ulm university universität
uulm

Universität Ulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für Datenbanken und Informationssysteme

Kombinatorische Testmethoden zur Analyse aktuarieller Software

Bachelorarbeit

in Informatik

vorgelegt von
Johannes Gabriel Sindlinger
am 23.10.2021

Gutachter

Prof. Dr. Manfred Reichert

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abstract

1 Einleitung

1.1 Motivation

1.2 Ziel der Arbeit

1.3 Verwandte Arbeiten

1.4 Struktur der Arbeit

2 Theoretische Grundlagen

Im folgenden Kapitel werden die grundlegenden Konzepte und Methoden für die Beantwortung der Fragestellung dieser Arbeit vorgestellt. Dabei folgt zunächst ein grundlegender Einblick in die Begrifflichkeiten der Qualitätssicherung, des Softwaretestens und der Testfallerzeugung. Im zweiten Abschnitt dieses Kapitels wird das Konzept des Combinatorial Testing im Genauen erläutert, welches die wesentliche Strategie bei der Erzeugung von Testfällen des in Kapitel 3 vorgestellten Anwendungsfalls darstellt.

2.1 Einführung ins Softwaretesten

Im Allgemeinen betrachtet ist das Softwaretesten ein Teilbereich der übergeordneten Qualitätssicherung, die 'alle qualitätsrelevanten Aktivitäten und Prozesse' zur Optimierung der Softwarequalität beinhaltet [LL10, S. 269]. Insbesondere stehen laut Ludewig und Lichter [LL10] bei der Qualitätssicherung diejenigen Maßnahmen im Vordergrund, die zu einem hohen Vertrauen in das Softwareprodukt bei allen beteiligten Stakeholdern beitragen sollen. Der Begriff der Qualität selbst kann dabei aus vielen verschiedenen Sichtweisen betrachtet werden, unter anderem ist eine Unterteilung in die Produktqualität und eine Projektqualität gebräuchlich [LL10, S. 66]. Dabei fokussiert sich die Produktqualität auf die Eigenschaften und Merkmale des letztendliche Endprodukts im Rahmen der zuvor definierten Anforderungen, die Projektqualität blickt eher auf den Prozess der Entwicklung und Wartung des jeweiligen Produkts auf Prozessebene [LL10, S. 66 ff.].

Auf Basis dessen lässt sich eine hierarchische Auflistung der verschiedenen Ebenen, die Qualitätssicherung umfassen, erstellen: Ludewig und Lichter [LL10, S. 271 ff.] führen organisatorische, konstruktive und analytische Maßnahmen als wesentliche Ebenen der Qualitätssicherung auf. Diese lassen sich im Hinblick auf die zuvor eingeführte Definition des

Qualitätsbegriffs insofern zuordnen, als organisatorische Maßnahmen sich im Wesentlichen auf die Prozessqualität fokussieren, analytische Maßnahmen eher der Produktqualität zugeordnet werden können und konstruktive Maßnahmen in beiden Bereichen des Qualitätsbegriffs eine Rolle spielen.

Als konkrete Beispiele gelten in Bezug auf konstruktive und organisatorische Maßnahmen unter anderem ein gelungenes Projektmanagement (organisatorisch) oder die präventive Schulung von Mitarbeiter (konstruktiv) [LL10, S. 272]. Die analytischen Methoden können zudem in nicht-mechanische und mechanische Verfahren unterteilt werden. Nicht-mechanische Analysen können beispielsweise manuelle Code-Reviews durch separate Programmierer sein, mechanische Analyse umfassen statische Analysen wie die Compiler-Analysen (beispielsweise Pufferüberläufe) und die Methoden des Softwaretestens. Da beim Softwaretesten im Gegensatz zu den statischen Analysen ein Testobjekt auf einem Rechner ausgeführt wird, spricht man in diesem Fall auch von dynamischen Softwaretests [SL19, S. 135]. Abbildung 2.1 zeigt die hierarchische Struktur der verschiedenen Komponenten der Qualitätssicherung im Software-Bereich.

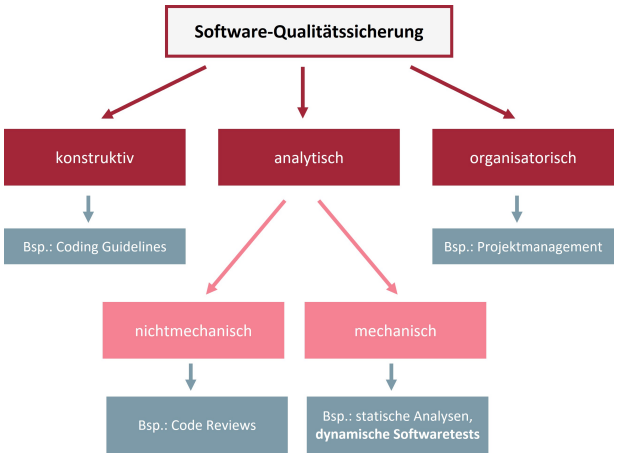

Abbildung 2.1: Die verschiedenen Komponenten der Qualitätssicherung in der Software-Entwicklung in der Übersicht [LL10, S. 271].

Als Teilgebiet der Qualitätssicherung ist das dynamische Softwaretesten also ein wesentlicher Bestandteil bei der Sicherstellung der Produktqualität, deren wichtigsten Anforderungskriterien der ISO-/IEC-Standard 25010 [Int11] als internationale Norm für Software und IT-Systeme zusammenfasst: Im Kern werden dabei von Softwareprojekten eine hohe Funktionalität, Effizienz, Sicherheit, Kompatibilität, Verlässlichkeit, Usability, Wartbarkeit und Portierbarkeit gefordert.

Dynamische Softwaretests besitzen demnach im übergeordneten Sinne die Aufgabe, diese Anforderungen zu prüfen und sicherzustellen. Im Konkreten bedeutet dies, dass dynamische Softwaretests anhand vordefinierter Testfälle mögliche Fehler vor der tatsächlichen Nutzung der Software aufdecken sollen und zudem überprüfen, ob ein Softwaresystem die an das System gestellte Spezifikationen erfüllt [Som12, S. 246]. Testfälle werden typischerweise entweder anhand eines systematischen Vorgehens künstlich erstellt oder ergeben sich aus Erfahrungswerten (vgl. dazu Unterabschnitt 2.1.2).

Der Begriff des Fehlers kann dabei im Rahmen des Testens von Software vielschichtig interpretiert werden und nimmt unterschiedliche Rollen ein: Ein Fehlverhalten (engl.: failure) bezieht sich auf ein unerwartetes Ergebnis eines Programms, ein Codefehler (engl.: defect) beschreibt die zugehörigen fehlerhaften Codezeilen und ein Denkfehler (engl.: error) ist der Auslöser eines Codefehlers. Alle drei Fehlerdefinitionen sind eng miteinander verknüpft: Ein Denkfehler kann bereits in der Konzeption dazu führen, dass trotz einer mutmaßlich korrekten Implementierung Codefehler und somit unterschiedliches Fehlverhalten auftreten kann. Gleichzeitig kann es auch bei der Implementierung durch menschliches Versagen zu Codefehlern kommen, die in der Konzeption korrekt waren und somit keinen Denkfehler darstellen. Abbildung 2.2 zeigt die logische Struktur dieser Begrifflichkeiten auf und greift diese Thematik auf.

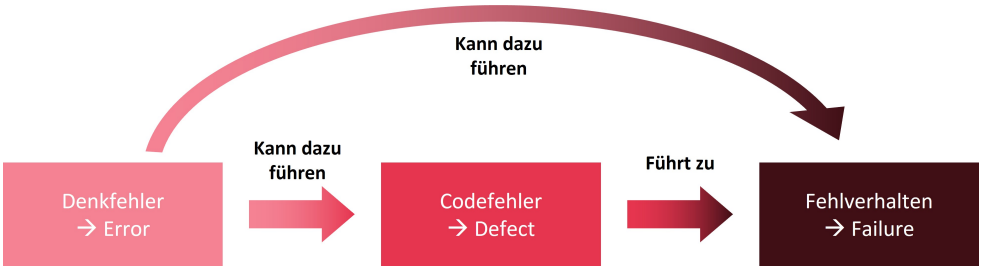

Abbildung 2.2: Ein Denkfehler bei der Implementierung führt zu einem Codefehler, welcher wiederum ein Fehlverhalten der Software auslöst.

Grundsätzlich gilt es zu berücksichtigen, dass dynamische Softwaretests nicht in der Lage sind, alle Fehler eines Systems vollumfänglich aufzudecken [Som12, S. 247]. Dafür sind die verwendeten Systeme meist zu komplex und vollumfängliche Tests, wie in Unterunterabschnitt 2.1.1 aufgeführt, nicht umsetzbar. Software-Engineering Pionier Edward Dijkstra stellte in diesem Zusammenhang bereits 1972 fest: 'Tests können nur die Anwesenheit von Fehlern aufzeigen, nicht ihre Abwesenheit' [DDH72].

Spillner et al. [SLS11, S. 8] fassen die Kernaufgaben, welche dynamische Softwaretests erfüllen sollen, in vier Aspekten zusammen:

- Entdeckung von Fehlverhalten innerhalb der betroffenen Software
- Schaffung von Vertrauen in das System bei allen betroffenen Stakeholdern
- Anregung frühzeitiger Analyse und Dokumentation in der Entwicklung und Wartung um Codefehler zu verhindern
- Messung von Software-Qualität durch Metriken wie zum Beispiel die Anzahl an Codezeilen, die zyklomatische Komplexität von McCabe [McC76] oder Halsteads-Softwaremetriken [Hal77]

2.1.1 Testfall-Design

Dynamische Softwaretests werden durch mehrere konkrete Tests an einem bestimmten Testsystem durchgeführt. Jeder Test unterliegt dabei einem spezifischen Testfall, der alle erforderlichen Komponenten zur Durchführung eines Tests bereithält. Diese sind die komplette Systemumgebung, Eingabedaten und das erwartete Verhalten beziehungsweise die erwartete Ausgabe des Systems [Sch12, S. 86].

Ein Testfall sollte dabei auf sinnvolle Art und Weise entwickelt werden: Dazu gehört laut Ludewig und Lichter [LL10, S. 480] eine eindeutig definierte Systemumgebung, ein systematisches Vorgehen bei der Wahl der Eingabedaten, klare Kriterien zur Testevaluation und eine detaillierte Dokumentation der Testergebnisse. Wenn diese vier Aspekte auf einen Testfall zutreffen, sprechen Ludewig und Lichter von einem 'systematischen Test'.

Eine der wesentlichen Herausforderungen des dynamischen Softwaretestens besteht darin, ein derartiges systematisches Vorgehen zu entwickeln, um Testfälle möglichst effizient zu erzeugen: Grundsätzlich ist das Ziel des Testfall-Designs mit möglichst wenig Aufwand, also einer geringen Anzahl an Testfällen, möglichst viele Fehler im Sinne eines Fehlverhaltens zu entdecken [LL10, S. 498]. Dabei sollte laut Ludewig und Lichter [LL10, S. 498] ein Testfall möglichst repräsentativ für viele andere Testfälle sein und zudem idealerweise sehr fehlersensitiv und redundanzarm.

Um diese Anforderungen abdecken zu können wurden in der Vergangenheit verschiedene Techniken ausgearbeitet, die ein systematisches und effizientes Testfall-Design vereinfachen

sollen. Dabei entstanden verschiedene Grundprinzipien, die sich aus einer unterschiedlichen Betrachtungsweise auf ein Softwaresystem ergeben. Diese sind nicht komplementär zu verstehen, sondern vielmehr ergänzend zur genauen Spezifizierung der Testfälle. Im Folgenden werden drei relevante Betrachtungsweisen vorgestellt: Stufenbasiertes Testen, Black Box- & White-Box-Tests und Abdeckungsbasiertes Testen.

Konkrete Testmethoden wie beispielsweise Combinatorial Testing (vgl. Abschnitt 2.2) oder die Äquivalenzklassenmethode (vgl. Unterabschnitt 2.1.2) lassen sich demnach meist unterschiedlichen Bereichen der aufgeführten Betrachtungsweisen zuordnen: So ist die Äquivalenzklassenmethode genauso wie das Combinatorial Testing auf fast allen Stufen des stufenbasierten Testens anwendbar, aber zugleich dem Black-Box-Ansatz zuzuordnen. Im Bereich des abdeckungsbasierten Testens spielt die Äquivalenzklassenmethode zudem bei der Abdeckung der Eingabewerte eine wichtige Rolle.

Stufenbasiertes Testen

Das stufenbasierte Testen ist eine der abstraktesten Betrachtungsweisen im Hinblick auf die Erstellung von Testfällen: Basierend auf verschiedenen Stufen innerhalb eines Softwareprojekts werden beim stufenbasierten Testen Testfälle passend zu den Anforderungen und Aktivitäten im jeweiligen Stadium des Projekts erarbeitet [AO08, S. 5].

