},\ldots,a_n)^T)$
$= det((a_1,\ldots,a_{k-1},a_k,a_{k+1},\ldots,a_n)^T)$
$+\,det((a_1,\ldots,a_{k-1},b_k,a_{k+1},\ldots,a_n)^T)$
für $k \in \{1,\ldots,n\}$

R7 *Addition von zwei Spalten zweier Matrizen*
$det((a^1,\ldots,a^{k-1},a^k+b^k,a^{k+1},\ldots,a^n))$
$= det((a^1,\ldots,a^{k-1},a^k,a^{k+1},\ldots,a^n))$

$+ det((a^1, \ldots, a^{k-1}, b^k, a^{k+1}, \ldots, a^n))$
für $k \in \{1, \ldots, n\}$

R8 *Gauß-Schritt in den Zeilen*
$det((a_1, \ldots, a_n)^T)$
$= det((a_1, \ldots, a_{j-1}, a_j + \beta a_i, a_{j+1}, \ldots, a_n)^T)$
für $\beta \in \mathbb{R}, i, j \in \{1, \ldots, n\}, i \neq j$

R9 *Gauß-Schritt in den Spalten*
$det((a^1, \ldots, a^n))$
$= det((a^1, \ldots, a^{j-1}, a^j + \beta a^i, a^{j+1}, \ldots, a^n))$
für $\beta \in \mathbb{R}, i \in \{1, \ldots, n\}, i \neq j$

R10 *Entwicklung in einer Zeile*
$det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j}\, det(A_{i,j})$
für $i \in \{1, \ldots, n\}$

R11 *Entwicklung in einer Spalte*
$det(A) = \sum_{i=1}^{n} (-1)^{i+j} a_{i,j}\, det(A_{i,j})$
für $j \in \{1, \ldots, n\}$

R12 *Multiplikation*
$det(A) \cdot det(B) = det(AB)$

In den Relationen kann man gut die wichtigsten Sätze und Rechenregeln für den Umgang mit Determinanten erkennen, z.B. sind die Relationen R10 und R11 direkt auf den Entwicklungssatz von Laplace zurückzuführen. Diese Auflistung legt es nahe, dass sich mathematisch gut untersuchte Objekte und Funktionen besonders gut für die Untersuchung des Metamorphischen Testens eignen, da Relationen besonders einfach abzuleiten sind.

Bisherige Untersuchungen hatten das Manko, zum einen nur wenige Relationen miteinander zu vergleichen, zum anderen waren diese Relationen oft strukturell sehr ähnlich. In der vorliegenden Arbeit werden dagegen Relationen untersucht, die strukturell sehr unterschiedlich sind. Das Ziel der Untersuchungen ist es, eventuell aus der Struktur der Relationen Rückschlüsse auf deren Effektivität zu ziehen.

5.1.2 Testeingaben

Die Eingaben lassen sich grob in zwei Klassen einteilen: quadratische und nicht quadratische Matrizen (K1). Da Determinanten nur für quadratische Matrizen definiert sind, sollte die SUT im Falle von nicht-quadratischen Eingaben die ungültige Eingabe behandeln, beispielsweise durch Werfen einer Exception.

Die Klasse der quadratischen Matrizen lässt sich weiter unterteilen:

K2 Matrizen ohne Elemente:
Die SUT sollte in diesem Fall ebenfalls einen Fehler anzeigen

K3 Matrizen mit einem Element:
Die Determinante ist in diesem Fall gleich dem einzigen Element

K4 singuläre Matrizen:
hier gilt immer $|A| = 0$ bzw. $|A| < \epsilon$, $\epsilon > 0$

K5 reguläre Matrizen mit mehr als einem Element:
In diesem Fall ist die Berechnung der Determinante eindeutig nicht trivial. Diese Teilklasse an Eingaben wird in der Studie verwendet werden, um die Metamorphischen Relationen zu untersuchen.

Um eine große Anzahl an (unterschiedlichen) Eingaben automatisch erzeugen zu können, wurde eine zufällige Eingabeerzeugung gewählt. Dabei wird jedes Element der Matrix als iid Realisierung einer im Intervall $[-1000, 1000]$ gleichverteilten Zufallsvariable erzeugt. Diese Art der Erzeugung hat die gewünschte Eigenschaft, dass die so erzeugte Matrix fast sicher regulär ist, denn wenn A singulär wäre, dann müßte sich mindestens eine Zeile als Linearkombination der übrigen Zeilen darstellen lassen. Es müssten also Konstanten $\lambda_i \neq 0$ existieren, so dass $\lambda_k a_k = \sum_{1 \leq i \leq n, i \neq k} \lambda_i a_i$ für mindestens ein $1 \leq k \leq n$ gilt. Dass dies der Fall ist, ist bei einer zufälligen Erzeugung der Matrizen sehr unwahrscheinlich. Mit anderen Worten, die Wahrscheinlichkeit dafür ist nahezu null.

Für die untersuchten Implementierungen spielt es allerdings keine Rolle, ob die übergebene Matrix regulär oder singulär ist. In beiden Fällen arbeitet der verwendete Algorithmus korrekt. Der Unterschied liegt in der Bestimmung der erwarteten Ausgaben des SUT. Für beliebige singuläre Matrizen als Eingaben ist die erwartete Ausgabe im Voraus bekannt (da in diesem Falle die Determinante gleich 0 ist), bei beliebigen regulären

Matrizen muss die erwartete Ausgabe erst (aufwändig) bestimmt werden. Im Falle von singulären Matrizen liegt also kein wirkliches Testproblem vor.

5.1.3 Ergebnisse der Studie

Zuerst wurden an den Originalimplementierungen überprüft, ob diese alle Eingabeklassen bis auf (K5) richtig behandeln. Die dazu benötigten Eingaben wurden ebenfalls automatisch erzeugt. Um singuläre Matrizen zu erhalten, wurde zuerst eine zufällige Matrix erzeugt. Anschließend wurden alle Elemente einer zufällig gewählten Zeile der Matrix mit 0 belegt. Alle untersuchten Implementierungen berechneten die erwarteten Ergebnisse. Nur ungültige Eingaben (K2) bereiteten einer einzigen Implementierung (I4) Probleme.

Anschließend wurden die Implementierungen mit Metamorphischen Relationen getestet. Für jede der gewählten Implementierungen wurde auf dieselbe Art und Weise vorgegangen. Jeder Mutant wurde mit allen zwölf Relationen untersucht. Dazu wurden für $n = 2, 4, \ldots, 20$ die Relation 100 mal überprüft. Die dazu benötigten Eingaben wurden teils (wie oben beschrieben) zufällig erzeugt und teils aus den zufällig erzeugten Matrizen so berechnet, dass die Eingaberelation erfüllt ist. Um z. B. die Eingaberelation von R12 zu erfüllen, wurden zwei Matrizen A, B zufällig erzeugt, die dritte Matrix erhält man aus A und B durch Multiplikation der beiden Matrizen.

Bei jedem der Metamorphischen Tests wurde zudem überprüft, ob die Originalimplementierung die jeweilige Metamorphische Relation erfüllt. Dies war selten nicht der Fall. Die Ursache war meistens eine fehlerhafte Implementierung der Metamorphischen Relation. Selten waren aber auch Rundungsfehler die Ursache. Wenn die Eingabe-Matrizen schlecht konditioniert sind (was bei zufälliger Erzeugung vorkommen kann), ist der verwendete Algorithmus bei großen Matrizen anfällig. Um Probleme mit Rundungsfehlern zu umgehen, können entweder kleinere Matrizen oder Matrizen mit Integerwerten als Elemente verwendet werden. Wenn eine Matrix aus Integer-Elementen besteht, ist auch die Determinante ein Integerwert. So können Probleme mit Gleitpunktarithmetik umgangen werden. Gegebenenfalls ist aber bei der Verwendung von Integer-Elementen auf einen möglichen Integer-Overflow zu achten. Probleme mit Rundungsfehlern traten allerdings nur sehr selten auf und gefährdeten zu keinem Zeitpunkt die Gültigkeit der gewonnenen Ergebnisse.

Jede Überprüfung einer Relation führt zu mindestens zwei Ausführungen des Mu-

tanten mit unterschiedlichen Eingaben. Ist die (Ausgabe-)Relation bei mindestens einer Überprüfung nicht erfüllt, gilt der Mutant als getötet. Um die Effektivität der Relation bestimmen zu können, wurde für jede erzeugte Eingabe das vom Mutanten berechnete Ergebnis zusätzlich mit dem Ergebnis der Originalimplementierung verglichen. Wenn die Relation durch den Mutanten nicht erfüllt wird, müssen die Ausgaben von Original und Mutant für mindestens eine der Eingaben voneinander abweichen, da die Originalimplementierung die Relation erfüllt.

Die Effektivität der Metamorphischen Tests könnte eventuell durch die Größe der Eingabematrizen beeinflusst werden. Um diesen Einfluss zu quantifizieren wurden die Tests mit unterschiedlichen Größen der Eingabematrizen durchgeführt. In jedem Durchlauf wurde die Matrixgröße fixiert und die Effektivität der Metamorphischen Tests bei dieser Matrixgröße bestimmt.

In Abbildung 5.1 sind die Ergebnisse für die Commons.Math-Bibliothek abgebildet. Die X-Achse beschreibt die Größe der Eingabematrizen, die Y-Achse die Effektivität der Relationen. Basierend auf der Abbildung kann man die Relationen grob in 4 Gruppen einteilen. Die Relation R12 ist am effizientesten, nahe am Optimum. In der zweiten Gruppe sind R3, R8, R9, R10, R11 mit einer Effektivität von mehr als 0.8. Die dritte Gruppe besteht aus R1, R4, R6 und R7 mit einer Effektivität zwischen 0.7 und 0.8. In der letzten Gruppe befinden sich R2 und R5. Die Effektivität der Relationen verändert sich kaum in Abhängigkeit von der Matrixgröße. Die Verwendung von Matrizen mit einer Größe von mehr als $n = 8$ würde unter Effektivität- und Laufzeitaspekten keinen Sinn machen.

Die Ergebnisse für die JAMA-Bibliothek sind in Abbildung 5.2 präsentiert. Die Relation R12 schneidet am besten ab, die Relationen R2 und R5 am schlechtesten. Die restlichen Relationen unterscheiden sich nicht so stark wie in Abb. 5.1, allerdings ist erkennbar, dass die Relationen R10 und R11 unter diesen am besten abschneiden und die Relationen R1, R4 und R7 in dieser Gruppe am wenigsten effizient sind.

Ein anderes Bild ergibt sich bei den Ergebnissen für die Implementierung I4 (Matrix-Klasse von Flangan) in Abb. 5.3. Die Effektivität der Relationen ist deutlich größer als bei allen anderen Implementierungen. Am besten schneidet wieder R12 ab, aber R2 und R5 sind nicht mehr die schlechtesten Relationen. Dafür verschlechtert sich die Effektivität von R4 sehr deutlich bei ansteigender Matrix-Größe. Alle Relationen liegen so dicht beisammen, dass eine Wertung eher problematisch ist.

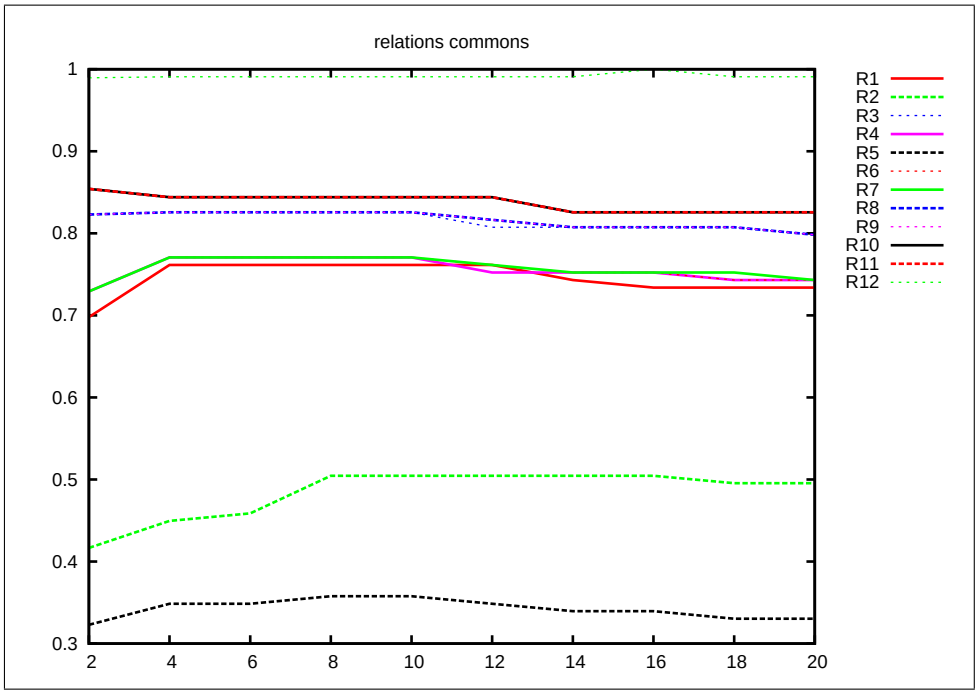


Abbildung 5.1: Effektivität der Relationen (Commons.Math)

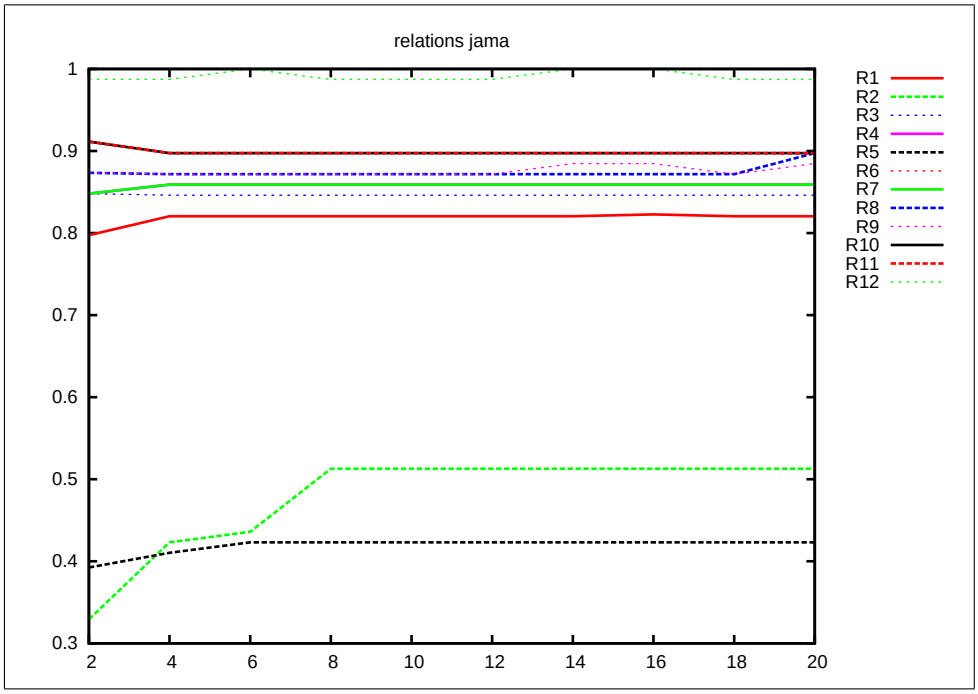


Abbildung 5.2: Effektivität der Relationen (JAMA)

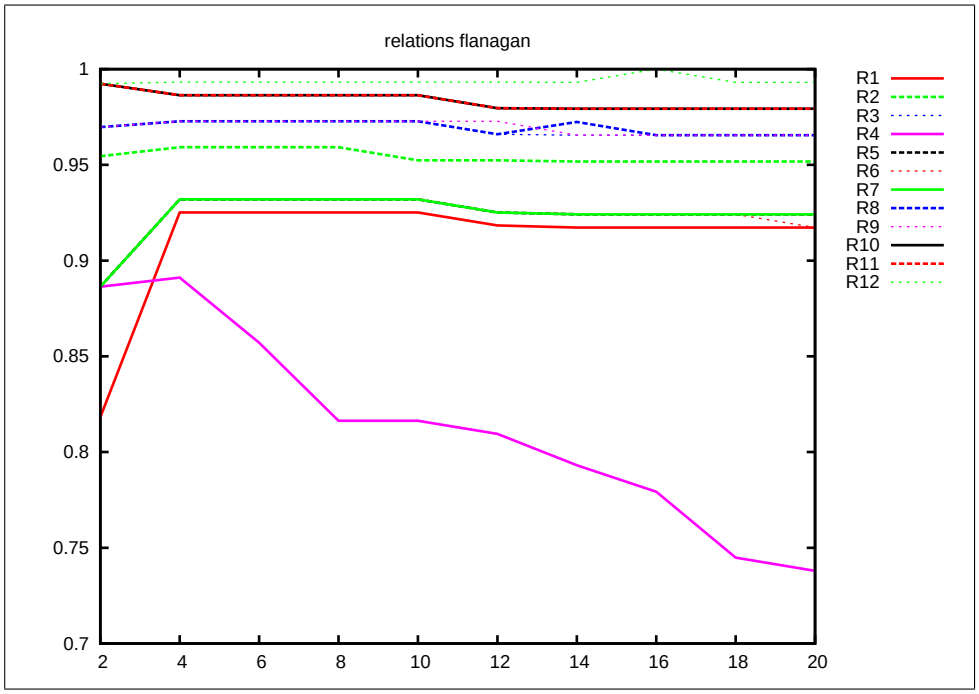


Abbildung 5.3: Effektivität der Relationen (Flanagan)

Bei den Ergebnissen zur Implementierung von Squire (Abb. 5.4) bietet sich wieder das Bild der ersten beiden Implementierungen. Die Relationen R2 und R5 sowie R1 schneiden am schlechtesten ab, R12 am besten, die anderen Relationen unterscheiden sich kaum in ihrer Effektivität.

