Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Evaluation des Einsatzes von lernfähigen Fuzz-Tests zur automatisierten Generierung von Systemtests

Christoph Braun

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Jasmin Ramadani, M.Sc.
Beginn am:	2016-12-19
Beendet am:	2017-06-19
CR-Nummer:	D.1.2, D.1.5, D.2.1, D.2.4, D.2.5, E.1, F.1.1

Kurzfassung

Automatisierte Systemtests sind die wohl gängigste Methode sinnvoll den Aufwand des Systemtests zu verringern. Die Tests zu schreiben, ist jedoch noch immer mit Aufwand verbunden und die Tests decken in der Regel nur eine Teilmenge der von Nutzern ausgelösten Szenarien ab. Fuzz-Tests sind Tests auf Basis zufällig generierter Eingaben. Sie versprechen bei langer Ausführungszeit mehr Nutzerszenarien bei geringerem Aufwand abzudecken. Um die Ausführungszeit zu verkürzen wird ein lernfähiger Algorithmus evaluiert und mit einem rein zufälligen Algorithmus verglichen. Die Evaluation zeigt, dass in der Theorie ein lernfähiger Algorithmus in der Lage ist, eine hohe Überdeckung der Szenarien zu erreichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application programming interface. 42

Car Capture-and-Replay. 20

CLI Command line interface. 25

Ldtp Linux Desktop Testing Project. 22

1. Einleitung

Diese Ausarbeitung befasst sich mit Systemtests und deren Automatisierung. Der Systemtest ist der einzige Test im Softwarezyklus, der Funktionalität des Prüflings vollständig testen kann[Joc13][S.491]. Er kann als Integrationstest auf höchster Ebene verstanden werden[Joc13][S.491]. Ein Integrationstest testet ob Programmteile richtig miteinander agieren [Joc13][S.491]. Beim Systemtest, sind diese Programmteile der Prüfling, andere Komponenten des Systems und vor allem anderen der Nutzer. Der Systemtest ist notwendig, richtig ausgeführt jedoch mit einem sehr hohen Aufwand verbunden [Joc13][S.62]. Selbst die bereits verwendeten Automatisierungstechniken, erfordern einiges an Implementierungsaufwand. Deshalb befasst sich diese Arbeit mit der automatischen Generierung der Testskripte, anhand der bereits eingesetzten Technologie und durch Fuzz-Tests. Fuzz-Tests sind Tests mit zufällig generierten Eingabewerten. Bei Black-Box Fuzz-Tests, die Nutzereingaben simulieren, spricht man häufig von „Monkey Tests“ Kapitel 4. Die zufälligen Eingaben, erzeugen viele Szenarien, die ein menschlicher Tester nicht testen würde [Nym00], jedoch auch viele redundante Szenarien. Deshalb wird evaluiert, ob durch einen lernfähigen randomisiserten Algorithmus die erzeugten Szenarien weniger redundant und vielfältiger werden. Die verwendete existierende Technologie sind die so genannten Page-Objects [sel17]. Sie erlauben es die Oberfläche des Prüflings zu automatisieren. Durch das ihnen zugrunde liegende Entwurfsmuster, bilden sie den möglichen Weg des Nutzers durch den Prüfling ab [sel17]. Diese Eigenschaft kann genutzt werden, um automatisch einen Zustandsgraphen zu erzeugen, auf dem die zufälligen Tests äußerst präzise ausgeführt werden können. Der Aufwand für die Vorbereitung des Verfahrens wurde ermittelt und ein geeigneter Algorithmus gefunden. Die Eigenschaften des Algorithmus wurden in einem Experiment geprüft Kapitel 6.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Automatische Systemtests: Hier werden die Anforderungen an Systemtest und ihre Automatisierung vorgestellt und der Aufwand erörtert.

Kapitel 3 – Zustand basiertes Testen: bietet eine Einführung in Zustandsbasiertes Testen und stellt eine Testbaum basierte Methode vor.

Kapitel 4 – Monkey-Tests: befasst sich mit den Monkey-Tests und stellt zwei unterschiedliche Ansätze.

Kapitel 5 – Das Verfahren: stellt die Problemdefinition. Die automatisierte Generierung des Zustandsgraphen und der Testfälle wird hier beschrieben.

Kapitel 6 – Lernfähigkeit: betrachtet wie der Algorithmus zur Testfallauswahl, durch Lernfähigkeit verbessert werden kann und beinhaltet einen Vergleich mit einem herkömmlichen Algorithmus.

Kapitel 7 – Bewertung: Hier wird das Verfahren nach einigen Kriterien bewertet. So soll eine Einschätzung der Eignung des Verfahrens ermöglicht werden.

Kapitel 8 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Automatische Systemtests

2.1. Anforderungen an Systemtests

Um zu klären, inwiefern automatisch generierte Systemtests, manuelle Tests ersetzen oder ergänzen können, ist es wichtig die allgemeinen Anforderungen an Systemtests zu spezifizieren. Zudem spielt die Einordnung der Tests in den Software-Prozess eine Rolle, wenn der Aufwand an Zeit und Ressourcen der Vorgehensweisen verglichen werden soll. Nach der Fertigung einer Spezifikation und der darin enthaltenen Anwendungsfälle, liegen alle grundlegenden, benötigten Informationen vor, um mit der Vorbereitung der Systemtests zu beginnen [Joc13, S.505]. Dadurch sind die Tests schon früh verfügbar, was gerade bei einem agilen Entwicklungsprozess entscheidend sein kann, da hier nach jeder Iteration und vor jedem Release[1] die relevanten Systemtests ausgeführt werden sollten. Ein weiterer Vorteil der entsteht, wenn die Testfälle erarbeitet werden sobald die Spezifikation vorliegt, ist die Überprüfung des Dokuments [And12][505].

Systemtests sind Black-Box-Tests und ziehen nur den Funktionsumfang des Prüflings in Betracht, nicht jedoch die Implementierung. Die Sicht des Testers auf den Prüfling ist hier also die Sicht des Nutzers oder des Kunden [And12][S.58]. Die Systemtests sollten diese Sicht widerspiegeln. Um die Systemtests in einem „systematischen Test“ verwenden zu können, müssen die Ergebnisse des Tests dokumentiert werden [Joc13][S.480]. Folglich müssen die Ergebnisse auch in einer von Menschen interpretier- und beurteilbaren Form vorliegen. Der Test soll zeigen, wie gut der Prüfling die funktionalen Anforderungen erfüllt [And12][S.59]. Dies ist für den Tester gerade dann einfach zu bewerten, wenn die Testergebnisse die Nutzersicht umsetzen und erkenntlich wird, was der Nutzer (im Test der simulierte Nutzer) getan hat, um das Fehlverhalten auszulösen. Wenn möglich, muss vor dem Test bereits ein Soll-Resultat definiert sein, gegen das die Testergebnisse validiert werden können [Joc13][S.482]. Die Spezifikation dient bei der Testvorbereitung als Anleitung für den Tester, welche Funktionen und Abläufe auszuführen sind. Sie stellt zudem auch das Soll-Resultat, anhand dessen der Tester erkennt, ob das Programm korrekt auf die Eingaben des Testfalls reagiert hat. Bei der Vorbereitung der Tests muss der Tester die Spezifikation also ausführlich studieren und kann Inkonsistenzen oder Unklarheiten entdecken [Joc13, S.504].

[1] Auslieferung an einen Kunden oder Veröffentlichung auf einem freien Markt.

Ähnlich wie bei Glass-Box oder Unit-Tests, können die ausgeführten Tests anhand einer Überdeckungsmetrik qualifiziert werden [Joc13, S.514ff].

Aus der Spezifikation direkt ergibt sich die **Funktionsüberdeckung**. Eine vollständige Funktionsüberdeckung ist erreicht, wenn alle in der Spezifikation definierten Funktionen ausgeführt werden. Benötigt wird für die Funktionsüberdeckung die Liste aller Funktionen und zugehörig eine Eingabegröße und eine erwartete Ausgabegröße, bzw. Reaktion des Prüflings [Joc13][S.506].

Die **Eingabeüberdeckung** ist deutlich komplexer, da sie für jede Funktion eine Ausführung mit jeder möglichen Eingabe bedeutet. Für Programme mit einer nicht trivialen Komplexität ist dies natürlich unmöglich. Daher verwendet man für die Eingaben Äquivalenzklassen [Joc13][S.506], wie sie auch im White-Box-Test gebräuchlich sind. Grob werden hier Eingaben, deren Auswirkungen auf die Funktion des Programms gleich sein sollen, in Äquivalenzklassen zusammengefasst. Es genügt nun eine Eingabe je Klasse auszuführen, um alle in der Äquivalenzklasse enthaltenen Werte abzudecken[And12, S.110]. Der Überdeckungsgrad kann dann anhand der abgedeckten Äquivalenzklassen, wie in Gleichung (2.1) ermittelt werden [And12][S.119].

$$\ddot{U}berdeckung = \frac{Anzahl\ getestete\ \ddot{A}quivalenzklassen}{\ddot{A}quivalenzklassen} \tag{2.1}$$

Die **Ausgabeüberdeckung** kann ähnlich wie die Eingabeüberdeckung durch Äquivalenzklassen erreicht werden. Wie der Name erahnen lässt, sind die Äquivalenzklassen jedoch bezogen auf die Ausgaben des Programms.

Neben diesen rein funktionalen Tests, bieten Systemtests noch die Möglichkeit, nicht funktionale Tests durchzuführen [And12][S.72]. Leistungsgrenzen z.B. die Anzahl an Aktionen auf der Nutzeroberfläche oder Mengengrenzen, wie besonders große Datenmengen als Eingabe, können getestet werden.

2.2. Aufwand in der Praxis und Motivation

2.2.1. Motivation

Um den Anforderungen an einen Systemtest gerecht zu werden, ist ein enormer Aufwand nötig. Die Vorbereitung der Tests anhand der Spezifikation, die Ausführung und die Analyse der Ergebnisse, kann große Teile der Ressourcen eines Projekts verschlingen. Der Systemtest ist jedoch der einzige Test, der die Funktionalität des gesamten Systems testet [Joc13][S.504]. Mit anderen Worten, ob das Projekt überhaupt seinen Zweck erfüllt. Ein Projekt, dass nicht die geforderte Funktionalität entwickelt, ist voraussichtlich zum Scheitern verurteilt. Der Systemtest ist somit der wohl wichtigste Test und alle anderen Tests, können als Ergänzung

des Systemtests[2] betrachtet werden [Joc13][S.504]. Die Kosten für Integration und Test können, wenn die Wartungskosten außen vor gelassen werden, etwa ein Viertel der Entwicklungskosten betragen [Joc13][S.62]. In einem iterativen Prozess, handelt es sich am Ende einer jeden Iteration um eine Integration der neu entwickelten Funktionalität. Diese muss selbstverständlich getestet werden. Da die Entwicklung in einem iterativen Projekt häufig mit dem ersten Release nicht abgeschlossen ist, sondern geplant vor Fertigstellung der vollständigen Funktionalität veröffentlicht wird, ist der Übergang von Erstentwicklung zur Wartung fließend. Die Tätigkeiten der Wartung sind „Korrektur, Anpassung und Erweiterung“ [Joc13][S.62]. Nach jeder dieser Tätigkeiten, muss ein System-, bzw. Regressionstest durchgeführt werden [And12][S.74]. Es müssen also nicht nur die Kosten des Systemtests in der Erstentwicklung, sondern zusätzlich die Wartungskosten beachtet werden. In diesem Fall liegen die Kosten für Wartung inklusive Integration und Test bereits bei etwa zwei Dritteln der Gesamtkosten des Projekts [Joc13][S.72]. Ein Teil dieser Kosten, entsteht durch den Systemtest. Abhängig davon wie groß der Funktionsumfang und wie kompliziert die Ausführung der Funktionalität, können manuelle Systemtests viele Entwicklerstunden benötigen.

Um den Aufwand zu verdeutlichen, ist hier ein Beispiel aus [And12][S.14] aufgeführt. Der Graph in Abbildung 2.1 zeigt den Kontrollflussgraph einer Anwendung mit vier Verzweigungen und einer Schleife um diese Verzweigungen herum. Die Schleife ergibt sich durch den Übergang von B zurück nach A. Um jeden Pfad durch die Verzweigungen einmal ausgeführt zu haben, benötigt man 5^1 Testfälle. Die Menge aller Kombinationen in einer Schleife ist natürlich unendlich, da die Schleife unendlich oft wiederholt werden kann. Begrenzt man nun die Anzahl der Schleifendurchläufe auf 20, ergibt sich noch immer $5^{20} + 5^{19} + \cdots + 5^1$ Testfälle [And12][S.14]. Bei einer Ausführungsdauer von fünf Minuten je Testfall, dauert eine manuelle Ausführung etwa $1.000.000.000$ Jahre [And12][S.14]. Sind die Tests automatisiert, dauert die Ausführung nur noch 19 Jahre, da im Beispiel von einer Laufzeit von fünf Mikrosekunden ausgegangen wird [And12][S.14]. Stellt jeder Pfad des Graphen eine Funktion des Programms dar, benötigt selbst eine einfache Funktionsüberdeckung des Prüflings 25 Minuten, da 5^1 Testfälle mit je fünf Minuten Dauer ausgeführt werden müssen. Eine automatisierte Funktionsüberdeckung ist in verschwindend geringer Zeit ausgeführt. Abgesehen davon, dass es in der Praxis unmöglich ist „vollständig zu testen“ [And12][S.15], zeigt es auch grob, wie viel Aufwand automatisierte Systemtests einsparen können.

Besonders für iterative Prozesse und in Anbetracht der Tatsache, dass der Aufwandsunterschied bei Regressionstests erneut ins Gewicht fällt, ist es lohnenswert die Ausführung der Systemtests weitestgehend zu automatisieren, solange dies unter Einhaltung der in Abschnitt 2.1 genannten Anforderungen an Systemtests möglich ist.

[2] In [Joc13] wird hier von „Black-Box-Test“ anstelle des Systemtests gesprochen. Die Aussage trifft jedoch auf den Systemtest, der durch Black-Box-Techniken ausgeführt wird, zu.

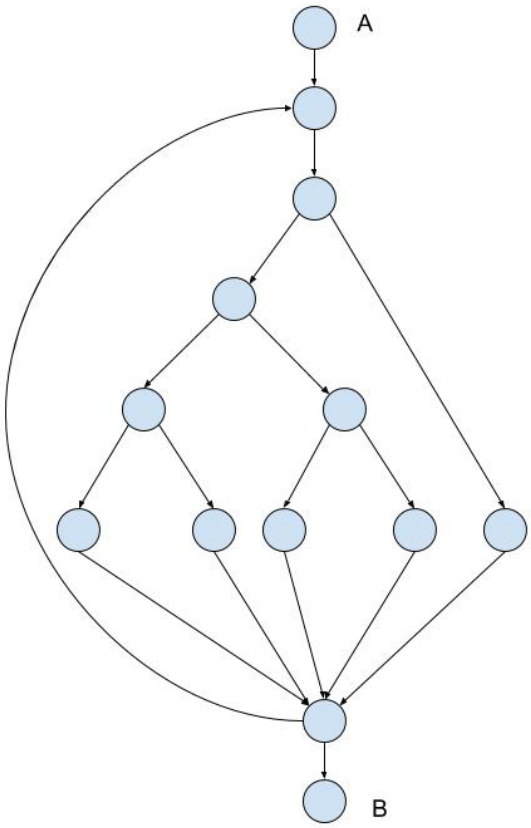

Abbildung 2.1.: Kontrollflussgraph einer Anwendung mit vier Verzweigungen [And12][S.14]

2.2.2. Aufwand der automatisierten Tests

Es existieren unterschiedliche Ansätze zur Automatisierung von Systemtests.

Capture-and-Replay Ein Test-Tool zeichnet die Eingaben an der Nutzeroberfläche, also Mausklicks und Tastatureingaben auf und speichert sie in einem Testskript ab [And12][S.210]. Diese Tools verfügen zwar über eine gewisse Robustheit gegen Änderungen der Nutzeroberfläche, bedürfen jedoch oftmals Nachbearbeitung bei Änderungen. Auch äußere Einflüsse von Simulatoren oder Netzwerkkomponenten im System müssen in die Skripts eingefügt werden, wenn sie nicht innerhalb des Programms gesteuert werden.

Oberflächenautomatisierung Es existieren zahlreiche Tools, welche die Automatisierung der Oberfläche rein programmatisch verwalten [Wik17b]. Sie nutzen Dienste der Betriebssysteme, um Objekte der Nutzeroberfläche zu erkennen [ldt17]. Im Gegensatz zu Capture-and-Replay Tools, müssen die Testskripts jedoch von Hand geschrieben werden. Die Objekt Repräsentanten können in einer objektorientierten Art und Weise in sogenannten „Page-Objects“ implementiert werden (siehe Abschnitt 2.3).