Dabei kann auf verschiedene Stufen klassischer Prozessmodelle in der Softwareentwicklung zurückgegriffen werden: Häufig dient dabei das V-Modell als Basis, das im Auftrag des Bundesministeriums für Verteidigung entwickelt wurde und in der Softwarewelt in angepasster Form verbreitete Anerkennung fand [LL10, S. 190]. Das V-Modell ist als eine Weiterentwicklung des von Royce [Roy87] 1970 entwickelten Wasserfallmodells zu verstehen: Dieses sieht vor, dass die Entwicklung eines Produkts im Allgemeinen in verschiedene Phasen unterteilt werden, bei der jede vorhergehende Phase essenzielle Grundlage der folgenden Phase ist [Som12, S. 57 f.]. Das V-Modell erweitert diesen Ansatz, indem aus jeder dieser Phasen des V-Modells Testfälle passend zu den jeweiligen Aktivitäten dieser Phase abgeleitet werden [LL10, S. 190].

Üblicherweise werden dabei die folgenden Phasen aus dem Wasserfall- und V-Modell übernommen und daraus resultierende Testfälle unterschieden [AO08, S. 5 f.]. Abbildung 2.3 visualisiert die hierarchische Struktur und das Zusammenspiel der verschiedenen Stufen des V-Modells:

- Anforderungsanalyse & Abnahmetests: Zu Beginn eines jeden Softwareprojekts werden die konkreten Anforderungen in Zusammenarbeit mit dem Auftraggeber herausgearbeitet. Diese Anforderungen werden in einem abschließenden Stadium eines Projekts anhand von Abnahmetests mittels echter Daten von tatsächlichen oder potenziellen Nutzern getestet.
- Systemarchitektur & Systemtest: Beim Systementwurf werden konkrete Designentscheidungen bezüglich der grundlegenden Architektur von Software beschlossen, insbesondere die Zuordnung bestimmter Anforderungen an unterschiedliche Hard- und Softwarekomponenten. Aus dieser Phase resultieren Systemtests, die sämtliche Komponenten des Projekts auf Basis der Architektur testen.
- Systementwurf & Integrationstest: Während der Systementwurf sich auf die grundlegende Architektur fokussiert, werden in der Phase des Subsystementwurfs die konkreten Strukturen und Komponenten der zu entwickelnden Software spezifiziert. Um die Interoperabilität dieser verschiedenen Komponenten zu testen, werden Integrationstests formuliert.
- Software-Design/Implementierung & Modul-/Unit-Tests: Die Implementierung umfasst die konkrete Umsetzung des Softwareentwurfs in Programmcode. Jede der einzelnen Komponenten, die meist unabhängig voneinander entwickelt werden, können mittels Unit-Tests bezogen auf kleine Einheiten wie Packages, Methoden oder Klassen getestet werden. Je nach Komplexität des Softwaresystems können weitere übergeordnete Module oberhalb der kleinsten Einheiten ausgearbeitet werden. Bevor letztlich mehrere Komponenten gemeinsam getestet werden, können einzelne Module vorgeschaltet in Modultests geprüft werden.

Die Umsetzung des stufenbasierten Testens muss jedoch nicht unbedingt anhand der vorgegebenen Stufen des V-Modells erfolgen, sondern erlaubt auch selbstdefinierte Phasenmodelle: Craig und Jaskiel [CJ02, S. 98 f.] führen beispielsweise bestimmte Produktrisiken, personelle oder zeitliche Anforderungen als wesentliche Faktoren bei der Stufenbildung auf.

White Box- & Black Box-Tests

Neben der Fokussierung auf die verschiedenen Phasen im Softwareentwicklungsprozess orientiert sich die Literatur bei der Erstellung von Testfällen in den meisten Fällen an der

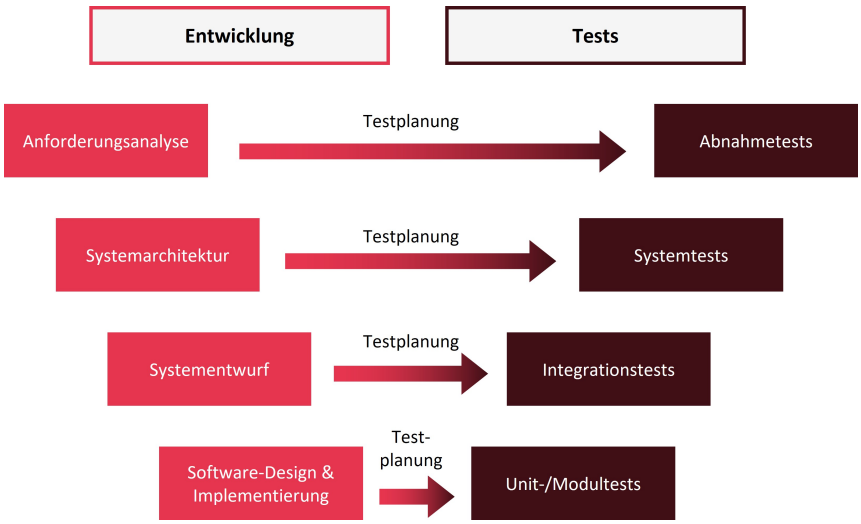

Abbildung 2.3: Das V-Modell dient als Grundlage für das stufenbasierte Testen. Verschiedene Phasen im Entwicklungsprozess einer Software werden mit unmittelbaren Testfällen verknüpft [CJ02, S. 101].

Unterscheidung zwischen Black-Box- und White-Box-Tests. Dabei bezieht sich der Begriff der White-Box beziehungsweise Black-Box darauf, inwiefern die Struktur der zu testenden Softwareeinheit bei der Erstellung der Testfälle bekannt ist.

Black Box-Tests Beim Black-Box-Testen ist die Struktur der zu testenden Softwareeinheit unbekannt, einzig und allein externe Beschreibungen der Software-Spezifikationen dienen als Grundlage für den Testfall [Sch12, S. 91]. Die Spezifikationen können dabei je nach Entwicklungsstadium im Stufenmodell (vgl. Unterunterabschnitt 2.1.1) unterschiedlich aussehen und somit zu verschiedenen Testfällen führen: So basieren Abnahmetests im Wesentlichen auf dem Black-Box-Ansatz, besitzen aber eine ganz andere Struktur als Unit-Tests, die ebenfalls als Black-Box-Tests durchgeführt werden können.

Grundsätzlich soll laut Schneider [Sch12, S. 91] im Rahmen von Black-Box-Tests jede Anforderungen an ein System getestet werden. Meist liegen diese nur in Form von Fließtext oder Tabellen vor und müssen dementsprechend in sinnvolle Testfälle umgewandelt werden [Sch12, S. 92]. Im Falle der Stufe der Anforderungsanalyse und Abnahmetests sind diese schriftlichen Beschreibungen beispielsweise textuelle Erläuterungen der Softwarefunktionalität, bei der Implementierung und Unit-Tests können dies unter anderem Methodenkommentare sein.

Verschiedene Methoden wie beispielsweise die Äquivalenzklassenmethode, die Grenzwertanalyse oder das zustandsbasierte Testen können den Black-Box-Tests zugeordnet werden(vgl. [Sch12, S. 94 ff.]). Details hierzu folgen in Unterabschnitt 2.1.2.

White Box-Tests Im Gegensatz zu den Black-Box-Tests setzen White-Box-Tests, oder auch Glass-Box-Tests genannt, voraus, dass die zu testende Softwareeinheit durchsichtig in ihrer Struktur ist und somit Testfälle auf Basis dessen erstellt werden können [Sch12, S. 91]. Konkret möchte man laut Schneider [Sch12, S. 91] bei White-Box-Tests kritische Stellen im Programmcode oder an den Schnittstellen verschiedener Softwarekomponenten anhand einer Analyse der vorhandenen Softwarestruktur entdecken.

Die meiste Verwendung finden White-Box-Test auf der Modul- und Implementierungsebene. Dort entsteht laut Schneider durch den hohen Grad an individueller Entwicklungsarbeit oftmals eine hohe Komplexität im Softwaresystem, welche zu schwer ermittelbaren Fehlverhalten im Code führen kann [Sch12, S. 108]. Daher soll laut Schneider [Sch12, S. 108 ff.] mit White-Box-Tests im Wesentlichen sichergestellt werden, dass alle geschriebenen Programmteile auch tatsächlich verwendet werden. Dies wird meist mittels Maßen für die Codeüberdeckung umgesetzt: Anweisungsüberdeckung, Zweigüberdeckung und Pfadüberdeckung sind hier die Stichworte, die in der Praxis eine hohe Relevanz besitzen [Sch12, S. 109 ff.].

Abdeckungsbasiertes Testen

Während sich die Literatur in der Vergangenheit bei der Testfallentwicklung im Wesentlichen auf die beiden zuvor aufgeführten Perspektiven der Phasen auf Prozessebene und der Sichtbarkeit der Struktur von Software fokussierte, rückt zunehmend eine weitere dritte Betrachtungsweise in den Fokus der Forschung: Auf Basis von verschiedenen Metriken, welche Abdeckungen bestimmter Eigenschaften des Softwaresystems messen, steht beim abdeckungsbasierten Testen vor allem die Quantifizierung der Güte eines Testverfahrens zur Sicherstellung der Softwarequalität im Fokus [CJ02]. Diese quantitativen Verfahren können unter anderem helfen, verschiedene Stakeholder eines Softwareprojekts von der Güte eines Testbestandes zu überzeugen [KKL+10].

Neben dem Ansatz, durch verlässliche Metriken Vertrauen in eine Menge von Testfällen zu schaffen, basiert das Konzept des abdeckungsbasierten Testens auf der Erkenntnis, dass ein

vollständiges Testen aller denkbaren Zustände, die ein Softwaresystem einnehmen kann, unmöglich ist [CJ02, S. 16 f.]. Konkret führen Craig und Jaskiel [CJ02, S. 16] das Beispiel eines Java Compilers an, dessen Limitierung durch die maximale Kapazität des Compiler-Parsers gegeben ist. Der Java Compiler kann theoretisch jede beliebige Zeichenfolge einer .java-Datei aufnehmen, versucht diese anhand seiner internen Logik zu interpretieren und daraus ein ausführbares Programm in Maschinencode zu erzeugen. Die Länge der Eingabe ist durch die Menge an Zeichen limitiert, die der zugehörige Parser aufnehmen kann. Dies entspricht im Falle von Java bei einer maximalen Dateimenge von 64 kB pro Methode [Dev21] und einer UTF-8-Kodierung mit 1 bis 4 Bytes pro Zeichen mindestens einer Anzahl von 16.000 Zeichen bei voller Auslastung der 64 kB. Berücksichtigt man nun noch alle denkbaren Kombinationen, die durch Permutieren der 144.697 aktuell im Unicode belegten Zeichen [uni] zustande kommen können, so ergibt sich eine Mindestanzahl von $16.000^{144.697}$ verschiedenen Kombinationen, die ein Java-Parser berücksichtigen müsste: Eine Dimension, die kein Rechner auf dieser Welt testen kann.

Dementsprechend können Testfälle meist nur einen Anteil aller denkbaren Kombinationen abdecken, sodass es sehr hilfreich ist, die Qualität einer Menge an Testfällen beurteilen zu können: Dies wird beim abdeckungsbasierten Testen durch die Nutzung von quantifizierbaren Metriken vorgenommen [CJ02, S. 17]. Jeder Testfall kann bei seiner Durchführung ein Testkriterium erfüllen oder auch nicht, sodass sich über die gesamte Menge aller zusammenhängender Testfälle ein Anteil ergibt, der die Erfüllung des Testkriteriums im Gesamten quantifiziert [CJ02, S. 17].

Konkret bedeutet dies: Für jedes Abdeckungskriterium lässt sich bestimmen, welche Eigenschaften die Menge aller Testfälle benötigt, um das Kriterium vollständig zu erfüllen. Bei Combinatorial Testing sind dies beispielsweise alle t-fachen Kombinationsabdeckungen (vgl. Unterunterabschnitt 2.1.1) der Eingabevariablen. Die Quantifizierung der Güte der Menge aller Testfälle ergibt sich schließlich durch die Quote der erfüllten Eigenschaften, die das Abdeckungskriterium fordert. Jeder Testfall kann dann seinen Teil dazu beitragen, dass das betroffene Kriterium erfüllt wird. Dabei kann es passieren, dass mehrere Testfälle den 'gleichen' Beitrag im Sinne des Testkriteriums liefern. In anderen Worten: Manche Testfälle sind überflüssig, da die geforderte Eigenschaft des Abdeckungskriteriums bereits durch einen anderen Testfall oder die Kombination mehrerer Testfälle abgedeckt wird.

Craig und Jaskiel [CJ02] unterscheiden vier wesentliche Kategorien von Abdeckungen im Bereich des dynamischen Softwaretestens.

- Graphen-basierte Abdeckung orientiert sich an verschiedenen Graphstrukturen, die sich im Rahmen eines Softwarekonzepts ergeben wie beispielsweise die Verzweigungen und Wiederholungen innerhalb des Programmcodes. Dabei können Metriken wie Knoten- oder Kantenabdeckungen zur Quantifizierung der Abdeckung herangezogen werden [CJ02, S. 27 ff.].