Die Untersuchung der Implementierung der GeoStoch-Bibliothek, abgebildet in Abb. 5.5, bestätigen weitgehend die bisherigen Ergebnisse. Ein Unterschied zu diesen ist jedoch die deutlichere Abhängigkeit der Effektivität von der Matrixgröße. Einzig die am besten abschneidende Relation R12 entzieht sich diesem Effekt.

Fasst man nun die Ergebnisse für die einzelnen Implementierungen zusammen, ergibt sich folgendes Bild. Die Relation R12 ist für alle Implementierung die beste Wahl, die Effektivität liegt nahe am Optimum und wird nicht von der Größe der Eingabe-Matrizen beeinflusst. Danach schneiden die Relationen R10 und R11 ebenfalls immer gut ab. Betrachtet man die am wenigsten effizienten Relationen, so findet man in allen Versuchen die Relationen R1, R2, R4 und R5 auf den hinteren Plätzen.

Auffällig bei den schlecht abschneidenden Relationen ist, dass die überprüften Eigenschaften (bis auf R1) von den Algorithmen benutzt werden, um das Ergebnis zu

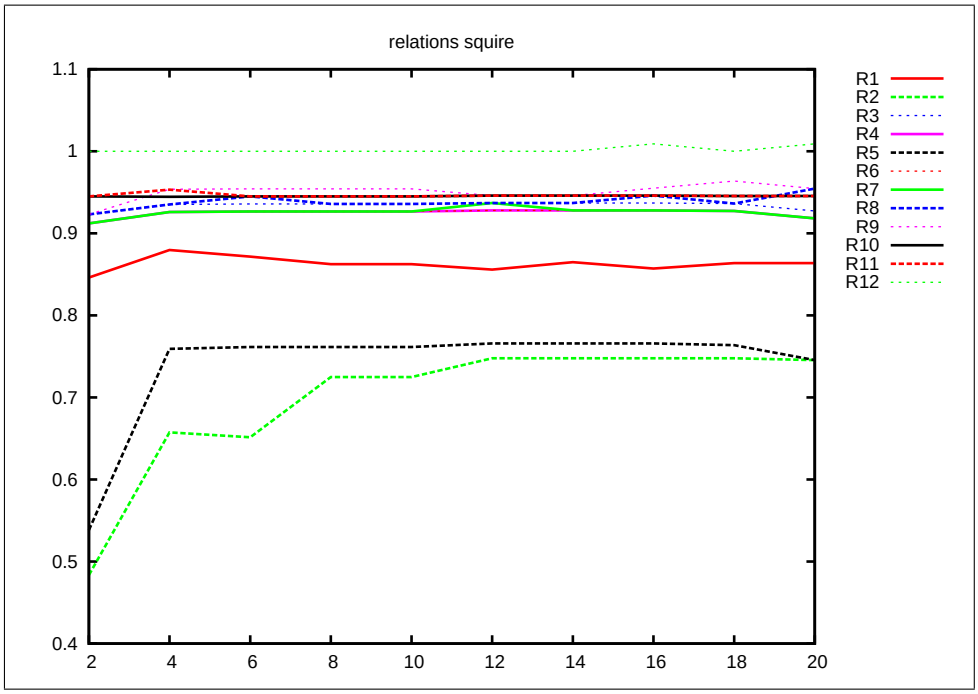


Abbildung 5.4: Effektivität der Relationen (Squire)

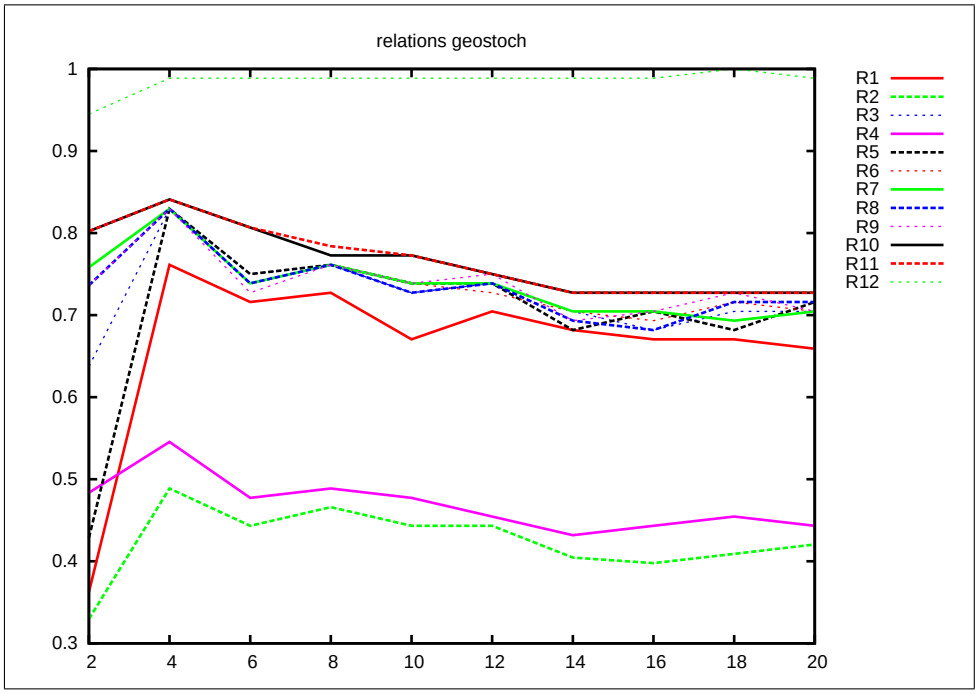


Abbildung 5.5: Effektivität der Relationen (GeoStoch)

berechnen. Das legt den Schluss nahe, dass solche Relationen für Testzwecke weniger geeignet sind. Die Pivotierung sorgt beispielsweise dafür, dass die Änderungen durch die Eingaberelation (z.B. Vertauschen zweier Spalten) wieder rückgängig gemacht wird, und beide Eingaben wieder zum gleichen internen Ablauf führen.

5.2 Metamorphisches Testen von Bildanalysesoftware

In einer weiteren Fallstudie wird nun untersucht, ob sich Metamorphisches Testen für das automatisierte Testen von Bildanalysesoftware eignet. Diese Art von Software ist nur schwer automatisiert zu testen, insbesondere fehlen in der Literatur automatisierbare Orakel. Üblicherweise werden Bildanalyseprogramme manuell mit Hilfe von im Laufe der Zeit akzeptierten Testbildern[2] getestet. Das Progamm wird auf diese Bilder angewandt und das Ergebnis der Bearbeitung wird dann von einem menschlichen Tester begutachtet. Dieses Vorgehen ist zum einen wegen der Tatsache problematisch, dass sogar erfahrenen Testern kleine subtile Fehler bei der Begutachtung entgehen können. Zum anderen lässt sich die Testentscheidung durch einen menschlichen Tester nicht automatisieren.

Neben der Evaluierung von Metamorphischen Relationen war es zusätzlich die Absicht dieser Fallstudie , einen vollständig automatisierten Test von Bildanalysesoftware zu beschreiben. In den folgenden Teilkapiteln wird nun dieser Ansatz präsentiert. Nach einer kurzen Einführung in die Grundbegriffe der Bildanalyse werden zunächst Mechanismen zur automatischen Erzeugung von Bildern beschrieben. Diese automatisch erzeugten Bilder werden im weiteren Verlauf der Fallstudie als Testeingaben verwendet. Anschließend werden Metamorphische Relationen vorgestellt, die später zum Test der Implementierungen verwendet werden sollen. Nach der Vorstellung der Studienobjekte werden dann die Resultate der Fallstudien präsentiert.

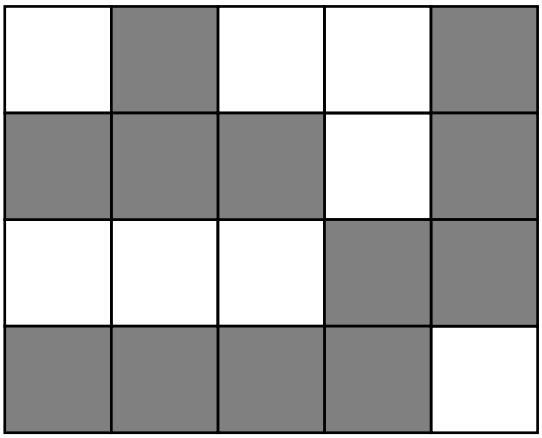
5.2.1 Bilder und Bildoperatoren

In diesem Kapitel werden nur die nötigsten Grundlagen zu Bildern und Bildoperatoren dargestellt. Details findet man u. a. in [56, 103].

Bilder, genauer gesagt zweidimensionale Bilder, werden als Matrix der Pixelwerte gesehen. Ein Bild der Größe $n_x \times n_y$ (Breite $\times$ Höhe) ist also eine Matrix $M \in \mathcal{R}^{n_y \times n_x}$, wobei $\mathcal{R}$ die Menge der zulässigen Pixelwerte ist. In dieser Arbeit werden Binärbilder (Schwarz-Weiß-Bilder) und Graustufenbilder betrachtet. Im Fall von Binärbildern ist $\mathcal{R} = \{0, 1\}$, wobei der Wert 0 für die Farbe Weiß und entsprechend 1 für Schwarz steht. Bei Graustufenbildern ist $\mathcal{R} = \{0, \ldots, g_{\max}\}$. Ein Beispiel für ein Binärbild und dessen Matrixdarstellung ist in Abb. 5.6 abgebildet.

Die Bearbeitung von Bildern erfolgt oft pixelweise. Die Matrixdarstellung des Bildes

[2] Eine Auflistung von Testbildern findet sich z. B. unter
http://www.cs.cmu.edu/afs/cs/project/cil/ftp/html/v-images.html



(a)

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(b)

Abbildung 5.6: A (a) Binärbild und dessen (b) Matrixdarstellung

impliziert dabei, dass das Pixel links oben im Bild als erstes bearbeitet wird, intuitiverweise würde man jedoch links unten beginnen. Legt man das Bild in ein Koordinatensystem ist also die Richtung der Y-Achse gegenüber des „normalen“ Koordinatensystems vertauscht.

Ein *Bildoperator* ist eine Funktion, die Bilder auf andere Bilder abbildet. Bildoperatoren spielen in dieser Fallstudie an zwei Stellen eine wichtige Rolle. Die untersuchten Programme implementieren (komplexe) Bildoperatoren, die mit unterschiedlichen Verfahren, u.a. mit Metamorphischem Testen, getestet werden sollen. Auch die beim Metamorphischen Testen verwendeten Relationen lassen sich als Bildoperator interpretieren. Im folgenden werden einfache Bildoperatoren dargestellt, die später für die Beschreibung von Metamorphischen Relationen benötigt werden.

Die folgenden Operatoren können auf beliebige Bilder $A = (a_{i,j}) \in \mathcal{R}^{n_y \times n_x}$ angewandt werden:

- *Drehung* eines Bildes A um 90 Grad gegen den Uhrzeigersinn:
  $R(A) = (r_{i,j}) \in \mathcal{R}^{n_x \times n_y}$ mit $r_{i,j} = a_{n_y-j-1,i}$.

- *Spiegelung* eines Bildes an der X-Achse:
  $M_y(A) = (m_{i,j}) \in \mathcal{R}^{n_y \times n_x}$ mit $m_{i,j} = a_{n_y-i-1,j}$.

- *Spiegelung* eines Bildes an der Y-Achse:
  $M_x(A) = (m_{i,j}) \in \mathcal{R}^{n_y \times n_x}$ mit $m_{i,j} = a_{i,n_x-j-1}$.

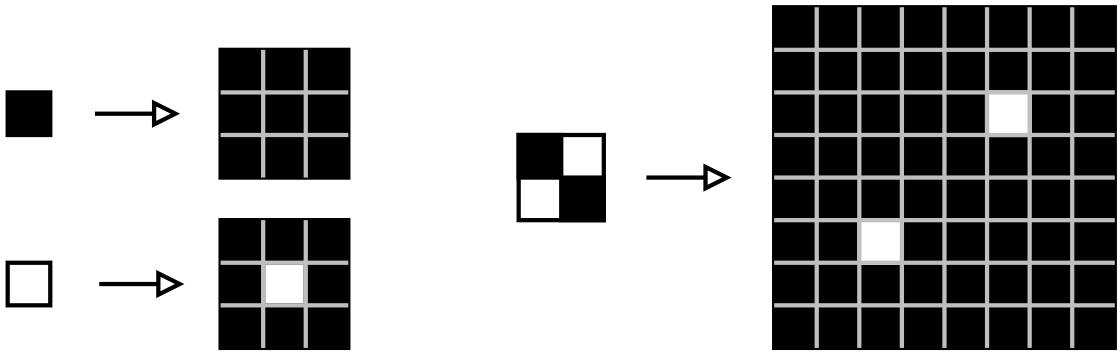


Abbildung 5.7: Illustration des Vergrößerungsoperators

- *Transposition*:
  $A^T = (b_{i,j}) \in \mathcal{R}^{n_y \times n_x}$ mit $b_{i,j} = a_{j,i}$.

Für Binärbilder sind zusätzliche Operatoren definiert:

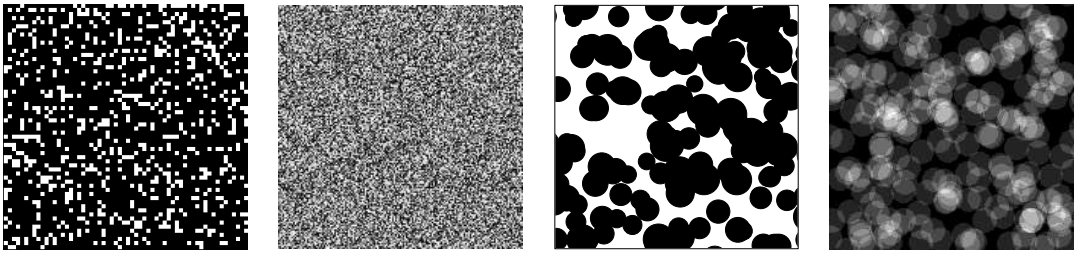
- der *Schnitt* zweier (Binär-)Bilder $A = (a_{i,j}),\ B = (b_{i,j}) \in \{0,1\}^{n_y,n_x}$:
  $A \cap B = (c_{i,j}) \in \{0,1\}^{n_y \times n_x}$ mit $c_{i,j} = a_{i,j} \wedge b_{i,j}$.
- die *Vereinigung* zweier (Binär-)Bilder $A = (a_{i,j}),\ B = (b_{i,j}) \in \{0,1\}^{n_y \times n_x}$:
  $A \cup B = (u_{i,j}) \in \{0,1\}^{n_y \times n_x}$ mit $u_{i,j} = a_{i,j} \vee b_{i,j}$.
- die *Vergrößerung* eines Binärbildes $E(A) = (e_{i,j}) \in \{0,1\}^{3n_y+2 \times 3n_x+2}$ mit

$$e_{i,j} = \begin{cases} a_{\lfloor i/3 \rfloor, \lfloor j/3 \rfloor} & \text{, falls } i \bmod 3 = 2 \text{ und } j \bmod 3 = 2 \\ 1 & \text{, sonst} \end{cases}$$

  Bei der Anwendung dieses Operators wird jedes Pixel mit 8 schwarzen Pixeln umgeben. Zusätzlich erhält das neue Bild einen schwarzen Rahmen von 1 Pixel Breite. Abb. 5.7 illustriert diesen Operator. Die linke Seite des Bilds zeigt das Erweitern eines Pixels um 8 Pixel. Die rechte Seite zeigt ein Ausgangsbild und das resultierende Bild nach Anwendung des Operators E.

5.2.2 Automatische Erzeugung von Bildern

Um die im nächsten Kapitel vorgestellten Algorithmen zu testen werden Bilder als Eingaben benötigt. Für Binär- und Graustufenbilder werden jeweils zwei Methoden vorgestellt, um zufällige Bilder zu erzeugen.



(a) (b) (c) (d)

Abbildung 5.8: Realisierungen des (a) *random binary* - Modells , (b) a random grayscale - Modells, (c) Booleschen Modells und (d) Booleschen Graustufen - Modells

random binary

Der einfachste Ansatz für Binärbilder ist es, Pixel zufällig schwarz oder weiß zu färben. Diese Methode hat einen einzigen Parameter, die Wahrscheinlichkeit $0 < p \leq 1$, mit der ein Pixel schwarz gefärbt wird. In der praktischen Durchführung wird jedes Pixel unabhängig von allen anderen Pixeln gefärbt. Offensichtlich ist bei diesem Modell der Anteil der schwarzen Pixel im Mittel gleich p. Dieser Ansatz wird im folgenden als *random binary - Modell* bezeichnet. Abb. 5.8(a) zeigt als Beispiel ein Bild der Größe 60×60 mit $p = 0.8$.

random grayscale

Analog dazu können auf einfache Weise Graustufenbilder (zufällig) erzeugt werden. Legt man die Anzahl der Graustufen $g_{\max}$ fest, kann die Farbe jedes Pixels unabhängig und (diskret) gleichverteilt auf $\{0, \ldots, g_{\max}\}$ gewählt werden. Abb. 5.8(b) zeigt ein Beispiel.

Boolesches Modell

Zusätzlich zu diesen beiden sehr einfachen Modellen zur Bilderzeugung werden in dieser Arbeit auch komplexere Methoden zur Bilderzeugung untersucht. Diese basieren auf dem sog. *Booleschen Modell*. Im Folgenden wird das Boolesche Modell kurz beschrieben. Details finden sich u.a. in [67, 81, 106]. Das Boolesche Modell gehört zur Klasse der sog. Keim-Korn-Modelle. Dieses besteht, wie der Name schon andeutet, aus zwei Komponenten. Die Körner sind geometrische Objekte, beispielsweise Kreise oder Linien. Das Modell erlaubt dabei auch, dass die Körner Zufallsvariablen sind, beispielsweise Kreise

mit zufälligem Radius.