In dieser Ausarbeitung, wird lediglich die Oberflächenautomatisierung weiter behandelt. Trotz der händischen Erzeugung der Page-Objects, hat das Verfahren einige Vorteile. Einer davon ist die Objektorientierung der Page-Objects und somit die hohe Wiederverwendbarkeit und die vermeintlich gute Wartbarkeit. Zudem kann mit einem Capture-and-Replay (Car) Tool nur genau die Funktionalität aufgezeichnet werden, die bereits vorhanden ist und vor allem auch korrekt funktioniert, da der Testfall ja bei der manuellen Ausführung aufgezeichnet wird. Diese Methode kann also ausschließlich zur Regression eines manuellen Tests zum Einsatz kommen oder erzeugt einen hohen Mehraufwand bei der händischen Bearbeitung der Testskripts [And12][S.211]. Eine automatisierte Oberfläche hingegen kann anhand der Spezifikation, noch bevor die Funktionalität umgesetzt ist, implementiert werden. Die Implementierung der Page-Objects lässt sich dadurch ideal in den Software-Prozess eingliedern und sofort nach Spezifizierung der Anforderungen durchführen [Joc13][S.504f]. Zudem lässt sich so der „Test-first“ Ansatz[3] aus dem Komponententest auf Systemtests anwenden. Auch eine bessere Lesbarkeit der Tests ist gegenüber den Car Tests zu erwarten. Die Testskripts der Car Tools beinhalten in der Regel die Bezeichner der Oberflächenkomponenten, anhand derer sie diese erkannt haben. Wenn keine Komponente erkannt wurde sogar Koordinaten. Die Page-Objects verstecken diesen Bezeichner Code vor den Tests und bieten eine von Menschenhand geschriebene Schnittstelle zur Nutzeroberfläche an Abschnitt 2.3.

Die Automatisierung der Systemtests, ermöglicht es, den Aufwand der Tests deutlich zu verringern. Tests die bei manueller Ausführung Minuten dauern, können innerhalb von Sekunden ausgeführt werden. Jedoch ist die Oberflächenautomatisierung selbst mit einigem Aufwand verbunden. Die Tests müssen vorbereitet und die Ergebnisse analysiert und verifiziert werden. Vorbereitend muss die Spezifikation gelesen und verstanden werden [Joc13][S.504f]. Die relevanten Testfälle werden anhand der spezifizierten Funktionalität und dem gewünschten Überdeckungskriterium ausgewählt [Joc13][S.504ff]. Noch während der Entwicklung sind die Page-Objects zu schreiben. Da ein System nur selten in seiner eigentlichen Einsatzumgebung getestet wird, werden etwaige Simulatoren benötigt, die diese Umgebung simulieren. Sobald die Page-Objects implementiert sind, müssen die Testskripts geschrieben werden. Nach deren Ausführung, können die Testergebnisse verifiziert werden. Möglicherweise gab es falsch-positive Testergebnisse, die von einem Fehler bei der Automatisierung oder einem inkorrekt geschriebenen Testfall ausgelöst wurden. Anschließend steht die Korrektur der Fehler an, nach der die Testausführung wiederholt wird, um sicherzustellen, dass die gefundenen behoben und keine neuen Fehler eingebaut wurden [Joc13][S.484f]. Dieser erneute Test ist der Regressionstest [Joc13][S.484f].

[3] Kurze Erläuterung in [And12][S.49].

2.3. Page-Objects

2.3.1. Funktionsweise

Um die Automatisierung von Systemtests zu realisieren, muss die Nutzeroberfläche des Prüflings programmatisch bedient werden können. Es existieren etliche Testframeworks, die das ermöglichen [Wik17b]. Sie bieten die Funktionalität, Oberflächenobjekte wie Fenster, Buttons oder Textfelder zu finden und zu bedienen. Die folgenden Frameworks sind Open-Source und für unterschiedliche Betriebssysteme geeignet. Sie stehen repräsentativ für eine Vielzahl an proprietären und Open-Source Lösungen, die Nutzeroberflächen automatisieren können.

ldtp [4]

Testet Windows, Gnome und Java Applications

Lizenziert unter GNU LGPL

Selenium [5]

Testet Web-Applikationen

Webdriver basiert

Unterstützt verteiltes Testen auf SeleniumGrid

Lizenziert unter Apache License 2.0

Appium [6]

Testet Web- und native Applikationen auf iOS und Android

Nutzt Selenium Webdriver Protokoll

Lizenziert unter Apache License 2.0

Weitere Android alternativen für Appium sind Espresso [7] und Robotium [8]. Es kann also davon ausgegangen werden, dass für alle gängigen Systeme, die zur Oberflächenautomatisierung benötigte Technologie zur Verfügung steht. Die Frameworks haben alle eine sehr ähnliche Funktionsweise. Sie erkennen Oberflächenelemente anhand von Bezeichnern oder vorkommenden Strings. Das Linux Desktop Testing Project (Ldtp) verwendet beispielsweise die AT-SPI Layer, über die das Betriebssystem Desktop Dienste anbietet [ldt17]. Dadurch hat das Ldtp Zugriff auf die Bezeichner der angezeigten Oberflächenelemente wie der angezeigten Fenster oder Buttons.

[4] https://ldtp.freedesktop.org/wiki/
[5] http://www.seleniumhq.org/
[6] http://appium.io/
[7] https://google.github.io/android-testing-support-library/docs/espresso/
[8] https://github.com/RobotiumTech/robotium/wiki

Um Testcode wiederverwendbar und lesbar zu machen, lohnt es sich den Framework-Code in eigene Page-Objects zu verpacken. Für Ldtp gibt es bereits einen Ansatz[9], der eine objektorientierte Programmierweise mit Ldtp erleichtern soll.

2.3.2. Page-Object Entwurfsmuster

Ausführlichere Informationen, wie objektorientierte Page-Objects implementiert werden können, finden sich auf der Selenium Website[10]. Die in dieser Ausarbeitung beschriebenen Page-Objects beruhen hauptsächlich auf dem dort aufgeführten Entwurfsmuster. Der Begriff Page, auf Deutsch übersetzt „Seite“ passt bei Seleniums Web Applikationen, bei denen es sich tatsächlich um Webseiten handelt gut. Aus Konsistenzgründen, werden im Folgenden auch Fenster einer Desktop Anwendung mit Seite bezeichnet. Bei der Erläuterung des Page-Object Entwurfsmusters, dürfte klar werden, dass die Implementierung von Seiten und Fenstern weitestgehend gleich sind. Jede Seite des Prüflings benötigt ein eigenes Page-Object [sel17]. Jedoch können auch einzelne Sektionen einer Seite, die häufig wiederverwendet werden eigene Page-Object Implementierungen haben [sel17]. Alle Methoden eines Page-Objects geben wiederum ein Page-Object zurück. Eine Ausnahme davon sind Methoden, die den Zustand der Seite überprüfen lassen. Beispielsweise kann ein Page-Object ein Textfeld anzeigen und eine öffentliche[11] $showTextField()$ Methode implementieren, die es erlaubt im Testfall Assertions[12] zu schreiben um zu überprüfen, ob die Seite den richtigen Wert anzeigt[sel17]. Die sonstigen öffentlichen Methoden repräsentieren hier die Eingabemöglichkeiten, welche die Seite dem Nutzer anbietet. Wichtig hierbei ist, dass das Page-Object low-level code des Testframeworks und die interne Funktionalität der Seite versteckt[sel17]. Bietet ein Dialog beispielsweise die Möglichkeit einen „Cancel“ und einen „Ok“ Button zu drücken, so findet das Testframework die beiden Nutzeroberflächenobjekte über ihre internen Bezeichner und erkennt, dass es sich um Buttons handelt, bietet also eine „click“ Methode an. Im entsprechenden Page-Object, sollte die Suche nach den Buttons und das Klicken in entsprechenden Methoden „pressCancel“ und „pressOK“ versteckt sein, die wiederum die folgenden Page-Objects zurück geben. So versteckt das Page-Object den Framework-Code und bietet nur die Bedienung des Page-Objects an. Durch den Aufruf der Methoden wird so der Weg des Nutzers durch das Programm modelliert [sel17]. Existieren Vorbedingungen, die bei gleichem Input zu unterschiedlichen Ergebnissen führen, müssen diese in unterschiedlichen Methoden implementiert werden [sel17]. Wichtig ist es auf das Timing zu achten und Funktionen einzubauen, die den Framework-Code auf etwaige Verzögerungen beim Aufbau der Nutzeroberfläche warten lassen. Auch bei Netzwerkkommunikation kann es zu Verzögerungen kommen wie eine Anfrage auf einen Backend-Server. Damit der Test auf diesem Page-Object nicht fehlschlägt und das nachfolgende Objekt gefunden

[9] https://ldtp.freedesktop.org/wiki/Object-Oriented_LDTP
[10] https://github.com/SeleniumHQ/selenium/wiki/PageObjects
[11] Öffentlich im Sinne der Datenkapselung.
[12] https://en.wikipedia.org/w/index.php?title=Assertion_(software_development)&oldid=779561246

werden kann, muss die in der Spezifikation des Prüflings angegebene Reaktionszeit des Servers gewartet werden.

3. Zustand basiertes Testen

3.1. Einführung in Zustand basiertes Testen

In diesem Kapitel wird ein zustandsbasiertes Verfahren zum Test einer Software beschrieben, wie es in [Cho78] vorgestellt wird. Der Test stellt eine Evaluation gegen die Spezifikation dar [Cho78]. Die Methode impliziert Nutzereingaben und Reaktionen des Prüflings, daher lassen sich mit diesem Test die funktionalen Anforderungen überprüfen. Das Verfahren kann also im Systemtest eingesetzt werden, da dieser eben diese Anforderungen testet [And12][S.58]. Verwendet wird ein endlicher Automat dessen Knoten die Zustände des Programmes repräsentieren und dessen Kanten entweder Stimuli oder Operationen sind [Cho78]. Stimuli sind Impulse von außerhalb der Prüflings, also beispielsweise Nutzereingaben. Operationen sind Aktionen innerhalb des Systems, die von Stimuli ausgelöst werden [Cho78]. Es können jedoch nicht alle Softwaresysteme auf diese Art dargestellt und getestet werden [Cho78]. Nicht eingebettete Software mit einer umfangreichen Nutzeroberfläche oder Command line interface (CLI), sollten jedoch einfach in Operationen und Stimuli darstellbar sein. Dies ist im Einzelfall zu prüfen.

Für den verwendeten Automaten werden folgende Annahmen getroffen[Cho78]:

Der Automat ist vollständig spezifiziert.

Der Automat ist minimal.

Der Automat hat einen Startzustand.

Der Automat hat nur erreichbare Zustände.

Vollständig spezifiziert, bezieht sich hier auf den Begriff der Automatentheorie. Es bedeutet, dass für alle Zustände Z und alle Eingabezeichen X die Zustandsübergangsfunktion δ und die Ausgabefunktion λ definiert sind [Stu17]. Es bedeutet insbesondere nicht, dass der Automat den vollständigen Funktionsumfang des Prüflings beinhaltet. Wird die „fertige“ Software getestet, muss diese Bedingung natürlich erfüllt sein. In einem iterativen oder agilen Prozess, sollte der Graph jedoch nur soweit die Spezifikation wiedergeben, wie der Prüfling im aktuellen Iterationsschritt implementiert sein soll. Ein Automat welcher die Zustände des Prüflings repräsentiert, muss ein deterministischer endlicher Automat sein. Ein Programm muss in einem Zustand immer gleich auf eine Nutzereingabe reagieren. Es ist klar, dass ein System das sich nicht deterministisch verhält nicht testbar ist, da es unmöglich ist die Soll-Resultate zu definieren. Auch beginnt der Automat immer in einem bestimmten Startzustand und hat eine

Menge von Endzuständen, die das getestete Programm beenden [Cho78]. Er lässt sich also wie ein deterministischer endlicher Automat über ein Tupel wie in folgender Gleichung (3.1) definieren [Sch92]:

Z *ist die Menge der Zustände*

Σ *ist das Eingabealphabet*

δ *ist die Übergangsfunktion*

z_0 *ist der Startzustand*

E *ist die Menge der Endzustände*

$$M = (Z, \Sigma, \delta, z_0, E) \tag{3.1}$$

Um den Prüfling zu testen werden diese Schritte auf dem Automaten ausgeführt [Cho78]:

1. Schätze maximale Anzahl der Zustände im korrekten Automaten
2. Erstelle Durchläufe durch den Automaten
3. Verifiziere Zustand gegen Schritt 2

Die Schätzung der maximalen Anzahl Zustände des korrekten Automaten ist von Menschen gemacht [Cho78]. Im Verlauf der Erläuterung wird klar werden, dass dies einen gewissen Einfluss auf die Zuverlässigkeit des Verfahrens hat.

Testdurchläufe

Die Testdurchläufe werden anhand eines Testbaumes und einer Testsequenz erstellt [Cho78]. Die Testdurchläufe sind eine Konkatenation von P und Z. P ist eine Menge von Eingabesequenzen. Für jeden Übergang von einem Zustand A_i zu einem Zustand A_j existieren Eingabesequenzen p und px in P, sodass der Automat von seinem Initialzustand A_{start} in den Zustand A_j gebracht wird. px ist die Eingabesequenz p gefolgt von einer weiteren Eingabe x [Cho78].

P kann auch als ein Testbaum dargestellt werden, der sich aus dem Automaten erzeugen lässt. Jeder von der Wurzel des Baumes ausgehende Pfad ist ein $p \in P$ [Cho78]. Erzeugung des Testbaumes:

1. Die Wurzel des Baumes wird ein Knoten, der den Startzustand des Zustandsautomaten repräsentiert. Er wird mit dem Initialzustand A_{start} gelabelt.
2. Erzeuge die Tiefenebenen des Baumes wie folgt:

a) Gehe die Eingabesymbole des vom Wurzelknote repräsentierten Zustandes durch. Hat der Zustand einen Übergang für ein Eingabesymbol, dann füge einen Kindknoten zum Baum hinzu, dessen Label der Folgezustand A_i ist. Der Ast im Baum wird mit dem Eingabesymbol gelabelt.

b) Wiederhole dies mit den Knoten in der nächsten Tiefenebene von links nach rechts. Existiert ein Knoten mit dem selben Label bereits in einer höheren Ebene, wird der Knoten ein Blatt. Sonst gehe vor wie beim Wurzelknoten.

Abbildung 3.1 zeigt einen Automaten und einen Baum der aus diesem erzeugt wurden. Zustand 1 ist der Startzustand und Zustand 3 ein Endzustand. In diesem Beispiel ist zu erkennen, dass der Testbaum endliche Pfade für den Zyklus des Automaten enthält. Der Pfad „a, a, a“ terminiert nicht im Knoten 1 in Ebene 3, obwohl dieser Knoten bereits im Testbaum ist, da der Knoten mit „a“ beschriftet ist. Entscheidend ist, dass es einen Pfad zu 1 gibt, der andere Eingabesymbole verwendet als der bereits im Testbaum vorhandene Pfad, weshalb der Knoten 1 noch einmal behandelt werden muss. Im aufgezeichneten Beispiel ist eindeutig, dass neue Eingabesymbole verwendet wurden, da 1 bisher nur als Startzustand im Baum aufgenommen wurde und keine Eingabe getätigt wurde. Anschließend terminiert der Pfad jedoch, da der Knoten 2 bereits mit dem Label „a“ vorhanden ist. Abhängig davon in welcher Reihenfolge die Eingabesymbole durch gegangen werden, ergibt sich eventuell ein anderer Baum. Z ist eine Menge die wie in Gleichung (3.2) angegeben definiert wird [Cho78]. W wird als das „Characterization-Set“ bezeichnet [Cho78]. Es ist eine Verknüpfung aus Zustand, Eingabesymbol und Reaktion eines jeden Zustands im Automaten. X ist das Eingabealphabet. Da für die geschätzte maximale Anzahl Zustände im korrekten Automaten m und die Anzahl Zustände im Automaten des Prüflings n gilt: $m \geq n$, ist X^{m-n} eine Menge von Eingaben größer oder gleich Null.

$$Z := W \cup X \cdot W \cdots \cup X^{m-n} \cdot W \tag{3.2}$$

Das „Characterization-Set“ des Automaten in Abbildung 3.1 ist tabellarisch in Abschnitt 3.1 dargestellt. Die linke Spalte zeigt die Zustände des Automaten. Die Spalten rechts davon, zeigen die Operationen, welche ausgeführt werden, wenn im Zustand der Reihe das Eingabesymbol der Spalte (hier a oder b) eingegeben wird. Um das Beispiel einfach zu halten, sind die Operationen nur Platzhalter die nach dem Fortschritt des Übergangs durch den Automaten benannt sind. Die Menge der Testdurchläufe $P \circ Z$ zu diesem Automaten ist dann:

- **{a, b}**
- a ∘ **{a, b}**
- b ∘ **{a, b}**
- a a ∘ **{a, b}**
- a b ∘ **{a, b}**
- a a a ∘ **{a, b}**
- a a b ∘ **{a, b}**

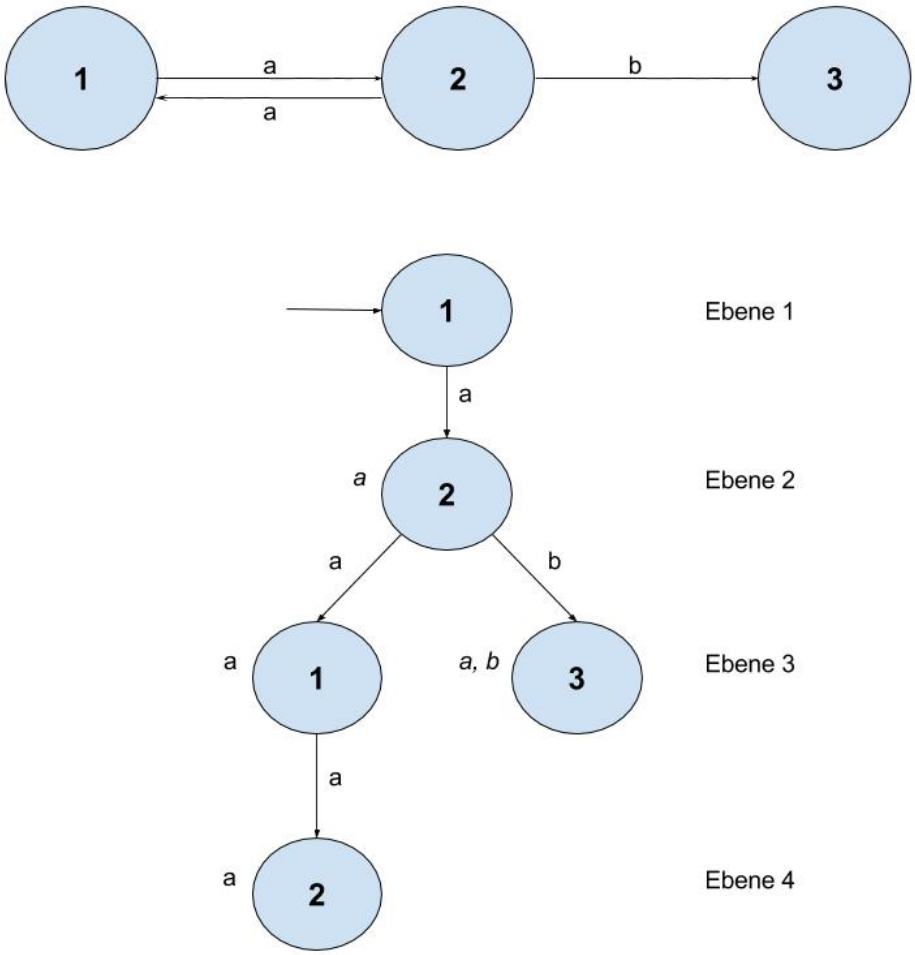

Abbildung 3.1.: Ein Automat und darunter der daraus erzeugte Testbaum.