- Logische Abdeckung basiert auf den Prinzipien der Prädikatenlogik, die sich analog zu Graphstrukturen auf verschiedene Bereiche eines Softwaresystems anwenden lassen, unter anderem die Verzweigungen innerhalb des Programmcodes oder die Umformulierung von Anforderungsspezifikation in Wenn-dann-Verbindungen [CJ02, S. 131 ff.]. Die Abdeckung der daraus abgeleiteten Testfälle können mit Methoden der Prädikatenlogik wie beispielsweise der Klausel- und Prädikatenabdeckung gemessen werden [CJ02, S. 106 ff.].

- Die Abdeckung der möglichen Eingabewerte berücksichtigt die Grundvoraussetzung jedes Softwaresystems: Jedes Programm verwendet in gewisser Weise Eingaben, sodass alle beliebigen Softwaretests auch Elemente des Eingabeuniversums verwenden [CJ02, S. 150]. Insbesondere dann, wenn die Struktur einer Software unbekannt ist, stellt die Analyse der möglichen Eingaben und Ausgaben gemeinsam mit der Anwendung von Erfahrungswerten die einzige Möglichkeit zum Testen von Softwareeinheiten dar. Im Vergleich zu den anderen abdeckungsbasierten Ansätzen stehen beim expliziten Testen der Eingabewerte die kombinatorischen Eingabemöglichkeiten im Vordergrund [CJ02, S. 150 ff.]. Combinatorial Testing ist dabei die relevanteste Methode zur Umsetzung dieser Strategie, welche im Folgenden in Abschnitt 2.2 ausführlich vorgestellt wird.

- Syntax-basierte Abdeckung verwendet die Ideen der Automatentheorie und benutzt dabei syntaktische Beschreibungen von zu testenden Softwareeinheiten, um Testfälle abzuleiten [CJ02, S. 170 ff.]. Im Zusammenhang mit Programmcode spielt vor allem der Begriff der Mutationen eine wichtige Rolle, bei dem elementare Codestücke durch leichte Veränderungen ersetzt werden und so Fehlverhalten provoziert wird. Als Metrik werden dabei zumeist die Abdeckung der Symbole und Produktionsregeln und weitere komplexere Kombinationen verwendet [CJ02, S. 172].

2.1.2 Beispiele für Teststrategien

Der folgende Abschnitt greift die grundlegende Herangehensweise in Bezug auf das dynamische Softwaretesten aus Unterabschnitt 2.1.1 auf und führt einige konkrete Beispiele für Teststrategien ein. Diese Beispiele beschränken sich auf mögliche Anwendungen im Zusammenhang mit dem in Kapitel 3 vorgestellten Anwendungsfall. Da dieser sich im Wesentlichen auf einen spezifikationsbezogenen Ansatz fokussiert, werden insbesondere die Techniken des White-Box-Testens (vgl. Unterunterabschnitt 2.1.1) an dieser Stelle ausgeklammert. Detaillierte Ausführungen zu den verbreiteten White-Box-Strategien der Anweisungs-, Zweig- und Pfadüberdeckung lassen sich in [Sch12, S. 108 ff.] nachlesen.

Da der Fokus dieser Arbeit auf Combinatorial Testing liegt, werden zudem die Prinzipien dieser Teststrategie nicht an dieser Stelle, sondern im folgenden, separaten Abschnitt 2.2 vorgestellt.

Äquivalenzklassenmethode

Da vollständiges Testen - wie zuvor erläutert - nicht möglich ist, erfordern systematische Tests Einschränkungen bei der Wahl der Parameter. Die Äquivalenklassenmethode löst dieses Problem derart, dass alle denkbaren Eingaben in verschiedene Äquivalenzklassen partitioniert werden [Sch12, S. 94]. Dabei sollten laut Schneider [Sch12, S. 94 ff.] die unterschiedlichen Klassen derart konstruiert werden, dass Vertreter aus derselben Klasse sich in Bezug auf das Fehlerverhalten innerhalb der Software identisch verhalten. Als Beispiel führt Schneider eine automatisierte Schwimmbadkasse auf, die für Jugendliche unter 18 Jahre einen vergünstigten Preis anbietet. In diesem Fall würde sich eine Äquivalenzklasse für die Variable 'Alter' durch die Menge 'Jugendlich' $= \{n < 18 \mid n \in \mathbb{N}\}$ und eine weitere Klasse 'Erwachsen' $= \{n \geq 18 \mid n \in \mathbb{N}\}$ ergeben.

Je nach Spezifikation und denkbaren Eingaben müssen die Äquivalenzklassen auch mögliche Falscheingaben berücksichtigen. Eine wesentliche Herausforderung bei Anwendung der Äquivalenzklassenmethode besteht laut Schneider darin, eine passende Auswahl der Klassen zu treffen, die einerseits nicht zu kleinteilig, andererseits aber auch nicht zu pauschalisierend sein sollte [Sch12, S. 95].

Grenzwertanalyse

Die Grenzwertanalyse basiert auf der Annahme, dass es häufig unmittelbar an den Grenzen verschiedener Eingabeparameter zu Fehlverhalten kommt, da diese oftmals nicht genau definiert sind oder von Programmierenden nicht berücksichtigt werden [SLS11, S. 120]. Dementsprechend überprüft die Grenzwertanalyse die Grenzen kritischer Werte und stellt somit eine Erweiterung der Äquivalenzklassenmethode dar [SLS11, S. 120 f.].

Für jede Äquivalenzklasse ergeben sich laut Spillner et al. [SLS11, S. 120 f.] automatisch kritische Grenzen, die mittels der Grenzwertanalyse abgedeckt werden können. Im Beispiel der Schwimmbadkasse würde dies bedeuteten, dass ein besonderes Augenmerk auf die Werte 17 und 18 gelegt werden sollte. Neben diesen offensichtlichen Grenzen der verschiedenen Äquivalenzklassen werden im Rahmen der Grenzwertanalyse häufig auch extreme, unerwartete Werte wie beispielsweise die maximal oder minimal mögliche Eingabe berücksichtigt werden [SLS11, S. 122]. Sowohl die Grenzwertanalyse als auch die Äquivalenzklassenmethode lässt sich auf allen Ebenen des Stufenmodells (vgl. Unterunterabschnitt 2.1.1) anwenden und ist in Bezug auf das abdeckungsbasierte Testen vor allem im Bereich der Abdeckung der Eingaben relevant.

Zustandsbasiertes Testen

Zustandsbasiertes Testen basiert auf der grundlegenden Annahme, dass sich jeder deterministische Programmfluss einer Software durch einen endlichen Automaten mit verschiedenen Zuständen und exakt definierten Zustandsübergängen darstellen lässt. Die Darstellung eines endlichen Automaten kann über ein Graph-basiertes Modell (vgl. Unterunterabschnitt 2.1.1) oder über eine Zustandsübergangstabelle erfolgen. Als Teststrategie eines Zustandsautomaten ergibt sich die Methode, dass jede einzelne Zelle der Zustandsübergangstabelle einen Testfall ergibt. Dadurch, dass die Transition der betroffenen Zustände im endlichen Automat eindeutig definiert sind, fällt die Ableitung der notwendigen Werte betroffener Parameter leicht.

Das zustandsbasierte Testen spielt insbesondere dann eine wichtige Rolle, wenn Systeme verschachtelte Hierarchien besitzen. Im Anwendungsfall dieser Arbeit (vgl. Kapitel 3) wird dies insbesondere bei Eingabemasken relevant, die nur durch die bestimmte Auswahl eines spezifischen Parameters zum Einsatz kommen und in allen anderen Fällen nicht.

(Adaptive) Random Testing

In einigen Fällen möchte man eine Softwareeinheit testen, bei der weder eine detaillierte Beschreibung der Spezifikation, noch der Programmcode selbst vorliegt. Dafür bietet sich die Verwendung des Zufallsprinzips an, welche die Grundlage des Random Testings ist. Dabei wird für das Eingabeuniversum eine Wahrscheinlichkeitsverteilung angenommen, aus welcher Eingabewerte zufällig entnommen werden [SLS11, S. 141 f.]. Grundsätzliche Vorteile einer zufallsbasierten Teststrategie ergeben sich laut Spillner [SLS11, S. 142] vor allem durch eine realitätsnahe Abbildung der tatsächlich verwendeten Werte der Softwareeinheit und durch die Möglichkeit, statistische Methoden zur Quantifizierung der Systemzuverlässigkeit anwenden zu können.

Unter anderem White und Cohen [WC80] konnten herausfinden, dass Fehlverhalten meist sehr gebündelt in Bereichen von Eingabewerten auftreten, sodass bereits durchgeführte Testfälle, die kein Fehlverhalten aufdeckten, möglichst weit im Eingabeuniversum von neuen Testfällen entfernt sein sollten [ABC+13]. Aus diesem Grund entwickelten Chen et al. [CLM04] die Methode des Adaptive Random Testing, welche versucht, die Testfälle möglichst weit und gleichmäßig über das Eingabeuniversum zu verteilen. Verschiedene Strategien und Algorithmen auf Basis von Distanzmaßen, Ausschlussverfahren oder evolutionären Algorithmen wurden in der Vergangenheit entwickelt, um Adaptive Random Testing in der Praxis umzusetzen [HXCL12].

Erfahrungsbasiertes Testen

Neben den aufgeführten systematischen Ansätzen Testfälle zu Erzeugung existiert eine weitere Methode zur Testfallerzeugung, die in der Anwendungspraxis eine bedeutende Rolle spielt: Erfahrungsbasiertes Testen kann vor allem Fehler aufdecken, die systematische Ansätze übersehen [SL19, S. 210]. Das Wissen und die Expertise der testenden Person bildet dabei die Grundlage, um mögliche Probleme innerhalb einer Softwareeinheit mit Tests aufzudecken. Spillner et al. [SL19, S. 213] betonen, dass erfahrungsbasiertes Testen nicht als erstes Mittel der Wahl bei der Erstellung von Testfällen verwendet werden sollte, sondern vielmehr als Ergänzung anderer systematischer Ansätze zu verstehen ist.

Insbesondere interagiert das erfahrungsbasierte Testen häufig mit anderen konkreten Testmethoden wie beispielsweise der Äquivalenzklassenmethode oder der Grenzwertanalyse: So stellt die Wahl der Äquivalenzklassen beziehungsweise die Wahl kritischer Grenzwerte eine

wesentliche Herausforderung der beiden Testmethoden dar, sodass an dieser Stelle Erfahrungswerte und Expertise besonders hilfreich sein können.

Erfahrungsbasiertes Testen lässt sich nur teilweise den drei zuvor aufgeführten Grundprinzipien des Testens zuordnen und kann selbst nur als 'semi-systematisches' Verfahren bezeichnet werden: Die Methode lässt sich nicht eindeutig als Black Box- oder White Box-Technik einstufen [SL19, S. 213], auch ein Abdeckungskriterium existiert, welches die Güte von erfahrungsbasierten Tests messen kann, existiert in den meisten Fällen nicht. Die Expertise erfahrenerer Tester*innen kann auf allen Ebenen des stufenbasierten Testens nützlich sein, kommt aber meist auf höheren Teststufen zum Einsatz, da auf den niedrigeren Ebenen genügend Informationen über die Spezifikationen, wie beispielsweise der Programmcode selbst, zur Verfügung stehen [SL19, S. 213].

In der Praxis haben sich in Bezug auf erfahrungsbasiertes Testen verschiedene Teilaspekte herausgebildet [SL19, S.210 ff.]:

- 'Error Guessing' (deutsch: Fehlerraten) basiert auf der Idee, Fehler aus der Vergangenheit oder in Zukunft zu erwartende Fehler bei der Testfallerstellung zu berücksichtigen [SL19, S.210 f.].
- Checklistenbasiertes Testen verwendet eine Checkliste zur Testfallerzeugung, die eine erfahrene Testperson in der Vergangenheit angelegt hat und die als Anleitung für zukünftige Testfälle dient [SL19, S. 211 f.]. In diesem Fall kann ein Abdeckungskriterium für die Güte einer Menge von Testfällen definiert werden, indem der Anteil der erfüllten Aspekte der Checkliste ermittelt wird [SL19, S. 211].
- Exploratives Testen ist ein Verfahren, das einen kontinuierlichen Prozess bestehend aus Testfallerzeugung und Auswertung jener Testfälle beschreibt [SL19, S. 211]. Über die Zeit hinweg entwickelt laut Spillner und Linz [SL19, S. 212] die testende Person zunehmend mehr Wissen und Verständnis über das Testobjekt und kann dieses für neue, verbesserte Testfälle anwenden. Darüber hinaus kann das Wissen des explorativen Testens angewendet werden, um systematische Tests zu entwickeln.