Die Keime beschreiben die (zufälligen) Lokationen der Körner im Bild. Beim Booleschen Modell wird ein stationärer zweidimensionaler Poisson-Prozess zur Beschreibung der Keime verwendet. Dieser ist definiert durch

Definition 4 *Poisson-Prozess*

Π *ist ein stationärer (zweidimensionaler) Poisson-Prozess mit Parameter* λ, $0 < \lambda < \infty$, *wenn*

- *die Anzahl der Punkte in einer beschränkten Borel-Menge* $B \subset \mathbb{R}^2$ *einer Poisson-Verteilung mit Parameter* $\lambda\mu(B)$ *folgt (wobei* $\mu(\cdot)$ *das Lebesgue-Maß ist),*
- *für disjunkte, beschränkte Borel-Mengen* $B_1, \ldots, B_n \subset \mathbb{R}^2$ *die Anzahl der Punkte in jeder Menge* B_i *(*$1 \leq i \leq n$*) unabhängige Zufallsvariablen sind.*

Das Boolesche Modell lässt sich dann wie folgt formal beschreiben:
Sei $\Pi = (X_1, X_2, \ldots)$ ein (stationärer) Poisson-Prozess mit Parameter $\lambda > 0$. Ferner sei $\Xi_1, \Xi_2, \ldots$ eine iid Folge von zufälligen kompakten Mengen, unabhängig von Π. Dann ist (wenn gewisse Integrabilitätsbedingungen erfüllt sind, siehe [67]) das Boolesche Modell gegeben durch

$$\Xi = \bigcup_{i=1}^{\infty} (X_i + \Xi_i)$$

Das Boolesche Modell kann leicht in ein Binärbild umgewandelt werden. Dazu betrachtet man eine eine Realisierung ξ von Ξ in einem Beobachtungsfenster W. Über dieses Beobachtungsfenster wird dann ein Gitter gelegt. Die Gitterpunkte entsprechen den Pixeln des Binärbilds. Für jeden Gitterpunkt wird dann überprüft, ob dieser in ξ liegt oder nicht. Man wählt also $A = (a_{i,j})$ mit $a_{i,j} = \mathbb{I}_{\{(i,j) \in \xi \cap W\}}$. Ein Beispiel für eine Realisierung eines Booleschen Modells ist in Abb. 5.8(c) gegeben. Zwischen dem Anteil der schwarzen Pixel im Bild p einerseits und dem Parameter λ des Booleschen Modells und der erwarteten Fläche der Körner $\mu(\Xi_i)$ kann man einen direkten Zusammenhang darstellen (siehe [81]). Es gilt:

$$p = 1 - e^{-\lambda\mu(\Xi_i)}$$

Wählt man Kreise mit festem Radius $r > 0$ als Körner des Booleschen Modells, ergibt sich aus obiger Formel $p = 1 - e^{-\lambda\pi r^2}$ bzw. für ein gegebenes p gilt $\lambda = -\ln(1-p)/\pi r^2$.

Auf diese Weise lassen sich das *random binary* - Modell und das Boolesche Modell einheitlich parametrisieren.

Graustufen-Version des Booleschen Modells

Auf Basis des Booleschen Modells lässt sich auf einfache Weise auch ein Graustufenbild erzeugen, indem man für jedes Pixel zählt, in wie vielen Realisierungen von (verschobenen) Körnern des Booleschen Modells sich das Pixel befindet. Formal beschrieben: Sei $\xi = \cup_{i=1}^{\infty}(x_i + \xi_i)$ eine Realisierung eines Booleschen Modells. Dann wird durch $A = (a_{i,j})$ mit

$$a_{i,j} = \sum_{k=1}^{\infty} \mathbb{I}_{\{(i,j)\in(x_k+\xi_k)\cap W\}}$$

ein Graustufenbild erzeugt. Ein Beispiel für ein durch diese Methode erzeugtes Bild ist in Abb. 5.8(d) gezeigt.

5.2.3 Untersuchte Algorithmen

In der Fallstudie werden drei unterschiedliche Algorithmen zur Bildanalyse untersucht. Diese Algorithmen werden im Folgenden kurz vorgestellt.

Euklidische Distanztransformation

Der erste Algorithmus berechnet die Distanztransformation von Binärbildern (siehe z. B. [14]). Die Distanztransformation ist ein Bildoperator, der ein Binärbild auf ein Graustufenbild abbildet. In diesem erhält jedes weiße Pixel den Wert 0, jedes Pixel an der Stelle, an der im Originalbild sich ein schwarzes Pixel befindet, erhält als Wert dessen Distanz zum nächsten weißen Pixel. Formal kann die Distanztransformation beschrieben werden durch eine Abbildung $D : \{0,1\}^{n_x,n_y} \to G^{n_x,n_y}$ mit

$$\begin{aligned} D(A) &= D((a_{i,j})) = (d_{i,j})_{0\le i<n_x,\, 0\le j<n_y} \\ d_{i,j} &= \min\{\mathrm{dist}((i,j),(k,l)) : a_{k,l} = 0, 0 \le k < n_y, 0 \le l < n_x\}, \end{aligned}$$

wobei $G = \{0, \ldots, g_{\max}\}$ Die Art der Distanzberechnung $\mathrm{dist}(\cdot,\cdot)$ kann dabei beliebig gewählt werden. In diesem Fall wurde die Euklidische Distanz $\mathrm{dist}_{\mathcal{E}}$ mit

$$\mathrm{dist}_{\mathcal{E}}((i,j),(k,l)) = \sqrt{\Delta_y^2(i-k)^2 + \Delta_x^2(j-l)^2},$$

(a) (b)

Abbildung 5.9: Ein (a) Binärbild und dessen (d) Euklidische Distanztransformation

gewählt. Die Konstanten Δ_x und Δ_y bezeichnen die Gitterabstände des Pixelgitters, werden aber üblicherweise auf den Wert 1 festgelegt. Um bei der Diskretisierung der obigen Formel die Gültigkeit der Dreiecksungleichung zu erhalten, wird das Ergebnis aufgerundet:

$$\mathrm{dist}_{\mathcal{E}'}((i,j),(k,l)) = \left\lceil \sqrt{\Delta_y^2(i-k)^2 + \Delta_x^2(j-l)^2} \right\rceil .$$

Die Implementierung der Euklidischen Distanztransformation wurde der GeoStoch-Bibliothek [109] entnommen und verwendet den in [30] beschriebenen Algorithmus. Abb. 5.9 zeigt ein Bild und dessen diskretisierte Euklidische Distanztransformation.

Lipschitzfilter

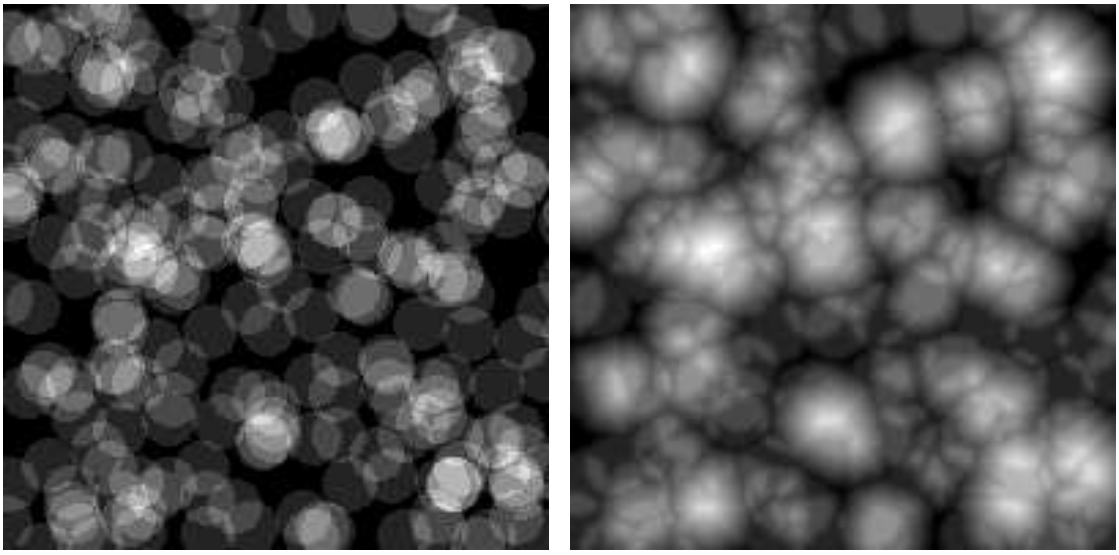
Eine weitere Möglichkeit zur Wahl von $\mathrm{dist}(\cdot,\cdot)$ ist die sog. Cityblock- oder Champfer-Distanz dist_C:

$$\mathrm{dist}_C((i,j),(k,l)) = \Delta_x|i-k| + \Delta_y|j-l|.$$

Für ein Binärbild ist eine Distanztransformation mit Cityblock-Distanz äquivalent zum sog. *Lipschitz-Filter* [14, 84]. Dieser ist definiert durch die Abbildung $L : G^{n_x,n_y} \to G^{n_x,n_y}$ mit

$$\begin{aligned} L(A) &= (l_{i,j}) \\ l_{i,j} &= \inf_{k,l}\{a_{k,l} + \mathrm{dist}_C((i,j),(k,l))\},\ 0 \le i,k < n_x \text{ und } 0 \le j,l < n_y. \end{aligned}$$

Der Lipschitz-Filter kann dazu verwendet werden, Schattierungen in Bildern zu erzeugen oder um verrauschte Hintergründe zu entfernen. Ein Beispiel ist in Abb. 5.10 gegeben.



(a) (b)

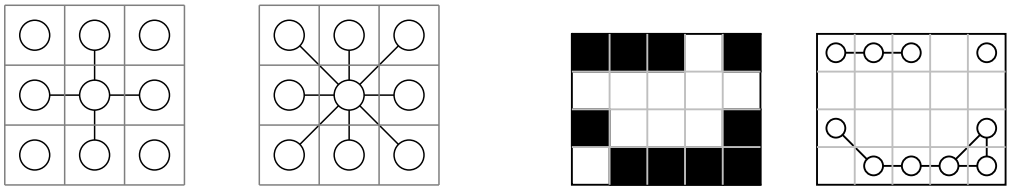
Abbildung 5.10: (a) Graustufenbild und (b) dasselbe Bild nach Anwendung des Lipschitzfilters

Färbung von Zusammenhangskomponenten

Als dritter Algorithmus wurde ein Verfahren zum Färben von Zusammenhangskomponenten in Binärbildern untersucht. Der Algorithmus betrachtet Pixel als Knoten in einem Graphen. Welche Pixel durch Kanten miteinander verbunden sind, wird durch sog. Nachbarschaften definiert. Die gebräuchlichsten Nachbarschaften sind die 4-Nachbarschaft und die 8-Nachbarschaft. Die 4-Nachbarschaft verbindet ein Pixel mit den vier Pixel, die sich oberhalb, unterhalb, links und rechts befinden. Die 8-Nachbarschaft verbindet ein Pixel mit 8 weiteren, die sich in einem 3×3-Quadrat um das Pixel befinden. Abb. 5.11(a) illustriert die Nachbarschaften. Links ist die 4-Nachbarschaft und rechts die 8-Nachbarschaft abgebildet.

Eine Zusammenhangskomponente besteht dann aus den schwarzen Pixeln, die über Nachbarschaftsbeziehungen miteinander verbunden sind. In Abb. 5.11(b) ist ein Beispielbild und dessen Interpretation als Graph zu sehen.

Im dem vom Beispiel-Bild in Abb. 5.11(b) abgeleiteten Graphen befinden sich drei Zusammenhangskomponenten. Würde statt der 8-Nachbarschaft die sog. 4-Nachbarschaft verwendet werden, würden sich vier Zusammenhangskomponenten im Graphen befinden, da dann das erste Pixel in der dritten Zeile nicht mit anderen Pixeln verbunden wäre.



(a) (b)

Abbildung 5.11: (a)Nachbarschaften und (b)die Interpretation eines Bildes als Graph (8-Nachbarschaft)

Der Algorithmus färbt jedes Pixel einer (ursprünglich schwarzen) Zusammenhangskomponente mit derselben Graustufe. In Abb. 5.12 ist ein Binärbild und das entsprechende durch Färbung der Zusammenhangskomponenten entstandene Graustufenbild abgebildet. Der in der Implementierung verwendete Algorithmus wurde ursprünglich in [93] veröffentlicht, die Implementierung orientiert sich jedoch an [101].

5.2.4 Metamorphische Relationen und andere Orakel

Basierend auf den in Kapitel 5.2.1 vorgestellten Bildoperatoren lassen sich Metamorphische Relationen definieren.

Für beliebige Bilder $A \in \mathcal{R}^{n_y,n_x}$ können die folgenden Relationen definiert werden.

R1 *Drehung gegen den Uhrzeigersinn um* 90°
$R(O(A)) = O(R(A))$

R2 *Spiegelung an der Y-Achse*
$M_x(O(A)) = O(M_x(A))$

R3 *Spiegelung an der X-Achse*
$M_y(O(A)) = O(M_y(A))$

R4 *Transposition*
$(O(A))^T = O(A^T)$

Offensichtlich sind diese Relationen nicht für alle Bildoperatoren $O(\cdot)$ gültig, aber alle im vorherigen Kapitel vorgestellten Operatoren erfüllen diese Eigenschaften. Im Allgemeinen erfüllen alle rotationsinvarianten Operatoren diese vier Metamorphischen Relationen. Da viele Bildoperatoren rotationsinvariant sind, wird in dieser Arbeit versucht,

(a) (b)

Abbildung 5.12: (a) Binärbild und (b) Graustufenbild mit dessen gefärbten Zusammenhangskomponenten

einen generischen Ansatz zum Testen von Bildoperatoren aufzustellen: Die Grundlage des Ansatz sollen die allgemein anwendbaren Relationen R1–R4 sein. Zusätzlich soll jede zu testende Implementierung mit weiteren speziellen Relationen oder Spezialfällen von Eingaben untersucht werden. Die Motivation für diesen Ansatz ist, dass jede Testtechik für sich betrachtet nicht alle Fehler in der Implementierung entdeckt. Durch die Kombination von allgemeinen Relationen und speziell auf die jeweilige Implementierung zugeschnittene Tests soll die Effektivität der Tests verbessert werden.

Im Folgenden werden deshalb für die vorgestellten Verfahren zur Bildanalyse weitere Relationen oder Spezialfälle vorgestellt, um diesen generischen Ansatz zu vervollständigen.

Testkriterien für die (Euklidische) Distanztransformation

Zusätzlich zu den bereits definierten Relationen kann man für die Euklidische Distanztransformation weitere Metamorphische Relationen definieren. Im Folgenden ist A ein beliebiges Binärbild. Dann gilt:

R5 *Vergrößerung des Bildes*
Sei $D(A) = (d_{i,j})$ und $D(E(A)) = (e_{i,j})$. Dann gilt $3d_{i,j} = e_{3i+2,3j+2}$ für jedes

$0 \leq i < n_y$ und $0 \leq j < n_x$.

Der Vergrößerungsoperator $E(\cdot)$ macht nur zur Verwendung in R5 wirklich Sinn. Er vergrößert den Abstand zum nächsten weißen Pixel genau um den Faktor 3. Dadurch kann ein einfacher Zusammenhang zwischen den Ausgaben für A und $E(A)$ hergestellt werden, der leicht (pixelweise) überprüft werden kann.

R6 *Schnitt zweier Bilder*
$D(A \cap B) = \min(D(A), D(B))$

Beweis:
Sei $D(A) = (a_{i,j})$, $D(B) = (b_{i,j})$, und $D(A \cap B) = (d_{i,j})$. Angenommen, es gilt dass $d_{i,j} > \min(a_{i,j}, b_{i,j})$ für mindestens ein (i, j) und dass o.B.d.A. $a_{i,j} = \min(a_{i,j}, b_{i,j})$. Dann muss ein weißes Pixel in A mit Abstand $a_{i,j}$ zum Pixel (i, j) existieren. Dieses Pixel an der Stelle (i, j) ist in $A \cap B$ ebenfalls weiß, nach der Definition des Schnitts zweier Bilder. Also gilt $d_{i,j} \leq a_{i,j}$, was einen Widerspruch zur Annahme darstellt. Angenommen, es gilt dass $d_{i,j} < \min(a_{i,j}, b_{i,j})$ für mindestens ein (i, j), dann muss ein weißes Pixel mit Abstand $d_{i,j}$ zum Pixel (i, j) in $A \cap B$ existieren. Das Pixel an dieser Stelle in A muss nun ebenfalls weiß sein. Also muss $a_{i,j} \leq d_{i,j}$ gelten, was wiederum ein Widerspuch zur Annahme ist. Daraus folgt die Behauptung.

R7 *Vereinigung zweier Bilder*
$D(A \cup B) \geq \max(D(A), D(B))$

Beweis:
Um R7 zu zeigen wird wie folgt vorgegangen: Sei $D(A) = (a_{i,j})$, $D(B) = (b_{i,j})$, und $D(A \cap B) = (d_{i,j})$. Angenommen, dass $d_{i,j} < \max(a_{i,j}, b_{i,j})$, und (o.B.d.A.) $a_{i,j} = \max(a_{i,j}, b_{i,j})$ ist. In diesem Fall muss ein weißes Pixel in $A \cap B$ existieren, dessen Abstand zum Pixel (i, j) $d_{i,j}$ beträgt. Das Pixel an dieser Stelle in A und B muss dann nach der Definition der Vereinigung zweier Bilder ebenfalls weiß sein. Also gilt $a_{i,j} \leq d_{i,j}$, was ein Widerspruch zur obigen Annahme ist. Offensichtlich kann die Gleichheit bei R7 nicht gelten. Sei B das Komplement von A, dann ist $d_{i,j} = \infty$ für alle Pixel von $D(A \cap B)$.