Eingabesymbole / Zustände	a	b
1	fortfahren	halt
2	zurück	fortfahren
3	Ende	Ende

Tabelle 3.1.: „Characterization-Set“ des in Abbildung 3.1 aufgezeichneten Automaten.

3.2. Verifikation

Um den aktuellen Prüfling zu verifizieren, gibt es zwei Möglichkeiten.

Test-Mode Zunächst werden die korrekten Abläufe anhand der Spezifikation erstellt. Diese geben das Verhalten des Systems bei einer bestimmten Sequenz von Eingaben wieder [Cho78]. Anschließend werden die Reaktionen des Systems damit verglichen.

Walk-Through Mode Es werden „Pfad-Programme“ geschrieben, die Eingabesequenzen und die erwarteten Reaktionen beinhalten und die Testdurchläufe ausführen. Die Korrektheit dieser Programme muss anhand der Spezifikation sichergestellt werden [Cho78].

3.3. Zuverlässigkeit und Überdeckung

Das Verfahren des zustandsbasierten Tests, kommt nur dann für die Anwendung in einem automatisierten Systemtest in Frage, wenn die Aussagekraft und Zuverlässigkeit der Testergebnisse den Anforderungen genügen. Die Verifikation des Prüflings, wie sie nach dem in [Cho78] vorgestellten Verfahren durchgeführt wird, kann ausgedrückt werden, als die Frage, ob der Automat des Prüflings und der aus der Spezifikation abgeleitete Vergleichsautomat $P \cdot Z$ äquivalent sind [Cho78]. Also ob sich die Automaten bei Eingabe der Sequenzen aus der Menge $P \cdot Z$ gleich verhalten. Das Theorem 1 drückt die Voraussetzungen für einen nachgewiesen korrekten Prüfling in bezug auf die im vorgestellten Verfahren auffindbaren Fehler aus. In [Cho78] ist das Theorem und dessen Beweis aufgeführt.

Theorem 1
Der Prüfling ist fehlerfrei genau dann, wenn der Automat des Prüflings und der korrekte Automat $P \cdot Z$ äquivalent sind und folgende Bedingungen gelten:

1. *Beide Automaten haben das selbe Eingabealphabet*
2. *Die Schätzung der maximalen Anzahl an Zuständen(m) ist korrekt*

Um das Verfahren vergleichen zu können, ist in der Abschnitt 3.3 eine aus [Cho78] abgeleitete Vergleichsmatrix abgebildet. „x“ bedeutet, dass alle Fehler dieses Typs gefunden werden, „/“ dass micht alle Fehler dieses Typs gefunden werden und „-“ bedeutet, dass dieser Fehlertyp nicht gefunden wird.

Operation Error Ein Automat führt bei einer Eingabe die falsche Operation aus. Zustand und Folgezustand sind jedoch richtig.

Transfer Error Ein Automat hat einen Übergang, der bei einer korrekten Eingabe zum falschen Folgezustand übergeht.

Fehlertyp / Überdeckung	Operation Error	Transfer Error	Extra States	Missing States
Branch Cover	x	-	-	-
Switch Cover	x	/	-	-
Boundary-Interior	x	/	-	-
n-switch cover	x	x	x	-

Tabelle 3.2.: Vergleichsmatrix Testüberdeckungstechniken

Extra States / Missing States Die Anzahl der Zustände der Automaten stimmt nicht über ein. Dies bedeutet immer einen Fehler, da das Verfahren von minimalen Zuständen ausgeht, deren Anzahl Zustände also gleich sein müssen, um äquivalent zu sein [Sch92][S.37].

Die Arten der Überdeckung hier im Vergleich sind Kontrollfluss orientiert und können deshalb mit dem hier vorgestellten Verfahren verglichen werden [Cho78]. Die Überdeckungen werden folgendermaßen erreicht [Cho78]:

Branch cover Eingabesequenzen bestehen aus Eingaben, die jeden Übergang des Automaten mindestens einmal durchlaufen.

Switch cover Eine Erweiterung der Branch cover. Die „Switches“ bestehen aus Übergangspaaren. Die „Switches“ für den Automaten in Abbildung 3.1 sind {12, 21, 23 } wobei 12 den Übergang von Zustand 1 zu Zustand 2 benennt.

Boundary-interior cover Eine Branch cover, jedoch wird zusätzlich jede Schleife des Automaten betreten und ein mal iteriert. Für den Automaten in Abbildung 3.1 genügt die Eingabesequenz {a, a, a, a, b} um die Überdeckung zu erreichen.

Das vorgestellte Verfahren findet im Gegensatz zu den anderen Überdeckungsstrategien alle Fehlertypen [Cho78]. Sind die Bedingungen in Theorem 1 erfüllt, werden sogar alle vorhandenen Fehler dieser Fehlertypen entdeckt. Das Verfahren ist also zuverlässig und valide [Cho78]. Bei der Vorbereitung und Ausführung von Tests, auch in Bezug zur Automatisierung, ist jedoch nicht nur die Zuverlässigkeit entscheidend. Oft stehen zum Test nur begrenzt Zeit und Ressourcen zur Verfügung, weshalb die Dauer und Kosten des Verfahrens eine Rolle spielen [And12][S.15ff]. Bei einem algorithmischen Verfahren wie dem hier vorgestellten, ist die Komplexität der Algorithmen eine Orientierungshilfe, anhand derer sich der Aufwand bewerten lässt.

3.4. Komplexität

Die Aufwändigen Schritte des Verfahrens sind die Konstruktion der Menge P und des „Characterization-set“ W. Um P zu bestimmen erzeugt man einen Testbaum aus dem Automaten und benennt alle Pfade des Baumes wie in Abschnitt 3.1 beschrieben. Dies geschieht

für alle Zweige die aus dem Automaten abgeleitet werden. Das sind etwa $n \times k$ wobei n die Anzahl der Zustände und k die Menge der Eingabesymbole ist [Cho78]. W lässt sich über P_k Tabellen erzeugen [Cho78], die alle Kombinationen von Eingaben darstellen. Ein minimaler Automat hat $n - 1$ solcher P_k Tabellen, die jeweils etwa $n \times k$ Schritte benötigen. Der nötige Aufwand um W zu erzeugen ist also etwa $(n - 1) \times n \times k$ also grob $n^2 \times k$ [Cho78]. Quadratische Laufzeit ist im Allgemeinen bei einem Algorithmus nicht wünschenswert. Jedoch wird nur vor Beginn der Systemtests und für den Regressionstest[1] der Baum aufgebaut und Eingabesequenzen bestimmt. Für den Regressionstest gilt dies auch nur, wenn ein vollständiger Systemtest erneut ausgeführt wird, was in der Praxis häufig aufgrund mangelnder Resourcen vernachlässigt wird [And12][S.75]. In jedem Fall, müssen die Fehler verursachenden Eingaben aus der ursprünglichen Testreihe ausgeführt werden. Bei einem vollständigen Regressionstest, müssen neue Sequenzen aus $P \cdot Z$ generiert werden. In einem iterativen Prozess, bedeutet dies zweimal eine Komplexität von $n^2 \times k$ und $n \times k$ je Iteration.

[1] Wird in einem Test ein Fehler gefunden, muss nach der Korrektur des Fehlers ein gleichwertiger Test erneut ausgeführt werden, um zu überprüfen, dass der Fehler behoben und keine neuen Fehler eingebaut wurden. Dieser Test wird Regressionstest genannt [And12][S.74f]

4. Monkey-Tests

4.1. Infinite Monkey Theorem

In diesem Abschnitt sollen stochastische Test Methoden behandelt werden. Dies sind Methoden, die zufällige Eingaben verwenden um den Prüfling zu testen. Genauer geht es um so genannte Monkey-Tests. Diese sollen vorgestellt werden. Das Prinzip der Monkey-Tests, geht auf das „Infinite Monkey Theorem“ zurück, dass Theorem 2 ist eine Übersetzung des Theorems. Der Affe ist eine Metapher für eine Maschine, die zufällige Eingaben tätigt [Wik17c].

Theorem 2
Ein Affe der für eine unendliche Zeit zufällige Tasten auf einer Schreibmaschine tippt, wird mit hoher Wahrscheinlichkeit einen Text, wie die gesammelten Werke von William Shakespeare hintereinander abtippen [Wik17c].

Dieses Theorem lässt sich auch auf Software Tests übertragen. Ein Affe, der unendlich lange Eingaben in ein Programm tätigt, führt mit hoher Wahrscheinlichkeit alle Testfälle aus, die auch ein menschlicher Tester ausführen würde. Mehr noch, ein Affe führt alle **relevanten** Testfälle aus, also nicht nur die von einem Tester ausgewählten, sondern auch all jene die von einem Nutzer in der Produktivumgebung ausgeführt werden. Natürlich ist es für einen Softwaretest in der Praxis unmöglich eine unendliche Zeit zu Testen. Jedoch selbst für große endliche Zeiträume ist die Wahrscheinlichkeit relevante Testfälle auszuführen sehr gering. Dem Affen des Theorems entsprechen die so genannten „Dumb Monkeys“ [Nym00]. Diese ignorieren wie ein Mensch die Software bedienen würde und wählen ihre Eingaben zufällig und ohne den Zustand des Prüflings zu kennen aus [Nym00]. Diese Tests können kostengünstig und sobald ein lauffähiger Prüfling existiert ausgeführt werden [Nym00]. Sie eignen sich für Belastungstests und finden „memory leaks“ [Nym00]. Zudem führen die „Dumb Monkeys“ komplexe Eingabesequenzen aus die ein Tester sich nicht ausdenken würde. Nyman schreibt dazu in [Nym00], dass „Dumb Monkeys nicht viele Fehler finden werden, jene die sie finden jedoch Abstürze und Aufhänger sind“, denn häufig treten diese Fehler nur bei komplexen und verketteten Eingaben auf. Für einen Systemtest und besonders einen systematischen Test, sind die „Dumb Monkeys“ jedoch nicht geeignet, da es ihnen an Struktur und Dokumentation fehlt. Auch führen sie eine Vielzahl an Eingabesequenzen aus, die keine Rolle beim Erreichen gewünschter Überdeckungskriterien spielen. Ein „Dumb Monkey“ erreicht eventuell erst nach einem Tag eine einfache Funktionsüberdeckung, die durch einen Menschen in Minuten oder wenigen Stunden erreicht werden kann.

4.2. Smart Monkeys

Um die Präzision des Affen zu erhöhen, können „Smart Monkeys“ verwendet werden [Nym00]. Diese verwenden ein Zustandsmodell des Prüflings und kennen die Funktionalität auf einfacher Ebene [Nym00]. In [Nym00] wird als Zustandsmodell eine Zustandstabelle genannt, diese sind jedoch äquivalent mit einem Zustandsgraphen wie im Zustand basierten Test aus Kapitel 3, da die Tabellen Zustände, Übergänge und Eingabesymbole enthalten. Durch die Kenntnis der Zustände, werden nur noch dem Zustandsmodell entsprechende Eingaben ausgeführt. Trotzdem erzeugen die „Smart Monkeys“ noch immer Eingabesequenzen, an die ein menschlicher Tester nicht denken wird, die trotz allem von Nutzern ausgeführt werden [Nym00]. Ein Grund dafür, dass „Smart Monkeys“ näher an den Eingaben eines echten Nutzers liegen als manuell geschriebene Tests ist, dass die manuellen Tests zwischen jedem Testfall zu einen bekannten Standartzustand zurückkehren [Nym00]. Damit löschen sie die Historie der Ausführung. In der Praxis ist es jedoch eher unwahrscheinlich, dass ein Nutzer den Prüfling startet, genau eine Funktion ausführt und das Programm dann, möglicherweise sogar durch einen Neustart, in eine Ausgangssituation zurück versetzt. Die „Smart Monkeys“ führen deutlich komplexere Sequenzen aus [Nym00] und können so vom Nutzer hervorgerufene Situationen simulieren und entdecken auch Fehlerkaskaden. In Abbildung 4.1 sind drei Mengen abgebildet, die den zu erwartenden Überdeckungsgrad, der durch die jeweiligen Verfahren erreicht werden wird, zeigen. Gültig ist dies jedoch nur, wenn die stochastischen Tests, also die „Smart Monkeys“ und die „Dumb Monkeys“ eine sehr lange Zeit ausgeführt werden. Die gestrichelt dargestellte Menge, stellt die voraussichtlich relevanten Testfälle dar. Das sind jene Testfälle, bei denen der Aufwand für das Finden in ökonomischer Relation zur Eintrittswahrscheinlichkeit und dem verursachten Schaden steht.

Der große Nachteil sind die hohen Kosten, die beim Erstellen des Zustandsmodells entstehen. N. Nyman schreibt in [Nym00], dass bei einem „Projekt mit moderater Komplexität“ ein Zustandsmodell mit 50000 Knoten benötigt wird. In einem iterativen Prozess, steigt die Zahl der Zustände konstant an, da neue Funktionalität und Änderungen übernommen werden müssen [Nym00]. Der Aufwand ist also nicht nur initial zu bewältigen, sondern zieht sich durch das gesamte Projekt. Die Kosten für die Vorbereitung und Ausführung der Tests, sollte die Kosten, die entstehen wenn ein Fehler auftritt, nicht übersteigen [And12][S.15]. Andernfalls ist der Test nicht „ökonomisch sinnvoll“ [And12][S.15]. Eben dies kann geschehen, wenn der Aufwand das Zustandsmodell zu erstellen und zu warten zu hoch ist [Nym00].

Ein weiterer Kostenfaktor sind die Testergebnisse. Die erwarteten Kosten eines Fehlers, ergeben sich aus dem finanziellen Schaden den er anrichtet und seiner Eintrittswahrscheinlichkeit [And12][S.15, S.178f]. Ein „Smart Monkey“ kann theoretisch hundert mal einen Dialog öffnen und wieder schließen. Anschließend, wird der Dialog fehlerhaft angezeigt. Die Kosten eines solchen Fehlers sind in den meisten Fällen verschwindend gering, da ein Nutzer kein derartiges Verhalten zeigt und der Fehler im Feld wohl nie eintreten wird. Trotzdem findet der Test den Fehler und einem Tester wird das Ergebnis vorliegen. Der Tester muss dann die Eingabesequenz nachvollziehen, den Test idealerweise wiederholen um die Reproduzierbarkeit zu verifizieren,

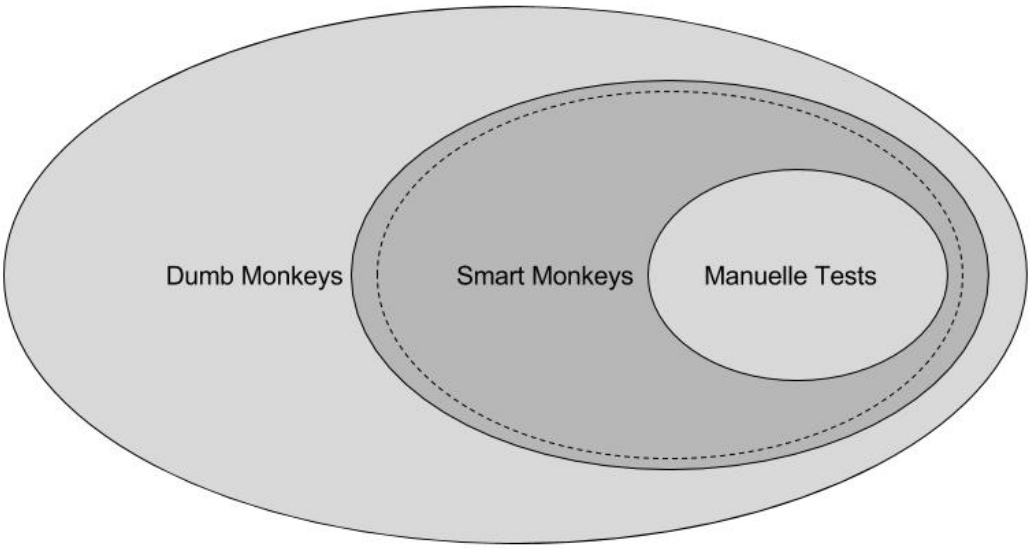

Abbildung 4.1.: Wahrscheinliche Testfallüberdeckung nach sehr langer Ausführungsdauer.

den Fehler dokumentieren und erst dann kann ein Verantwortlicher den Fehler priorisieren. Die Wirtschaftlichkeit des Aufwands, für das Bearbeiten der Testergebnisse eines „Smart Monkey“ Tests insgesamt, ist von folgenden Faktoren abhängig:

1. Lesbarkeit und Nachvollziehbarkeit der Testergebnisse
2. Anzahl der „irrelevanten“[1] Fehler
3. Anzahl der falsch-positiven Testergebnisse

Letzteres hängt stark vom Prüfling und seiner Komplexität ab. Das Timing ist erfahrungsgemäß immer eine häufige Fehlerursache, selbst bei manuell geschriebenen Tests. Ein weit verteiltes System mit viel Netzwerkkommunikation, läuft häufig Gefahr aufgrund von Verzögerungen Fehler zu erzeugen, die in Wirklichkeit keine Fehler des Systems selbst, sondern der Testinfrastruktur bedeuten.