2.2 Combinatorial Testing

Combinatorial Testing (deutsch: Kombinatorisches Testen) ist analog zu den Beispielen aus Unterabschnitt 2.1.2 eine Strategie zur systematischen Erzeugung von Testfällen. Da

diese Methode in dieser Arbeit im Mittelpunkt steht, wird diese nun an dieser Stelle ausführlicher als die zuvor aufgeführten Teststrategien vorgestellt. Zunächst sollen dabei die grundlegende Motivation für Combinatorial Testing und die wesentlichen Prinzipien im Fokus stehen. Anschließend werden verschiedene Metriken zur Quantifizierung der Güte eines Testbestandes im Zusammenhang von Combinatorial Testing erläutert und verschiedene Algorithmen und Tools zur Anwendung von Combinatorial Testing vorgestellt.

2.2.1 Einführung in Combinatorial Testing

Als wesentliche Motivation für die Verwendung kombinatorischer Methoden bei der Erzeugung von Testfällen ergibt sich wie bei fast allen Testfallstrategien auch die Erkenntnis, dass vollständige Tests in der Realität nicht umsetzbar sind (vgl. Abschnitt 2.1). Combinatorial Testing greift diese Tatsache auf und versucht auf effiziente Art und Weise das Eingabeuniversum einer zu testenden Softwareeinheit möglichst sinnvoll mittels kombinatorischer Methoden im Sinne von quantifizierbaren Metriken (vgl. Unterunterabschnitt 2.1.1) abzudecken. Dabei kann Combinatorial Testing auf allen Ebenen des in Unterunterabschnitt 2.1.1 aufgeführten Stufenmodells zum Einsatz kommen, wie verschiedene Beispiele in diesem Abschnitt aufzeigen werden. Zudem lässt sich Combinatorial Testing als Black-Box-Technik einstufen, da die Struktur der zu testenden Softwareeinheit für den Testablauf irrelevant ist.

Grundannahme beim Combinatorial Testing ist es, dass selten die Eingaben einzelner Parameter eines Testprüflings für Fehler verantwortlich sind, sondern vielmehr die Interaktion einiger weniger Parameter zu häufigen Fehlern führt [KKL+10]. Kuhn et al. [KWG04] konnten in einer Analyse verschiedener Software-Systeme aufzeigen, dass die Interaktion von sechs oder weniger Parameter für annähernd 100 Prozent der auffindbaren Softwarefehler verantwortlich sind. Abbildung 2.4 zeigt ein einfaches Beispiel aus [KKL+10], bei welchem nur die Kombination aus den Parametern 'Druck' < 10 und 'Volumen' > 300 zu einem Fehlverhalten führt. Bei einer zufälligen Wahl der Testparameter könnte es dazu kommen, dass dieses Fehlverhalten nicht aufgedeckt wird.

Um diesem Problem zu begegnen, hat sich laut Kuhn et al. [KKL+10] in der Anwendungspraxis das sogenannte Pairwise-Testing etabliert, bei dem alle Kombinationen von Paaren der Eingabeparameter getestet werden. Im Beispiel von Abbildung 2.4 bedeutet dies, dass alle vier paarweise Kombinationen aus 'Druck' < 10, 'Druck' ≥ 10, 'Volumen' > 300, 'Volumen' ≤ 300 getestet werden sollten. Der Ansatz des Combinatorial Testing

```
1                    if ('Druck' < 10) {
2                        // do something
3                        if ('Volumen' > 300) {
4                            // faulty code! BOOM!
5                        } else {
6                            // good code, no problem
7                        }
8                    } else {
9                        // do something else
10                   }
```

Abbildung 2.4: Beispiel für die Interaktionsproblematik beim Testen verschiedener Parameter: Nur bei der Kombination 'Druck' < 10 und 'Volumen' > 300 kommt es zu einem Fehlverhalten [KKL+10].

greift diese Idee auf und erweitert die Idee des Pairwise-Testing derart, dass die Zahl des zu prüfenden Interaktionsparameters variabel ist [KKL+10]. Konkret entspricht ein Combinatorial Testing-Ansatz mit Interaktionsparameter $t = 2$ dem Pairwise-Konzept, bei $t = 3$ muss jede denkbare Dreierkonstellation der Eingabeparameter mindestens in einem Testfall berücksichtigt werden.

Aus mathematischer Sicht lassen sich daraus eine Mindest- und Maximalanzahl durchzuführender Tests ($T_{\min}$, $T_{\max}$) anhand der Wahl des Interaktionsparameter t, der Anzahl n aller Eingabeparameter x_i und der Anzahl y_i der verschiedenen Werte, die jeder Parameter x_i annehmen kann, ermitteln: Unter der Annahme, dass der Interaktionsparameter t größer als die Anzahl n der vorhandenen Parameter ist, stellt das Produkt der t größten Werte y_i eine untere Schranke für die durchzuführenden Tests dar. Seien also die y_i derart geordnet, dass $y_1 \geq y_2 \geq \cdots \geq y_n$. Dann gilt:

$$T_{\min} = \prod_{i=1}^{t} y_i \tag{2.1}$$

Eine obere Schranke für die Anzahl der Tests ist unterdessen die Anzahl aller denkbaren Kombinationen der Eingabeparameter:

$$T_{\max} = \prod_{i=1}^{n} y_i \tag{2.2}$$

Eine große Spannweite zwischen $T_{\min}$ und $T_{\max}$ entsteht insbesondere dann, wenn die Differenz zwischen der Anzahl der Parameter n und des Interaktionsparameters t groß wird. Im Gegensatz dazu liegen $T_{\min}$ und $T_{\max}$ nah beisammen, wenn n und t beinahe identisch sind. Combinatorial Testing kann seine Vorteile vor allem dann ausspielen, wenn die Differenz zwischen $T_{\min}$ und $T_{\max}$ sehr groß ist: Durch die Fokussierung auf die Interaktion von t Parametern können in einem Testfall mehrere Parameterkombinationen gleichzeitig abgedeckt werden, was die Anzahl der tatsächlich benötigten Tests in Relation zu $T_{\max}$ sehr nahe an der unteren Schranke $T_{\min}$ hält.

Das in Tabelle 2.1 aufgeführte Beispiel aus [KKL+10] verdeutlicht dieses Prinzip: Die Tabelle zeigt eine mögliche Reihe an Testfällen für $n = 10$ binäre ($y_i = 2\ \forall i = 1, \ldots, n$) Eingabevariablen A bis J bei einer Abdeckung aller Dreierkombinationen, also $t = 3$. Eine derartige Tabelle mit einer vollständigen Abdeckung aller t-fachen Kombinationen der Eingabeparameter nennt man im Kontext des Combinatorial Testing auch Covering Array oder Orthogonal Array.

Würde man alle möglichen Kombinationen der 10 Variablen abdecken wollen, würde sich eine Anzahl von $T_{max} = 2^{10} = 1024$ Testfällen ergeben. Dadurch, dass nur die Kombinationen aus drei Parametern berücksichtigt werden müssen, kann die Anzahl der Testfälle auf 13 reduziert werden. Beispielhaft wird in den drei unterschiedlichen roten Farbtönen anhand der Parameterkombinationen A-B-C, D-E-G und H-I-J aufgezeigt, inwiefern mit einem Testfall mehrere Konstellationen abgedeckt werden. Dieser Ansatz ließe sich auf alle $\binom{n}{t} = \binom{10}{3} = 120$ Parameterkombinationen ausweiten, sodass die 13 Testfälle in Tabelle 2.1 alle Binärkonstellationen aus drei Variablen beinhalten. Es zeigt sich also, dass in diesem Fall die tatsächlich benötigte Anzahl an Testfällen nahe an der unteren Schranke $T_{\min} = 2^3 = 8$ liegt, insbesondere im Vergleich zu $T_{\max}$.

Im Allgemeinen konnten Cohen et al. [CDFP97] aufzeigen, dass bei n verschiedenen Variablen, die jeweils y verschiedene Werte annehmen können, die Anzahl der notwendigen Tests T zur Abdeckung aller t-Kombinationen proportional zur folgenden Größe wächst:

$$T \sim y^t \cdot \log n \tag{2.3}$$

Konfigurationstests & Tests der Eingabewerte Combinatorial Testing wird im Allgemeinen in zwei Teildisziplinen unterschieden, die beide auf die zuvor aufgeführten kombinatorischen Prinzipen zurückgreifen [KKL+10]. Konfigurationstests fokussieren sich auf die

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Tabelle 2.1: Ein Covering Array für die Kombination aller Dreierkombinationen bei der Wahl von 10 binären Eingabevariablen A-J. Die verschiedenen Rotfärbungen zeigen die vollständigen Abdeckungen der Variablenkombinationen A-B-C, D-E-G und H-I-J [KKL+10]

verschiedenen Möglichkeiten, die ein Softwaresystem in Bezug auf Konfigurationsparameter einnehmen kann und welche Wechselwirkungen bei der Kombination dieser Parameter auftreten können [KKL+10]. Typischerweise sind damit laut Kuhn et al. [KKL+10] Softwaresysteme gemeint, die auf verschiedenen Betriebssystemen, verschiedenen Browsern oder unter verschiedenen Standards agieren müssen und somit eine hohe Interoperabilität erfordern.

Im Gegensatz dazu orientieren sich die Tests der Eingabewerte darauf, welche konkrete Parameter in eine spezifische Komponente oder ein Softwaresystem eingegeben werden können [KKL+10]. Dies können beispielsweise die Auswahlfelder einer Eingabemaske wie im Anwendungsfall in Kapitel 3 oder die möglichen Variablen einer Methode wie in Abbildung 2.4 betreffen.

Eine besondere Herausforderung beim Testen der Eingabewerte besteht laut Kuhn et al. [KKL+10] darin, dass Eingabeparameter meist sehr viele unterschiedliche Werte annehmen können und oftmals auch Eingaben aus kontinuierlichen Zahlenbereiche möglich sind. Da die Anzahl der Tests mit der Anzahl an möglichen Werten für jede Variable wächst (vgl. Gleichung 2.3), würde die Berücksichtigung aller möglichen Eingaben in solchen Fällen zu einer unüberschaubaren Menge an Testfällen führen.

Um diesem Problem zu begegnen, sollten laut Kuhn et al. [KKL+10] unter anderem die Strategien der Äquivalenzklassenbildung und der Grenzwertanalyse (vgl. Unterabschnitt 2.1.2) als Methoden zur Reduzierung der Testfälle herangezogen werden. Insgesamt sollte die Anzahl verschiedener Werte beziehungsweise Klassen pro Variable unter zehn bleiben [KKL+10]. Da die Anzahl der verschiedenen Parameter n nur logarithmisch in das Wachstum der Testfälle einfließt (vgl. Gleichung 2.3), ist dieser Wert im Vergleich weniger kritisch zu betrachten [KKL+10].

2.2.2 Maße für Testabdeckung

Als einer der vier hauptsächlichen Vertreter des abdeckungsbasierten Testens (vgl. Unterunterabschnitt 2.1.1) erfordert das Testen der Eingabewerte wie im Falle von Combinatorial Testing Metriken, welche die Güte einer Menge von Testfällen bestimmen kann.

Im Folgenden werden die wesentlichen Metriken vorgestellt, die sich im Zusammenhang mit kombinatorischen Testmethoden etabliert haben. Die ersten beiden Metriken sind dabei eher rein theoretischer Natur und spielen bei der praktischen Anwendung von Combinatorial Testing eine untergeordnete Rolle. Die Ausführungen werden durch folgendes Beispiel, angelehnt an Craig und Colesky [CJ02, S. 160 ff.], mit drei verschiedenen Variablen a, b, c, den möglichen Werten $a = \{A, B\}, b = \{1, 2, 3\}, c = \{x, y\}$ und der folgenden beispielhaften Menge T an Testfällen bekräftigt:

Vollständige Kombinationsabdeckung Auch wenn vollständiges Testen in verschiedener Hinsicht meist nicht durchführbar ist (vgl. Unterabschnitte 2.1.1) lässt sich anhand der Idealvorstellung eines vollständigen Tests eine Metrik im kombinatorischen Sinne ableiten: Die vollständige Kombinationsabdeckung bezieht sich auf alle denkbaren Kombinationen der Eingabeparameter und besitzt damit die obere Schranke für Combinatorial Testing $T_{\max}$ als Bezugsgröße [CJ02, S. 160]. Im Beispiel der Variablen a, b, c wären dies alle $2 \cdot 3 \cdot 2 =$

a	b	c
A	1	x
A	2	x
A	3	y
B	1	y
B	3	x

Tabelle 2.2: Testbestand T

12 denkbaren Konstellationen. Die vollständige Kombinationsabdeckung des Testbestands T liegt demnach bei $\frac{5}{12} \approx 42\%$.