R8 *untere Schranke*
$D(A) \geq A$

Die Gültigkeit der Relation R8 ist offensichtlich, da die Distanztransformation

jedem weißen Pixel den Wert 0 zuweist (er bleibt also weiß) und jedem schwarzen Pixel eine Wert größer oder gleich 1.

Die Relationen R5 – R7 haben allerdings den Nachteil, dass sie nicht nur für die Euklidische Distanztransformation, sondern auch für beliebige andere Distanztransformationen gültig sind. Die folgenden Relationen R8 und R9 werden benötigt, um die Euklidische Distanztransformation von anderen Distanztransformationen unterscheiden zu können.

R9 *lokale Schranke*

Sei $D(A) = d(i,j)$. Dann gilt: $|d_{i,j} - d_{i+k-1,j+l-1}| \leq b_{k,l}$ für alle $k,l \in \{0,1,2\}$ und jedes innere Pixel mit $1 \leq i < n_y - 1$ und $1 \leq j < n_x - 1$, wobei

$$B = (b_{k,l}) = \begin{pmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & 0 & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{pmatrix}.$$

Die Relation R9 ergibt sich aus der Dreiecksungleichung und der Definition der Euklidischen Distanz. Die Differenz der Werte der Distanztransformation an einem Pixel und dessen umgebenden Pixeln in einem 3×3-Quadrat kann durch die Werte in B abgeschätzt werden. Da allerdings die Distanzen bezüglich der Maximums-Distanz immer größer oder gleich denen bezüglich der Euklidischen Distanz sind, kann R9 diese beiden Distanztransformationen nicht voneinander unterscheiden.

Die Relationen R8 und R9 haben eine sehr einfache Struktur. Deshalb wird angenommen, dass nur wenige Fehler durch diese Relationen entdeckt werden. Da die Relationen aber bestimmte Eigenschaften der Euklidischen Distanztransformation überprüfen, wird angenommen, dass sie in Kombination mit anderen Relationen durchaus die Effektivität der Tests verbessern können.

S1 *spezielle Eingaben*

Zusätzlich zu den beschriebenen Metamorphischen Relationen kann die Euklidische Distanztransformation mit einem alternativen Ansatz automatisiert getestet werden. Sind die weißen Pixel in einem Bild bekannt, so kann die Distanztransformation nach der Definition berechnet werden. Allerdings hängt die Laufzeit dann sehr stark von der Anzahl der weißen Pixel ab, da für jedes schwarze Pixel die minimale Distanz zu allen weißen Pixeln bestimmt werden muss. Erzeugt man jedoch

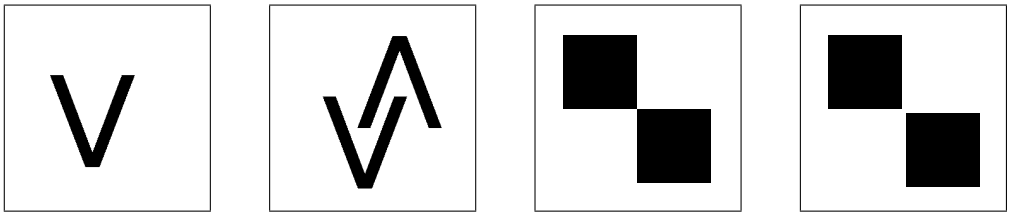


Abbildung 5.13: Spezielle Bilder zum Testen des Färbens von Zusammenhangskomponenten

ein Binärbild mit nur wenigen weißen Pixeln und speichert deren Position, so kann die Ausgabe der SUT einfach überprüft werden. Genau genommen handelt es sich in diesem Fall um ein automatisierbares *Solved example*-Oracle (siehe [12]).

Testkriterien für das Färben von Zusammenhangskomponenten

Für das Testen des Algorithmus' zum Färben von Zusammenhangskomponenten war es nicht möglich, weitere Metamorphische Relationen zu bestimmen. Stattdessen wurden von Hand spezielle Bilder erzeugt, um mit diesen als Testeingaben besondere Fehlerfälle abzudecken. Abb. 5.13 zeigt diese Bilder. Für alle ist es sehr einfach, die Zusammenhangskomponenten zu identifizieren und gefärbte Bilder zu erstellen.

Der untersuchte Algorithmus tastet das Bild zweimal ab. Der erste Durchlauf läuft von oben nach unten durch das Bild, der zweite von unten nach oben. Das Bild S2 dient nun dazu, zu testen, ob der erste Durchlauf die beiden zuerst erkannten Komponenten (die Spitzen des „V“) dann tatsächlich zu einer einzigen Zusammenhangskomponente zusammensetzt. Das Bild S3 überprüft zusätzlich den zweiten Durchlauf des Algorithmus.

Die Bilder S4 und S5 sollen die Korrektheit der verwendeten Nachbarschafts-Definition prüfen. Die beiden Blöcke in S4 sind nach der 8-Nachbarschaft miteinander verbunden, es gibt also nur eine Zusammenhangskomponente. Bezüglich der 4-Nachbarschaft sind diese Blöcke aber nicht verbunden. Verwendet der Algorithmus also eine andere Definition von Zusammenhang, wird der Algorithmus zwei Zusammenhangskomponenten identifizieren. Im Bild S5 sollten dagegen immer zwei Zusammenhangskomponenten identifiziert werden.

5.2.5 Empirische Studie

In der im Folgenden präsentierten empirischen Studie sollen die beschriebenen Algorithmen mit Hilfe der dafür entwickelten Methoden getestet werden. Von besonderem Interesse sind dabei zwei Fragestellungen. Wie weit beeinflussen die gewählten Methoden zur Erzeugung der Eingaben (Bilder) die Testergebnisse? Was kann über die Effektivität der eingesetzen Orakel bzw. Testmethode gesagt werden?

Für die Studie wurden Implementierungen der Algorithmen aus unterschiedlichen Quellen verwendet.

1. **EuclidDT**: Die Implementierung wurde der GeoStoch-Bibliothek [109] entnommen und besteht aus 301 LOC. Der verwendete Algorithmus wurde in [30] beschrieben.

2. **ConnectedC8**: Das zweite Studienobjekt, eine Implementierung zur Färbung von Zusammenhangskomponenten, stammt aus der Bibliothek Image Processing API, die am Ostfold College [96] entwickelt wurde. Diese Implementierung besteht aus 215 LOC. In der Studie wurde der „klassische“ Algorithmus mit 8-Nachbarschaft verwendet (siehe [93, 101]).

3. **Lipschitz**: Die Implementierung des Lipschitz-Filters [105] ist ein Plugin für die Bildbearbeitungs-Software ImageJ [88]. Sie besteht aus 320 LOC. In der Studie wurden die Parameter
   `m_Down=false`, `m_TopHat=false` und `m_Slope=10` verwendet. Da die Berechnungen für andere Parameterwerte sehr ähnlich sind, sollte dies keine Einschränkung für die Studie darstellen.

Aus den Implementierungen wurden mit Hilfe von MuJava Mutanten erstellt. Tabelle 5.2 gibt eine Übersicht über die Anzahl der erzeugten Mutanten und die Anzahl der äquivalenten Mutanten. Die zur Originalimplementierung äquivalenten Mutanten wurden durch manuelle Inspektion des Quellcodes bestimmt.

Untersuchung der Einflüsse der Eingabeerzeugung

Im Folgenden werden für jede Implementierung die Einflüsse der Eingabeerzeugung untersucht. Dazu werden für unterschiedliche Parameterwerte mit Hilfe der beschriebenen

Implementierung	Mutanten insg.	äquivalent
EuclidDT	1334	93
ConnectedC8	549	54
Lipschitz	1041	101
insgesamt	2924	248

Tabelle 5.2: Übersicht der Mutanten der Bildoperatoren

Verfahren zur Eingabeerzeugung Bilder erzeugt. Die daraus resultierenden Ausgaben der Mutanten werden mit den Ausgaben des Originalprogramms verglichen.

EuclidDT
Um die Parameter der Verfahren optimal zu wählen, wurden sowohl die Bildgröße als auch der Anteil der schwarzen Pixel im Bild variiert. Um die Wahl zu vereinfachen wurden nur quadratische Bilder betrachtet. Als Seitenlängen wurden 10, 20, 30, 40, 50 und 60 Pixel gewählt. Der Anteil der schwarzen Pixel wurde sowohl für das Boolesche Modell als auch für den *random binary*-Ansatz auf 10%, 20%, 50%, 80%, 90% gesetzt, für den *random binary*-Ansatz zusätzlich auf 99%. Im Booleschen Modell wurden Kreise mit Radius 1 Zehntel der Seitenlänge des Bilds als Körner verwendet. Für jeden Mutanten wurden durch jedes Verfahren 1000 Eingaben erzeugt. Wichen die Ausgaben für dasselbe Bild von Mutant und Originalimplementierung voneinander ab, wurde der Mutant als getötet klassifiziert.

Tabelle 5.3 zeigt die Anzahl der durch Vergleich mit der Originalimplementierung getöteten Mutanten. Die Eingaben wurden durch das Boolesche Modell erzeugt, Spalten und Zeilen geben die unterschiedlichen Parameterwerte an. Wird die Bildgröße 60×60 und ein Anteil von 90% schwarzer Pixel im Bild gewählt, so werden 1222 Mutanten getötet. Für eine Bildgröße von 30×30 Pixeln werden bereits 1221 Mutanten getötet. Berücksichtigt man zudem die von der Bildgröße abhängige Laufzeit der Tests, ist die Wahl Bildgröße 30×30 Pixel und 90% Anteil schwarzer Pixel im Bild die effizienteste Wahl, da die Laufzeit des Algorithmus quadratisch von der Bildgröße abhängt.

Tabelle 5.4 zeigt die Ergebnisse bei Verwendung des *random binary*-Ansatzes zur Eingabeerzeugung. Auch hier können bestenfalls 1221 Mutanten getötet werden, allerdings ist dazu ein Anteil schwarzer Pixel von 99% und eine Bildgröße von 40×40 Pixeln notwendig. Eine Bildgröße von 30×30 sollte aber auch bei dieser Art der Eingabegenerierung ausreichend sein, da nur ein Mutant weniger getötet wird und die Reduktion

	Anteil schwarzer Pixel im Bild				
Bildgröße	10%	20%	50%	80%	90%
10	1105	1136	1148	1162	1165
20	1137	1143	1159	1214	1218
30	1154	1158	1214	**1221**	1221
40	1206	1214	1218	1221	1221
50	1215	1219	1221	1221	1221
60	1219	1220	1221	1221	1222

Tabelle 5.3: Anzahl der durch Vergleich mit der Originalimplementierung getöteten Mutanten (Bild-Erzeugung durch Boolesches Modell, nicht-äquivalent: 1241 Mutanten)

der Bildgröße auf 30×30 die Laufzeit der Tests etwa halbiert.

	Anteil schwarzer Pixel im Bild					
Bildgröße	10%	20%	50%	80%	90%	99%
10	917	1068	1132	1151	1158	1164
20	1004	1091	1139	1156	1191	1215
30	1033	1098	1139	1155	1208	**1220**
40	1050	1096	1139	1155	1212	1221
50	1057	1101	1138	1159	1214	1221
60	1065	1097	1139	1162	1217	1221

Tabelle 5.4: Anzahl der durch Vergleich mit der Originalimplementierung getöteten Mutanten (Bild-Erzeugung durch *random binary*-Ansatz, nicht-äquivalent: 1241 Mutanten)

Der alternative Testansatz S1 wurde separat untersucht, da dieser eine eigene Methode zur Erzeugung der Eingaben erfordert. Für diesen Testansatz stellte sich ebenfalls eine Bildgröße von 30×30 Pixel als optimal heraus. Bei der Anzahl der zufällig im (schwarzen) Bild plazierten weißen Pixel ist $k = 15$ als optimal anzusehen.

ConnectedC8

Für die Optimierung der Parameterwahl zum Testen von ConnectedC8 wurde das gleiche Vorgehen gewählt wie bei EuclidDT. Allerdings ist zum Vergleich der Ausgaben ein pixelweiser Vergleich nicht zielführend, da nicht garantiert ist, dass eine Zusammenhangskomponente in unterschiedlichen Programmaufrufen mit derselben Graustufe gefärbt wird. Deshalb wurde ein Isomorphismus zum Vergleich von zwei Ausgaben

angewendet. Dabei wird geprüft, ob die Pixel, welche das Originalprogramm zu einer Zusammenhangskomponente zuordnet, in der Ausgabe des Mutanten ebenfalls zu einer einzigen Zusammenhangskomponente gehören, also in derselben Graustufe gefärbt sind, welche aber nicht dieselbe Graustufe wie in der Ausgabe des Originalprogramms sein muss.

Sei also $A = (a_{i,j}) \subset \mathcal{R}^{n_y \times n_x}$ die Ausgabe des Originalprogramms und $B = (b_{i,j}) \subset \mathcal{R}^{n_y \times n_x}$ die Ausgabe eines Mutanten. Dann gilt $A \cong B$ genau dann, wenn

$$a_{i,j} = a_{l,k} \Leftrightarrow b_{i,j} = b_{l,k}$$

mit $1 \leq i, l \leq n_y$ und $1 \leq j, k \leq n_x$ gilt.

Wird das Boolesche Modell zur Eingabeerzeugung verwendet, wird die maximale Anzahl an getöteten Mutanten bereits bei einer Bildgröße von 10×10 Pixeln bei einem Anteil schwarzer Pixel von 50% erreicht, insgesamt 473 Mutanten konnten so getötet werden (siehe Tabelle 5.5). Der *random binary*-Ansatz zeigt sich hier als einfacher handhabbar, da eine Vielzahl von Kombinationen aus Bildgröße und Schwarz-Anteil zu dieser maximalen Anzahl an getöteten Mutanten führt (siehe Tabelle 5.6).

	Anteil schwarzer Pixel im Bild					
Bildgröße	10%	20%	50%	80%	90%	99%
10	469	471	**473**	471	469	398
20	470	471	473	472	471	456
30	470	471	473	472	471	446
40	470	471	473	472	471	463
50	470	471	473	472	471	460
60	470	471	473	473	471	463

Tabelle 5.5: Anzahl der von ConnectedC8 abgeleiteten und durch Vergleich mit der Originalimplementierung getöteten Mutanten (Bild-Erzeugung durch Boolesches Modell, nicht-äquivalent: 495 Mutanten)

Lipschitz

Tabelle 5.7 zeigt die Anzahl der getöteten Mutanten für Lipschitz. Für die Versuche wurden Bilder der Größe 30×30 Pixel verwendet. Die erste Zeile zeigt die Anzahl der getöteten Mutanten bei Verwendung des random grayscale-Ansatzes zur Eingabeerzeugung mit Anzahl der Graustufen $g_{\max} = 25$. Benutzt man das Boolesche Modell zur

	Anteil schwarzer Pixel im Bild					
Bildgröße	10%	20%	50%	80%	90%	99%
10	468	**473**	473	472	465	403
20	473	473	473	472	468	404
30	473	473	473	472	466	407
40	473	473	473	471	467	406
50	473	473	473	471	467	414
60	473	473	473	471	466	414

Tabelle 5.6: Anzahl der von ConnectedC8 abgeleiteten und durch Vergleich mit der Originalimplementierung getöteten Mutanten (Bild-Erzeugung durch *random binary*, nicht-äquivalent: 495 Mutanten)

Erzeugung der Eingaben, so hängt die Anzahl der getöteten Mutanten kaum vom Parameter p ab. In den weiteren Untersuchungen wurde deshalb $p = 10\%$ gewählt. Mit dieser Wahl der Parameter konnten 864 von 940 Mutanten getötet werden.

p	Boolesches Model (Graustufe)	Random Grayscale
–	–	**857**
10%	**864**	–
25%	864	–
50%	864	–
75%	861	–
90%	857	–

Tabelle 5.7: Lipschitz: Anzahl der getöteten Mutanten (insg. nicht-äquivalent: 940 Mutanten)

Untersuchung der Orakel

Im Folgenden wird nun die Effektivität der in Kapitel 5.2.4 vorgeschlagenen Orakel untersucht. Zur Erzeugung der Eingaben werden die im vorherigen Abschnitt ermittelten Parameterwerte verwendet, um eine optimale Effektivität der Orakel zu erreichen.

EuclidDT

Wie im vorherigen Kapitel beschrieben, wurde für beide Eingabemodelle eine Bildgröße von 30×30 Pixeln gewählt. Für das Boolesche Modell wurde zudem der Parameter $p = 90\%$, für *random binary* $p = 99\%$ gewählt. Tabelle 5.8 zeigt die Anzahl der durch

Orakel	Boolesches Modell	Random Binary
R1	1211 (744)	1210 (744)
R2	1163 (743)	1157 (743)
R3	1192 (742)	1190 (741)
R4	1208 (739)	1202 (741)
R5	1182 (739)	1177 (738)
R6	1097 (741)	1081 (738)
R7	1061 (738)	990 (741)
R8	752 (741)	752 (740)
R9	1079 (732)	1075 (737)
S1	1218 (739)	

Tabelle 5.8: Anzahl der durch die vorgeschlagenen Orakel getöteten Mutanten (insg. nicht-äquivalent: 1241 Mutanten)

die einzelnen Orakel getöteten Mutanten. Da S1 ein eigenes Modell zur Erzeugung der Eingaben beinhaltet, sind die Ergebnisse für S1 in der Tabelle leicht abgesetzt aufgeführt. In Klammern ist jeweils die Anzahl der Mutanten angegeben, bei deren Test ein Laufzeitfehler (Exception) aufgetreten ist. Um diese Mutanten als fehlerhaft zu identifizieren wird kein komplexes Orakel benötigt, ein einfacher Smoketest reicht dazu aus.