Die Anzahl der „irrelevanten“ Fehler die gefunden werden, hängt von der Auswahl der Testfälle ab. Ein „Smart Monkey“, der 1000 mal die selbe Aktion ausführt, ist höchstwahrscheinlich weit weg vom Verhalten eines echten Nutzers. Führt er jedoch 10 bis 50 mal eine Aktion aus, simuliert er vermutlich Situationen, die ein Nutzer herbeiführen wird. Welches Nutzerprofil relevant ist, kommt auf den Prüfling und seine spätere Anwendung an.

[1] Natürlich sollte das Ziel sein möglichst alle Fehler des Systems zu beheben. Wie bereits erwähnt, existieren jedoch Fehler, bei denen der Aufwand für die Fehlerfindung und Korrektur nicht im Verhältnis zu den Fehlerkosten steht. Diese Fehler sind hier unter „irrelevanten“ Fehlern zu verstehen.

Die Lesbarkeit der Testergebnisse ist wohl der schwerwiegendste Faktor für die Kosten eines „Smart Monkeys“, zumal er die Auswirkungen der anderen Hauptfaktoren mindern kann. Dies liegt trivialer Weise daran, dass falsch-positive und irrelevante Testergebnisse schneller erkannt werden, wenn der Tester die Ergebnisse gut nachvollziehen kann. Die Studie in [CMM+12] beschäftigt sich mit der Lesbarkeit von zufällig generierten Tests im Vergleich zu manuell geschriebenen. Jedoch handelt es sich um Unit-Tests und nicht um Systemtests. Verglichen wurde die Genauigkeit und Effizienz zweier Gruppen mit vergleichbaren Fähigkeiten, also die Anzahl der korrigierten Fehler und die Anzahl der korrigierten Fehler pro Minute [CMM+12]. Eine Gruppe verwendete die zufällig erzeugten Tests, die andere die von Menschen geschriebenen. Der Hauptunterschied der manuellen Tests zu den randomisierten bestand darin, dass die randomisierten Tests automatisch generierte, nichtssagende Bezeichner verwendeten, jedoch weniger komplex waren. Da die randomisierten Tests in allen Versuchen besser abschnitten, schließen die Autoren der Studie, dass die Komplexität bei der Analyse der Testfälle der größte Faktor für die Nachvollziebarkeit der Tests war [CMM+12]. Um diese Studie auf die „Smart Monkeys“ zu übertragen, muss jedoch beachtet werden, dass es sich hier um Black-Box-Tests handelt. Der Tester hat also keine Einsicht, was mit den Objekten, die hinter den Bezeichnern stehen geschieht. Alles was der Tester beim Systemtest weiß, ist was auf der Oberfläche passiert. Zur Nachvollziehbarkeit der Tests, ist zusätzlich zur Komplexität vermutlich wichtig, dass er die Bezeichner aus den Testergebnissen Objekten der Nutzeroberfläche zuordnen kann. Der Schluss, der sich aus der Studie für Systemtests schließen lässt, ist trotz der größeren Rolle der Bezeichner, dass die Komplexität der Testfälle die Nachvollziehbarkeit entscheidend beeinflusst.

Mit „Smart Monkeys“ lassen sich also viele Situationen testen, die ein Nutzer herbeiführt, in manuellen Tests jedoch nicht ausgeführt werden. Gerade in komplexen Systemumgebungen kommt dieser Vorteil zum tragen [Nym00]. Die Kosten für das Erstellen des Zustandsmodells sind jedoch sehr hoch und auch die Auswertung der Testergebnisse kann, vor allem aufgrund der hohen Komplexität der Eingabesequenzen, die „Smart Monkeys“ unökonomisch werden lassen. Sie eigenen sich also nur für leicht automatisierbare Prüflinge [Nym00].

5. Das Verfahren

5.1. Problemdefinition

Um Ressourcen in der Software Entwicklung zu sparen, sollen Zustand basierte Systemtests automatisch generiert werden. Im Idealfall erzielen die automatisch generierten Tests eine andere, mindestens genauso große Überdeckung wie manuell ausgeführte oder manuell erzeugte automatisierte Tests und erfüllen die in Kapitel 3 genannten Anforderungen. Dafür in Frage kommen automatisierte stochastische Tests, die „Smart Monkeys“ verwenden, um anhand eines Zustandsmodells randomisierte Tests auf dem Prüfling auszuführen. Diese Testmethode hat zwei große Einschränkungen.

1. Die Kosten der „Smart Monkeys“.
2. Das implizite Soll-Resultat.

Die Kosten der „Smart Monkeys“, insbesondere die für den Aufbau des Zustandsgraphen, sind in Kapitel 4 näher erläutert. Da die Testscripts nicht von Hand geschrieben werden, kennen die Tests keine explizit angeführten Soll-Resultate. Ein von Hand geschriebener Test, enthält implizit die Abläufe und erwarteten Page-Objects und explizit Assertions [Wik17a], mit denen der Tester die erwarteten Reaktionen weiter spezifizieren kann. Auto generierte Tests haben lediglich die Informationen, die im Zustandsmodell enthalten sind [Nym00]. In herkömmlichen Zustandsgraphen, wie sie von T. S. Chow in [Cho78] verwendet werden, sind Kontrollflussinformationen enthalten. Das Soll-Resultat ist in diesen Graphen sehr umfangreich. Der Aufbau eines solchen Zustandsgraphen bedarf jedoch eines hohen Aufwands[1] [Cho78].

Gesucht ist also ein Verfahren, dass den Aufwand der Testvorbereitung minimiert, den Umfang des impliziten Soll-Resultats maximiert und ein Testauswahlverfahren verwendet, das möglichst nahe an echtem Nutzerverhalten ist und trotzdem Aussagen zum Überdeckungsgrad zulässt. Zudem sollen die Testergebnisse möglichst relevante Fehler erkennen. Für die letzte Eigenschaft soll in Kapitel 6 untersucht werden, wie effektiv Lernalgorithmen eingesetzt werden können, um irrelevante Pfade durch den Testgraphen zu eliminieren.

[1] Algorithmus in quadratischer Laufzeit und das manuelle Erzeugen des Zustandsautomaten aus der Spezifikation.

5.2. Erzeugung eines Zustandsgraphen aus Page-Objects

Im Kapitel 3 wird der Systemtest basierend auf dem Zustandsgraph des Prüflings behandelt. Von der Validität und Korrektheit des Verfahrens, wird in diesem Abschnitt ausgegangen. Um den Aufwand, der durch die Vorbereitung des Verfahrens entsteht zu minimieren, sollen die in Kapitel 2 vorgestellten Page-Objects verwendet werden, um den Zustandsgraph des Programms automatisch zu generieren. Erhoffen lassen sich dadurch geringere Kosten und ein reduzierter Aufwand für den Systemtest. Ausgangspunkt des Verfahrens ist ein Programm mit einer Nutzeroberfläche, dass den Prüfling darstellt. Es liegt die Spezifikation des Prüflings vor, eventuell sogar bereits eine Implementierung.

Aus der Spezifikation abgeleitet, hat ein Softwareentwickler bereits Page-Objects geschrieben, anhand derer der gewünschte Programmablauf in Form von Eingaben und den Reaktionen des Programms simuliert werden kann. Diese Page-Objects zu implementieren ist bereits mit hohem Aufwand verbunden. Man beachte jedoch, dass die Page-Objects auch für manuell geschriebene automatisierte Tests benötigt werden, die Oberflächenautomatisierung[2] verwenden. Es entsteht dadurch also zunächst kein Mehraufwand im Vergleich zur konventionellen Testautomatisierung. Das Page-Object Entwurfsmuster, induziert durch die Page-Object erzeugenden Methoden, den Kontrollfluss der Nutzeroberfläche in die Page-Object Klassen. Eine Aktion auf Objekt A, führt zu Objekt B als Reaktion. Um sich diesen Umstand zu Nutze zu machen, soll nun anhand des Quellcodes dieser Page-Objects ein Zustandsautomat abgeleitet werden. Dazu muss ein Programm existieren, das im Folgenden als Erzeugerprogramm bezeichnet wird. In diesem Zustandsautomaten stellen die Page-Objects die Knoten und die Methoden der Page-Objects die Übergänge dar. Dem Erzeugerprogramm muss eine (abstrakte) Page-Object Klasse, hier $AbstractPageObject$ vorliegen, welche die Superklasse aller Page-Objects darstellt. Es werden dann nur Klassen beachtet, die Subklassen dieser Klasse sind. Zudem muss ein Page-Object als „Root“ Knoten angegeben werden. Dieses muss das erste bei Programmstart angezeigte Page-Object repräsentieren, damit der Ablauf der Tests der echten Programmausführung entspricht. Zu beachten ist, dass sich die Übergänge und Knoten leicht von denen in [Cho78] vorgestellten unterscheiden. Dort sind die Knoten die Zustände und Übergänge sind Operationen oder Stimuli [Cho78]. Da die Page-Objects keinen Einblick in die Implementierung des Programmcodes gewähren, können aus ihnen nicht ohne weiteres Operationen abgeleitet werden. Ohnehin sollen die erzeugten Tests Black-Box Tests und ohne Kenntnis von den Abläufen im Programmcode, sein [Joc13]. Die Zustände werden statt vom internen Zustand des Programms, von der dem Nutzer angebotenen Funktionalität abgeleitet. Operationen sind Eingaben des Nutzers und Stimuli sind externe Einflüsse wie Interaktionen mit Simulatoren oder Datenbankzugriffe. Der Zustandsgraph, der aus den Page-Objects erzeugt wird, setzt also eine Abstraktionsebene höher an, da der Nutzer als intern und Netzwerkeinflüsse als extern betrachtet werden, anstatt die Operationen im Programm als intern und die Eingaben des Nutzers als externe Stimuli zu betrachten [Cho78]. Die Überprüfung der Programmfunktionalität läuft

[2] siehe Abschnitt 2.2.2

auf beiden Automaten gleich ab. Eine Eingabe wird getätigt, die Nutzeroberfläche ist in einem neuen Zustand, dieser Zustand wird verifiziert. Wichtig beim Schreiben der Page-Objects für das Erzeugerprogramm ist, dass jede Eingabe, die für den Zustand des Programms relevant ist, in einer Methode stattfindet, die die erwartete Reaktion kennt und über das zurückgegebene Page-Object repräsentiert. Nur so erkennt das Erzeugerprogramm, dass der Prüfling über die Eingabe in einen neuen Zustand übergeht.

5.2.1. Analyse der Page-Object Klassen

Das Page-Object Entwurfsmuster erlaubt nur öffentliche Methoden, deren Rückgabewert erneut ein Page-Object sind, außer sie dienen der Kontrolle der angebotenen Dienste auf dieser Seite [sel17]. Für die Erzeugung des Zustandsautomaten, sind zunächst jedoch nur die Page-Object erzeugenden Methoden relevant. Sei der Knoten K_1 nun der „Root“ Knoten also das gegebene erste Page-Object. Das Erzeugerprogramm betrachtet nun alle Methoden dieses Page-Objects und erzeugt von K_1 ausgehende Kanten für jede, die ein Objekt das von $AbstractPageObject$ erbt, als Rückgabewert hat. Die Kanten münden dann in Knoten $K_2 ... K_n$, welche die jeweils von der Methode erzeugten Page-Objects repräsentieren. Das Erzeugerprogramm wiederholt dies dann rekursiv in den Page-Objects der Knoten $K_2 ... K_n$ und deren Kindknoten. Ausgehend von K_1 geht es so durch alle Page-Objects, bis für alle Methoden in allen Page-Objects Kanten existieren. Führt eine Kante zu einer Page-Object-Klasse, die bereits untersucht wurde, entstehen Zyklen. Ein zusätzlicher Fall, sind Methoden, die das Programm beenden sollen. Diese haben alle eine Kante auf denselben Endknoten, der kein Page-Object repräsentiert.

Externe Stimuli

Bei einer Programmausführung, können externe Stimuli das Programm beeinflussen. Dies geschieht entweder automatisch, also ein Server oder Simulator sendet eine Anfrage an das Programm, welches dann darauf reagiert oder der Nutzer setzt aktiv eine Anfrage ab, die meist auch eine Reaktion des Angesprochenen Simulators hervorruft. Wird beispielsweise eine Nachricht an den Prüfling gesendet und dieser soll einen Dialog als Reaktion darauf öffnen, muss der externe Stimuli in einer Methode des Page-Objects verpackt sein. Diese erzeugt das Page-Object des Dialogs, da der Aufruf neuer Page-Objects durch die öffentlichen Methoden ausgelöst wird [sel17]. Werden die Tests manuell geschrieben, löst ein Tester im Test aktiv die Anfrage des Simulators aus. Da die Tests jedoch nicht geschrieben werden, muss die Interaktion mit dem Simulator in den Page-Object Methoden geschehen, damit das Erzeugerprogramm sie in den Kontrollfluss aufnimmt.

Abbildung 5.1 zeigt ein Beispiel eines Login Dialogs. Zuerst wird ein Login Fenster angezeigt (1) in dem der Nutzer Nutzername und Passwort eingeben muss. Bei einem klick auf $Login$ werden die Login Daten an einen Server weiter geleitet. Sind die Nutzerdaten valide, öffnet

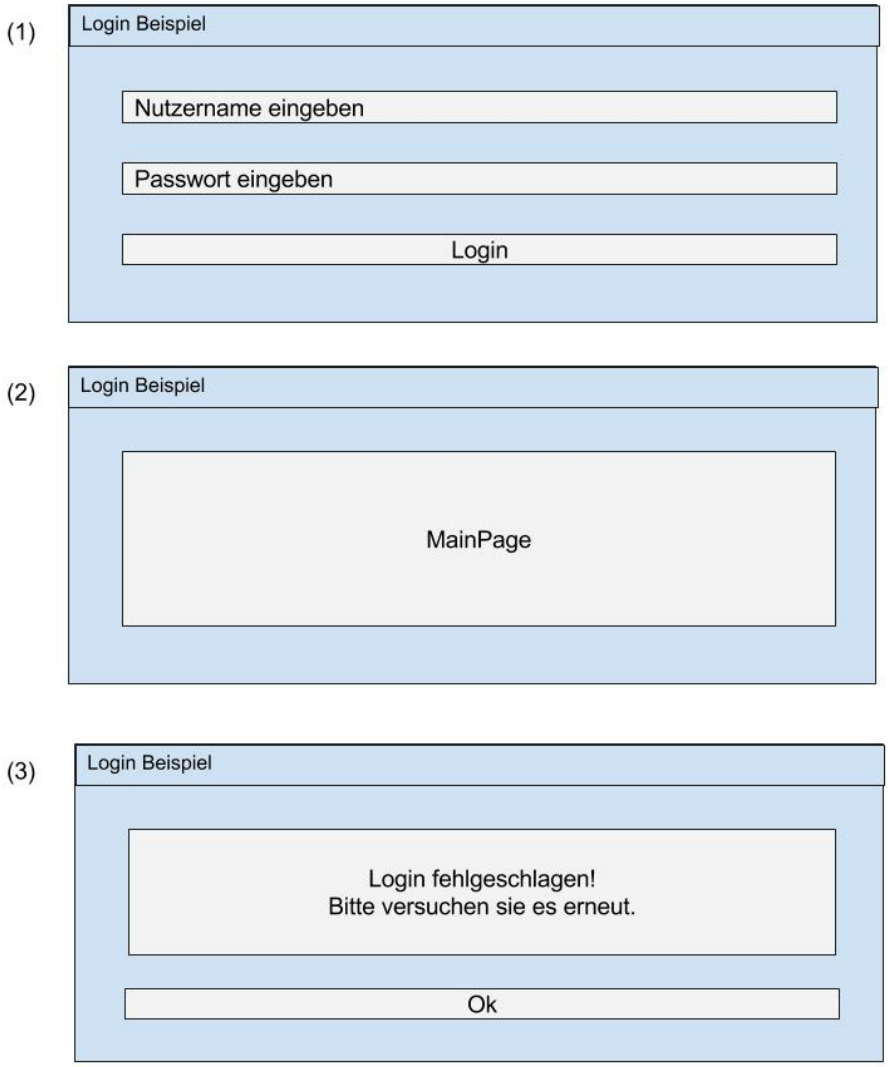

Abbildung 5.1.: Beispiel eines Login Dialogs

sich die $MainPage$ (2). Sind die Nutzerdaten falsch, öffnet sich der $Loginfehlgeschlagen$ Dialog (3).