Einfache Variablenabdeckung Die einfache Variablenabdeckung markiert in gewisser Weise die gegensätzliche Extreme zur vollständigen Abdeckung aller möglichen Kombinationen bei der vollständigen Kombinationsabdeckung: Bei der einfachen Variablenabdeckung ergibt sich die Bezugsgröße zur Bestimmung der Testfallabdeckung dadurch, dass für jede Variable isoliert betrachtet jeder mögliche Wert mindestens einmal vorkommen sollte [CJ02, S. 160 f.]. Der Testbestand T erfüllt dieses Kriterium offensichtlich zu 100%, bereits die drei Testfälle $(A, 1, x), (B, 2, y), (A, 3, x)$ würden dafür ausreichen [CJ02, S. 160 f.].

Base Choice-Abdeckung Beim Ansatz der Base-Choice-Abdeckung wird laut Craig und Jaskiel [CJ02, S. 162] für jeden Eingabeparameter eine Basiswahl festgelegt, anhand derer ein Basistestfall erstellt wird. Zusätzliche Testfälle sollten darüber hinaus durch Verändern eines einzelnen Parameters und Festhalten der Basiswerte aller anderen Parameter erzeugt werden. Im Beispiel der Variablen a, b, c könnte man beispielsweise die Kombination $(A, 1, x)$ als Basiswahl festlegen. Dann enthält eine vollständige Base-Choice Testabdeckung die Testfälle $(A, 1, x)$, $(B, 1, x)$, $(A, 2, x)$, $(A, 3, x)$, $(A, 1, y)$ [CJ02, S. 162]. Tatsächlich vorhandene Testfälle in der Menge T sind nur die Basiswahl selbst und $(A, 2, x)$, sodass eine Testabdeckung von $\frac{2}{5} = 40\%$ erreicht wird.

t-fache Kombinationsabdeckung Die t-fache Kombinationsabdeckung orientiert sich am Grundprinzip des Combinatorial Testing, welches in Unterabschnitt 2.2.1 vorgestellt

wurde: Alle Kombinationen von t verschiedenen Variablen sollten idealerweise abgedeckt werden [KKL+10]. So entspricht die t-fache Kombinationsabdeckung dem Anteil der t-fachen Kombinationen, die eine Menge von Testfällen abdeckt [KKL+10]. Tabelle 2.1 zeigt eine Menge von Testfällen, die eine vollständige 3-fache Kombinationsabdeckung erfüllen.

Wird eine Menge an Testfällen, wie in Unterabschnitt 2.2.1 beschrieben, auf Grundlage des Interaktionsparameters t erstellt, erfüllt diese automatisch zu 100 % die t-fache Kombinationsabdeckung [KKL+10]. Für $t = 2$ spricht man auch von der paarweisen Kombinationsabdeckung, da dort alle Werte der Eingabevariablen paarweise kombiniert werden [KKL+10]. Im Beispiel der Variablen a, b, c existieren für $t = 2$ drei verschiedene 2-fach-Kombinationen $\{(a, b), (a, c), (b, c)\}$ von denen im Testbestand T lediglich die Paarung (a, c) alle möglichen Optionen beinhaltet. Bei (a, b) fehlt die Option $(B, 2)$ und bei (b, c) die Option $(2, y)$. Dies entspricht einer Abdeckung von $\frac{2}{3} \approx 66,7\%$.

Allgemein entspricht die Anzahl der zu prüfenden t-fachen Kombinationen K_t dem Binomialkoeffizienten aus der Anzahl der Eingabeparameter und dem Interaktionsparameter t:

$$K_t = \binom{n}{t}$$

Um festzustellen, ob eine der $\binom{n}{t}$ Parameterkombinationen vollständig in einem Testdatensatz vorhanden ist, müssen alle Variablen-Wert-Konfigurationen für jeden Parameter geprüft werden. Beim Testbestand T sind dies genau 16 Stück. Die Berechnung der Anzahl all jener Variablen-Wert-Konfigurationen wird im folgenden Abschnitt zur Variablen-Wert-Konfiguration aufgegriffen.

$(t + k)$-fache Kombinationsabdeckung Jede Menge an Testfällen, die einen hohen Anteil einer t-fachen Kombinationsabdeckung aufweist, wird auch einen gewissen Anteil einer $(t + k)$-fachen Kombinationen abdecken [KKL+10]. Insbesondere dann, wenn verschiedene Mengen von Testfällen auf Basis einer t-fachen Kombinationsabdeckung erstellt wurden und somit in dieser Hinsicht eine Abdeckung von 100% erfüllen, wird laut Kuhn et al. [KKL+10] mittels der$(t + k)$-fachen Kombinationsabdeckung eine Vergleichbarkeit ermöglicht: Neben der absoluten Anzahl an Testfällen, welche zur vollständigen Abdeckung vonnöten sind, können unter anderem die $(t+1)$-fache oder $(t+2)$-fache Kombinationsabdeckung als Vergleichskriterium zwischen zwei Mengen von Testfällen herangezogen werden.

Der Testbestand T besitzt wie zuvor aufgeführt eine 2-fache Kombinationsabdeckung von ungefähr 67,7 %. Prüft man nun die Abdeckung bezüglich $(t+1) = 3$ ergibt sich eine Abdeckung von 0%, da die einzig vorhandene Dreierkonstellation (a, b, c) offensichtlich nicht vollständig vorhanden ist.

Variablen-Wert-Konfigurationsabdeckung Bei der Betrachtung der t-fachen Kombinationsabdeckung wird für jeden Eingabeparameter x_i lediglich geprüft, ob alle t-fachen Kombinationen abgedeckt sind und dabei nicht berücksichtigt, wie viele Kombinationen zu einer vollständigen Abdeckung im Hinblick auf die jeweiligen Variablen x_i fehlen würden: Die Variablen-Wert-Konfigurationsabdeckung greift laut Kuhn et al. [KKL+10] diese Problematik auf und prüft für jede t-fache Variablenkombination den Anteil der abgedeckten Kombinationsmöglichkeiten. Wird beispielsweise $t = 2$ gewählt, existieren bei binären Eingabeparametern vier verschiedene Kombinationsmöglichkeiten für jedes Paar an Parameter.

Im konkreten Beispiel bedeutet dies, dass im Testbestand T für $t = 2$ insgesamt $K_t = \binom{n}{t} = 3$ verschiedene Parameterpaarungen existieren und damit $2 \cdot (3+2) + 3 \cdot 2 = 16$ verschiedene Variablen-Wert-Konfigurationen abzuprüfen sind. Die vier Kombinationen von (a, c) sind vollständig vorhanden, bei (a, b) und (b, c) fehlt jeweils eine der sechs Optionen. Dementsprechend liegt die Variablen-Wert-Konfiguration hier bei $\frac{14}{16} \approx 87,5\%$. Zum Vergleich: Der Wert der 2-fachen Kombinationsabdeckung liegt bei 67,7%.

Seien wie in Abschnitt 2.2 n die Anzahl der verschiedenen Parameter x_i, y_i die zu einem Eingabeparameter x_i gehörende Anzahl an verschiedenen Werten und t der Interaktionsparameter. Allgemein lässt sich dann die Anzahl aller möglichen Variablen-Wert-Kombinationen V_t folgendermaßen bestimmen:

$$V_t = \sum_{i=1}^{n-t+1} y_i \cdot \sum_{Z \in U_i} \prod_{z \in Z} z \quad \text{wobei}$$

$$U_i = \{U \mid U \in \mathcal{P}(Y_i) \wedge |V| = t-1\} \quad \text{und}$$

$$Y_i = \{y_j \mid x_j \text{ ist Eingabeparameter mit } y_j \text{ verschiedenen Werten} \wedge i < j\}$$

Diese Berechnungsvorschrift ist wie folgt zu verstehen: Es wird über die Eingabewerte x_i iteriert und dabei für jeden Parameter die verbleibende Restanzahl an t-fachen Kombina-

tionen ermittelt. Der Abbruch der Iterationen erfolgt bereits bei $n-t+1$, da ab dort keine weiteren Kombinationen möglich sind.

Konkret entspricht Y_i der Menge, welche die Anzahl der unterschiedlichen Werte y_i der noch nicht bearbeitenden x_i beinhaltet - also diejenigen x_j für die $j > i$ gilt. Im Beispiel der Variablen a, b, c wäre dies bei der Wahl von $x_1 = a$ die Menge $\{2, 3\}$. Aus dieser Menge werden nun in U_i alle Kombinationen, die gemeinsam mit dem aktuell ausgewählten x_i eine t-fache Kombination darstellen können, ausgewählt. Dies geschieht über die Bildung der Potenzmenge und der Prüfung, ob die Mächtigkeit einer Teilmenge $t-1$ entspricht. x_i selbst sorgt dafür, dass aus der Zahl $t-1$ eine t-fache Kombination folgt. Jede Menge Z in U_i beinhaltet dann $t-1$ Zahlen, welche für die mögliche Anzahl verschiedener Parameterwerte stehen. Gemeinsam mit den y_i möglichen Eingabewerten für den 'Haupteingabeparameter' zu diesem Zeitpunkt, x_i, ergeben sich dann für die betroffene Menge Z genau $y_i \cdot \prod_{y \in Z} y$ t-fach-Kombinationen aus x_i und den x_j, die zu den $y_j \in Z$ gehören. Abschließend gilt es die Summe über alle Kandidaten Z in U_i zu bilden, ehe die nächste Iteration folgt.

Über die Variablen-Wert-Konfiguration hinaus definieren Kuhn et al. [KKL+10] die (p, t)-Vollständigkeit als ein weiterführendes Maß zur Quantifizierung der kombinatorischen Testgüte. Die (p, t)-Vollständigkeit wird dabei definiert als der Anteil der $K_t = \binom{n}{t}$ Parameterkombinationen, deren Variablen-Wert-Konfigurationsabdeckung mindestens p überschreiten.

Im Testbestand T fehlen in Bezug auf $t = 2$ einzig bei den Parameterpaaren (a, b) und (b, c) eine der jeweils sechs Kombinationen, sodass diese Kombinationen für $p \leq \frac{5}{6}$ die Anforderung an die (p, t)-Vollständigkeit erfüllen. Zudem erfüllt die Kombination (a, c) aufgrund der vollständigen Variablen-Wert-Abdeckung für jedes p die (p, t)-Vollständigkeit. Wählt man also beispielsweise $p = 75\%$, liegt die (p,t)-Vollständigkeit des Testbestands T insgesamt bei 100%, da alle drei Parameterpaare die Variablen-Wert-Konfigurationsabdeckung von 75% überschreiten.

2.2.3 Algorithmen zur Testfallerzeugung

In der Vergangenheit wurden verschiedene Algorithmen entwickelt, die eine Menge an Testfällen mit kombinatorischer Abdeckung, also im Wesentlichen Covering Arrays, erstellen können. Khalsa und Labiche [KL14] konnten in einer Metaanalyse im Jahr 2014

75 verschiedene Algorithmen und Tools entdecken, die auf unterschiedliche Art und Weise kombinatorische Methoden zur Testfallerzeugung anwenden. Diese Arbeit markiert die zum aktuellen Zeitpunkt größte Übersicht über die vorhanden Algorithmen und Tools zu Combinatorial Testing.

Laut Khalsa und Labiche [KL14] lassen sich die Algorithmen zur Erstellung von Testmengen bei Combinatorial Testing zwei grundlegenden Kategorien zuordnen, den Test-basierten und Parameter-basierten Methoden: Bei der Test-basierten Variante wird ein Testfall derart erzeugt, dass dieser in einem Schritt möglichst viele einfache t-fache Konfigurationen (vgl. Unterabschnitt 2.2.2) abdeckt und alle Eingabeparameter berücksichtigt [KL14]. Beispiel hierfür ist der AETG-Algorithmus (vgl. Unterunterabschnitt 2.2.3).

Im Gegensatz dazu berücksichtigen laut Khalsa und Labiche [KL14] Parameter-basierte Algorithmen zunächst nur t Eingabeparameter und erstellen für diesen Fall eine Testmenge mit vollständiger t-facher Abdeckung. Anschließend werden die Testfälle dann so erweitert, dass diese mit Werten der $n - t$ fehlenden Parametern belegt werden. Falls notwendig, werden weitere Testfälle auf dem Weg zu einer vollständigen t-fachen Abdeckung ergänzt. Diese Erweiterung kann beispielsweise als Greedy-Verfahren oder anhand rekursiver, algebraischer Methoden vorgenommen werden [KL14]. Ein Beispiel hierfür ist der IPOG-Algorithmus (vgl. Unterunterabschnitt 2.2.3).

Ungeachtet dieser Unterteilung konnten Khalsa und Labiche [KL14] fünf verschiedene Algorithmen-Klassen den vorhanden Combinatorial Testing Tools und Algorithmen zuordnen:

- Greedy-Verfahren: Eine optimale Lösung in Bezug auf eine Bewertungsfunktion versucht man bei Greedy-Verfahren durch Iterationsschritte lediglich auf Basis der lokal verfügbaren Informationen zu finden [Sch01, S. 185]. In Bezug auf Combinatorial Testing bedeutet dies, dass ausgehend von einer bestehenden Menge an Testfällen weitere Testfälle möglichst viele der noch nicht berücksichtigten Kombinationen abdecken [KL14] sollten. Greedy-Verfahren machen den Großteil (53 %) der von Khalsa und Labiche [KL14] entdeckten Algorithmen und Tools für Combinatorial Testing aus.