Bei der Durchführung der Tests wurde jede Metamorphische Relation maximal 1000 Mal bei jedem Mutanten überprüft. Sobald die Relation einmal nicht erfüllt wurde, wurde der Test abgebrochen und der Mutant als getötet gezählt.

Die Relation R1 stellt sich in dieser Untersuchung als die effizienteste Entscheidungsregel heraus, dicht gefolgt von R4. Auch die Relationen R2, R3 und R5 liefern noch akzeptable Ergebnisse. Die Relationen R6, R7, R8 und R9 sind jedoch weniger geeignet, um Fehler in den getesteten Implementierungen zu erkennen. Am erfolgreichsten ist jedoch der Ansatz S1, aber auch mit diesem gelingt es nicht, alle Mutanten, die durch Vergleich mit der Originalimplementierung getötet wurden (1222 Mutanten), als fehlerhaft zu erkennen.

Um die Erkennungsrate weiter zu verbessern, wurden die Ergebnisse der einzelnen Relationen miteinander kombiniert. Dazu wurden die Ergebnisse von jeweils zwei Relationen bzw. von S1 miteinander verglichen. Ein Mutant wurde dann als getötet markiert, wenn er von mindestens einer der Entscheidungsregeln als getötet markiert wurde. Diese Art der Kombination ist also keine „echte“ Kombination der Entscheidungsregeln, sondern wird aus den Einzelergebnissen erzeugt. Das hat den Vorteil, dass diese Kombi-

	R1	R2	R3	R4	R5	R6	R7	R8	R9	S1
R1	1211	1212	1212	1211	1212	1212	1212	1215	1212	**1221**
R2		1163	1207	1212	1199	1166	1165	1168	1195	1220
R3			1192	1212	1202	1193	1193	1197	1197	1220
R4				1208	1211	1211	1211	1212	1209	1220
R5					1182	1186	1185	1186	1186	1220
R6						1097	1101	1104	1159	1218
R7							1061	1068	1141	1218
R8								752	1090	1218
R9									1079	1218
S1										1218

Tabelle 5.9: Anzahl der durch paarweise Kombination von Relationen getöteten Mutanten (EuclidDT)

nationen im Nachhinein aus vorhandenen Ergebnissen erzeugt werden können, ohne die Tests erneut durchführen zu müssen.

Tabelle 5.9 zeigt die Ergebnisse dieser Kombinationen. Dabei zeigt sich, dass die Anzahl der getöteten Mutanten teilweise erheblich steigt. Besonders die Kombination von für sich genommen wenig effizienten Relationen verbessert die Ergebnisse. Kombiniert man die Relation R1 mit S1, so erreicht man dieselbe Anzahl an getöteten Mutanten wie durch Vergleich mit der Originalimplementierung. Die Kombination von R2 – R5 mit S1 tötet 1220 Mutanten. Damit töten diese Relationen ebenfalls Mutanten, die durch S1 nicht getötet wurden.

ConnectedC8

Für die Untersuchung der Mutanten von ConnectedC8 wurden im vorherigen Kapitel die Parameter Bildgröße 10×10 Pixel und p=50% ermittelt. Tabelle 5.10 zeigt die Anzahl der durch die Relationen R1 – R4 und die manuell erzeugten Eingaben S2 – S5 getöteten Mutanten. Wieder ist R1 die für sich alleine genommen effizienteste Entscheidungsregel, besser sogar als die Tests, welche auf manuell erzeugten Eingaben und Referenzausgaben beruhen. Wie in der Untersuchung von EuclidDT zeigt sich jedoch auch hier, dass die maximal mögliche Anzahl an getöteten Mutanten nur durch die Kombination mehrerer (genauer gesagt zweier) Entscheidungsregeln, R1 und S3, erreicht wird. Besonders hervorzuheben ist die Kombination von R1 und S5. Auch hier wird die maximal mögliche Anzahl von 473 getöteten Mutanten erreicht, obwohl der Test mit S5 alleine „nur“ 293

	R1	R2	R3	R4	S2	S3	S4	S5
R1	459	459	459	459	465	**473**	465	**473**
R2		448	454	459	457	470	457	470
R3			450	459	457	469	457	469
R4				452	458	466	460	466
S2					365	416	374	413
S3						415	420	415
S4							256	310
S5								293

Tabelle 5.10: Anzahl der durch paarweise Kombination von Relationen getöteten Mutanten (ConnectedC8)

	S2	S3	S4	S5
R1	375	421	385	293
R2	365	415	385	295
R3	367	415	385	293
R4	372	424	256	295

Tabelle 5.11: Anzahl der Mutanten von ConnectedC8, die durch S2–S5 und deren „follow-ups“ getötet wurden

Mutanten tötet.

Bei manuell erzeugten Eingaben bietet es sich an, diese als Eingaben für den Metamorphischen Test zu verwenden, also eine „echte“ Kombination beider Techniken vorzunehmen. Dabei werden ausgehend von S1–S5 durch die Eingaberelationen weitere Bilder erzeugt. Dann wird die Ausgaberelation verwendet, um den Test zu entscheiden. Tabelle 5.11 zeigt die Ergebnisse dieser Kombination. Dieses Vorgehen scheint nicht wirklich erfolgsversprechend zu sein, da nur relativ wenige Mutanten getötet werden. Einzig für S4 verbessern sich die Ergebnisse deutlich.

Lipschitz

Für die Untersuchung der Relationen R1 – R4 wurde die Bildgröße 10×10 Pixel gewählt. Tabelle 5.12 zeigt die Resultate der Untersuchung. Die Anzahl der Mutanten, die durch Werfen einer Exception oder durch Überschreiten eines Zeitlimits von 10 Sekunden bei der Testausführung getötet wurden, ist in Klammern angegeben.

Wie in den vorhergehenden Untersuchungen ist die Relation R1 die effizienteste, ge-

Relation	Boolesches Modell (Graustufe)	Random Grayscale
R1	825 (429)	814 (428)
R2	788 (428)	779 (428)
R3	793 (429)	783 (428)
R4	805 (430)	790 (427)

Tabelle 5.12: Anzahl der getöteten Mutanten (insg. nicht-äquivalent: 940 Mutanten)

folgt von R4, R3 und R2. Die Ergebnisse unter Verwendung des (Graustufen-) Booleschen Modells scheinen leicht besser zu sein, als die unter Verwendung des einfacheren random grayscale - Modells. Allerdings erlaubt die dünne Datenlage keine definitiven Schlüsse. Die Unterschiede sind so klein, dass sie evtl. durch geschicktere Wahl der Parameter ausgeglichen werden könnten.

Wie in den anderen Untersuchungen wurde ebenfalls eine Kombination der Einzelergebnisse vorgenommen, es wurden jedoch keine Verbesserungen damit erzielt. Dies ist jedoch nicht verwunderlich, da auch in den vorherigen Untersuchungen die Kombinationen dieser Relationen kaum besser als R1 abgeschnitten haben. Für nutzbringende Kombinationen sind offensichtlich weitere Relationen oder manuell ermittelte Spezialfälle wie S1–S5 notwendig.

Kapitel 6

Beurteilung von Metamorphischem Testen

Die empirischen Studien zur Untersuchung der Effektivität von Metamorphischem Testen im vorherigen Kapitel haben gezeigt, dass Metamorphisches Testen eine sehr effektive Testmethode sein kann. In diesem Kapitel werden deshalb aus den Ergebnissen der Studien Hinweise für die effektive Anwendung von Metamorphischem Testen präsentiert. Die Hinweise betreffen zum einen die Erzeugung der Testeingaben, zum anderen die Auswahl und Anwendung von Metamorphischen Relationen.

Anschließend wird als Beispiel für die praktische Anwendung von Metamorphischem Testen ein allgemeiner Ansatz zum Testen von Bildanalyse-Software vorgestellt.

6.1 Strategien zur Eingabeerzeugung

In den empirischen Studien wurden unterschiedliche Methoden zur automatischen Erzeugung von Testeingaben vorgestellt. Dabei wurden durchweg Methoden des Zufallstests verwendet.

Ein oft verwendetes Argument gegen die Verwendung des Zufallstests ist der Effekt, dass bestimmte Fehler praktisch nicht zu entdecken sind. Die Ursache dafür ist, dass die Ausführungspfade, die zu einem Fehlverhalten führen, nur bei sehr speziellen Eingaben ausgeführt werden. Die Wahrscheinlichkeit, dass genau diese Eingaben durch zufällige Generierung erzeugt werden, ist oft nahezu gleich null.

Die in der vorliegenden Arbeit verwendeten Strategien zur zufälligen Erzeugung er-

lauben fast durchweg eine Steuerung der Erzeugung durch einzelne Parameter. Deshalb wurde versucht, durch die Wahl einer optimalen Parametrisierung möglichst viele Fehler zu finden. In den Untersuchungen der Bildverarbeitungsalgorithmen war es so möglich, selbst im schlechtesten Fall mehr als 90 % der Fehler bzw. der fehlerhaften Mutanten auch als solche zu identifizieren. Im Falle von EuclidDT wurden nur 1.3% (20 Mutanten) der nicht-äquivalenten Mutanten nicht entdeckt. Bei ConnectedC8 und Lipschitz waren es 3.4% (15 Mutanten) bzw. 8.1% (76 Mutanten).

Interessanterweise spielt die Art der zufälligen Erzeugung bei richtiger Parametrisierung anscheinend keine Rolle. Die Wahl der Parametrisierung ist daher der entscheidende Faktor für einen erfolgreichen Einsatz des Zufallstests. In den Untersuchungen der Bildanalyseverfahren war diese jedoch oft intuitiv motivierbar. Beispielsweise beim Test der Euklidischen Distanztransformation war es notwendig, dass große Werte der Distanztransformation tatsächlich vorkommen. Deshalb ist ein großer Anteil schwarzer Pixel im Bild notwendig. Beim Färben von Zusammenhangskomponenten ist eine ausreichend große Anzahl dieser Zusammenhangskomponenten im Bild notwendig. Für zu kleine Werte von p sind es zu wenige. Mit steigendem p nimmt diese zu, ab einem bestimmten Wert von p aber wieder ab, weil Komponenten wieder verbunden werden. Die Wahl von $p = 50\%$ ist somit plausibel erklärbar.

In den durchgeführten Studien war der Zufallstest ein geeignetes Mittel zur automatischen Erzeugung von Testeingaben. Da aber im schlechtesten Fall ca. 10% der Mutanten nicht getötet werden konnten, sollte der Zufallstest nicht als einzige Methode zur Erzeugung von Testeingaben verwendet werden. Eine Möglichkeit ist es, die Programmteile zu identifizieren, die nur mit sehr geringer Wahrscheinlichkeit durch den Zufallstest erzeugt werden und dann die Eingaben zu bestimmen, die für die Ausführung dieser Teile sorgen. Dabei können z.B. Werkzeuge zur Messung von Überdeckungen (etwa Cobertura, siehe [27]) verwendet werden. Zur Entscheidung der Tests können dann entweder das Metamorphische Testen oder manuell erzeugte, erwartete Ausgaben verwendet werden.

6.2 Effektivität von Metamorphischem Testen

6.2.1 Kriterien für die Auswahl von Metamorphischen Relationen

Zuerst ist festzustellen, dass die Methode des Metamorphischen Testens an sich sehr gut funktioniert. Es wurden für jede der untersuchten Anwendungen Relationen gefunden,

die es ermöglichen, einen großen Anteil der Fehler tatsächlich zu entdecken. Allerdings ist die Effektivität der einzelnen Relationen sehr unterschiedlich. In der Studie zur Determinantenberechnung wurden Relationen gefunden, deren Effektivitätsmaß nahe bei 1 liegt. Diese Relationen sind also fast genauso gut wie das bestmögliche Orakel. Gleichzeitig konnten andere Relationen aber nur 30 %-40 % der fehlerhaften Mutanten töten. Das Kernproblem des Metamorphischen Testens ist es daher, geeignete Relationen zu finden. Es wäre also ein großer Fortschritt, die Effektivität einer Metamorphischen Relation ohne aufwändige Untersuchungen grob beurteilen zu können. Aus den Ergebnissen der Fallstudien in der vorliegenden Arbeit wurden vier einfache Kriterien hergeleitet, die das erlauben. Die Kriterien wurden zunächst aus der ersten Fallstudie abgeleitet und dann nach der zweiten Fallstudie weiter verfeinert. Vorläufige Resultate zu Auswahlkriterien für Metamorphische Relationen wurden bereits in [70] veröffentlicht.

1. *In der Implementierung verwendete Relationen*:
   In diese Klasse fallen Relationen, die von der Implementierung benutzt werden. Beispielsweise die Eigenschaft, dass sich die Determinante beim Vertauschen zweier Zeilen der Matrix nicht ändert, wird massiv vom Gauß-Algorithmus bei der Pivotierung verwendet.

   Ist die Implementierung nicht bekannt, kann der Tester natürlich nur Vermutungen über den verwendeten Algorithmus der Implementierung anstellen. Bei mathematisch beschriebenen Problemen existieren aber oft nur wenige unterschiedliche Algorithmen zur Lösung, was die Abschätzung der Effektivität der Relation eventuell doch ermöglicht.

   Effektivität der Relation: sehr schlecht

2. *Gleichheitsbeziehungen*:
   Relationen der Form $P(i_1) = P(i_2)$ sind nicht in der Lage Fehler zu entdecken, die aus einer additiven bzw. multiplikativen Konstante bestehen. Wird statt $P(i)$ der Wert $P' = c \cdot P(i) + a$ berechnet, ist es nicht möglich, mit einer solchen Relation den Fehler zu entdecken, da die Fehler beim Vergleich eliminiert werden:

   $$P'(i_1) \stackrel{?}{=} P'(i_2) \Rightarrow c \cdot P(i_2) + a \stackrel{?}{=} c \cdot P(i_2) + a \Rightarrow P(i_1) = P(i_2)$$

   Damit ist die Relation erfüllt, obwohl die Programmausgabe $P'(\cdot)$ nicht dem korrekten Ergebnis $P(\cdot)$ entspricht.

Effektivität der Relation: schlecht

3. *Gleichungen*:
   In diese Klasse fallen Relationen der Form

$$P(i_1) = f(P(i_2), \ldots, P(i_n)),$$

   mit $n \geq 2$ und einer Funktion $f : \mathcal{O}^{n-1} \rightarrow \mathcal{O}$. Die Effektivität von Relationen dieser Art sind nur schwer einzuschätzen. Ist $f(\cdot)$ linear, so hat die Relation ebenfalls Schwierigkeiten, Fehler durch multiplikative Konstanten zu erkennen. Die Effektivität ist aber üblicherweise höher als bei Gleichheitsbeziehungen. Ist $f(\cdot)$ nichtlinear, scheint die Effektivität im Allgemeinen gut zu sein.

   Effektivität der Relation: gut

4. *Semantisch reiche Relationen*:
   Mit dieser Beschreibung sind Relationen gemeint, welche die Spezifikation so möglichst eindeutig beschreiben. Diese Relationen sind dadurch gekennzeichnet, dass keine oder nur sehr wenige andere Spezifikationen diese Relation ebenfalls erfüllen. Die Relation R12 in der Studie über die Determinanten-Berechnung fällt in diese Klasse.

   Effektivität der Relation: sehr gut

Es ist zu beachten, dass diese Kriterien nur eine Abschätzung der Effektivität der Relationen bieten. Die obige Klassifikation ist zudem eher konservativ. Es kann also durchaus vorkommen, dass eine Relation, die zu einer vermeintlich schlechten Kategorie gehört, trotzdem sehr effektiv ist. Aus den Fallstudien folgt beispielsweise, dass auch als schlecht klassifizierte Relationen bis zu 50% der fehlerhaften Mutanten töten können. Generell ist aber aus der Klassifikation abzuleiten, dass als schlecht klassifizierte Relationen nicht als einziger Test verwendet werden sollten. In Kombination mit anderen Relationen oder Testverfahren sind aber auch diese „schlechten“ Relationen in der praktischen Anwendung verwendbar.

Die Auswahl einer einzelnen Relation oder weniger Relationen bietet sich vor allem an, wenn nur begrenzte Ressourcen für die Tests zur Verfügung stehen, da auf diese Weise die knappen Ressourcen für die potentiell effektivsten Tests verwendet werden können. Sind jedoch genügend Ressourcen vorhanden, um alle bekannten Relationen zu

überprüfen, bietet sich die Kombination der Relationen als Methode zur Steigerung der Effektivität an.

6.2.2 Kombinationen von Metamorphischen Relationen

In den Fallstudien zur Untersuchung Metamorphischer Relationen wurde auch die Kombination von Relationen als Mittel zur Steigerung der Effektivität des Metamorphischen Testens vorgestellt. Die Kombination erfolgt durch logische Verknüpfung der Ergebnisse der Tests. Ein kombinierter (metamorphischer) Test schlägt genau dann fehl, wenn mindestens eine der an der Kombination beteiligten Relationen nicht erfüllt ist. Im Kontext der Mutationsanalyse bedeutet dies, dass ein Mutant genau dann von einer Kombination von Relationen $R_1, \ldots, R_n$ getötet wurde, wenn der Mutant von mindestens einer der Relationen $R_1, \ldots, R_n$ getötet wurde.