Abbildung 5.2 zeigt einen Automaten, der den aus den Page-Objects gewonnenen Kontrollfluss zeigt, wenn keine Methoden für unterschiedliche Reaktionen der Simulatoren geschrieben wurden. Das $Login$ Fenster Page-Object hat lediglich eine Methode zum Login mit den gegebenen Daten. Im Tests selbst, wird dann bestimmt, welche Nutzerdaten eingegeben werden und welche Reaktion erwartet wird. Auch wenn unterschiedliche Zustände des Servers getestet werden sollen, muss dies dann im Test geschehen. Der Server muss dort aktiv beeinflusst werden, wenn etwa getestet werden soll, wie sich der Prüfling verhält, wenn der Server nicht

Listing 5.1 Login Methoden die unterschiedliche Reaktionen hervorrufen.

```
public MainPage loginWithCorrectCredentials() {
      userNameField.enter("EchterNutzer");
      passwordField.enter("EchtesPasswort");
      loginButton.click();
      wait(4);
      return new MainPage();
}

public LoginFailedPage loginWithIncorrectCredentials() {
      userNameField.enter("FalscherNutzer");
      passwordField.enter("FalschesPasswort");
      loginButton.click();
      wait(4);
      return new LoginFailedPage();
}
```

erreicht werden kann. Der Graph aus Abbildung 5.2 eignet sich also nicht für einen vollständigen automatisierten Test, da in der Regel mehr Szenarios als nur ein korrekter Login getestet werden sollen [Joc13][S.505].

Abbildung 5.3 zeigt einen Automaten, dessen Page-Objects die für alternative Testfälle nötigen Methoden implementieren. Der $LoginScreen$ bietet eine Methode für korrekte und eine Methode für falsche Login Daten an. Eine Methode führt zur $MainPage$, eine andere zur $Loginfehlgeschlagen$ Seite. Zudem gibt es eine Methode die Nutzerdaten eingibt und versucht sich einzuloggen, wenn der Server nicht erreichbar ist. Dieser landet ebenfalls auf der $Loginfehlgeschlagen$ Seite. Von dort führt eine Methode für den Klick auf Ok wieder zurück zur $Login$ Seite.

In Listing 5.1 sind die beiden zusätzlichen Methoden des $LoginScreen$ Page-Objects skizziert, die für unterschiedliche Login Daten nötig sind. Die erwartete Reaktion auf einen erfolgreichen Login, ist das Erscheinen der MainPage. Daher gibt die $loginWithCorrectCredentials()$ Methode ein $MainPage$ Objekt zurück. Die Nutzerdaten werden über das verwendete Framework zur Automatisierung in die Textfelder eingegeben und anschließend auf den $Login$ Knopf geklickt. Eine $wait(s)$ Funktion zeigt dem Testframework an, dass eine Zeit s gewartet werden soll, bis das korrekte Erscheinen des Page-Objects überprüft wird. Sollte die Reaktionszeit des Programms oder des Servers größer als s sein, wird der Test fehlschlagen. Die $loginWithIncorrectCredentials()$ Methode geht gleich vor, gibt aber falsche Nutzerdaten ein und gibt deshalb eine $LoginFailedPage$ zurück.

Die Methoden die auf den von den Zugrunde liegenden Testframeworks basieren, wie die $enter(Strings)$ Methode oder $click()$, sind hier exemplarisch zu betrachten. Die Signatur und auch die Funktionalität, hängt im Einzelfall vom verwendeten Testframework ab. Das Testframework sucht in den Konstruktoren der Page-Objects immer nach dem entsprechenden Oberflächenelement. Wird es nicht gefunden, verhält sich der Prüfling offenbar falsch und der Test schlägt fehl.

Listing 5.2 Server nicht erreichbar Methode

```
public loginFailedScreen loginWhenBackendUnavailable() {
      userNameField.enter("EchterNutzer");
      passwordField.enter("EchtesPasswort");
      server.setUnavailable();
      loginButton.click();
      wait(4);
      return new LoginFailedPage();
```

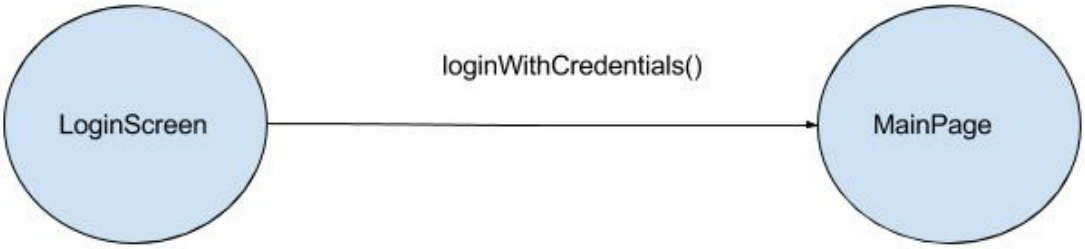

Abbildung 5.2.: Automat des Login Dialogs

Es soll nun zusätzlich das Verhalten des Prüflings bei nicht erreichbarem Server getestet werden, wie es in Abbildung 5.3 abgebildet ist. Es muss sich also der Zustand des Servers ändern. In der $loginWhenBackendUnavailable()$ Methode geschieht dies über den Aufruf der $server.setUnavailable()$ Methode. Listing 5.2 zeigt die entsprechende Methode.

Der zum Testen verwendete Server oder Server-Simulator muss eine Programmierschnittstelle oder Application programming interface (API) anbieten, die solche Zustandsänderungen programmatisch erlaubt und diese Methode implementiert. Der Zustand in den das Programm gerät, wenn ein Login aufgrund eines inaktiven Servers fehlschlägt, repräsentiert ebenfalls das Page-Object $loginFailedScreen$. Die Aktion muss wieder in einer Methode implementiert sein, die dieses Page-Object erzeugt [sel17].

Wenn eine Methode einen bestimmten Zustand des Backend-Servers erwartet, muss sie über die API sicherstellen, dass dieser auch der aktuelle Zustand des Servers ist. Erwähnenswert für die Implementierung dieser zusätzlichen Methoden ist, dass ein Ausgangszustand spezifiziert sein sollte. In diesen Zustand müssen externe Komponenten vor jedem Testdurchlauf versetzt werden, um sicherzustellen, dass Tests nicht aufgrund vorheriger Testausführungen und API Aufrufe fehlschlagen. Zusätzlich sollte jede Methode, die einen bestimmten Zustand von externen Komponenten erwartet, diesen durch einen API Aufruf sicher stellen. Für die Methoden in Abschnitt 5.2.1 bedeutet dies ein Aufruf einer $server.setAvailable()$ Methode.

Über eine Simulatoren API, lässt sich auch der Fall realisieren, in dem Anfragen von einem Server aus an den Prüfling gehen, ohne dass der Prüfling diese ausgelöst hat. Dieses Szenario kann beispielsweise im Zusammenhang mit Datenbankanwendungen relevant werden, wenn

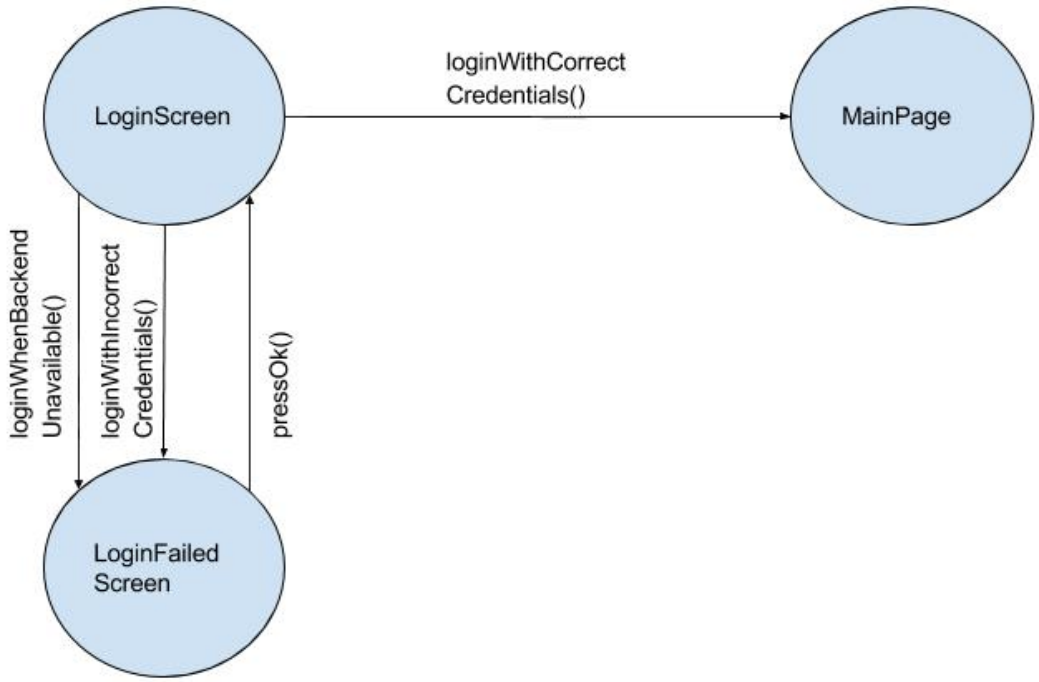

Abbildung 5.3.: Automat eines Login Dialogs mit Backend-Server Manipulation

die Datenbank eine „Push-Connection“[3] zum Prüfling aufbaut. Wenn vom Prüfling eine bestimmte Reaktion erwartet wird, muss nur in der Methode, die das neue Page-Object erzeugt, die entsprechende API Methode aufgerufen werden, um das „push event“ der Datenbank auszulösen.

Es zeichnet sich ab, dass dieses Verfahren einen Mehraufwand beim Schreiben der Page-Objects erfordert. Zudem steigt die Komplexität der Tätigkeit deutlich an. Die Entwickler müssen für jedes Page-Object die möglichen Vorbedingungen in Betracht ziehen und abschätzen können, ob diese sich in unterschiedlichen Zuständen äußern. Durch die steigende Komplexität, steigt vermutlich auch die Fehleranfälligkeit. Daher auch die Menge der falsch-positiven Testergebnisse und damit die Kosten für das Testverfahren.

Validität

Die Validität des erzeugten Zustandsgraphen, hängt von der Korrektheit der geschriebenen Page-Objects ab. Wenn sich die Entwickler an das Page-Object Entwurfsmuster halten, ist der entstehende Graph auch ein Zustandsgraph der Oberfläche des Prüflings, dessen angebotene Funktionalität, mit der eines nach [Cho78] erzeugten Zustandsgraphen übereinstimmt. Um zu zeigen, dass diese Aussage wahr ist, muss die Repräsentanten der Knoten und Kanten

[3] https://en.wikipedia.org/wiki/Push_technology

des Zustandsgraphen betrachtet werden. Die Knoten repräsentieren den aktuellen Zustand. Dieser zeichnet sich durch die ausgehenden Übergänge, also die Operationen und Stimuli, die an ihm ausgeführt werden können, aus [Cho78]. Die aus den Page-Objects abgeleiteten Zustände bieten ihrerseits alle im aktuellen Zustand verfügbaren Funktionen an. Die Stimuli werden in den Methoden des Page-Objects durch die API ausgelöst. Sie sind also in Page-Object erzeugenden Methoden versteckt, die im Zustandsgraph, wie auch die Stimuli in einem nach [Cho78] erstellten Graph, als Kanten repräsentiert werden. Die Kanten des hier erzeugten Zustandsgraph, sind jedoch potentiell eine Kombination aus mehreren Operationen und Stimuli. Eine Methode die einen Text eingibt, den Zustand eines Servers ändert und dann auf einen Button klickt, wird als eine Kante zum resultierenden Page-Object dargestellt, obwohl sie eigentlich zwei Operationen und einen externen Stimulus ausführt. Diese vermeintliche Minimierung, sorgt jedoch eventuell für einen deutlich größeren Graph. Damit die Menge der repräsentierten Operationen und Stimuli gleich bleibt, muss für jede Kombination aus Operationen und Stimuli an einem Knoten eine Methode und ein resultierendes Page-Object geschrieben werden. Auch wenn der Entwickler, der die Page-Objects implementiert, eventuell irrelevante Kombinationen erkennen und eliminieren kann, bedeutet dies doch einen deutlichen Mehraufwand bei der Implementierung und einen größeren Zustandsgraph.

5.3. Auswahl der Testfälle

Nachdem der Zustandsgraph aus den Page-Objects generiert wurde, müssen die Testfälle ausgewählt werden. Diese sollen qualitativ anhand eines Überdeckungskriteriums bewertbar sein und Eingabesequenzen verwenden, die nahe an den Eingaben echter Nutzer liegen. Simplere Kriterien, wie eine „Branch cover“, „Switch cover“ oder eine „Boundary-interior cover“ erfüllen die

5.3.1. Chow's Methode

In Kapitel 3 wird Chows Methode aus [Cho78] vorgestellt. Die Methode verspricht alle Fehlertypen des Zustandsgraphen zu entdecken und findet unter bestimmten Bedingungen garantiert alle diese Fehler [Cho78]. Eine dieser Bedingungen ist jedoch, dass der Tester die Anzahl der Zustände im korrekten Graphen richtig schätzt. Eine menschliche Schätzung ist in einem automatisierten Verfahren natürlich fehl am Platz. Durch die Erzeugung des Zustandsgraphen aus den Page-Objects, existiert jedoch ein Graph der im Idealfall genau die richtige Anzahl Zustände hat. Die Page-Objects werden schließlich aus der Spezifikation abgeleitet und bilden, wenn sie vollständig und korrekt implementiert wurden, den spezifizierten Kontrollfluss ab. Garantiert, ist die Korrektheit in der Praxis jedoch genauso wenig, wie bei Chows herkömmlicher Methode. Statt eine Abhängigkeit von einer Menschlichen Schätzung[4], besteht eine

[4] siehe Kapitel 3 und [Cho78]

Abhängigkeit von der Implementierung der Page-Objects. Diese werden von Menschenhand geschrieben und Menschen machen Fehler. Wenn die Page-Objects für ein Erzeugerprogramm wie in Abschnitt 5.2 geschrieben werden, also auch die Vorbedingungen beachtet werden müssen und durch diese zusätzliche Page-Objects entstehen, steigt die Komplexität der Implementierung stark an. Gerade die für Chows Methode benötigte Anzahl der Zustände ist davon betroffen. Zudem kann keine allgemeine Aussage darüber getroffen werden, wie hoch die Schnittmenge der getesteten Eingabesequenzen mit jenen ist, die ein Nutzer tätigt. Es ist hier keine Frage der Wahrscheinlichkeit oder der Ausführungsdauer, wie es bei stochastischen Methoden wie „Smart Monkeys“ der Falls ist, sondern kommt auf den Prüfling individuell an. Es kommt darauf an, wie häufig bei der Anwendung Schleifen im Kontrollfluss genommen werden. Am Beispiel in Abbildung 3.1 und einem Besipeil in [Cho78][Fig. 3] erkennt man, dass die Methode Schleifen häufig nur ein mal durchläuft. In der Praxis, kann sich ein Programm nach einem Schleifendurchlauf jedoch anders verhalten als zuvor. Trotz alledem ist Chows Methode auch für den automatisch generierten Zustandsgraphen eine legitime Methode, die für den Test eines Prüflings gut geeignet sein kann.

5.3.2. Randomisiert

Die Testfallauswahl beginnt am Startzustand des Zustandsgraphen, wählt eine zufällige Kante am Startzustand aus und führt die Eingabe dieser Kante aus. Ist der kommende Zustand der erwartete Zustand, wird wieder eine Kante gewählt. Kommt ein nicht erwarteter Zustand, bricht der Test als fehlgeschlagen ab und gibt die getätigten Eingaben als ausgeführten Testfall aus. Eine solche vollständig randomisierte Testfallauswahl bedeutet, dass Theoretisch jeder Testfall ausgeführt werden kann, im Voraus jedoch keine Aussage getroffen werden kann, welche Fälle abgedeckt werden. Die Testausführung kann so auch ewig dauern und die Eingabesequenz, die zum aufgetretenen Fehler geführt hat, ist lang und schwer nach zu vollziehen. Ein großer Vorteil sind die geringen Kosten, da außer dem Zustandsgraph keine Vorbereitung nötig ist und mit einer gewissen Wahrscheinlichkeit Nutzerszenarios ausgeführt werden, die sonst nie getestet würden. Ein Nachteil ist, dass eine Vielzahl irrelevanter Tests ausgeführt werden. Auch kann keine Aussage zur Überdeckung getroffen werden, weshalb ein solcher Ansatz höchstens als zusätzlicher Test der Zuverlässigkeit ausgeführt werden kann [Joc13][S.523]. Ein anderer Ansatz ist die Eingabesequenzen zuvor zu bestimmen. So kann im Voraus eine Aussage zur Überdeckung getroffen und nach einem Spezifizierten Kriterium eine neue Sequenz gewählt werden. Es empfiehlt sich hier ein schwaches Überdeckungskriterium, wie Zustandsüberdeckung zu wählen, da sonst die Vorteile des zufälligen Tests (nicht bedachte Eingabesequenzen) zunichte gemacht werden. Um die Eingabesequenzen nachvollziehbarer zu machen, sollte bei der Testfallauswahl bei jeder Kante eine geringe Wahrscheinlichkeit bestehen, dass der Test beendet wird oder die Länge der Testpfade begrenzt sein. Zustandsgraph und Prüfling werden dann in den Anfangszustand versetzt. Je größer der Zustandsgraph, desto geringer muss die Wahrscheinlichkeit sein, da sonst weit verzweigte Bereiche des Graphen nicht ausreichend überdeckt werden.