- Meta-Heuristiken: Im Allgemeinen wird bei heuristischen Algorithmen versucht, das Auffinden einer optimalen Lösung eines Optimierungsproblems durch Zuhilfenahme 'problem-spezifischer Informationen' [Sch01, S. 319], sogenannten 'Heuristiken', zu beschleunigen [Sch01, S. 319]. Im Konkreten werden im Kontext von Combinatorial

Testing unter anderem die Methoden der genetischen Algorithmen, der Partikelschwarmoptimierung oder des Simulated Annealing verwendet [KL14]. Im Vergleich zu Greedy-Methoden sind laut Khalsa und Labiche [KL14] Meta-Heuristiken meist langsamer in ihrer Ausführung, liefern aber häufig bessere Lösungen im Sinne einer kleineren Menge an benötigten Testfällen.

- Adaptive Random-/Adhoc-Verfahren: Diese Kategorie der Algorithmen-Klassen für Combinatorial Testing fokussiert sich im Grundsatz auf eine zufallsgesteuerte Erzeugung von Testfällen [KL14]. Adhoc-Verfahren erzeugen laut Khalsa und Labiche [KL14] Testfälle auf Grundlage einer zuvor angenommenen Wahrscheinlichkeitsverteilung. Bei Adaptive Random-Methoden wird durch ein Distanzmaß, beispielsweise dem Hamming-Abstand, gewährleistet, dass die erzeugten Testfälle sich nicht zu stark überschneiden und so eine geringe Menge an Testfällen zur t-fachen Abdeckung ausreicht [KL14].

- Algebraische Verfahren: Algebraische Verfahren erzeugen Testfälle anhand einer vorgegebenen mathematischen Funktion oder vordefinierten mathematischen Regeln [KL14]. Unter anderem werden dabei auch rekursive Methoden verwendet, um die Komplexität der Problematik auf kleinere Teilprobleme zu reduzieren.

- Hybride Methoden: Die Konzepte verschiedener zuvor aufgeführten Konzepte zu vereinen, steckt als Grundidee hinter den hybriden Verfahren [KL14]. Ein Beispiel hierfür ist der modifizierte IPOG-D-Algorithmus (vgl. Unterunterabschnitt 2.2.3).

Neben den unterschiedlichen Strategien zur Testfallerzeugung existieren zwischen den verschiedenen Algorithmen und Tools laut Khalsa und Labiche [KL14] wesentliche Unterschiede unter anderem in der maximal unterstützten Höhe des Interaktionsparameters t und der Möglichkeit Bedingungen an das Testsystem zu stellen, sodass gewisse kombinatorische Möglichkeiten ausgeschlossen werden können. Zudem besitzt nicht jeder Algorithmus die Möglichkeit sogenannte Mixed Covering Arrays abzudecken, also Testfälle, bei denen jeder Eingabeparameter eine unterschiedliche Anzahl an Werten einnehmen kann [KL14].

Im Folgenden werden beispielhaft relevante Vertreter der Algorithmen zur Erstellung von Covering Arrays vorgestellt. Diese decken nur einen Bruchteil der existierenden Methoden zur Testmengen-Erstellung dar und fokussieren sich demnach insbesondere auf diejenigen Algorithmen, die im Rahmen dieser Arbeit eine wichtige Rolle spielen.

AETG

Als mitunter erste Wissenschaftler überhaupt beschäftigten sich Cohen et al. [CDFP97] 1997 mit den Methoden des Combinatorial Testing und der Frage, wie auf effiziente Art und Weise Covering Arrays erstellt werden können. Der von Cohen et al. [CDFP97] entwickelte Algorithmus und das zugehörige AETG System waren über lange Zeit das de facto Standardtool in Bezug auf kombinatorische Testmethoden. Das kommerzielle AETG System ist jedoch in seiner ursprünglichen Form nicht mehr verfügbar, soll aber aufgrund seiner historischen Bedeutung kurz vorgestellt werden.

Der AETG-Algorithmus ist ein Greedy-Verfahren, das grundsätzlich eine beliebige Höhe des Interaktionsparameters t abdecken kann [CDFP97]. Die tatsächliche Realisierung des AETG-Systems wurde jedoch lediglich für eine paarweise Kombinationsabdeckung ($t = 2$) entwickelt [KL14]. Darüber hinaus besitzt das AETG System die Möglichkeit unerwünschte Kombinationen durch explizite Angabe von 'verbotenen Tupeln' auszuschließen [CDFP97, KL14].

Der konkrete Algorithmus, der als Test-basierter Algorithmus einzustufen ist (vgl. Unterabschnitt 2.2.3) soll im Folgenden kurz erläutert werden. Eine detailliertere Beschreibung der Vorgehensweise lässt sich bei Cohen et al. [CDFP97] finden.

Unter der Annahme, dass insgesamt n Parameter geprüft wurden und bereits r Testfälle erzeugt wurden, entsteht der $(r + 1)$-te Testfall durch folgendes Prinzip:

1. Zufällig wird ein Parameter x^* gewählt und für diesen Parameter der Wert z ermittelt, der bisher durch die wenigsten t-fachen Kombinationen abgedeckt ist. Anschließend werden die übrigen Parameter zufällig als Folge $x_1 := x^*, \ldots, x_n$ geordnet.

2. Nun wird für jeden Parameter x_i mit $1 < i \leq n$ der Wert des $(r + 1)$-ten Testfalls folgendermaßen bestimmt: Angenommen es seien bereits die Werte für $x_1, \ldots x_j$ festgelegt, dann ergibt sich der Wert des Parameters x_{j+1} dadurch, dass für jeden möglichen Wert von x_{j+1} geprüft wird, wie viele neue t-fachen Überdeckungen in Kombination mit den bereits gewählten Werten von $x_1, \ldots, x_j$ entstehen würden und davon wird schließlich das Maximum ausgewählt.

IPOG

Der IPOG-Algorithmus stellt als Parameter-basierte Variante ein alternatives Greedy-Verfahren zum AETG-Algorithmus dar, das insbesondere im ACTS-Tool (vgl. Unterunterabschnitt 2.2.4) eine wesentliche Rolle spielt. IPOG wurde 2007 von Lei et al. [LKK+08] vorgestellt und stellt eine Erweiterung des allgemeinen IPO-Algorithmus [LT98] dar, der die grundlegende Idee der Parameter-basierten Erzeugung (vgl. Unterabschnitt 2.2.3) in einfachster Form für paarweises Testen ($t = 2$) umsetzt. IPOG steht dabei für 'In-Parameter-Order-Generalization' [LKK+08].

Sei n die Anzahl aller Eingabeparameter x_i, t der Interaktionsparameter und T die aus dem Algorithmus resultierende Menge an Testfällen. Dann funktioniert der IPOG-Algorithmus im Grundsatz folgendermaßen. Eine vollständige Beschreibung inklusive Pseudocode lässt sich bei Lei et al. [LKK+08] nachlesen:

1. Die Menge der ausgegebenen Testfälle T wird als leere Menge initialisiert und die Eingabeparameter werden nach der Anzahl ihrer möglichen Werte absteigend sortiert. Dann werden zunächst für die ersten t Parameter alle Kombinationen gebildet und in die Menge T eingefügt. Falls $n > t$ folgt nun eine iterative Erweiterung der Testfälle in T für jeden Parameter x_j mit $t < j \leq n$, also die Parameter, die unter den ersten t Parametern nicht dabei waren.

2. In jeder Iteration wird für den aktuellen Parameter x_i eine Menge π berechnet, die alle t-fachen Kombinationen mit den bereits berücksichtigten Parametern j mit $1 \leq j < i$ beinhaltet, die abgedeckt werden müssen. Man beachte, dass die Testmenge T bei der i-ten Iteration bereits eine vollständige t-fache Abdeckung für die Parameter $x_1, \ldots, x_{i-1}$ besitzt. Die fehlenden Testfälle zur Abdeckung von $x_1, \ldots, x_i$ werden nun in zwei Schritten vorgenommen, die Lei et al. als 'horizontales' und 'vertikales Wachstum' [LKK+08] bezeichnen.

3. Beim horizontalen Wachstum wird für jeden bereits existierenden Testfall ein zusätzlicher Wert für den Parameter x_i angefügt und dabei jener Wert ausgewählt, der in der zuvor erstellten Kombinationsmenge π die meisten Kombinationen abdecken kann. Neu abgedeckte Parameterkombinationen werden stets aus π entfernt.

   Bei der folgenden vertikalen Erweiterung wird für alle noch verbleibenden t-fachen Kombinationen in der Menge π grundsätzlich ein neuer Testfall erstellt, bei dem die Werte der aktuell gewählten Parameter der t-fach Kombination durch die jeweiligen

Werte der t-fach Kombination vorgegeben werden und alle anderen Werte auf sogenannte 'Don't Cares' gesetzt werden. Dies soll gewährleisten, dass die Werte der nicht betroffenen Variablen für weitere Erweiterungen flexibel bleiben. Denn neben der Erzeugung eines Testfalls wird stets bei jeder Iteration des vertikalen Wachstums geprüft, ob eine Kombination durch Anpassungen derartiger 'Don't Cares' abgedeckt werden kann und somit kein neuer Testfall erstellt werden muss.

Der IPOG-Algorithmus zählt während seiner Ausführung alle denkbaren t-fachen Kombinationen auf und besitzt laut Lei et al. [LKK+08] die Zeitkomplexität von $O(y_{max}^{t+1} \cdot n^{t-1} \cdot \log n)$, wobei n die Anzahl der Eingabeparameter, t der Interaktionsparameter und y_{max} die Anzahl an unterschiedlichen Werten des Parameters ist, der die meisten verschiedenen Werte annehmen kann. Dementsprechend ist der allgemeine IPOG-Algorithmus insbesondere bei sehr großen Systemen ineffektiv [LKK+08], was zu verschiedenen Erweiterungen des Algorithmus führte.

Der IPOD-Algorithmus [LKK+08] vereint die Methoden des allgemeinen IPOG-Algorithmus und eines algebraisch-rekursiven Ansatzes, um bei der Erstellung der Testfälle nicht alle Kombinationsmöglichkeiten explizit aufzählen zu müssen. Im Konkreten wird dabei der erste Schritt des zuvor erläuterten IPOG-Algorithmus, also die initiale Erstellung einer t-fachen Abdeckung für die ersten t Parameter, mittels einer rekursiven Methode optimiert. Ein Verfahren von Chateauneuf [CCK99] für die effiziente Erstellung von Covering Arrays mit Interaktionsparameter $t = 3$ dient dabei als Grundlage. Insbesondere bei einer großen Anzahl an verschiedenen Werten für die Parameter x_i agiert der IPOD-Algorithmus laut Lei et al. [LKK+08] wesentlich schneller als der IPOG-Algorithmus: Anhand des Beispiels von $n = 20$ unterschiedlichen Eingabeparameter mit je vier verschiedenen Werten und dem Interaktionsparameter $t = 5$ konnten Lei et al. aufzeigen, dass der IPOD-Algorithmus lediglich 3% der benötigten Zeit zur Ausführung im Vergleich zum IPOG-Algorithmus benötigt. Zugleich fiel die resultierende Menge an Testfällen zur vollständigen fünffachen Abdeckung um 51 % größer aus.

Einen anderen Ansatz zur Zeit- und Testfalloptimierung wird bei den IPOG-F- und IPOG-F2-Algorithmen [FLL+08] gewählt, die sich neben einigen kleineren Optimierungen des IPOG-Algorithmus auf die effiziente horizontale Erweiterung im IPOG-Algorithmus fokussieren. Mittels der Methoden des dynamischen Programmierens wird laut Forbes et al. [FLL+08] die bestmögliche Abdeckung wesentlich schneller gefunden als beim ursprünglichen IPOG-Algorithmus. Während der allgemeine IPOG-Algorithmus alle $\binom{n-1}{i-1}$ Kombinationsoptionen für den neu hinzugefügten Parameter x_i überprüft, wird der horizontale

Erweiterungsschritt bei den beiden IPOG-F-Algorithmen anhand von Werten aus zwei Tabellen ermittelt, die Daten der zuvor gespeicherten Kombinationen beinhalten. Zudem können so mehr Informationen bei der bestmöglichen Wahl des 'Erweiterungswertes' berücksichtigt werden, was sich letztlich in einer kleineren Menge an Testfällen zur vollständigen t-fachen Abdeckung bemerkbar macht [FLL+08].

IPOG-F und IPOG-F2 unterscheiden sich durch die Verwendung einer Heuristik bei der Auswahl der horizontalen Erweiterung innerhalb der Tabellen des Ansatzes der dynamischen Programmierung bei IPOG-F2: IPOG-F2 ist demnach wesentlich effizienter in der Rechenleistung und benötigt weniger Speicherplatz, erstellt aber größere Mengen an Testfällen im Vergleich zu IPOG-F [FLL+08].