Das Testergebnis der Kombination beruht also auf den Testergebnissen der einzelnen Relationen. Die Kombination kann auf unterschiedliche Arten gebildet werden. Die erste Möglichkeit ist, die beteiligten Relationen nacheinander zu überprüfen und den Test abzubrechen, sobald eine Relation nicht erfüllt wurde. Werden effektive Relationen zuerst überprüft, kann oft auf eine Überprüfung von anderen Relationen verzichtet werden. Das dürfte die für die Tests benötigte Zeit deutlich reduzieren. Die zweite Möglichkeit ist, alle Relationen zu überprüfen, und dann anschließend die Kombination aus den Ergebnissen der einzelnen Relationen zu bilden. Dieser Ansatz hat den Vorteil, dass die Relationen parallel, also quasi gleichzeitig überprüft werden können. Wenn die Effektivität der verwendeten Relationen nicht abgeschätzt werden kann und deswegen alle Relationen überprüft werden müssen, kann durch die Parallelisierung der Tests erheblich Zeit gespart werden.

Generell sollte darauf geachtet werden, möglichst viele und strukturell unterschiedliche Relationen zu verwenden, da die Effektivitätskriterien nur grobe Richtlinien sind und deshalb nicht genau prognostiziert werden kann, welche Relationen wirklich effektiv sind. Zudem ist es sinnvoll, Metamorphisches Testen mit anderen Testmethoden zu kombinieren, wie beispielsweise beim Test der Distanztransformation mit dem Ansatz S1 geschehen.

6.3 Automatisiertes Testen von Bildverarbeitungssoftware

In der in Kapitel 5.2 vorgestellten Fallstudie wurden drei unterschiedliche Implementierungen von Algorithmen zur Bildanalyse bzw. -verarbeitung untersucht. Ursprünglich diente auch diese Studie zur Beurteilung der Effektivität von Metamorphischem Testen. Im Laufe der Untersuchung stellte sich jedoch heraus, dass immer dieselben vier Relationen $R1$ - $R4$ am effektivsten waren. Allerdings waren diese vier Relationen in keinem der untersuchten Anwendungsfälle in der Lage, alle fehlerhaften Mutanten zu töten.

Aus diesem Grund wurden zusätzlich zu den allgemeingültigen Metamorphischen Relationen weitere Relationen und Testtechniken verwendet, um die Effektivität der Tests zu verbessern. Im Falle der Distanztransformation wurden dazu eine besondere Art der Eingabeerzeugung und eine damit verbundene Testauswertung verwendet, im Falle der Färbung der Zusammenhangskomponenten wurden manuell erstellte Testfälle verwendet. Bei den zusätzlich zum Metamorphischen Testen gewählten Testtechniken handelt es sich immer um sog. *Testen mit Spezialfällen* (siehe [50, 54]). Dabei werden Eingaben verwendet, für die es besonders einfach ist, die tatsächlichen Ausgaben des SUT zu überprüfen. In beiden untersuchten Implementierungen konnten dann durch die Kombination der Testtechniken alle Mutanten getötet werden, die schon durch den Vergleich mit der Originalimplementierung als fehlerhaft klassifiziert wurden.

Die Ergebnisse der Studie führen zu folgendem allgemeinen Ansatz zum automatisierten Testen von Bildanalyse-Software. Zum Testen der Implementierung wird eine Kombination aus Metamorphischem Testen mit den in Kapitel 5.2 beschriebenen Relationen $R1$ - $R4$ und Testen mit Spezialfällen vorgeschlagen. Die initialen Eingaben werden durch die ebenfalls in Kapitel 5.2 beschriebenen Methoden zur Erzeugung zufälliger Bilder generiert. Bei der Verwendung dieser Methoden ist auf die richtige Parametrisierung der Generatoren zu achten. Gegebenenfalls sind Experimente zum Optimieren der Parametrisierung notwendig.

In der präsentierten Studie waren nur wenige Spezialfälle notwendig, um die Effektivität der Methode zu verbessern. Dadurch wird selbst dann, wenn die erwarteten Ausgaben der Spezialfälle manuell erstellt werden müssen, nur wenig Zeit für die Erstellung benötigt.

Der vorgeschlagene Ansatz hat gegenüber dem bisher üblichen rein manuellen Testen von Bildanalyse-Software den klaren Vorteil, dass die Testentscheidung aufgrund objektiver Kriterien erfolgt, und nicht durch visuelle Inspektion der Ausgaben des SUT.

Das Metamorphische Testen kann vollständig automatisiert werden und benötigt damit nach der Programmierung der Tests bei wiederholtem Testen kaum Zeit für die Testdurchführung. Beim Testen mit Spezialfällen kann zumindest die Testausführung und die Testauswertung ebenfalls automatisiert werden. Insgesamt erhält man durch den vorgeschlagenen Ansatz also einen nahezu vollständig automatisierten und damit reproduzierbaren Test von Bildanalysesoftware.

Kapitel 7

Empirische Untersuchungen statistischer Methoden im Softwaretest

7.1 Testentscheidung durch statistische Tests

Obwohl statistische Hypothesentest oft zum Treffen von Testentscheidungen vorgeschlagen werden, fehlten bisher systematische Untersuchungen zur Effektivität dieser Testmethode. Neben der formalen Beschreibung und Begründung der dazu verwendeten Vorgehensweise ist deshalb eine solche fundierte Untersuchung ein wichtiger Bestandteil der vorliegenden Arbeit. Die Ergebnisse der folgenden Fallstudien wurden vorab bereits in [49] veröffentlicht.

Die Fallstudie besteht aus zwei Teilen. Im ersten Teil sollen die in Kapitel 3.2 vorgeschlagenen Methoden erprobt werden. Dabei kommt zunächst ein wenig realitätsnaher Untersuchungsgegenstand zum Einsatz. Dieser wurde aber gewählt, um möglichst wenig störende Einflüsse auf die Untersuchungen zu haben. Im zweiten Teil sollen dann die vorgeschlagenen Methoden auf ein realitätsnahes Studienobjekt angewandt werden. Damit soll die praktische Anwendbarkeit des Vorgehens gezeigt werden.

Das erste Studienobjekt implementiert die inverse Verteilungsfunktion der Standard-Normalverteilung. Der Quellcode stammt wie schon eine Implementierung der Determinantenberechnung aus der Commons.Math-Bibliothek des Apache Jakarta-Projekts (siehe [4]). Diese Bibliothek steht als Open Source zur Verfügung. Die zweite Implemen-

tierung simuliert Tessellationen (siehe Kapitel 3.3.1) und wurde der Geostoch-Bibliothek (siehe [109]) entnommen.

7.1.1 Studie 1: Normalverteilung

Die Implementierung der inversen Verteilungsfunktion der Standard-Normalverteilung Φ^{-1} ist eine deterministische Funktion, erst durch zufällige Eingaben wird aus dem SUT randomisierte Software. Zum Testen wird die Eigenschaft

$$X \sim \mathcal{U}(0,1) \Rightarrow P = \Phi^{-1}(X) \sim \mathcal{N}(0,1)$$

ausgenutzt. Wenn eine auf dem Intervall $(0,1)$ gleichverteilte Zufallsvariable als Eingabe benutzt wird, ist die Ausgabe des SUT eine standard-normalverteilte Zufallsvariable.

Für den Test auf Normalverteilung stehen natürlich auch Tests zur Verfügung, welche die oben formulierte Nullhypothese direkt prüfen, beispielsweise der Shapiro-Wilk-Test (siehe [15]. Bei diesen Tests kann aber der β-Fehler nicht analytisch abgeschätzt werden. Deshalb ist nicht sicher, ob die propagierte Methode der Wahrscheinlichkeitsverstärkung anwendbar ist. Falls ein solcher Test angewandt werden sollte, müsste erst durch Simulationen überprüft werden, ob $\beta < \frac{1}{2}$ gilt.

Die Implementierung von Φ^{-1} besteht aus 800 LOC Code. Da Φ^{-1} nicht analytisch berechnet werden kann, muss Φ^{-1} näherungsweise berechnet werden. Die Implementierung orientiert sich dabei an dem in [80] vorgestellten Verfahren. Aus der Implementierung wurden mit MuJava insgesamt 1280 Mutanten erzeugt. Bei der Anwendung von statistischen Verfahren wird ein Mutant dann als getötet markiert, wenn das in Kapitel 3.2.6 zusammengefasste Vorgehen den Mutanten als fehlerhaft bewertet. Ob diese Bewertung auch zutrifft, ist ein Gegenstand der folgenden Untersuchung.

Die Implementierung ist eine deterministische Funktion. Deshalb kann das normale *gold standard oracle* mit der Originalimplementierung als Referenzimplementierung angewendet werden, um festzustellen welche Mutanten zum Originalprogramm äquivalent sind. Aus diesem Grund wurden die Mutanten nicht von Hand auf Äquivalenz zum Originalprogramm untersucht. Statt dessen wurde empirisch überprüft, ob ein Mutant für eine bestimmte Anzahl an Eingaben, hier 1000, dieselbe Ausgabe wie das Originalprogramm berechnet.

Für diese Untersuchung wurden zwei unterschiedliche Methoden zur Eingabeerzeu-

gung verwendet. Zum einen wurden die Eingaben zufällig erzeugt. Dabei wurde die Gleichverteilung auf dem Intervall $[0, 1]$ verwendet. Zum anderen wurden die Eingaben deterministisch gewählt, indem das Intervall $[0, 1]$ äquidistant zerlegt wurde, also $x_0 = 0$, $x_{n+1} = x_n + 10^{-4}$. Bei Verwendung von gleichverteilten Eingaben wurden 204 Mutanten als nicht-äquivalent klassifiziert. Bei Eingaben auf dem äquidistanten Gitter lieferten 208 Mutanten vom Original abweichende Ausgaben.

Die Untersuchung der SUT besteht aus zwei Teilen. Im ersten Teil wird die Effektivität der Tests gegen die theoretischen Referenzwerte für den Erwartungswert untersucht, im zweiten Teil werden die Tests gegen eine Alternativimplementierung durchgeführt. Im ersten Teil der Untersuchung wird, wie in Kapitel 3.2.6 beschrieben, ein Gauss-Test mit Nullhypothese $H_0 : \mathbb{E}(X) = 0$ verwendet. Da X normalverteilt ist, wird zudem ein Test mit χ^2-verteilter Prüfgröße mit Nullhypothese $H_0 : Var(X) = 1$ benutzt.

Bei der Untersuchung der Tests gegen theoretische Referenzwerte wurde $\alpha = \beta = \frac{1}{3}$ gewählt. Die Anzahl der Wiederholungen wurde auf $R = 501$ bzw. $k = 250$ festgesetzt.

Tabelle 7.1 zeigt die Ergebnisse der Tests gegen die theoretischen Referenzwerte. Die erste Spalte zeigt die durch den Test mit Nullhypothese $H_0 : \mathbb{E}(P) = 0$ getöteten Mutanten. Die zweite Spalte gibt die Anzahl der Mutanten an, die durch den Test mit Nullhypothese $Var(P) = 1$ getötet wurden. Die dritte Spalte zeigt die Anzahl der durch die Kombination der Tests getöteten Mutanten. Dort werden die Mutanten gezählt, die entweder durch den ersten oder durch den zweiten Test getötet wurden. Die letzte Spalte zeigt die empirisch bestimmte Genauigkeit der Tests. Für jede Testdurchführung wurde dazu δ_i nach (3.5) aus α, β und n bestimmt. Der Eintrag in der Tabelle ist dann gegeben durch $\delta_{max} = \max\{\delta_i,\, 1 \leq i \leq R\}$.

	#Mutanten, getötet durch Test mit H_0			
n	$\mathbb{E}(P) = 0$	$Var(P) = 1$	Kombination	δ_{max}
50	74	106	148	0.199
100	85	114	159	0.140
200	95	139	186	0.099
300	97	141	189	0.081
500	107	148	194	0.063
1000	111	150	194	0.044
1500	111	150	194	0.036

Tabelle 7.1: Ergebnisse für die Tests gegen theoretische Referenzwerte – Studie 1

Die Tests wurden mehrmals durchgeführt. Bei jeder Testdurchführung wurden dann unterschiedliche Stichprobenumfänge n von $n = 50$ bis $n = 1500$ verwendet. Wie zu erwarten erhöhte sich die Anzahl der getöteten Mutanten und die empirisch bestimmte Genauigkeit der Tests nimmt zu. Allerdings wird die maximale Anzahl an getöteten Mutanten schon für $n = 500$ erreicht. Eine weitere Erhöhung des Stichprobenumfangs sorgt daher nur für eine Verlängerung der für die Tests benötigten Zeit, bringt aber keine besseren Ergebnisse. Betrachtet man nun die Effektivität der Tests, erhält man im besten Fall für den Test des Erwartungswerts $m_{\text{eff}} = 0.54$, den Test der Varianz $m_{\text{eff}} = 0.74$ und für die Kombination $m_{\text{eff}} = 0.95$. Es ist aber auch festzustellen, dass kein Mutant fälschlicherweise getötet wurde. Zumindest empirisch gesehen ist die vorgeschlagene Methode also ein partielles Orakel.

Tabelle 7.2 zeigt die Ergebnisse des zweiten Teils der Studie, die aus Tests gegen eine Referenzimplementierung besteht. In diesem Fall wurde die Originalimplementierung als Referenz genommen. Diese Implementierung wurde schon zur Erzeugung der Mutanten verwendet. Zusätzlich zu den Gauss-Tests zum Vergleichen der Erwartungswerte der Stichproben bzw. der absolut zentrierten Stichproben wurde ein F-Test zum Vergleich der Varianzen der beiden Stichproben verwendet. Damit soll überprüft werden, ob der Test auf Gleichheit der Erwartungswerte der absolut zentrierten Stichproben als Ersatz für einen üblicherweise nicht verfügbaren allgemein anwendbaren Test zum Vergleich der Varianzen geeignet ist.

Genau wie im ersten Teil der Studie wurden die Tests mit unterschiedlichen Stichprobenumfängen durchgeführt. Die maximale Anzahl an getöteten Mutanten wurde bereits für $n = 500$ erreicht. Die einzelnen Tests haben eine leicht geringere Effektivität als beim Vergleich mit den theoretischen Werten im ersten Teil. Der Test auf gleichen Erwartungswert in beiden Stichproben erreichte $m_{\text{eff}} = 0.51$, der Varianztest $m_{\text{eff}} = 0.73$ im besten Fall. Erstaunlich erfolgreich war der Test der linearen Variabilität mit $m_{\text{eff}} = 0.94$. Allerdings mussten auch hier, wie bei den Tests gegen theoretische Referenzwerte, unterschiedliche Tests kombiniert werden, um dieselbe Anzahl an getöteten Mutanten zu erreichen. Die Kombination von Varianz-Test und Test auf gleiche lineare Variabilität zeigt, dass beide Tests dieselben Mutanten töten, da die kombinierte Anzahl an getöteten Mutanten immer dem Maximum der Anzahl der getöteten Mutanten der einzelnen Tests entspricht.

n	Anzahl der Mutanten, getötet durch Test mit H_0			Kombinationen der Tests				δ_{max}
	$\mathbb{E}(P) = \mathbb{E}(S)$ (a)	$Var(P) = Var(S)$ (b)	$\mathbb{E}(P') = \mathbb{E}(S')$ (c)	(a) & (b)	(a) & (c)	(b) & (c)	(a) & (b) & (c)	
50	50	94	93	137	136	102	145	0.274
100	67	108	146	150	149	151	153	0.205
200	83	119	170	164	172	175	177	0.140
300	91	140	178	189	187	183	190	0.112
500	95	145	189	193	194	189	194	0.086
1000	105	147	192	194	194	192	194	0.063
1500	105	149	192	194	194	192	194	0.052

Tabelle 7.2: Ergebnisse der Tests gegen eine Referenzimplementierung – Studie 1

7.1.2 Studie 2: Tessellationen

Die erste Studie diente primär einer ersten Untersuchung der vorgeschlagenen Methoden unter vereinfachten Bedingungen. Die zweite Studie soll nun anhand einer komplexeren und realitätsnäheren Implementierung die Brauchbarkeit des vorgeschlagenen Verfahrens unterstreichen.

Die Ausgaben des SUT sind nun nicht mehr nur Zahlen, sondern komplexe Objekte. Diese Objekte beschreiben Tessellationen, sie repräsentieren also ein geometrisches Objekt. Dieses komplexe Objekt muss in Zahlen transformiert werden, bevor die präsentierten statistischen Tests anwandt werden können. Dazu bieten sich gewisse Kenngrößen $\lambda_1, \ldots, \lambda_4$ für Tessellationen an. Über den Parameter $\gamma > 0$ des SUT können die theoretischen Werte für die Kenngrößen nach [66, 73] bestimmt werden:

$$\begin{aligned}
\lambda_1 &= \frac{1}{\pi}\gamma^2 \text{ (mittlere Anzahl von Knoten pro Flächeneinheit)} \\
\lambda_2 &= \frac{2}{\pi}\gamma^2 \text{ (mittlere Anzahl von Kanten pro Flächeneinheit)} \\
\lambda_3 &= \frac{1}{\pi}\gamma^2 \text{ (mittlere Anzahl von Zellen pro Flächeneinheit)} \\
\lambda_4 &= \gamma \text{ (mittlere Gesamtlänge der Kanten pro Flächeneinheit)}
\end{aligned}$$

Für jede Realisierung bzw. Programmausgabe werden diese Kenngrößen bestimmt und können dann durch Gauss-Tests mit den theoretischen Werten verglichen werden.

Das SUT besteht aus insgesamt 1200 LOC. Aus der Implementierung wurden dann 2362 Mutanten mit MuJava erzeugt, allerdings konnten nur 1926 Mutanten übersetzt werden. Von diesen terminierten 570 durch Werfen einer Exception. Weitere 36 Mutanten terminierten nicht nach 30s Ausführung und wurden damit als nicht-terminierend klassifiziert.

Zur Durchführung der Tests wurde wieder $\alpha = \beta = \frac{1}{3}$ verwendet, die Anzahl der Wiederholungen wurde auf $R = 501$ festgelegt. Als Parameter des SUT wurde $\gamma = 0.05$ gewählt.

Dieses SUT ist, im Gegensatz zum SUT in Studie 1, tatsächlich randomisierte Software. Deshalb sind „klassische“ Testtechniken nicht anwendbar. Insbesondere der Vergleich mit der Originalimplementierung ist so nicht mehr möglich. Um dennoch eine Vergleichsgröße zu bekommen, wurde ein separater Testlauf mit der sog. *fixed seed* Technik (siehe

[11]) durchgeführt. Dabei wird der verwendete Zufallszahlengenerator bei jedem Testdurchlauf bei Original und Mutant mit demselben Wert initialisiert. Dadurch liefert der Zufallszahlengenerator dieselbe Folge von „Zufallszahlen“. Die Anwendung wird dadurch deterministisch und die Ausgaben von Mutant und Original können verglichen werden. Auf diese Weise wurden insgesamt 139 Mutanten identifiziert, die vom Original abweichende Ausgaben produzierten.

Tabelle 7.3 zeigt nun die Ergebnisse der Tests gegen die theoretischen Werte der Kenngrößen. Jede Kenngröße wurde separat untersucht indem durch einen (asymptotischen) Gauss-Test überpüft wurde, ob der Erwartungswert der Kenngröße der Programmausgaben dem theoretischen Wert entspricht. Zudem wurden die Ergebnisse dieser Tests miteinander kombiniert. Die Spalten 2–5 der Tabelle zeigen die Anzahl der durch die Tests der einzelnen Kenngrößen getöteten Mutanten. Spalte 6 gibt die Anzahl der durch die Kombination der Tests getöteten Mutanten an.

In dieser Studie reichte bereits ein Stichprobenumfang von $n = 300$ aus, um die maximale Anzahl an getöteten Mutanten zu erreichen. Dabei ist festzustellen, dass die Kombination der Tests der einzelnen Kenngrößen kaum Verbesserungen bringt. Ein Vergleich mit dem durch fixe Initialisierung des Zufallszahlengenerators gewonnenen Tests zeigt zudem, dass die Effizienz der Methode stark zu wünschen übrig lässt. Die Kombination der Tests erreicht gerade einmal $m_{\text{eff}} = 0.47$. Dieser Wert entspricht in etwa dem der Tests für den Erwartungswert in der ersten Studie. Die theoretischen Resultate zu Tessellationen erlauben aber keine Varianz-Tests, deshalb fehlt deren Beitrag bei der Kombination der Tests.

n	λ_1	λ_2	λ_3	λ_4	Alle	δ_{max}
100	56	58	56	56	64	0.00506
200	58	60	60	58	63	0.00387
300	59	61	61	61	66	0.00328
500	60	62	62	60	66	0.00250
1000	61	62	62	61	66	0.00180

Tabelle 7.3: Ergebnisse der Tests gegen theoretische Referenzwerte – Studie 2

In Studie 1 wurde festgestellt, dass der Test auf gleiche lineare Variabililtät ein brauchbarer Ersatz für Tests auf gleiche Varianz zu sein scheint. Im nächsten Schritt wurden deshalb Tests gegen eine Referenzimplementierung durchgeführt. Tabelle 7.4 präsentiert deren Resultate. Die Tabelle führt nur noch die Ergebnisse für die jeweili-

ge Kombination der Tests auf, da der erste Teil der Untersuchung keine wesentlichen Unterschiede zwischen den Ergebnissen für die einzelnen Kenngrößen gezeigt hat.

n	$\mathbb{E}(P) = \mathbb{E}(S)$	$\mathbb{E}(P') = \mathbb{E}(S')$	Kombination	δ_{max}
100	58	57	58	0.00766
200	55	56	57	0.00537
300	58	65	65	0.00451
500	60	63	66	0.00347
1000	60	63	66	0.00251

Tabelle 7.4: Ergebnisse der Tests gegen eine Referenzimplementierung – Studie 2

Im Gegensatz zur vorherigen Studie bleibt die Effektivität der untersuchten Methode hinter den Erwartungen zurück. Auch der Test auf gleiche lineare Variabilität trägt nicht zur Verbesserung der Ergebnisse bei. Das kann zwei Gründe haben. Entweder sind die verwendeten Kenngrößen ungeeignet um Tests zu konstruieren, oder die Tests selbst, besonders der Test auf gleiche lineare Variabilität, sind ungeeignet. Für ersteres spricht die Tatsache, dass bei der Berechnung der Kenngrößen ein großer Teil der Informationen verloren gehen. Die Kenngrößen beschreiben nur Teile der Tessellation, andere Teile wie Lage und Verteilung der Knoten gehen aber verloren. Für den zweiten Grund spricht die Tatsache, dass die Kenngrößen selbst Poisson-verteilt sind. Bei der Poisson-Verteilung sind aber Erwartungswert und Varianz gleich, dadurch überprüft ein Varianz-Test oder ein vergleichbarer Test keine weiteren Charakteristika der Verteilung.

7.2 Statistisch-Metamorphische Tests

In diesem Kapitel soll eine kurze empirische Untersuchung des in Kapitel 3.3 vorgestellten Verfahrens der Statistisch-Metamorphischen Tests erfolgen. Die Ergebnisse der folgenden Studie wurden, wie auch das Verfahren selbst, bereits in [47] publiziert.

Da sich nach einigen Experimenten herausstellte, dass sich aus dem einführenden Beispiel in Kapitel 3.3.1 keine aussagekräftigen Daten gewinnen lassen, wurde erneut auf den „freundlichen“ Fall der Standard-Normalverteilung bzw. deren inverser Verteilungsfunktion Φ^{-1} zurückgegriffen.

Trotz der freundlichen statistischen Eigenschaften dieses Untersuchungsgegenstandes hat dieser einen deutlichen Nachteil. Da nur ein einziger Parameter benötigt wird, konnte nur eine einzige Metamorphische Relation identifiziert werden. Für X gleichverteilt auf dem Intervall $[0, 1)$ und $a, b > 0$ gilt

$$a\Phi^{-1}(X) + b\Phi^{-1}(X) \stackrel{d}{=} \sqrt{a^2 + b^2}\Phi^{-1}(X)$$

(Faltungsstabilität der Normalverteilung, siehe beispielsweise [15]). Da nur diese eine Relation identifiziert wurde, konnte dementsprechend auch keine vergleichende Untersuchung mehrerer Relationen wie beispielsweise in Kapitel 5 durchgeführt werden. In der folgenden Untersuchung wurde zur Vereinfachung $a = 3$ und $b = 4$ gewählt.

Für diese Untersuchung wurde jedoch auf eine andere Implementierung zurückgegriffen. Die Implementierung wurde [62] entnommen, einer Java-Bibliothek für Simulationen. Aus der aus 90 LOC bestehenden Implementierung wurden mit MuJava insgesamt 306 Mutanten erzeugt. Die Mutanten wurden anschließend mit drei unterschiedlichen Testmethoden untersucht. Da es sich bei der Implementierung um eine deterministische Funktion handelt, konnte der Vergleich mit der Originalimplementierung als Orakel verwendet werden. Zudem wurden Tests mit statistischen Hypothesentests gegen die Originalimplementierung (wie in Kapitel 3.2 vorgestellt) und Statistisch-Metamorphische Tests durchgeführt. Um die Wahrscheinlichkeit für Fehlentscheidungen zu reduzieren, wurden jeweils $R = 871$ Wiederholungen durchgeführt.

Durch den Vergleich mit der Originalimplementierung wurden 256 Mutanten getötet, 77 davon durch Werfen einer Exception. Eine anschließende manuelle Inspektion des Quellcodes der 50 nicht getöteten Mutanten ergab, dass 35 Mutanten tatsächlich äquivalent zur Originalimplementierung sind, und dass bei den übrigen 15 Mutanten der

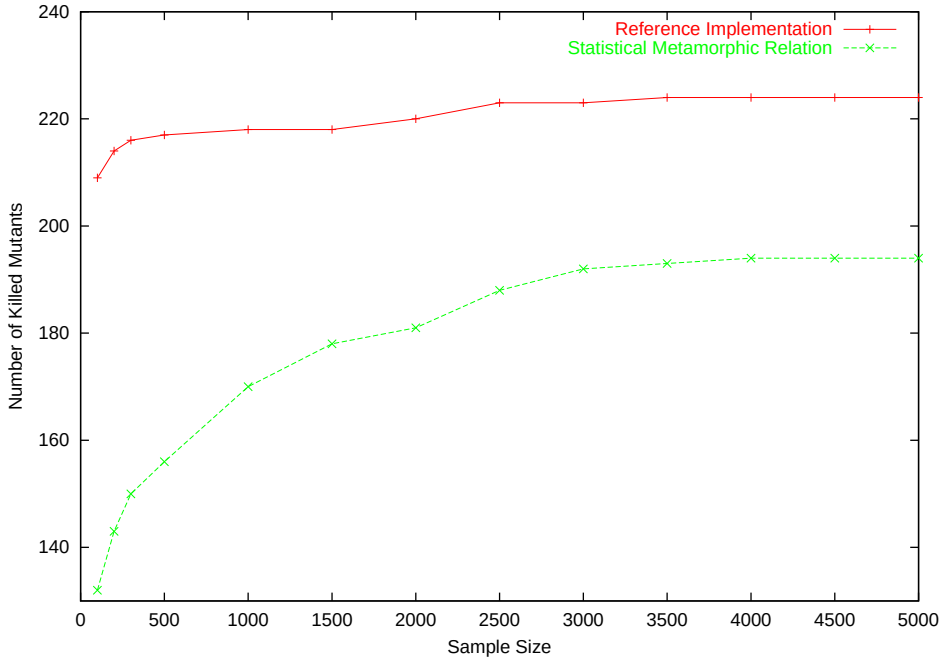


Abbildung 7.1: Vergleich von Testen mit statistischen Methoden und Statistisch-Metamorphischem Testen.

Pfad, welcher den Fehler enthält nur mit einer Wahrscheinlichkeit nahe bei Null ausgeführt wird.

In Abbildung 7.1 werden nun die Ergebnisse der anderen beiden Testverfahren miteinander verglichen. Die obere Linie zeigt die Anzahl der durch Testen mit statistischen Hypothesentests getöteten Mutanten, die untere Linie die durch Statistisch-Metamorphische Test getöteten Mutanten. Die x-Achse der Abbildung ist der bei den Tests verwendete Stichprobenumfang n. Wie zu erwarten, steigt bei beiden Verfahren die Anzahl der getöteten Mutanten mit steigendem Stichprobenumfang an. Beim Verfahren mit statistischen Hypothesentests stabilisiert sich die Anzahl der getöteten Mutanten ab einem Stichprobenumfang von $n = 3500$ bei 224 Mutanten ($m_{\text{eff}} = 0.855$). Der Ansatz der Statistisch-Metamorphischen Tests tötet ab $n = 4000$ konstant 194 Mutanten ($m_{\text{eff}} = 0.758$).

Eine manuelle Inspektion der Mutanten ergab, dass die Mutanten, die nicht durch den Test mit statistischen Hypothesentests getötet wurden, eine Funktion f berechnen, die zwar nicht mit Φ^{-1} übereinstimmt, aber die trotzdem die (durch die statistischen

Tests überprüften) Eigenschaften erfüllt.

Die Mutanten, die zwar durch statistische Hypothesentests, aber nicht durch statistisch-metamorphische Tests getötet wurden, weichen entweder nur am Rand oder nur sehr geringfügig von der Originalimplementierung ab. Diese können ebenfalls getötet werden, wenn bei den Tests ein größerer Stichprobenumfang verwendet wird. Das würde aber den Aufwand für die Durchführung der Tests deutlich erhöhen und damit die Wirtschaftlichkeit der vorgeschlagenen Methode reduzieren.

7.3 Diskussion statistischer Methoden im Softwaretest

Die im vorherigen Kapitel durchgeführten Versuche haben gezeigt, dass sich statistische Methoden zum Testen von randomisierter Software generell eignen. Im besten Fall konnte eine beachtliche Effektivität von $m_{\text{eff}} = 0.95$ erreicht werden. Die Effektivität hängt jedoch sehr stark von einer geeigneten Wahl der statistischen Hypothesentests ab. Wenn nicht unter guten Bedingungen getestet wird, sinkt die Effektivität deutlich. Das ist im Falle des Tests der Simulation von Tessellationen geschehen.

Bei statistischen Untersuchungen beschränkt oft das vorhandene Datenmaterial bzw. der Stichprobenumfang die Genauigkeit der gewonnenen Aussagen. Beim Testen von Software ist das nicht der Fall, da eine fast beliebig große Datenmenge erzeugt werden kann. Der Stichprobenumfang wird nur durch die zur Verfügung stehenden Ressourcen beschränkt. Deshalb sollte es auch unproblematisch sein, asymptotische Tests durchzuführen, da durch den großen Stichprobenumfang die Konvergenz der Teststatistiken gewährleistet werden kann.

Testen mit statistischen Hypothesentests und Statistisch-Metamorphisches Testen erfordert eine große Anzahl an Programmausführungen. Dadurch benötigt diese Art des Testens wesentlich mehr Ressourcen als beispielsweise Metamorphisches Testen, selbst wenn bei diesem ebenfalls zufällig erzeugte Eingaben verwendet werden. Aus Gründen der (Laufzeit-)Effizienz sollte deshalb versucht werden, wenn möglich deterministische Testverfahren zu verwenden.

In der vorliegenden Arbeit wurden Gauss-Tests verwendet, da diese nicht nur im Allgemeinen eine gute Trennschärfe besitzen, sondern weil wie in Kapitel 3.2.6 erwähnt auch die Wahrscheinlichkeit für einen β-Fehler analytisch abgeschätzt werden kann. Für andere Tests ist dies im Allgemeinen nicht möglich. Oft wurden jedoch empirische Untersuchungen dazu durchgeführt bzw. können vor den eigentlichen Softwaretests mit den

dazu vorgesehenen Tests durchgeführt werden. Wenn aus diesen Untersuchungen abgeleitet werden kann, dass die Voraussetzung $\max\{\alpha, \beta\} < \frac{1}{2}$ erfüllt ist, können auch diese Tests zum Treffen der Testentscheidung verwendet werden.

Mit der neu formulierten Testbedingung $P \stackrel{d}{=} S$ ist auch ein neuer Begriff von Programmäquivalenz verbunden. Die Gleichheit von Implementierung und Spezifikation bezieht sich jetzt nur noch auf statistische Eigenschaften. Man kann also jetzt von statistischer Programmäquivalenz sprechen. Es kann nun vorkommen, dass Implementierungen als statistisch äquivalent zur Spezifikation angesehen werden, die deterministisch etwas völlig anderes als gefordert berechnen. Fehler der Implementierung können daher „im Rauschen untergehen“. Wenn, wie in den Fallstudien geschehen, nur einzelne Charakteristika der Verteilung geprüft werden, ist dieser Effekt für einen guten Teil der nicht entdeckten Fehler verantwortlich. Bisher wurde aber noch nicht untersucht, inwieweit durch die Wahl von statistischen Tests dieser Effekt minimiert werden kann.

Das Statistisch-Metamorphische Testen hat sich zumindest in der durchgeführten Untersuchung als brauchbare Testtechnik herausgestellt. Besonders wenn benötigte Referenzwerte oder eine Referenzimplementierung nicht zur Verfügung stehen kann Statistisch-Metamorphisches Testen einen großen Teil der Mutanten töten. Da bisher andere Testtechniken in diesem Fall nicht zur Verfügung standen, ist dies eine deutliche Verbesserung.

Kapitel 8

Zusammenfassung und Ausblick

Große Softwaresysteme enthalten fast zwangsläufig Fehler. Das Finden und Beseitigen von Fehlern ist für einen großen Teil des Aufwandes beim Entwickeln von Softwaresystemen verantwortlich. Neben anderen Techniken ist das Testen von Software eine der Aktivitäten zum Finden von Fehlern. Testen ist wohl die Technik, die dazu am häufigsten eingesetzt wird. Testen ist (oder sollte) Bestandteil eines jeden modernen Vorgehensmodells zum Entwickeln von Software sein.

Beim Testen wird das zu testende System unter festgelegten Bedingungen ausgeführt und das Verhalten des Systems beobachtet. Ein Fehler wird dann festgestellt, wenn das Verhalten des Systems von der vorher festgelegten Spezifikation abweicht.

Beim Testen gibt es zwei Grundprobleme: zum einen die Auswahl der Testeingaben, zum anderen das Treffen einer Testentscheidung. Bei dieser Testentscheidung muss festgestellt werden, ob sich das zu testende Programm konform zur Spezifikation verhält.

In der vorliegenden Arbeit wurden unterschiedliche Methoden zum Treffen einer Testentscheidung (Orakel) vorgestellt, entwickelt und untersucht. Bei den Untersuchungen ging es darum, festzustellen wieviele Fehler eine Testmethode entdeckt, also wie effektiv eine Testmethode ist. Dazu wurden bekannte Resultate zur Untersuchung der Güte einer Testeingabemenge so abgewandelt, dass auch die Güte einer Testentscheidung beurteilt werden kann.

Zur Untersuchung der vorgestellten Orakel wurde überwiegend Software aus Open Source-Projekten verwendet. Das zeigt, dass die vorgeschlagenen Methoden geeignet sind, praktisch relevante Software zu testen. Aus den Ergebnissen der empirischen Untersuchungen wurden zudem Hinweise abgeleitet, wie die Effektivität der vorgestellten

Orakel gesteigert werden kann.

Alle Testmethoden erlauben zudem, die Testentscheidung automatisch, also ohne menschliches Zutun, treffen zu können. Die Automatisierung des Testvorgangs hat viele Vorteile: die Tests werden reproduzierbar und der Aufwand für die Testdurchführung sinkt deutlich. Sind automatisierte Tests einmal implementiert, können diese ohne großen Aufwand erneut durchgeführt werden. Auch die Erweiterung und Anpassung der Tests wird erleichtert.

Wie ein solcher automatisierter Test aufgebaut werden kann, wird in einer der Untersuchungen von Testmethoden demonstriert. In der Untersuchung wird eine Vorgehensweise zum automatisierten Testen von Bildverarbeitungssoftware präsentiert. Dazu werden nicht nur die benötigten Orakel, sondern auch Techniken zum automatischen Erzeugen von Bildern als Testeingaben vorgestellt. In den Untersuchungen der vorgestellten Methoden hat nicht nur gezeigt, dass Bildverarbeitungssoftware automatisiert getestet werden kann, sondern auch, dass die vorgestellten Testmethoden effektiv einsetzbar sind.

Beiträge der vorliegenden Arbeit

Die wesentlichen Beiträge der vorliegenden Arbeit sind:

- die Entwicklung einer Methode zum Bewerten von Orakellösungen
- die Beschreibung eines allgemein anwendbaren Vorgehens zum Testen von Software mit statistischen Hypothesentests
- die Beschreibung eines neuen Testansatzes, dem sog. Statistisch-Metamorphischen Testen
- systematische Untersuchungen von Orakellösungen, insbesondere des Metamorphischen Testens
- die Herleitung von Kriterien zur Auswahl von effektiven Metamorphischen Relationen
- Hinweise für die effektive Anwendung von Metamorphischem Testen, basierend auf den Ergebnissen empirischer Untersuchungen
- die Präsentation eines Ansatzes zum vollständig automatisierten Test von Bildverarbeitungssoftware

Ausblick

Das Forschungsgebiet „Software-Test-Orakel“ hat lange Zeit nur wenig Aufmerksamkeit erhalten. Auch die vorliegende Arbeit kann nur einen kleinen Teil dieses Gebiets bearbeiten. Weitere Forschung in diesem Gebiet ist deshalb notwendig.

Aus der vorliegenden Arbeit ergeben sich zwei weitere Forschungsrichtungen. Zum einen die Weiterentwicklung von Methoden zur Bewertung von Orakellösungen und zum anderen die Entwicklung und Untersuchung weiterer Orakellösungen.

Die in der vorliegenden Arbeit vorgestellten Bewertungsmethoden können sicherlich noch weiterentwickelt werden. Wichtiger wäre aber eine Überprüfung der vorgestellten Methode anhand von „echten“ Fehlern. Die Modellannahmen der Bewertungsmethode wurden bisher unzureichend geprüft und die Ergebnisse dieser Prüfungen sind leider nicht eindeutig.

Die vorgestellten Orakellösungen wurden im Rahmen der vorliegenden Arbeit ausschließlich an Software untersucht, die aus dem wissenschaftlichen Umfeld stammt. Die Software konnte immer durch mathematischen Strukturen beschrieben werden. Es wäre nun interessant zu untersuchen, ob die vorgestellten Methoden auch auf den Test von Software übertragen werden können, für die eine mathematische Beschreibung nicht existiert. Falls das nicht der Fall ist, müsste nach anderen Orakellösungen gesucht werden.

Literaturverzeichnis

[1] V. D. Agrawal. When to use random testing. *IEEE Transactions on Computers*, 27:1054–1055, 1978.

[2] P. Ammann and J. Knight. Data Diversity: An Approach to Software Fault Tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[3] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411. ACM, 2005.

[4] Apache Software Foundation. Apache Commons.Math. http://commons.apache.org/math/.

[5] A. Avritzer and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, 1995.

[6] V. Basili and B. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.

[7] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, 1987.

[8] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2000.

[9] K. Beck. *Test-driven Development: By Example.* Addison-Wesley Professional, 2003.

[10] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, pages 85–103. IEEE Computer Society Washington, DC, USA, 2007.

[11] J. Bible and G. Rothermel. A unifying framework supporting the analysis and development of safe regression test selection techniques. Technical Report 99-6011, Oregon State University, 1999.

[12] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional, 1999.

[13] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM (JACM)*, 42(1):269–291, 1995.

[14] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(5):529–533, 1995.

[15] G. Casella and R. Berger. *Statistical inference.* Duxbury Press Belmont, CA, USA, 1990.

[16] F. Chan, T. Y. Chen, S. C. Cheung, M. Lau, and S. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the 1998 IASTED Conference in Software Engineering*, pages 191–197. Acta Press, 1998.

[17] W. Chan, S. Cheung, J. Ho, and T. Tse. Reference Models and Automatic Oracles for the Testing of Mesh Simplification Software for Graphics Rendering. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)-Volume 01*, pages 429–438. IEEE Computer Society Washington, DC, USA, 2006.

[18] W. Chan, S. Cheung, and K. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, pages 470–476. IEEE Computer Society Washington, DC, USA, 2005.

[19] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications.

In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, pages 241–249, Los Alamitos, California, 2005. IEEE Computer Society Press.

[20] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, 1978.

[21] T. Chen, F. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *Proceedings of the 5th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2004)*, pages 128–134, 2004.

[22] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[23] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: A case study. In *Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2002)*, pages 327–333. IEEE Computer Society, 2002.

[24] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering*, pages 569–583, Madrid, Spain, 2004. Polytechnic University of Madrid.

[25] T. Y. Chen, T. Tse, and Z. Zhou. Fault based testing in the absence of an oracle. In *Proceedings of the 25th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2001)*, pages 172–178. IEEE Computer Society, 2001.

[26] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279, 2000.

[27] Cobertura Project. A coverage analysis tool for java. http://cobertura.sourceforge.net/.

[28] A. Cockburn. *Surviving object-oriented projects: a manager's guide.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[29] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[30] P. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14(3):227–248, 1980.

[31] J.-M. Dautelle. JScience. http://jscience.org/.

[32] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.

[33] A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In *Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, volume 2399 of *Lecture Notes in Computer Science*, pages 211–212. Springer-Verlag, 2002.

[34] R. Doong and P. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, 1994.

[35] R. Dunn. *Software defect removal.* McGraw-Hill, Inc. New York, NY, USA, 1984.

[36] C. Erhardt. *Static Code Analysis in Multi-Threaded Environments.* PhD thesis, Ulm University, 2007.

[37] M. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[38] M. Flanagan. Java library. http://www.ee.ucl.ac.uk/∼mflanaga/java/Matrix.html.

[39] F. Fleischer. *Analysis and Fitting of Random Tessellation Models. Applications in Telecommunication and Cell Biology.* PhD thesis, Ulm University, 2007.

[40] J. Forrester and B. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 59–68, 2000.

[41] P. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 153–162. ACM Press New York, NY, USA, 1998.

[42] C. Gloaguen, F. Fleischer, H. Schmidt, and V. Schmidt. Simulation of typical Cox–Voronoi cells with a special regard to implementation tests. *Mathematical Methods of Operations Research (ZOR)*, 62(3):357–373, 2005.

[43] C. Gloaguen, F. Fleischer, H. Schmidt, and V. Schmidt. Fitting of stochastic telecommunication network models via distance measures and Monte–Carlo tests. *Telecommunication Systems*, 31(4):353–377, 2006.

[44] C. Gloaguen, F. Fleischer, H. Schmidt, and V. Schmidt. Modelling and simulation of telecommunication networks: Analysis of mean shortest path lengths. In R. Lechnerova, I. Saxl, and V. Benes, editors, *Proceedings of the 6th International Conference on Stereology, Spatial Statistics and Stochastic Geometry*, pages 25–36. Union of Czech Mathematicians and Physicists, Prague, Czech Republic, 2006.

[45] A. Gotlieb. Exploiting symmetries to test programs. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*, pages 365–374. IEEE Computer Society, 2003.

[46] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert. Benchmarking testing strategies with tools from mutation analysis. In *Proceedings of the first Workshop on A Benchmark for Software Testing (TestBench'08)*. IEEE Digital Library, 2008.

[47] R. Guderlei and J. Mayer. Statistical metamorphic testing - testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proceedings of the 7th International Conference on Quality Software (QSIC 2007)*, pages 404–409. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[48] R. Guderlei and J. Mayer. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal on Software Engineering and Knowledge Engineering*, 17(6):757–781, 2007.

[49] R. Guderlei, J. Mayer, C. Schneckenburger, and F. Fleischer. Testing randomized software by means of statistical hypothesis tests. In *Proceedings of SOQUA '07: Fourth international workshop on Software quality assurance*, pages 46–54, New York, NY, USA, 2007. ACM.

[50] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis.*, pages 206–215, 1988.

[51] R. Hamlet. Testing Programs with the Aid of a Compiler. *IEEETransactions on Software Engineering*, 3(4):279–290, 1977.

[52] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[53] R. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, 53(10):1330–1342, 2004.

[54] W. Howden. Functional program testing. In *Proceedings of the IEEE Computer Society's Second International Computer Software and Applications Conference (COMPSAC'78)*, pages 321–325, 1978.

[55] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

[56] B. Jähne. *Digital Image Processing.* Springer-Verlag, 6th edition, 2005.

[57] JAMA Project. Java matrix package (JAMA). http://math.nist.gov/javanumerics/jama/.

[58] S. Johnson. *Lint, a C Program Checker.* Bell Telephone Laboratories, 1977.

[59] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[60] J. Knight and P. Ammann. An experimental evaluation of simple methods for seeding program errors. In *Proceedings of the 8th international conference on Software engineering*, pages 337–342. IEEE Computer Society Press Los Alamitos, CA, USA, 1985.

[61] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[62] P. L'Ecuyer. Stochastic Simulation in Java homepage. http://www.iro.umontreal.ca/ simardr/ssj/.

[63] P. S. Loo and W. K. Tsai. Random testing revisited. *Information and Software Technology*, 30:402–417, 1988.

[64] Y.-S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, pages 352–366, 2002.

[65] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[66] R. Maier and V. Schmidt. Stationary iterated tessellations. *Advances in Applied Probability*, 35:337–353, 2003.

[67] G. Matheron. *Random sets and integral geometry.* Wiley Series in Probability and Mathematical Statistics, 1975.

[68] J. Mayer. On testing image processing applications with statistical methods. In *Proceedings of Software Engineering 2005 (SE 2005)*, volume P-64 of *Lecture Notes in Informatics*, pages 69–78, Bonn, Germany, 2005. Köllen Druck+Verlag GmbH.

[69] J. Mayer and R. Guderlei. Test oracles using statistical methods. In *Proceedings of the First International Workshop on Software Quality (SOQUA 2004)*, volume P-58 of *Lecture Notes in Informatics*, pages 179–189, Bonn, Germany, 2004. Köllen Druck+Verlag GmbH.

[70] J. Mayer and R. Guderlei. An Empirical Study on the Selection of Good Metamorphic Relations. In *Proceedings of the 30th Annual International Computer*

Software and Applications Conference, 2006 (COMPSAC 06)., volume 1, pages 475–484, 2006.

[71] J. Mayer and R. Guderlei. On random testing of image processing applications. In *Proceedings of the 6th International Conference on Quality Software (QSIC 2006)*, pages 85–92. IEEE Computer Society, 2006.

[72] J. Mayer, V. Schmidt, and F. Schweiggert. A unified simulation framework for spatial stochastic models. *Simulation Modelling Practice and Theory*, 12(5):307–326, 2004.

[73] J. Mecke. Parametric representation of mean values for stationary random mosaics. *Mathematische Operationsforschung und Statistik Series Statistics*, 15:437–442, 1984.

[74] B. Meek and K. Siu. The effectiveness of error seeding. *ACM SIGPLAN Notices*, 24(6):81–89, 1989.

[75] A. Memon and Q. Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 8–17, 2004.

[76] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2000.

[77] B. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54. ACM Press New York, NY, USA, 2006.

[78] J. Miller and C. Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[79] H. Mills. On the Statistical Validation of Computer Programs. Technical report, IBM FSD Report, 1970.

[80] R. Milton and R. Hotchkiss. Computer Evaluation of the Normal and Inverse Normal Distribution Functions. *Technometrics*, 11(4):817–822, 1969.

[81] I. Molchanov. *Statistics of the Boolean model for practitioners and mathematicians*. Wiley, Chichester, NY, 1997.

[82] D. Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–101. ACM Press, New York, NY, USA, 2001.

[83] D. Monniaux. Abstraction of expectation functions using Gaussian distributions. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes In Computer Science*, pages 161–173. Springer-Verlag, 2002.

[84] P. Moreau and C. Ronse. Generation of shading-off in images by extrapolation of lipschitz functions. *Graphical Models and Image Processing*, 58(4):314–333, 1996.

[85] J. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application.* McGraw-Hill, Inc. New York, NY, USA, 1987.

[86] G. Myers. *The Art of Software Testing.* John Wiley & Sons Inc, 1979.

[87] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Software Engineering Notes*, 29(4):55–64, 2004.

[88] National Institutes of Health. ImageJ homepage, 2007. http://rsb.info.nih.gov/ij/.

[89] J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):5–20, 1992.

[90] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55, 2000.

[91] T. Ostrand and E. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300, 1984.

[92] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.

[93] J. Pfaltz. Sequential Operations in Digital Picture Processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

[94] H. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[95] W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the IEEE Western Electronic Show and Convention (WESCON)*, pages 1–9. IEEE Computer Society Press Los Alamitos, CA, USA, 1970.

[96] P. Rusås. Java Image Processing API homepage, 2004. http://www.ia.hiof.no/∼por/imageprocAPI/version2/.

[97] H. Schmid. *Konzeption einer pragmatischen Testmethodik für den Test von eingebetteten Systemen*. PhD thesis, Ulm University, 2003.

[98] U. Schöning. *Theoretische Informatik kurz gefasst*. BI-Wiss.-Verl., 1992.

[99] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[100] H. Sevcikova, A. Borning, D. Socha, and W.-G. Bleek. Automated testing of stochastic systems: A statistically grounded approach. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 215–224. ACM, 2006.

[101] L. Shapiro and G. Stockman. *Computer vision*. Prentice Hall Upper Saddle River, NJ, 2001.

[102] D. Slutz. Massive stochastic testing of SQL. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB 1998)*, pages 618–622, 1998.

[103] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, 2nd edition, 2004.

[104] J. Squire. A Java matrix class. http://www.csee.umbc.edu/∼squire/download/Matrix.java.

[105] M. Stencel and J. Janacek. Homepage of the Lipschitz filter ImageJ plugin, 2006. http://rsb.info.nih.gov/ij/plugins/lipschitz/index.html.

[106] D. Stoyan, W. S. Kendall, and J. Mecke. *Stochastic Geometry and Applications*. John Wiley & Sons, Chichester, 1995.

[107] P. Thevenod-Fosse, H. Waeselynck, Y. Crouzet, and T. LAAS-CNRS. An experimental study on software structural testing: deterministic versus random input generation. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 410–417, 1991.

[108] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 458–466, Los Alamitos, California, 2004. IEEE Computer Society Press.

[109] Ulm University, Dept. of Stochastics and Dept. of Applied Information Processing. GeoStoch homepage. http://www.geostoch.de.

[110] W. Weil and J. Wieacker. Densities for Stationary Random Sets and Point Processes. *Advances in Applied Probability*, 16(2):324–346, 1984.

[111] E. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[112] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)*, pages 20–24. IEEE Computer Society, 2003.

[113] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, Japan, 2004. Software Engineers Association.

Summary

All large software systems contain bugs. Therefore the identification and the removal of these bugs is an important and time consuming task, responsible for a large slice of the software development schedule. The testing of software is the most prominent method to improve software quality. It is part of any modern software development process.

Testing is the execution of a software system under defined conditions. The system is monitored during execution and its output is collected. A bug is identified if the behaviour of the program (i.e. its output) does not correspond to its specification.

There are two main problems in software testing. The first problem is the choice of the test input, the second problem is the so-called oracle problem. An oracle is a mechanism to obtain reference values and to compare the actual output of the system under test to these reference values. Often, there is no oracle available.

In the present thesis, several oracles are presented and discussed. Some of the presented oracles were newly developed. Additionally, a new method for the measurement of the effectiveness of an oracle was developed. This effectiveness measure is based on a well-known test adequacy criterion, the so-called mutation analysis technique.

The presented oracle methods were examined using software from open source projects, showing that these methods can be applied to real life software. The experiments lead to the conclusion that the proposed methods are effective. The results of the examination were also used to derive guidelines for the improvement of the effectiveness of the proposed methods.

The presented methods can be applied automatically. That means that the test decisions are made without human intervention. Therefore, the test results are reproducible and unbiased. The automation of the testing process also reduces the necessary amount of time for running and rerunning the tests. This does not only reduce the costs for testing, testing can also be done more frequently. That leads to an improvement of the

software quality.

The automated software testing is illustrated in one of the empirical studies. There, an approach for the testing of imaging software is presented. The approach contains not only method for the automatic generation of images as test input, but also the necessary oracles are presented. The presented approach is not limited to a few image operators, it can be applied to a broad set of image operators. The empirical studies reveal that the proposed approach is very effective. As imaging software is usually tested manually, the presented approach clearly demonstrates the possibilities of the methods proposed in the present thesis.

Recapitulating, the main contributions of the present thesis are:

- the definition of a new effectiveness measure for oracles
- the presentation of a generally applicable approach for the usage of statistical hypothesis tests in order to make a test decision
- the development of a new testing approach, the so-called statistical metamorphic testing
- systematic empirical evaluation of various oracle methods, especially the so-called metamorphic testing
- the description of rules for the choice of effective metamorphic relations
- hints for the effective usage of metamorphic testing
- the presentation of a fully automated approach for the testing of imaging software