Überdeckung

Sollen randomisierte Tests als systematische Tests eingesetzt werden, müssen "die Eingaben systematisch ausgewählt"werden [Joc13][S.480]. Das dies bei stochastischen Tests natürlich nicht geschieht, liegt auf der Hand. Jedoch lässt sich anhand der Wahrscheinlichkeit erahnen, ob die sonst systematisch gewählten Testfälle ausgeführt wurden oder nicht. Die Tests werden auf einem Testgraph ausgeführt. Im folgenden wird gezeigt, wie die Wahrscheinlichkeit einer Kantenüberdeckung errechnet werden kann. Dabei wird klar, von welchen Faktoren die Aussagekraft eines randomisierten Tests abhängt. Sei K die Menge aller Kanten in einem Pfad zu einer Kante k_n. Die Wahrscheinlichkeit $P(k_n)$, dass k_n durch einen Testfall ausgewählt wird, ist die Wahrscheinlichkeit der vorhergehenden Kante multipliziert mit der Wahrscheinlichkeit, dass am Knoten N von dem k_n ausgeht, mit der Menge ausgehender Kanten A_N, die Kante k_n ausgewählt wird:

$$P(k_n) = P(k_{n-1}) * \frac{1}{A_N} \tag{5.1}$$

Durch die Abhängigkeit von A_N, erkennt man, dass die Wahrscheinlichkeit einer Kantenüberdeckung stark von der Komplexität des Graphen Abhängt. Je mehr Kanten aus N gehen, desto geringer ist die Wahrscheinlichkeit, dass k_n ausgewählt wird. Für jede Vorgängerkante potenziert sich die Komplexität. Für eine durchschnittliche Komplexität von zwei Kanten je Knoten, ergibt sich eine Wahrscheinlichkeit $P(k_n icht)$, eine Kante in x Versuchen nicht ab zu decken wie in Gleichung (5.2):

$$P(k_{nicht}) = (1 - P(k_n))^x = (1 - (\frac{1}{2})^l)^x \tag{5.2}$$

Wobei l die Länge des Pfades zu k_n ist. Bei einer so geringen Komplexität und einer geringen Pfadtiefe, strebt die Wahrscheinlichkeit den Pfad nicht abgedeckt zu haben sehr schnell gegen null. Bei einer Pfadtiefe von $l = 3$, ist die Wahrscheinlichkeit Beispielsweise bereits nach zehn Durchgängen bei etwa 75%. Sieht man sich jedoch die in Abbildung 5.4 abgebildete Menüleiste einer Open-Source Software[5] für Endnutzer an, wird klar, dass eine solch geringe Komplexität nur selten existiert. Bei einer Tiefe von eins, also bereits im Hauptfenster der Anwendung, hat allein die Menüleiste eine Komplexität von 37. Das bedeutet, um nur alle Optionen der Menüleiste mit einer Wahrscheinlichkeit von 75% abzudecken, benötigt man etwa 50 Durchgänge (siehe Gleichung (5.3)).

$$(1 - \frac{1}{36})^{50} \approx 0,244 \tag{5.3}$$

Um komplexere Szenarios auszuführen, sollte die Pfadlänge einiges länger sein als die der manuell geschriebenen Tests. Führt ein manueller Test eine Funktion aus, sollte der randomisierte mindestens doppelt so lang sein, um mehrere Funktionen ausführen zu können. Die

[5] http://www.jabref.org

Abbildung 5.4.: Menüleiste der Open-Source Software Jabref

Überdeckungswahrscheinlichkeit je Testfall steigt dann im Vergleich zu der obigen Gleichung ein wenig an, da durch rückwärtige Pfade mehrere ausgehende Kanten eines Knoten ausgeführt werden können. Die Ausführungszeit steigt jedoch auch, weshalb die Überdeckung je Zeit, bzw. die Überdeckung je Aufwand etwa gleich bleiben sollte. Im Anhang A.1.1 sind Eckdaten eines echten Software Projekts zu finden. Dort dauert ein automatisierter Testfall im Schnitt etwa 40 Sekunden. Ein Testlauf mit den wie oben genannten 50 Durchgängen und einer doppelten Pfadlänge, also 80 Sekunden, dauert dann etwas mehr als eine Stunde. Die Wahrscheinlichkeit noch nicht einmal die Hauptfunktionalität der Menüleiste ausgeführt zu haben, ist dann immer noch bei etwa 25%.

Folgerung

Es zeigt sich also, dass ein großer Aufwand nötig ist, um eine hohe Überdeckung mit akzeptabler Wahrscheinlichkeit zu erreichen. Auch wenn die Wahrscheinlichkeit sagt, dass die Menge der nicht überdeckten Kanten bei einer hohen Zahl an Durchgängen gegen null strebt, bleibt immer noch eine Restwahrscheinlichkeit, dass Kanten nicht überdeckt bleiben. Diese Restwahrscheinlichkeit gilt es weiter zu minimieren, ohne den Aufwand weiter in die Höhe zu treiben. Ein möglicher Ansatz hierfür sind Lernalgorithmen. Anhand eines Algorithmus, der die zunächst zufälligen Entscheidungen betrachtet, kann eine Gewichtung vorgenommen werden, um die Wahrscheinlichkeit, nicht überdeckte Kanten auszuführen, zu erhöhen. Wie ein solcher Algorithmus ausgeprägt sein könnte, ist in Kapitel 6 ausgeführt.

6. Lernfähigkeit

In diesem Kapitel geht es darum, die kostengünstigen vollständig randomisierten Tests präziser einzusetzen. Schleifendurchläufe und Verzweigungen, die ein manueller Test vermutlich nicht ausführen würde, werden trotz höherer Präzision ausgeführt und die Testfälle nach einem Überdeckungskriterium bewertet. Der Lernprozess und der entsprechende Algorithmus werden vorgestellt.

6.1. Lernvorgang

Der Lernvorgang soll die Redundanz der Testfallauswahl verringern. Er soll also dafür sorgen, dass die Testpfade sich besser über den Zustandsgraph des Prüflings verteilen und verhindern, dass der Algorithmus zu häufig in den selben Schleifen hängen bleibt. Bei vollständig randomisierter Auswahl kann dies geschehen. Die erzeugten Testpfade sind zudem durch vom Tester bestimmte Werte begrenzt. Für jeden Testdurchgang, ist die Anzahl der generierten Testfälle, sowie die Pfadtiefe vorgegeben, um die Lesbarkeit der Testergebnisse im Vergleich zu herkömmlichen „Monkey Tests“ zu erhöhen (siehe Abschnitt 5.3.2). Zu komplexe Testpfade, ergeben schlecht nachvollziehbare Testresultate, sollte ein Fehler gefunden werden [CMM+12]. Der hier vorgestellte Algorithmus, soll weiterhin eine stochastische Natur haben, um die in Abschnitt 5.3.2 und Kapitel 4 vorgestellten Vorteile dieser Algorithmen zu bewahren. Um seine Präzision zu erhöhen, soll sich die Wahrscheinlichkeit, mit der ein bestimmter Pfad ausgewählt wird jedoch anpassen. Der Algorithmus geht vom Startknoten aus durch den Graph. An jedem Knoten wird zunächst mit gleich verteilter Wahrscheinlichkeit eine Kante ausgewählt. Bei einem Knoten N mit drei ausgehenden Kanten ist die Wahrscheinlichkeit für jede Kante $1/3$. Diese Wahrscheinlichkeit, wird nun durch eine gewichtete Wahrscheinlichkeit abgelöst, welche für eine Kante E mit der Gewichtung W_E durch die Gleichung (6.1) bestimmt wird. Wobei W_{gesamt} die Summe aller Kantengewichte der ausgehenden Kanten an N ist.

$$P(E) = \frac{W_E}{W_{gesamt}} \tag{6.1}$$

Damit der Algorithmus seltener redundante Eingabesequenzen ausgibt kann, muss der er speichern, wie häufig er bereits welche Kante besucht hat. Dafür merken sich die Kanten im Graph die Anzahl der Besuche V_E. So sinkt die Wahrscheinlichkeit, dass der Algorithmus in die selbe Schleife läuft, die er bereits besucht hat. Da jedoch eine ausreichende Restwahrscheinlichkeit für solche Schleifendurchläufe bleiben soll, da auch ein Nutzer eine derartige

Sequenz ausführen kann, wird die Anzahl der Besuche erst am Ende eines jeden Durchgangs inkrementiert. Zudem sollen, auch wenn eine Schleife in einer Testsequenz sehr häufig durchlaufen wurde, in folgenden Tests Kombinationen mit dieser Schleife getestet werden. Um die Wahrscheinlichkeit, dass die Schleife wieder aufgerufen wird, groß genug zu halten, wird nicht die Anzahl der Aufrufe einer Kante gespeichert, sondern die Anzahl der Testsequenzen, in denen die Kante aufgerufen wurde.

Die Besuchszahl V_E soll abhängig von der Menge der generierten Testfälle bisher sein. Bei einer hohen Zahl an Durchgängen, werden zu Beginn häufig besuchte Kanten wieder relevant, wenn die restlichen Kanten abgedeckt wurden und Kombinationen der Verzweigungen und Schleifen getestet werden können. Die Wahrscheinlichkeit für bereits besuchte Kanten muss also wieder steigen, wenn viele Testsequenzen ausgeführt wurden, ohne diese zu besuchen. Nur so erhält sich der Vorteil der stochastischen Tests, dass komplexe Szenarien, die relevant aber nicht manuell getestet sind, ausgeführt werden [Nym00].

Wie der randomisierte Algorithmus, soll der lernfähige zunächst vollständig zufällig, mit gleich verteilter Wahrscheinlichkeit Kanten auswählen. Daher ist der Startwert des Gewichts einer Kante V_E die Anzahl der ausgehenden Kanten. Im ersten Testdurchgang ergibt sich dann für den Knoten N mit drei Ausgehenden Kanten wie oben eine Wahrscheinlichkeit von $1/3$ je Kante, da die Gleichung (6.2) gilt.

$$P(E) = \frac{W_E}{W_{gesamt}} = \frac{3}{9} = \frac{1}{3} \tag{6.2}$$

Bei komplexeren Graphen, ergibt sich durch diese Gewichtung jedoch das Problem, dass Teilgraphen, die an Knoten beginnen, die Teil einer Schleife sind oder viele Pfade zu ihren eingehenden Kanten haben, mit hoher Wahrscheinlichkeit nicht oder nur sehr selten besucht werden. Grund dafür ist, dass die vorherigen Kanten mit hoher Wahrscheinlichkeit häufig besucht werden und deren Gewichtung dann verringert wird. Jedes mal, wenn eine Testsequenz in einen Teilgraph führt, sinkt die Wahrscheinlichkeit, dass dieser noch einmal besucht wird. Dies geschieht unabhängig davon, wie gut der Teilgraph abgedeckt ist. Um eine minimale Abdeckung wahrscheinlicher zu machen, soll die Dekrementierung des Kantengewichts nur dann statt finden, wenn keine neue Kante auf dem Pfad lag. Führt ein Pfad das erste mal in den Teilgraph, wird der Weg dort hin nicht unwahrscheinlicher. Führt der Pfad erneut hinein, steigt jedes mal die Wahrscheinlichkeit, eine der nicht besuchten Kanten im Teilgraph zu besuchen, da andernfalls die bereits besuchten Kanten an Gewicht verlieren. Solange regelmäßig neue Kanten entdeckt werden, steigt die Wahrscheinlichkeit diesen Teilgraph zu besuchen, im Vergleich zu bereits vollständig abgedeckten Teilgraphen. Wird also eine neue Kante mit $V_E = 0$ besucht, ist das Gewicht aller Kanten die in diesem Testlauf besucht wurden $W_k = W_k$. Wenn nicht, wird das Gewicht verringert um den Faktor der Besuche im Verhältnis zur Zahl der Durchgänge bisher. Die Gewichtung einer Kante W_E, wenn keine neue Kante besucht wurde, errechnet sich also wie in Gleichung (6.3), wobei D die Anzahl der Testdurchläufe bisher ist. Das Gewicht muss nach jedem Testdurchlauf neu berechnet werden.

$$W_E = W_E * \frac{V_E}{D} \tag{6.3}$$

In Abschnitt 6.2 wird der lernfähige Algorithmus mit dem rein zufälligen Vorgehen verglichen und auf die erhofften Vorteilen untersucht.

6.2. Vergleich der Algorithmen

Im Folgenden ist meine Untersuchung des Algorithmus erläutert. Zunächst der Versuchsaufbau, dann die Variationen der Durchführung und im Anschluss die Folgerungen, die aus den Ergebnissen gewonnen werden können.

6.2.1. Versuchsaufbau

Der Versuch wurde anhand eines Graphen und zweier Algorithmen in Java ausgeführt. Die Algorithmen sind ein zufälliger und ein lernfähiger Algorithmus. Der Graph ist ein gerichteter Graph mit Schleifen, wie er als Kontrollfluss eines Prüflings vorkommen könnte. Die Implementierung des Graphen besteht aus zwei unterschiedlichen Klassen. Die „Edge“ Klasse und die „Node“ Klasse. Um die Auswertung der Ergebnisse einfacher zu gestalten, wurden extra Objekte für die Kanten verwendet, statt sie wie häufig implizit durch die Verknüpfung der Knoten darzustellen. Die Kanten wissen von welchem Knoten sie ausgehen und in welchem Knoten sie enden. Sie speichern die Anzahl der Besuche und ihre Gewichtung in einer „visits“ und einer „heat“ Variable. Beide Klassen sind in Anhang A.2 zu finden. Nach jedem Testdurchlauf mit Gewichtung wird die „visits“ Variable aller besuchter Knoten inkrementiert. Danach wird die Gewichtung jedes Knotens neu berechnet. Wenn keine neue Kante besucht wurde und die boolesche Variable „newEdgeVisited“ den Wert $false$ hat, wird die Gewichtung wie in Gleichung (6.3) errechnet. Der Versuchsaufbau führt folglich den lernfähigen Algorithmus wie in Abschnitt 6.1 beschrieben aus.

Es wurden drei Komplexitätsstufen des Graphen verwendet, um die Auswirkungen der Komplexität auf die Ergebnisse der Algorithmen zu untersuchen. Abbildung 6.1 zeigt eine Visualisierung der Datenstruktur. In Komplexitätsstufe zwei, kamen die grünen Knoten und Kanten hinzu. In der dritten Komplexitätsstufe die gestrichelten Kanten. Der Wurzelknoten ist der mit „r“ beschriftete Knoten. In jeder Komplexitätsstufe wurden vier Versuche ausgeführt. Beide Algorithmen wurden mit jeweils 20 und 50 Testdurchläufen ausgeführt. Das erste mal mit einer Pfadtiefe von 20 und das zweite mal mit einer Pfadtiefe von 25. Die Pfadtiefe bestimmt, nach wie vielen besuchten Kanten ein Testdurchlauf zu Ende ist. 20 und 25 wurden gewählt, da die Pfade dann länger als der kürzeste direkte Pfad zum am weitesten von der Wurzel entfernten Knoten [1] sind und trotzdem eine möglichst geringe Länge haben. Das Ziel sind schließlich für einen

[1] Von der Wurzel bis zu Knoten 17 ist die Pfadlänge zehn. Bei zehn Schritten ist die Wahrscheinlichkeit den direkten Pfad zu gehen $\frac{1}{3} * \frac{1}{2} * \frac{1}{2} * \frac{1}{4} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{3} * \frac{1}{3} = \frac{1}{1728}$. Das Ereignis, dass ein Pfad mit $Länge \leq 20$ zu 17 führt ausgewählt wird, hat eine deutlich höhere Wahrscheinlichkeit.