Laut Forbes et al. [FLL+08] ergibt sich für beide IPOG-F-Varianten eine ähnliche Worst-Case-Zeitkomplexität wie beim allgemeinen IPOG-Algorithmus. Durch experimentelle Untersuchungen konnten die Autoren jedoch aufzeigen, dass sowohl IPOG-F als auch IPOG-F2 erhebliche Ersparnisse in der Durchführungszeit im Vergleich zu IPOG einbringen. IPOG-F spart zusätzlich rund 5% der notwendigen Testfälle gegenüber IPOG ein [FLL+08].

CASA

CASA [GCD09, GCD11] steht für 'Covering Arrays by Simulated Annealing' und stellt ein meta-heuristisches Verfahren zur Erstellung vom Covering Arrays dar. CASA basiert auf den Grundsätzen des Simulated Annealing, was erstmals von Stevens [Ste99] im Zusammenhang von Covering Arrays erwähnt wurde. Garvin et al. [GCD09] erweiterten das Grundkonzept von Stevens derart, dass Bedingungen des zu testenden Systems formuliert werden konnten und diese bei der Testfallerzeugung berücksichtigt wurden. 2011 ergänzten sie ihr erweitertes Konzept durch weitere Verbesserungen des verwendeten Algorithmus im Hinblick auf die Menge an erzeugten Testfällen, die möglichst gering ausfallen sollte [GCD11]. CASA wurde von der University of Nebraska-Lincoln verwaltet, konnte jedoch nach den Recherchen im Rahmen dieser Arbeit nicht mehr als anwendbares Tool aufgefunden werden. Jedoch ist der Algorithmus in das web-basierte CTWedge-Tool (vgl. Unterunterabschnitt 2.2.4) eingebettet.

Im Allgemeinen fußt Simulated Annealing auf dem Prinzip der lokalen Verbesserungsstrategien [Sch01, S. 330]: Ausgehend von einem zufälligen Startzustand wird die unmittelbare

Nachbarschaft jenes Zustands untersucht und dabei lokale Verbesserungsschritte zum optimalen Ergebnis mithilfe einer Kostenfunktion vorgenommen. Dieses Prinzip, das auch unter dem Konzept des Hill Climbing bekannt ist [Sch01, S.327], verharrt jedoch bei lokalen Optima, sodass nicht unbedingt die bestmögliche globale Lösung gefunden wird [Sch01, S.327]. Simulated Annealing löst dieses Problem, indem mit einer zeitlich abnehmender Wahrscheinlichkeit auch Lösungen akzeptiert werden, die schlechter als zuvor ermittelte Teillösungen sind. Eine detailliertere Beschreibung dieses Grundprinzips lässt sich in Schöning finden [Sch01, S.329 ff.].

Der von Garvin et al. [GCD11] entwickelte CASA-Algorithmus, der auf das Konzept des Simulated Annealing zurückgreift, besitzt zwei grundlegende Komponenten, welche die Autoren als 'Outer Search' und 'Inner Search' bezeichen:

1. Die 'Outer Search' bildet den Rahmen des Algorithmus, der über einen möglichen Bereich eine Binärsuche für die optimale Anzahl an Testfällen T durchführt. Im Vorfeld wird daher eine untere Schranke $T_{\min}$ und $T_{\max}$ vorgegeben, die diesen Rahmen bilden. Für jeden möglichen Kandidaten N zwischen $T_{\min}$ und $T_{\max}$ wird dann in der 'Inner Search' versucht eine Testmenge zu bilden. Falls eine Testmenge T mit vollständiger Abdeckung gefunden wurde, wird die obere Grenze innerhalb der Binärsuche angepasst und versucht eine Testabdeckung mit geringerer Anzahl an Testfällen zu erreichen.

2. In der 'Inner Search' findet der Prozess des Simulated Annealing statt: Die Kostenfunktion, die es zu optimieren gilt, ist die Anzahl an nicht abgedeckten Kombinationen, die idealerweise 0 betragen soll. Auf Basis einer Startkonfiguration, die anhand vorheriger Iterationen der 'Outer Search' abgeleitet wird, werden im Folgen Nachbarschaftsveränderungen in Form von Anpassungen einzelner Werte eines Parameters vorgenommen. Dabei wird stets geprüft, inwiefern sich die Anzahl der abgedeckten Kombinationen verändert und unter dem Prinzip des Simulated Annealing mit einem 'Cooldown' der Akzeptanz-Wahrscheinlichkeit für schlechtere Lösungen Anpassungen durchgeführt.

Ergänzende Erweiterungen dieses grundlegenden Algorithmus nahmen Garvin et al. [GCD11] insbesondere im Hinblick auf die effiziente Einbettung von Bedingungen an das Testsystem vor, zudem verändert der optimierte CASA-Algorithmus in der 'Inner Search' nicht nur einzelne Werte eines Parameters in einem Updateschritt, sondern ganze t-fache Mengen.

Dies bewirkt, dass die Iterationsschritte größer ausfallen und so vor allem nahe der Optimalitätsgrenze von keinen fehlenden Kombinationen weniger Schritte benötigt werden, um das Optimum zu finden [GCD11]. Außerdem beobachteten Garvin et al., dass bei einer fehlgeschlagenen 'Inner Search' - also kein passendes Covering Array mit N Testfällen konnte erstellt werden - eine reine Binärsuche in der 'Outer Search' frühzeitig geringe Werte für N als Optimallösung ausschließt. Als Lösung dafür setzt der modifizierte CASA-Algorithmus auf eine einseitige Einschränkung als Alternative zur Binärsuche [GCD11].

In einer experimentellen Untersuchung konnten Garvin et al. [GCD11] aufzeigen, dass der CASA-Algorithmus im Vergleich zu einer modifizierten Version des AETG-Algorithmus (vgl. Unterunterabschnitt 2.2.3) durchschnittlich 25 % weniger Konfigurationen zur vollständigen Testabdeckung erzeugt. Zudem konnten die Autoren ermitteln, dass insbesondere bei großen Systemen mit längerer Durchführungszeit der Algorithmen CASA schneller Ergebnisse liefert als der verglichene mAETG-Algorithmus.

2.2.4 Tools zur Testfallerzeugung

Die Landschaft der verfügbaren Tools und Algorithmen zur Anwendung von Combinatorial Testing ist trotz einer derart ausführlichen Übersicht, wie Khalsa und Labiche [KL14] ausarbeiteten, recht unübersichtlich. Viele der insgesamt 75 Algorithmen und Tools, die Khalsa und Labicher finden konnten, sind lediglich in Form von Pseudocode zugänglich und können daher nicht unmittelbar auf einen konkreten Anwendungsfall umgemünzt werden. Des Weiteren gibt es einige kommerzielle Anwendungen, die im Rahmen dieser Arbeit nicht berücksichtigt werden. Eine weitere Einschränkung der verfügbaren Tools für den in Kapitel 3 vorgestellten Anwendungsfall ergibt sich dadurch, dass viele Tools lediglich paarweise Abdeckungen ($t = 2$) ermöglichen [KL14]. Eine Übersicht über viele der aktuell nutzbaren Tools lässt sich bei Czerwonka [Czeb] finden.

Ungeachtet dessen sind die zur Verfügung stehenden Informationen einiger dort aufgeführter Tools limitiert, sodass die verwendeten Algorithmen nur teilweise bekannt sind. Unter anderem gehört zu dieser Kategorie das Tool Allpairs [Bac12], bei dem lediglich der Quellcode bekannt ist und keine weiteren Informationen vorliegen. Allpairs lässt sich in die Algorithmen der Adhoc-Kategorie einordnen und ist nach Aussagen des Autors Bach [Bac12] wesentlich ineffizienter als andere Algorithmen.

Die wenigen Tools mit einer fundierten wissenschaftlichen Grundlage, die frei zu Verfügung stehen, einen Interaktionsparameter größer als $t = 2$ berücksichtigen können und zudem zum aktuellen Zeitpunkt abgerufen werden können, sind PICT (vgl. Unterunterabschnitt 2.2.4), ACTS [YLKK13] und CTWedge [GR].

ACTS

Das 'Advanced Combinatorial Testing System' (ACTS) [YLKK13] ist eine vom US National Institute of Standards and Technology und der Universität von Texas entwickelte Software zur Erzeugung von Testmengen nach den Prinzipien des kombinatorischen Testens. ACTS wurde erstmalig 2006 vorgestellt, damals noch unter dem Namen FireEye [LKK+07]. Es existieren insgesamt drei verschiedene Arten ACTS in der Praxis einzusetzen [YLKK13]: Eine grafische Benutzeroberfläche, eine Kommandozeilen-Schnittstelle und eine Java-Programmierschnittstelle (API). Abbildung **??** zeigt beispielhaft die grafische Oberfläche von ACTS nach Erstellung eines Covering Arrays. ACTS selbst wurde in Java geschrieben.

ACTS ist im Gegensatz zu vielen anderen Tools zur Erstellung von Covering Arrays sehr flexibel in der Art und Weise, wie das zu testende System auszusehen hat und auf welche Art und Weise Testfälle generiert werden sollen: Grundsätzlich kann ACTS Mixed Covering Arrays erstellen, was bedeutet, dass die Eingabeparameter eine unterschiedliche Anzahl an verschiedenen Werten annehmen können. Zudem unterstützt ACTS Bedingungen an das Testsystem: Mittels einer speziellen Syntax können bestimmte Parameterbeziehungen durch aussagenlogischen Ausdrücke ausgeschlossen werden. Diese Syntax orientiert sich im Wesentlichen an typischen logischen Operationen 'Und', 'Oder', 'Nicht' und arithmetischen Vergleichsoperatoren '$<$','$>$','$\leq$', '$\geq$' (vgl. dazu [YLKK13]). Für die Umsetzung dieses Konzepts ist in ACTS ein externes Framework namens Choco [sol] integriert, das für jeden erzeugten Testfall überprüft, ob die zuvor formulierten Bedingungen verletzt werden.

Unter Berücksichtigung dieser Bedingungen erstellt ACTS Covering Arrays bis zu einer Abdeckung des Interaktionsparameters $t = 6$. Dass ACTS keine Abdeckung höherer Werte für den Interaktionsparameter ermöglicht, liegt darin begründet, dass Kuhn et al. [KWG04] in einer experimentellen Untersuchung aufzeigen konnten, dass fast alle Fehler von Softwaresystemen auf die Interaktion von sechs oder weniger Parameter zurückzuführen sind (vgl. Abschnitt 2.2).

ACTS ermöglicht es zudem für verschiedene Gruppen von Eingabeparameter einen unterschiedlichen Wert für den Interaktionsparameter zu fordern und ermöglicht es in gewisser Weise verschiedene Parameterbeziehungen zu priorisieren. In der Literatur wird dies 'Mixed Strength Test Generation' genannt [YLKK13, Cze06]. Wird beispielsweise der Interaktionsparameter $t = 2$ gewählt und es ist jedoch bekannt, dass die Parameter x_1, x_3, x_4 und x_7 in ihrer Interaktion besonders kritisch zu betrachten sind, so kann für diese Parametermenge bei ACTS eine 'Relation' [YLKK13] definiert werden, bei der eine Abdeckung mit $t = 3$ gefordert wird. ACTS besitzt eine interne Logik, die zunächst die verschiedenen Relationen verknüpft und eventuell überflüssige Kombinationen herausfiltert. Erst danach erstellt ACTS eine Übersicht über die zu erstellenden Kombinationen und bildet ein Covering Array [YLKK13].

Bei der Erstellung von Covering Arrays haben die Anwender*innen von ACTS die Wahl zwischen verschiedenen Algorithmen: ACTS wurde konzeptionell auf die Nutzung der unterschiedlichen IPOG-Algorithmen ausgerichtet (vgl. Unterunterabschnitt 2.2.3) und besitzt dementsprechend die Auswahlmöglichkeiten des klassischen IPOG-Algorithmus [LKK+08], des IPOG-D-Algorithmus [LKK+08] und der beiden IPOG-F-Algorithmen [FLL+08] (vgl. Unterunterabschnitt 2.2.3). Darüber hinaus implementierten die Entwickler von ACTS einen zufallsbasierten Algorithmus namens 'PaintBall', zu dem jedoch im Rahmen der Recherche dieser Arbeit keine detaillierten Informationen ermittelt werden konnten. Zudem existiert bei ACTS die Möglichkeit ein Covering Array im Sinne einer Base Choice-Abdeckung zu erstellen, wie sie in Unterabschnitt 2.2.2 vorgestellt wurde.

Neben der Erzeugung einer Menge von Testfällen zur vollständigen t-fachen Abdeckung besteht bei ACTS die Möglichkeit ein bestehendes Covering Array bei Anpassungen der Parameter und deren Werten zu ergänzen ohne dabei eine komplett neue Testfallmenge erstellen zu müssen. Allgemein bietet ACTS für jede Menge an Testfällen zudem die Möglichkeit zu prüfen, ob und inwiefern eine t-fache Abdeckung erreicht wird.

PICT

PICT [Cze06, Czea] wurde 2006 erstmalig von Microsoft-Mitarbeiter Jacek Czerwonka vorgestellt und basiert auf dem AETG-Algorithmus (vgl. Unterunterabschnitt 2.2.3). Das Tool wird über die Kommandozeile angesteuert, nimmt dabei eine Textdatei mit der Spezifikation des Testsystems als Eingabe an und gibt die Menge der erzeugten Testfälle auf der Kommandozeile aus [Czea]. Czerwonka [Cze06] betont, dass sich das entwickelte Tool

verstärkt auf die (Rechen-)Effizienz und die Nutzbarkeit im praktischen 'Test-Alltag' ausrichtet und dementsprechend keine grundlegend neuen Methoden zur Generierung von Covering Arrays beinhaltet. Konkret bedeutet dies: PICT besitzt eine Reihe zusätzlicher Eigenschaften und Funktionalitäten im Vergleich zur allgemeinen Erstellung von Covering Arrays wie sie beispielsweise im Rahmen vom AETG-Algorithmus durchgeführt werden.

So veränderte Czerwonka den Algorithmus der Erstellung der Covering Arrays des AETG-Ansatzes derart, dass PICT auf Basis eines fest definierten Seeds, also einem vordefinierten Startwert für die Erstellung pseudozufälliger Zahlen, arbeitet und keine zufällige Komponente besitzt (vgl. im Gegensatz dazu Unterunterabschnitt 2.2.3).

Neben dieser Änderung am Algorithmus selbst fügte Czerwonka verschiedene Optionen ein, welche die Menge der zu erzeugenden Kombinationen, die als Grundlage des Algorithmus dient, flexibel hält [Cze06]. Konkret bedeutet dies: Der Interaktionsparameter t kann bei PICT bis zu einem Wert von $t = 6$ beliebig gewählt werden [KL14], zudem unterstützt das Tool das Prinzip der 'Mixed Test Strength Generation' (vgl. Unterunterabschnitt 2.2.4) [Cze06]. PICT erlaubt es den Anwendenden darüber hinaus eine Hierarchie zu definieren, sodass tiefer positionierte Komponenten nur auf ihrer jeweiligen Hierarchiestufe kombiniert werden können und somit die Anzahl der Testfälle im Gesamten reduziert werden kann [Cze06].

Wie auch ACTS ist PICT in der Lage, Bedingungen des zu testenden Systems zu berücksichtigen: Mittels einer 'IF-THEN'-Logik kann spezifiziert werden, welche Voraussetzungen die resultierenden Testfälle erfüllen sollen. PICT wandelt diese aussagenlogischen Formeln intern in eine Menge an auszuschließenden Kombinationen um, die dann beim modifizierten AETG-Erzeugungsalgorithmus exkludiert werden [Cze06]. Eine weitere Ergänzung von PICT ist die Formulierung von 'Seeds': Dies sind Testfälle, die in jedem Fall im Covering Array erscheinen müssen. Zudem unterstützt PICT explizit sogenanntes 'Negative Testing', also gezieltes Testen auf Fehlverhalten, und die Gewichtung von Werten einzelner Parameter, die bei 'Don't Cares' im Algorithmus berücksichtigt wird.

CTWedge

Viele der Tools zur Erstellung von kombinatorischen Testmengen, die in der Vergangenheit vorgestellt wurden, sind Domänen-spezifisch: Dies bedeutet, dass diese meist als Desktop-Anwendungen oder als spezielle Plugins für existierende Software entwickelt wurden und

dementsprechend nicht universell einsetzbar sind [GR18]. Um diesem Problem zu begegnen, entwickelten Gargantini und Radavelli [GR18] ein Web-basiertes Tool für Combinatorial Testing [GR], das die Erstellung von Covering Arrays über einen Server vornimmt und der Zugriff über das Internet erfolgt. Dies wird auch als 'Software as a Service' bezeichnet [GR18]. Abbildung **??** zeigt einen Screenshot von CTWedge aus Nutzersicht.

Über eine selbst definierte Syntax, die anhand des domänenspezifischen Modellierungsframeworks Xtext [EB10] geparst wird, können die Parameter und ihre Werte definiert werden. CTWedge unterstützt genauso wie ACTS die Formulierung von Bedingungen an das Testsystem, die ebenfalls in einer selbst definierten Syntax aufgeschrieben werden müssen. Details zur Syntax können bei Gargantini und Radavelli [GR18] und anhand vordefinierten Beispielen des frei verfügbaren CTWedge-Tools [GR] entnommen werden.

Gargantini und Radavelli verwenden bei der Erzeugung der Testfälle keine eigenen Algorithmen, sondern greifen auf die Schnittstellen anderer Tools zurück: Im Konkreten haben die beiden Autoren die Schnittstelle von ACTS (vgl. Unterunterabschnitt 2.2.4) und CASA (vgl. Unterunterabschnitt 2.2.3) integriert.

3 Anwendungsfall

3.1 Versicherungstechnische Grundlagen

3.2 Implementierung

3.3 Ergebnisse

4 Diskussion

5 Fazit

Literaturverzeichnis

[ABC+13] Anand, Saswat ; Burke, Edmund K. ; Chen, Tsong Y. ; Clark, John ; Cohen, Myra B. ; Grieskamp, Wolfgang ; Harman, Mark ; Harrold, Mary J. ; McMinn, Phil ; Bertolino, Antonia u. a.: An orchestrated survey of methodologies for automated software test case generation. In: *Journal of Systems and Software* 86 (2013), Nr. 8, S. 1978–2001

[AO08] Ammann, Paul ; Offut, Jeff: *Introduction to software testing*. New York : Cambridge University Press, 2008. – ISBN 9780521880381

[Bac12] Bach, J: *Allpairs test case generation tool*. `https://www.satisfice.com/download/allpairs`. Version: 2012. – Zuletzt aufgerufen: 26.10.2021

[CCK99] Chateauneuf, MA ; Colbourn, Charles J. ; Kreher, Donald L.: Covering arrays of strength three. In: *Designs, Codes and Cryptography* 16 (1999), Nr. 3, S. 235–242

[CDFP97] Cohen, David M. ; Dalal, Siddhartha R. ; Fredman, Michael L. ; Patton, Gardner C.: The AETG system: An approach to testing based on combinatorial design. In: *IEEE Transactions on Software Engineering* 23 (1997), Nr. 7, S. 437–444

[CJ02] Craig, Rick ; Jaskiel, Stefan: *Systematic software testing*. Boston : Artech House, 2002. – ISBN 9781580535083

[CLM04] Chen, Tsong Y. ; Leung, Hing ; Mak, IK: Adaptive random testing. In: *Annual Asian Computing Science Conference* Springer, 2004, S. 320–329

[Czea] Czerwonka, Jacek: *Microsoft/PICT: Pairwise Independent Combinatorial Tool*. `https://github.com/microsoft/pict`. – Zuletzt aufgerufen: 03.11.2021

[Czeb] Czerwonka, Jacek: *Pairwise testing.* `https://www.pairwise.org/`. – Zuletzt aufgerufen: 26.10.2021

[Cze06] Czerwonka, Jacek: Pairwise testing in real world. In: *24th Pacific Northwest Software Quality Conference* Bd. 200 Citeseer, 2006

[DDH72] Dahl, Ole-Johan ; Dijkstra, Edsger W. ; Hoare, Charles Antony R.: *Structured programming.* Academic Press Ltd., 1972

[Dev21] Deva, Prashant: *Method Size Limit in Java - DZone Java.* `https://dzone.com/articles/method-size-limit-java`. Version: Feb 2021. – Zuletzt aufgerufen: 29.10.2021

[EB10] Eysholdt, Moritz ; Behrens, Heiko: Xtext: implement your language faster than the quick and dirty way. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, S. 307–309

[FLL+08] Forbes, Michael ; Lawrence, Jim ; Lei, Yu ; Kacker, Raghu N. ; Kuhn, D R.: Refining the in-parameter-order strategy for constructing covering arrays. In: *Journal of Research of the National Institute of Standards and Technology* 113 (2008), Nr. 5, S. 287

[GCD09] Garvin, Brady J. ; Cohen, Myra B. ; Dwyer, Matthew B.: An improved meta-heuristic search for constrained interaction testing. In: *2009 1st International Symposium on Search Based Software Engineering* IEEE, 2009, S. 13–22

[GCD11] Garvin, Brady J. ; Cohen, Myra B. ; Dwyer, Matthew B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. In: *Empirical Software Engineering* 16 (2011), Nr. 1, S. 61–102

[GR] Gargantini, Angelo ; Radavelli, Marco: *Combinatorial Testing Web-based Editor and Generator.* `https://foselab.unibg.it/ctwedge/`. – Zuletzt aufgerufen: 26.10.2021

[GR18] Gargantini, Angelo ; Radavelli, Marco: Migrating combinatorial interaction test modeling and generation to the web. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* IEEE, 2018, S. 308–317

[Hal77] Halstead, Maurice H.: *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., 1977

[HXCL12] Huang, Rubing ; Xie, Xiaodong ; Chen, Tsong Y. ; Lu, Yansheng: Adaptive random test case generation for combinatorial testing. In: *2012 IEEE 36th Annual Computer Software and Applications Conference* IEEE, 2012, S. 52–61

[Int11] International Organization for Standardization, International Electrotechnical Commission: Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software quality models / International Organization for Standardization. Geneva, CH, März 2011. – Standard

[KKL+10] Kuhn, D R. ; Kacker, Raghu N. ; Lei, Yu u. a.: Practical combinatorial testing. In: *NIST special Publication* 800 (2010), Nr. 142, S. 142

[KL14] Khalsa, Sunint K. ; Labiche, Yvan: An orchestrated survey of available algorithms and tools for combinatorial testing. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering* IEEE, 2014, S. 323–334

[KWG04] Kuhn, D R. ; Wallace, Dolores R. ; Gallo, Albert M.: Software fault interactions and implications for software testing. In: *IEEE transactions on software engineering* 30 (2004), Nr. 6, S. 418–421

[LKK+07] Lei, Yu ; Kacker, Raghu ; Kuhn, D R. ; Okun, Vadim ; Lawrence, James: IPOG: A general strategy for t-way software testing. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)* IEEE, 2007, S. 549–556

[LKK+08] Lei, Yu ; Kacker, Raghu ; Kuhn, D R. ; Okun, Vadim ; Lawrence, James: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. In: *Software Testing, Verification and Reliability* 18 (2008), Nr. 3, S. 125–148

[LL10] Ludewig, Jochen ; Lichter, Horst: *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken.* Heidelberg : Dpunktverlag, 2010. – ISBN 9783898646628

[LT98] Lei, Yu ; Tai, Kuo-Chung: In-parameter-order: A test generation strategy for pairwise testing. In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)* IEEE, 1998, S. 254–261

[McC76] McCabe, Thomas J.: A complexity measure. In: *IEEE Transactions on software Engineering* (1976), Nr. 4, S. 308–320

[Roy87] Royce, Winston W.: Managing the development of large software systems: concepts and techniques. In: *Proceedings of the 9th international conference on Software Engineering*, 1987, S. 328–338

[Sch01] Schöning, Uwe: *Algorithmik*. Heidelberg, Berlin : Spektrum, Akad. Verl, 2001. – ISBN 3827410924

[Sch12] Schneider, Kurt: *Abenteuer Softwarequalität : Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. Heidelberg : Dpunkt.verlag, 2012. – ISBN 9783898647847

[SL19] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest : Aus- und Weiterbildung zum Certified Tester Foundation Level nach ISTQB-Standard*. Heidelberg : dunkt.verlag, 2019. – ISBN 9783960885016

[SLS11] Spillner, Andreas ; Linz, Tilo ; Schaefer, Hans: *Software Testing Foundations : A Study Guide for the Certified Tester Exam : Foundation Level, ISTQB compliant*. Santa Barbara, CA : Rocky Nook, 2011. – ISBN 9781933952789

[sol] *Choco Solver*. `https://choco-solver.org/`. – Zuletzt aufgerufen: 03.11.2021

[Som12] Sommerville, Ian: *Software-Engineering*. München, Harlow, u.a. : Pearson, Higher Education, 2012. – ISBN 9783868940992

[Ste99] Stevens, Brett: *Transversal covers and packings*. University of Toronto, 1999

[uni] *Unicode® Version 14.0 Character Counts*. `https://www.unicode.org/versions/stats/charcountv14_0.html`. – Zuletzt aufgerufen: 29.10.2021

[WC80] White, Lee J. ; Cohen, Edward I.: A domain strategy for computer program testing. In: *IEEE transactions on software engineering* (1980), Nr. 3, S. 247–257

[YLKK13] Yu, Linbin ; Lei, Yu ; Kacker, Raghu N. ; Kuhn, D R.: Acts: A combinatorial test generation tool. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* IEEE, 2013, S. 370–375

Ehrenwörtliche Erklarung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den 25.03.2020 _____

(Unterschrift)