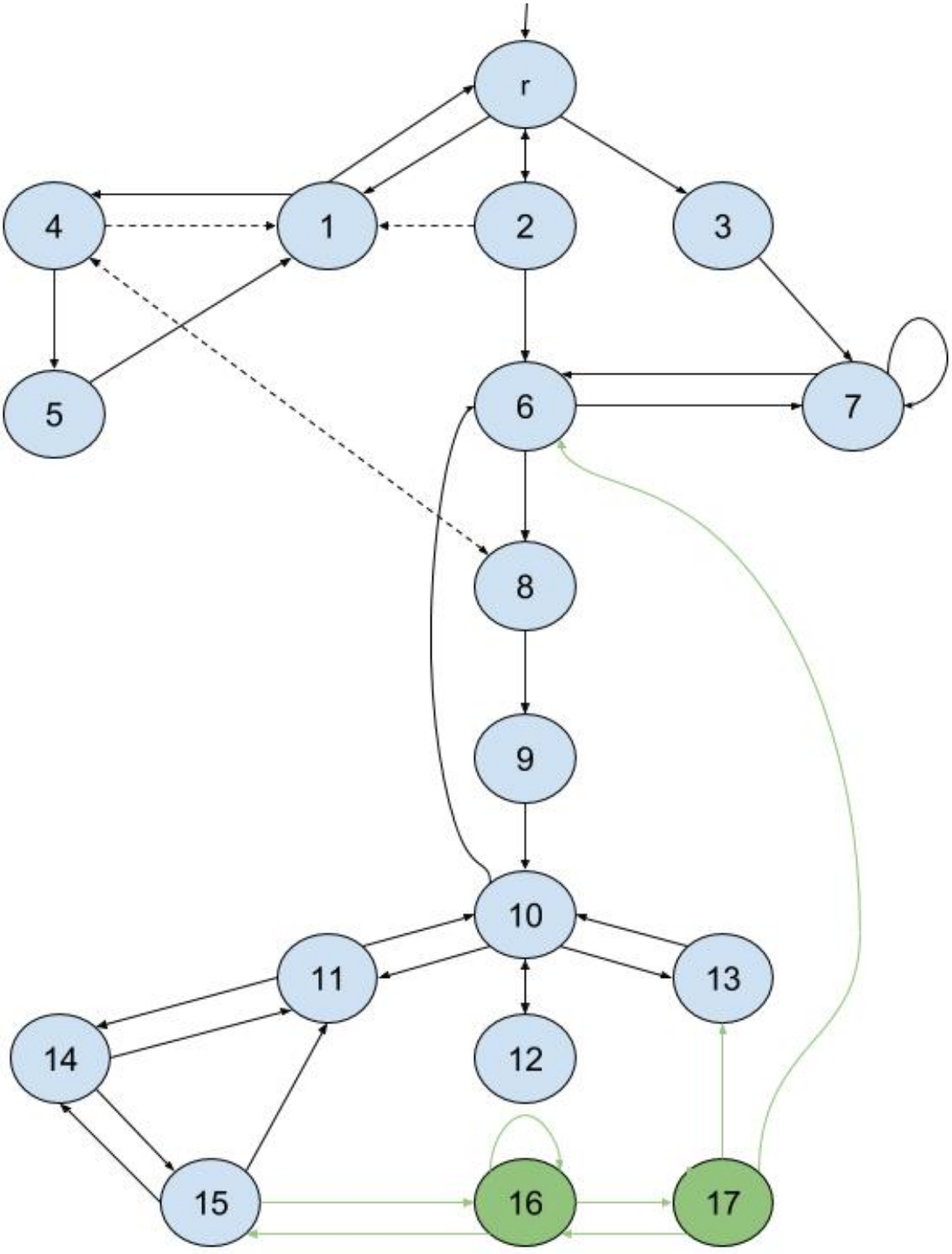

Abbildung 6.1.: Gerichteter Graph mit Schleifen

Menschen interpretierbare Ergebnisse. Auch die Anzahl Testdurchläufe ist so gewählt, dass bei einer Ausführungsdauer von 40 Sekunden je Testsequenz, der Testdurchlauf maximal etwa eine halbe Stunde dauert. Der lernfähige Algorithmus führt die Gewichtung 20 bzw. 50 mal, also für jeden Testdurchlauf aus. Nach 20 bzw. 50 Testdurchläufen wird die Gewichtung des Graphen wieder zurück gesetzt, so dass der Versuch unabhängig wiederholt werden kann.

Am Ende jedes Testfalls wurde ausgegeben, wie viele Kanten noch nicht besucht wurden. Die Verteilung der Testsequenzen auf den Graphen, wird also anhand einer Kantenüberdeckung gemessen. Trotz einer gewünschten Varianz der Testfälle, soll in ökonomisch sinnvoller Zeit, eine sonst systematisch angestrebte Überdeckung erreicht werden. Die Kantenüberdeckung ist ein gutes Indiz dafür, ob die Pfade den Graphen gut verteilt überdeckt oder ob sie sich auf einzelne Teilgraphen und Schleifen beschränkt haben. Die Ergebnisse der Tests liegen gezielt in einem Grenzbereich. Sprich aufgrund der Komplexität des Graphen, der gewählten Pfadtiefe und der Anzahl Testdurchläufe, ist die Wahrscheinlichkeit hoch, dass Kanten nicht besucht werden. Die Ergebnisse ungleich null lassen sich dann quantitativ leichter bewerten, als jeden einzelnen Pfad zu vergleichen. Der gewählte Versuchsaufbau führte im ersten Test, zu Ergebnissen zwischen null und maximal zwölf nicht überdeckten Kanten.

6.2.2. Ergebnisse

Versuch 1

Jedes Experiment wurde 30 mal wiederholt und Anschließend Mittelwert und Median ausgerechnet. Die Ergebnisse der Versuchsreihen sind in Abschnitt 6.2.2, Abschnitt 6.2.2, Tabelle 6.3 aufgeführt. Die Tabellen sind zweigeteilt. Die oberen drei Zeilen, sind jeweils die Ergebnisse des lernfähigen Algorithmus. Die Zeilen vier bis sechs sind die des zufälligen. Die Spalten beinhalten zuerst die Experimente mit 20 Testdurchläufen jeweils. Zunächst mit der Pfadtiefe 20 dann mit 25 Kanten tiefen Pfaden. Die letzten beiden Spalten sind die selben Experimente mit jeweils 50 Testdurchläufen. Angegeben sind jeweils Mittelwert und Median um eine etwaige Rolle von Ausreißern erkenntlich zu machen.

Aus den Tabellen ist klar ersichtlich, dass der lernfähige Algorithmus in den ausgeführten Versuchen deutlich besser bei der Kantenabdeckung abschneidet. Beide Algorithmen decken in Komplexitätsstufe eins recht zuverlässig den Graphen ab. Bei 50 Testdurchläufen, bleibt beim lernfähigen Algorithmus im Schnitt keine Kante übrig. Selbst bei nur 20 Durchläufen, ist das Ergebnis mit 0,133 und 0,033 deutlich besser als das des zufälligen Algorithmus. Der lernfähige Algorithmus ist hier sogar etwa um den Faktor neun besser als der Zufällige. Jedoch fällt auf, dass der Faktor, um den der lernfähige Algorithmus besser ist sinkt, je mehr Durchgänge und je höher die Pfadtiefe. Dies hängt jedoch damit zusammen, dass für beide Algorithmen der Wert gegen null strebt. Dies ist zu erwarten, denn je mehr Ausführungen, desto höher ist die Wahrscheinlichkeit, dass alle Kanten besucht werden (siehe Theorem 2). Es lässt sich jedoch bereits Ableiten, dass zumindest bei niedriger Anzahl Durchläufe eine bessere Kantenüberdeckung durch den lernfähigen Algorithmus erreicht wurde.

Lernend	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	0,133	0,033	0	0
Median	0	0	0	0
Zufällig	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	1,100	0,300	0,200	0,100
Median	1	0	0	0

Tabelle 6.1.: Versuch auf der Komplexitätsstufe eins

Lernend	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	2,967	0,933	0,333	0,367
Median	3	0,5	0	0
Zufällig	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	5	4,5	3,033	1,8
Median	4,5	4	2,5	1

Tabelle 6.2.: Versuch auf der Komplexitätsstufe zwei

Versuch 2

In Versuch zwei wurden die Experimente aus Versuch eins, mit Pfadtiefe 25 wiederholt. In jedem Experiment wurde die Menge der Durchläufe um 30 erhöht. Die Experimente wurden jeweils 100 mal wiederholt und der Mittelwert errechnet. Graphen zeigen für Komplexität zwei und drei, bis zu welchem Punkt der lernfähige Algorithmus Vorteile bietet. In Abbildung 6.2 ist erkenntlich, dass für Komplexitätsstufe drei, bereits nach etwa 110 Durchläufen die Wahrscheinlichkeit eine Kantenüberdeckung erreicht zu haben gleich ist, da sich die Durchschnittswerte dort schneiden. In Komplexitätsstufe zwei, trifft das erst nach etwa 180 Durchläufen zu. Die Unterschiede können hier durchaus der stochastischen Natur der Algorithmen zugerechnet werden und sind vernachlässigbar. Die gesamt Wahrscheinlichkeit ist bei so geringen Unterschieden, wie sie ab 180 Durchläufen zu vermerken sind, gleichwertig.

Lernend	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	5,133	3,067	1,133	0,700
Median	5	2	0	0
Zufällig	20 / 20	20 / 25	50 / 20	50 / 25
Mittelwert	6,633	5,333	3,7	1,933
Median	7	5	3	2

Tabelle 6.3.: Versuch auf der Komplexitätsstufe drei

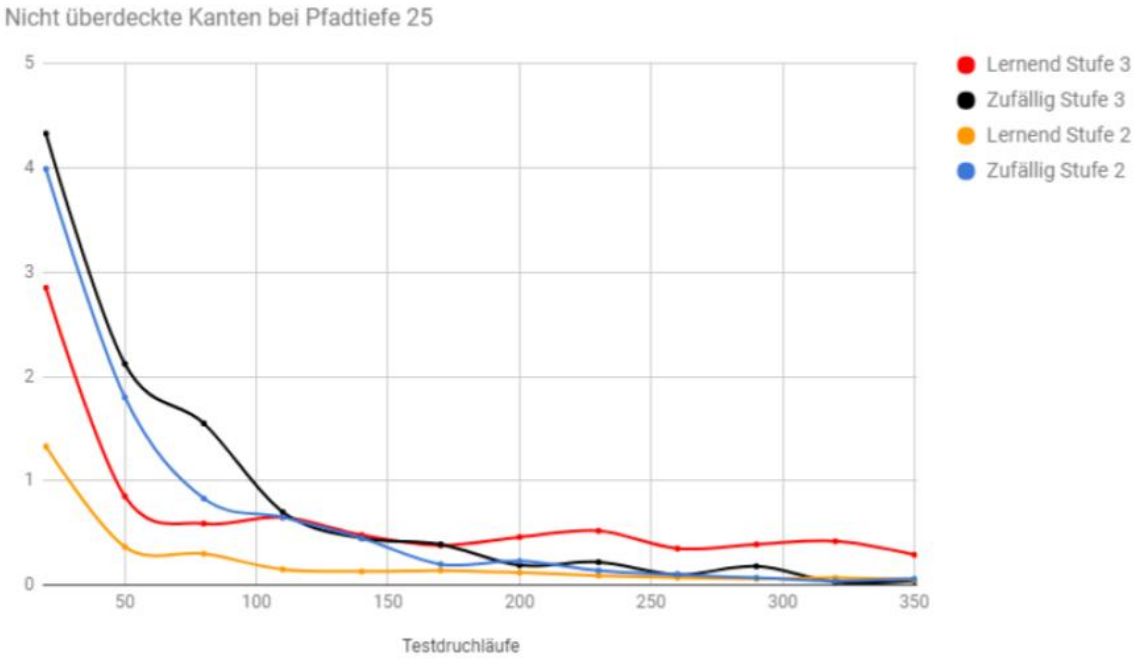

Abbildung 6.2.: Durchschnittlich übrige Kanten bei Pfadtiefe 25

Versuch 3

In einer weiteren Versuchsreihe, sollte die Verteilung der Kanten in einzelnen Durchläufen ermittelt werden. Hierfür wurde die durchschnittliche Anzahl unterschiedlicher besuchter Kanten je Durchlauf gemessen. Das Experiment wurde bei einer Pfadtiefe von 25 auf Komplexitätsstufe zwei ausgeführt. Die Ergebnisse sind in Abschnitt 6.2.2 aufgeführt. Hier zeigt sich, dass nahezu kein Unterschied existiert. Beide Algorithmen besuchen je Testlauf also etwa gleich viele unterschiedliche Kanten. Der zufällige Algorithmus, schneidet sogar etwas besser ab als der lernfähige.

Versuch 4

Der letzte Versuch, zielte darauf ab, die Überdeckung weit vom Startknoten entfernter Pfade zu betrachten. Hierfür wurde die Kante „17-13“ betrachtet und die durchschnittliche Anzahl der Besuche der Kante je Testdurchlauf gemessen. Die in Abbildung 6.3 abgebildeten Werte sind der Durchschnitt der Ergebnisse aus 100 Ausführungen des Versuchs. Die betrachtete Kante ist die am weitesten vom Wurzelknoten entfernte Kante. Auf Komplexitätsstufe zwei ist die Wahrscheinlichkeit des direkten Pfades der Länge 11, beim zufälligen Algorithmus $\frac{1}{1944}$, auf Stufe drei ist sie $\frac{1}{3888}$. Entsprechend schneidet der zufällige Algorithmus ab. Die Aufrufe

Durchläufe	Lernender Algorithmus	Zufälliger Algorithmus
20	13.81	13.88
40	13.14	13.22
60	12.51	12.52
80	11.90	11.89
100	11.45	11.64
120	11.10	11.16
140	10.68	11.3
160	10.84	11.15
180	10.36	10.72
200	10.50	10.98
220	10.20	10.19

Tabelle 6.4.: Besuchte unterschiedliche Kanten je Durchlauf

der Kante steigen mit zunehmenden Durchläufen für den lernfähigen Algorithmus annähernd exponentiell an. Für den zufälligen Algorithmus, ist ein schwacher, linearer Zuwachs zu verzeichnen.

6.2.3. Folgerung

Die Untersuchungen des lernfähigen Algorithmus, zeigen welche Faktoren für die Präzision der Testfälle ausschlaggebend sind.

Komplexität des Graphen

Menge der Testdurchläufe

Tiefe der Pfade

In Versuch ein und zwei zeigt sich dass beide Algorithmen schneller eine Kantenüberdeckung erreichen, wenn die Komplexität minimal, Pfadtiefe und Durchläufe maximal sind. In der Praxis liegt die Komplexität des Graphen und die Pfadtiefe, gar nicht oder nur bedingt in der Hand der Tester. Die Komplexität liegt allein an der Beschaffenheit des Prüflings, die Testfalltiefe sollte begrenzt sein, damit die Testergebnisse nachvollziehbar und Fehlersituationen reproduzierbar bleiben.

Im Vergleich schneidet der lernfähige Algorithmus, besonders bei geringem Aufwand, besser ab als der Zufällige. Der Aufwand ergibt sich aus Ausführungsdauer und Komplexität. In Abbildung 6.2 ist auch zu erkennen, dass die Anfangswerte der Kurven des lernenden Algorithmus weiter auseinander liegen als die des zufälligen. Die Komplexität hat also auf den Lernenden einen stärkeren negativen Effekt als auf den zufälligen. Es lässt sich vermuten, dass

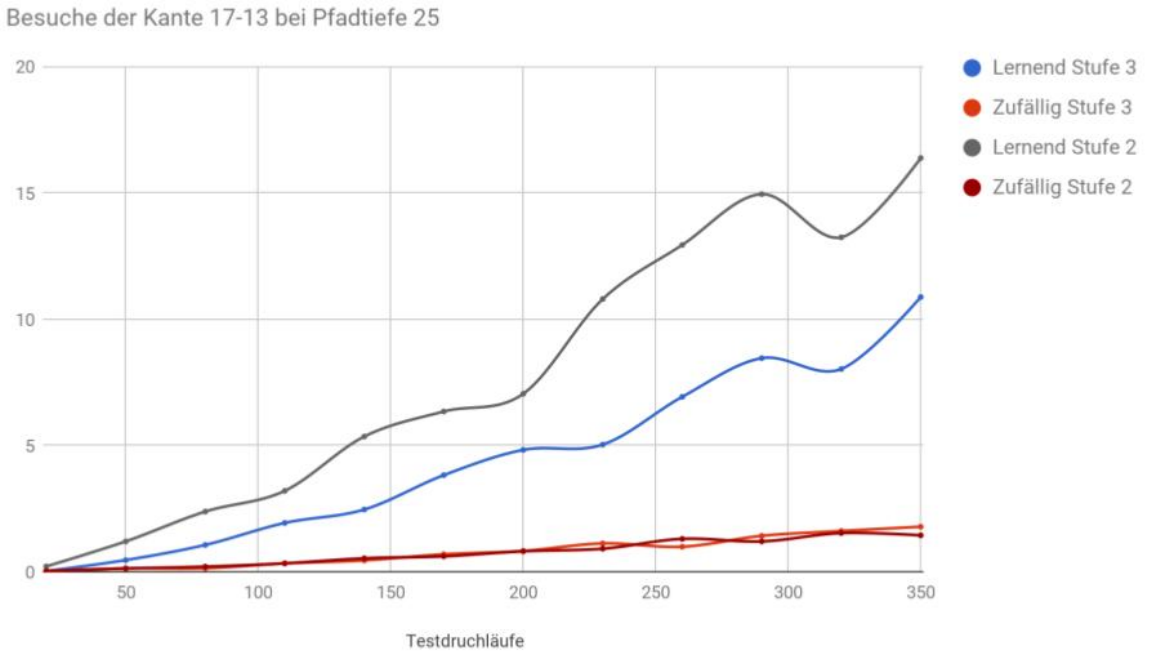

Abbildung 6.3.: Besuche einer weit entfernten Kante

der anfängliche Vorteil des lernenden Algorithmus ab einer gewissen Komplexität nicht mehr existiert.

Im dritten Versuch, wurde festgestellt, dass die Anzahl unterschiedlicher Kanten in einem erzeugten Testpfad, bei beiden Algorithmen etwa gleich ist. Dies deutet darauf hin, dass beide etwa gleich häufig Schleifen ausführen. Würde ein Algorithmus häufiger Schleifen durchlaufen als der andere, würde er weniger unterschiedliche Kanten je Pfad besuchen. Die Algorithmen verhalten sich bezüglich Schleifen also gleich.

Versuch vier widmete sich der wohl wichtigsten Eigenschaft der Algorithmen. Ideal ist ein Algorithmus der bei einem bestimmten Aufwand möglichst alle Teile des Programms gleich testet, also die Testfälle über den gesamten Graphen verteilt. Dazu wurde die am weitesten von der Wurzel entfernteste Kante betrachtet. In **??** zeigt sich eindeutig, dass der lernende Algorithmus deutlich häufiger weit entfernte Kanten besucht. Das zeugt davon, dass der Algorithmus die gewählten Pfade weiter streut als der zufällige Algorithmus. Der zufällige Algorithmus hat also eine höhere Wahrscheinlichkeit, weit verzweigte Teilbäume nur unzureichend zu testen, während der lernende Algorithmus bei erneuter Ausführung deutlich wahrscheinlicher weit entfernte Kanten besucht. Die Zuname an Besuchen ist sogar annähernd exponentiell, während die des zufälligen Algorithmus nur leicht linear steigt. Jedoch zeigt sich auch hier, dass die Komplexität des Graphen eine große Rolle für die Verteilung des lernfähigen Algorithmus ist. Wie der Algorithmus bei einem umfangreichen Kontrollflussgraph eines komplexen Prüflings

abschneiden würde, gilt es noch zu prüfen. Durch die exponentielle Zunahme, sollte er jedoch selbst dann besser abschneiden als der zufällige Algorithmus.

7. Bewertung

Das in dieser Ausarbeitung vorgestellte Verfahren lässt sich in zwei Abschnitte teilen. Die Erzeugung eines Zustandsgraphen anhand modifizierter Page-Objects und die automatisierte Testfallgenerierung mithilfe eines lernfähigen Algorithmus. Das Ergebnis sollen, mit manuell geschriebenen Systemtests vergleichbare, automatisch generierte Systemtests sein. Die Bewertungen dieses Kapitels werden für beide Abschnitte getrennt vorgenommen.

7.1. Laufzeit und Arbeitsaufwand

7.1.1. Erzeugung des Zustandsgraphen

Die für die Erzeugung des Zustandsgraphen notwendige Laufzeit, ist abhängig von der Menge der Page-Objects. Für jedes Page-Object, müssen die ausgehenden Kanten gefunden werden. Eine Untersuchung der Page-Objects eines echten Softwareprojekts (siehe Anhang A.1.1) ergab, dass für die betrachteten 30 Tests, 45 neue Page-Objects nötig wären, um alle im Test erzeugten Zustände zu implementieren. Für dieses Projekt würde das, hoch gerechnet auf 370 Systemtestfälle, 555 neue Page-Objects bedeuten. Für die Laufzeit des Erzeugerprogramms ist das mit moderner Hardware unbedenklich. Wenn jedes Page-Object mit jedem Page-Object verbunden ist bedeutet das einen Worst-case in quadratischer Laufzeit, was bei einigen hundert Elementen jedoch kein Problem darstellt. Selbst wenn die Berechnung wenige Minuten dauern sollte, so wird sie doch nur ein mal vor dem Systemtest ausgeführt und nur wenn Änderungen am Kontrollfluss vorgenommen wurden [1]. Bedenklicher sind die neuen Page-Objects für den Arbeitsaufwand. Es fällt ein mehr als doppelt so hoher Aufwand für das schreiben der Page-Objects an. Dieser Mehraufwand sollte mit den Vorteilen der zufälligen Testfallauswahl im Verhältnis stehen.

7.1.2. Auswahl der Testfälle

Die Auswahl mit Hilfe eines lernfähigen Algorithmus wie er in Kapitel 6 beschrieben wird, erfordert keinen bis minimalen Aufwand für die Durchführung, da diese vollautomatisch

[1] Z.B durch neue Funktionalität. Bei Regressionstests, fällt der Aufwand nicht mehr an.

abläuft. Die Auswertung der Testergebnisse wiederum, kann einiges an Aufwand hervorrufen. „Smart Monkeys“ erzeugen eine sehr lange Eingabesequenz, die es nachzuvollziehen gilt, sollte ein Fehler gefunden werden [Nym00]. Die vorgestellte Methode kürzt diese Sequenzen auf eine für Menschen leicht nachvollziehbare Länge, die abhängig von der Pfadtiefe des Prüflings festgelegt werden muss. Zudem streut der Algorithmus die Testpfade weiter über den verwendeten Zustandsgraphen. So lassen sich relevante Testfälle und Ergebnisse erhoffen, da Sequenzen in die Tiefe in der Regel eher echten Anwendungsfällen entsprechen.

Die Laufzeit des Algorithmus ist vom Tester selbst fest zu legen. Der Algorithmus kann Tage aber auch nur Minuten ausgeführt werden. Zu bedenken ist jedoch der entstehende Aufwand durch (hoffentlich relevante) Testergebnisse. Auch hängt die Überdeckung des Graphen von der Ausführungsdauer ab.

7.2. Aussagekraft und Überdeckung

Soll das Verfahren mit anderen Testverfahren verglichen werden, kann die Aussagekraft der Testergebnisse und die mögliche Überdeckung des Prüflings untersucht werden.

7.2.1. Erzeugung des Zustandsgraphen

Die Qualität der Page-Objects, hat einen großen Einfluss auf die Aussagekraft der auf ihnen ausgeführten Tests. Sollen die Page-Objects nicht nur die Nutzeroberfläche, sondern auch den Zustand des Prüflings repräsentieren, nimmt die Komplexität der Implementierung zu. Die Entwickler müssen für jedes Page-Object die Vor- und Nachbedingungen kennen, um Übergänge in jeden möglichen Zustand erzeugen zu können. Ein übersehener Zustand führt zu einem nicht getesteten, jedoch spezifizierten Anwendungsfall. Gerade für sicherheitskritische Anwendungen ist dies ein hohes Risiko. Wenn die Page-Objects anhand der Spezifikation, korrekt und vollständig implementiert werden können, stellen sie einen vollständigen Zustandsgraphen für die Software dar, auf dem ein konventioneller, Zustand basierter Test (siehe Kapitel 3) ausgeführt werden kann. Das Verfahren hat dann die selbe Aussagekraft wie dieser.

7.2.2. Auswahl der Testfälle

Die Aussagekraft der Tests, hängt stark von der Überdeckung des Zustandsgraphen ab. Selbst wenn ein korrekter Graph existiert, kann auf den Grad der Korrektheit der Software nur vertraut werden, wenn der Graph auch ausreichend überdeckt ist. Zufällige Tests versprechen bei genügend langer Ausführungsdauer eine hohe Wahrscheinlichkeit alle relevanten Szenarios auf dem Graphen auszuführen. Diese Aussage lässt sich vom „infinite monkey theorem“ ableiten [Wik17c]. Erste Versuche mit dem lernfähigen Algorithmus in Abschnitt 6.2 haben gezeigt, dass durch dynamische Gewichtung der Kanten eine Streuung der Überdeckung erreicht

werden kann. Die Aussagekraft der Tests, ist der Wahrscheinlichkeit entsprechend hoch. Jedoch kann es aufgrund der stochastischen Natur der Tests immer noch passieren, dass wichtige Szenarios nicht ausgeführt werden. Zudem wurde das Verfahren auf einem Graphen mit geringer Komplexität ausgeführt. Um eine aussagekräftige Überdeckung von komplexeren Graphen zu erreichen, muss die Laufzeit entsprechend hoch sein.

7.3. Skalierbarkeit

Die Aussagekraft des Verfahrens als vollwertiger Systemtest ist fragwürdig. Jedoch lässt es sich problemlos zu Oberflächentests oder Zuverlässigkeitstests herunter skalieren. Wenn die Page-Objects nach dem herkömmlichen Entwurfsmuster geschrieben sind, repräsentieren sie die Nutzeroberfläche des Prüflings [sel17]. Aus diesen Page-Objects lässt sich genauso ein Graph erzeugen, wie aus Page-Ojects die den Zustand des Programms beinhalten. Die Aussagekraft der Tests ist dann natürlich nicht mehr mit der eines Systemtests vergleichbar. Die Oberfläche kann trotzdem kostengünstig anhand zufälliger Tests auf ihre Funktionalität getestet werden. Lediglich Tests die externe Stimuli benötigen, können nicht automatisch ausgeführt werden. Sollte die Komplexität des Prüflings zu hoch sein, als das der lernfähige Algorithmus eine gute Überdeckung erreichen kann, dienen seine Ergebnisse dennoch als zusätzliche Zuverlässigkeitstests.

8. Zusammenfassung und Ausblick

Das vorgestellte Verfahren ist ein zweigeteiltes skalierbares Verfahren zur Automatisierung von Systemtests. Es verwendet Page-Objects die nach dem Page-Object Entwurfsmuster implementiert wurden [sel17]. Um einen Zustand basierten Test auf dem Prüfling ausführen zu können, wird aus den Page-Objects ein Zustandsgraph generiert. Dazu muss das Entwurfsmuster erweitert werden und ein Page-Object für jeden Zustand implementiert sein. Das Verfahren macht sich dann zu nutze, dass eine stochastische Testfallauswahl sehr komplexe und doch relevante Eingabesequenzen erzeugt, die bei manuellen Tests nicht ausgeführt werden [Nym00], um die Testüberdeckung des Prüflings zu erhöhen. Die zufällig generierten Tests orientieren sich an den „Monkey Tests“ [Nym00]. Um die Menge der ausgeführten Tests möglichst nahe an die Menge der von Nutzern ausgeführten Szenarios zu bringen, wird ein lernfähiger Algorithmus eingesetzt, der die Eingabesequenzen besser auf dem gesamten Zustandsgraphen zu verteilt. Die Tests selbst müssen nicht geschrieben werden und erzeugen keinen Aufwand. Durch den lernfähigen Algorithmus werden vermeintlich relevante und auch lesbare Testergebnisse erzeugt. Das Verfahren erfordert jedoch einen hohen Aufwand beim Schreiben der Page-Objects. Auch die Überdeckung des Graphen ist nur bei häufiger Ausführung der Tests wahrscheinlich. Wie das Verfahren an einer echten Software abschneidet, ist noch nicht erprobt. Es lässt sich aber erahnen, dass aufgrund des hohen Aufwands für die Implementierung der Page-Objects, eine vollständige Umsetzung des Zustandsgraphen nicht ökonomisch sinnvoll ist. Eine mögliche Anwendungsweise könnte jedoch eine teilweise Umsetzung des Zustandsgraphen sein. Vollautomatisch getestet wird dann nur ein Teilgraph, der sich leicht in Page-Objects abbilden lässt. Die Methode scheint eher zusätzlich zu manuell geschriebenen Systemtests einsetzbar zu sein, als diese ersetzen zu können.

Ausblick

Um den Einsatz des Verfahrens weiter zu evaluieren, sollte es an einem echten Prüfling angewendet werden. Geeignet ist eine Software, für die bereits Page-Objects nach dem korrekten Entwurfsmuster und manuell geschriebene Systemtests existieren. Für diese sollten dann, den Zustand repräsentierende, Page-Objects komplett neu geschrieben werden. Gleichzeitig müssen die bereits existierenden an die Anforderungen des Verfahrens angepasst werden. So kann der Aufwand für die neuen Page-Objects und für eine Anpassung ermittelt werden. Anschließend können die zufällig generierten Tests mit den bereits geschriebene im Einsatz

verglichen werden. Dabei sollte die Schwere der gefundenen Fehler, die Rate in der Fehler gefunden werden und die ausgeführten Testpfade betrachtet werden.

Ein Ansatz zur weiteren Verbesserung der Testfallauswahl ist die Verwendung eines neuronalen Netzwerks. Ein solches Netz könnte die Gewichtung der Kanten des Zustandsgraphen anhand von gemessenen Überdeckungskriterien durchführen. So könnte der Algorithmus mit der Zeit lernen, wie er den Prüfling am besten überdeckt. Auch die Relevanz und Menge der gefundenen Fehler könnte das Netzwerk auswerten. So wäre es möglich, Teilgraphen in denen häufig Fehler auftreten intensiver zu testen als solche, die selten Fehler beinhalten. Ein solches Netzwerk könnte unabhängig davon angewandt werden, ob der Zustandsgraph automatisch generiert oder aus der Spezifikation abgeleitet wurde.

A. Anhang

A.1. Untersuchtes Projekt

A.1.1. Eckdaten

Das untersuchte Beispielprojekt ist eine native Android Endnutzeranwendung eines deutschen Softwarehauses. Es dient lediglich dazu Größenordnungen und Aufwand abschätzen zu können. Die verwendeten Daten sind lediglich aus einer Anwendung erhoben, sind also nicht repräsentativ. Dennoch erlauben sie eine grobe Schätzungen.

LOC Projekt ohne Tests 203.000

Anzahl Systemtestfälle 370

Anzahl Page-Objects ≈ 70

Dauer automatisierter Tests $\approx 4h$

Dauer je Testfall $\approx 40s$

A.1.2. Daten exemplarischer Testfall

Eine Sektion der Testfälle, die den selben Simulator und einen Server verwendet.

19 Tests

675 LOC in den Tests

1 Simulator

1 Server

A.2. Code des Versuchs

A.2.1. Node

```
public class Node {
   private List<Edge> in;
   private List<Edge> out;

   public Node() {

      in = new ArrayList<>();
      out = new ArrayList<>();
   }

   public int getNumberOfOutgoingEdges() {
      return out.size();
   }

   public List<Edge> getOut() {
      return out;
   }
   public void addToOut(Edge edge) {
      this.out.add(edge);
   }
   public void addToIn(Edge edge) {
      this.in.add(edge);
   }
}
```

A.2.2. Edge

```
public class Edge {

   private Node from;
   private Node to;

   private int visits = 0;
   private double heat = 0;

   private String name;

   public Edge(Node from, Node to, String name) {
      this.from = from;
      this.to = to;
      this.name = name;
      from.addToOut(this);
      to.addToIn(this);
   }

   public Node getFrom() {
```

```
      return from;
   }

   public Node getTo() {
      return to;
   }

   public int getVisits() {
      return visits;
   }

   public void setVisits(int visits) {
      this.visits = visits;
   }

   public double getHeat() {
      return heat;
   }

   public void setHeat(double heat) {
      this.heat = heat;
   }

   public String getName() {
      return name;
   }
}
```

Literaturverzeichnis

[And12] T. L. Andreas Spillner. *Basiswissen Softwaretest*. Dpunkt.Verlag GmbH, 11. Sep. 2012. XXIV S. ISBN: 3864900247. URL: http://www.ebook.de/de/product/19361935/andreas_spillner_tilo_linz_basiswissen_softwaretest.html (zitiert auf S. 17–21, 25, 30, 31, 34).

[Cho78] T. Chow. „Testing Software Design Modeled by Finite-State Machines“. In: *IEEE Transactions on Software Engineering* SE-4.3 (Mai 1978), S. 178–187. DOI: 10.1109/tse.1978.231496. URL: https://doi.org/10.1109%2Ftse.1978.231496 (zitiert auf S. 25–27, 29–31, 37, 38, 43–45).

[CMM+12] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, P. Tonella. „An Empirical Study About the Effectiveness of Debugging when Random Test Cases Are Used“. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, S. 452–462. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337277 (zitiert auf S. 36, 49).

[Joc13] H. L. Jochen Ludewig. *Software Engineering*. Dpunkt.Verlag GmbH, 11. Apr. 2013. XXI S. ISBN: 3864900921. URL: http://www.ebook.de/de/product/20483151/jochen_ludewig_horst_lichter_software_engineering.html (zitiert auf S. 15, 17–19, 21, 38, 41, 45, 46).

[ldt17] ldtp. *ldtp Tutorial*. [Online; accessed 25-May-2017]. 2017. URL: https://ldtp.freedesktop.org/ldtp-tutorial.pdf (zitiert auf S. 20, 22).

[Nym00] N. Nyman. „Using monkey test tools“. In: *Software Testing & Quality Engineering Magazine* (2000), S. 18–21 (zitiert auf S. 15, 33, 34, 36, 37, 50, 60, 63).

[Sch92] U. Schöning. *Theoretische Informatik kurz gefasst*. 5.Auflage. Wissenschaftsverlag Mannheim, 1992 (zitiert auf S. 26, 30).

[sel17] selenium. *selenium page object pattern*. [Online; accessed 25-May-2017]. 2017. URL: https://github.com/SeleniumHQ/selenium/wiki/PageObjects (zitiert auf S. 15, 23, 39, 42, 61, 63).

[Stu17] I. U. Stuttgart. *Äquivalenz von Automaten*. [Online; accessed 30-May-2017]. 2017. URL: http://www.iris.uni-stuttgart.de/lehre/eggenberger/eti/23_Reduktion/Reduktion.htm (zitiert auf S. 25).

[Wik17a] Wikipedia. *Assertion (software development) — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2017]. 2017. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Assertion_(software_development)&oldid=779561246%7D (zitiert auf S. 37).

[Wik17b] Wikipedia. *Comparison of GUI testing tools — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-May-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_GUI_testing_tools&oldid=781179472 (zitiert auf S. 20, 22).

[Wik17c] Wikipedia. *Infinite monkey theorem — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-June-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Infinite_monkey_theorem&oldid=784217653 (zitiert auf S. 33, 60).

Alle URLs wurden zuletzt am 15. 06. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift