ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Modellbasierte Testautomatisierung Direktionaler Benutzeroberflächen am Beispiel von In-Vehicle Infotainment Systemen

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

Vorgelegt von:
Daniel Mauser aus Karlsruhe

2015

Amtierende Dekanin: Prof. Dr. Tina Seufert
Gutachter: Prof. Dr. Franz Schweiggert
Gutachter: Prof. Dr. Helmut Partsch
Tag der Promotion: 08.02.2016

Kurzfassung

Die mangelnde Praxistauglichkeit der modellbasierten Automatisierung von Abnahmetests direktionaler Bedienoberflächen verhindert nach wie vor einen Einsatz im industriellen Kontext. Die zentrale Einschränkung ist die bisher nicht adressierte Robustheit gegenüber Abweichung zwischen dem erwarteten und dem tatsächlichen Verhalten. Während menschliche Tester auf solche Abweichungen reagieren und den Testablauf anpassen können, muss die automatische Durchführung abgebrochen werden. Die dadurch verursachte, unkalkulierbare Verzögerung des Testfortschritts stellt ein erhebliches Risiko bei der Projektplanung dar. Bisherige Forschungsansätze vernachlässigen die Analyse von Fehlersymptomen und deren Auswirkungen auf die Praxistauglichkeit eines Testverfahrens gänzlich. Dadurch erweist sich der eigentliche Zweck der Testautomatisierung, Fehler aufzudecken, als wesentliche Limitierung.

In der vorliegenden Arbeit wird ein Verfahren zur modellbasierten Testautomatisierung vorgestellt, dass die Fortführung der Durchführung auch im Falle von Abweichungen erlaubt. Als Diskussionsgrundlage der Arbeit dienen In-Vehicle Infotainment Systeme als komplexe Ausprägung direktionaler Oberflächen. Anhand der Analyse von Fehlerberichten werden die häufigsten Fehlersymptome aktueller direktionaler Oberflächen klassifiziert und deren Auswirkungen auf die automatische Durchführung diskutiert. Die testrelevanten Informationen werden durch eine Kombination aus Zustandsautomaten und Programmcode modelliert und so alle, für den Test erforderlichen Bedienschritte und Verifikationsinformationen abgeleitet. Das vorgestellte Verfahren ermöglicht es dem Testrahmen im Falle eines Fehlers die von dem Symptom betroffenen Teile des Testfalls auf Basis des tatsächlichen Systemzustands erneut zu generieren, falls dadurch die Durchführung fortgeführt werden kann. Die Wirksamkeit der automatischen Fehlerbehandlung wird durch ein Experiment bestätigt. Das vorgestellte Verfahren beschleunigt die Behebung von Fehlern, die mit Hilfe der Mutationsanalyse auf Basis der Fehlersymptome in einen Prüfling eingebracht wurden.

Danksagung

Mein größter Dank gilt meinem Doktorvater Professor Doktor Franz Schweiggert, der mich aus der richtigen Mischung aus Freiheit und Führung durch die Höhen und Tiefen der wissenschaftlichen Arbeit geleitet hat, zuletzt sogar trotz seines Ruhestands.

Bedanken möchte ich mich zudem bei den Kollegen bei Daimler in Sindelfingen und in Ulm für die Unterstützung durch Ihr Fachwissen und Ihrem Gemeinschaftsgefühl. Insbesondere möchte ich mich bei André Berton bedanken, der die Grundlage geschaffen hat, innerhalb des Konzerns selbstbestimmt und wissenschaftlich zu arbeiten. Bei Christoph Steglich möchte ich mich dafür bedanken, dass er durch die Übernahme bürokratischer Notwendigkeiten Raum geschaffen sowie zahlreiche praktische Erfahrungen und pragmatische Standpunkte beigetragen hat.

Besonderer Dank gilt auch den Doktoranden bei Daimler und der Universität Ulm. Hervorheben möchte ich die enge und freundschaftliche Zusammenarbeit mit Felix, Frank, Hansjörg, Mark und Sven und die vielen schönen Erfahrungen, gemeinsam und doch für sich zu promovieren. Dankbar bin ich zudem Hans, Leo, Tamara sowie Alexander und Aya, die als Werkstudenten, Masteranden und Bacheloranden direkt und durch Ihre Lebensfreude indirekt zu dieser Arbeit und der schönen Zeit der Entstehung beigetragen haben.

Schließlich bedanke ich mich ganz besonders herzlich bei Alexandra, meiner Familie und meinen Freunden, die mein Leben täglich bereichern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1

Einleitung

Datenverarbeitende Geräte sind im heutigen Alltag allgegenwärtig. Insbesondere eingebettete Systeme verbreiten sich rapide. Viele Anwender sind auf deren Funktionen angewiesen, um auf Daten zuzugreifen oder Maschinen zu bedienen. Damit steigt die Bedeutung der Benutzeroberfläche als zentrales Element der Interaktion zwischen Mensch und Maschine. Während vor einiger Zeit diese Schnittstellen hauptsächlich Hardwareschalter waren, werden diese heute in vielen Bereichen durch Software-Implementierungen ersetzt, da diese vielseitiger umgesetzt und sogar zur Laufzeit einfach verändert werden können (siehe Kapitel 2.1). Bedienoberflächen unterscheiden sich durch deren Mittel der Ein- und Ausgabe, den so genannten Eingabe- bzw. Ausgabemodalitäten. Bisher am weitesten verbreitet ist die Kombination aus graphischen Ausgabegeräten und haptischer Eingabe, wie sie bspw. bei Smartphones verwendet wird. Eine weitere Variante sind Schnittstellen, die eine Auswahl aus einer Reihe an Optionen durch einen Fokuszeiger ermöglichen nehmen, dessen Position der Benutzer mit Hilfe von Richtungskommandos verändern kann (siehe 2.1.2). Diese Oberflächen werden in dieser Arbeit als Bedienoberflächen mit direktionalem Bedienkonzept bezeichnet und finden bei der Menüführung von Fotokameras, Fernsehern, In-Vehicle-Infotainment Systemen Anwendung. Abbildung 1.1 zeigt die direktionale Bedienoberfläche des In-Vehicle

Abbildung 1.1.: *BMW In-Vehicle Infotainment*

Infotainment Systems, das BMW in der Modellreihe 7er verwendet. Die Optionen auf gleicher Ebene sind untereinander angeordnet; mit den Richtungskommandos links und rechts erreicht man das übergeordnete bzw. das untergeordnete Menü.

Auf die Verbreitung von Software-basierten Oberflächen müssen auch die Maßnahmen der Qualitätssicherung angepasst werden. Gerade für die Bedienung eingebetteter Systeme ist eine komplexe Logik erforderlich, die den Prozess der Entwicklung vor neue Herausforderungen stellt. Trends wie die Erweiterung der Funktionalität mit Hilfe weiterer Software oder eine flexible Adaption der Bedienoberfläche können diese Komplexität weiter verschärfen (siehe Abschnitt 2.1). Dadurch gewinnt während der Entwicklung der Bedienoberflächen die Testphase immer stärker an Bedeutung. Wie bei anderen Softwareartefakten werden auch bei Entwicklung von Bedienoberflächen Tests auf unterschiedlichen Abstraktionsebenen durchgeführt. Diese Ebenen reichen von der Überprüfung einzelner Softwarekomponenten bis hin zur Abnahme des Gesamtsystems. Die Ebenen lassen sich gut durch das so genannte V-Modell veranschaulichen (siehe Abbildung 1.2). Während bei der Entwicklung schrittweise ein technisches Konzept erstellt wird, das die fachlichen Vorgaben erfüllen soll, wird während des Testens auf denselben Abstraktionsebenen sichergestellt, dass die Anforderungen durch die Implementierung erfüllt wurden.

Sowohl während der Entwicklung als auch während des Testens ist die Arbeitsteilung von fachlichen und technischen Experten erforderlich. Fachliche Experten erstellen Anforderungskataloge, technische Experten realisieren ein System, das diese fachlichen Anforderungen erfüllen soll. In diesem Prozess kommt es an zwei Stellen zu einer Übergabe. Im V-Modell kann diese Übergabe während der Entwicklung nach der Anforderungsdefinition und während des Tests vor der Abnahme verortet werden.

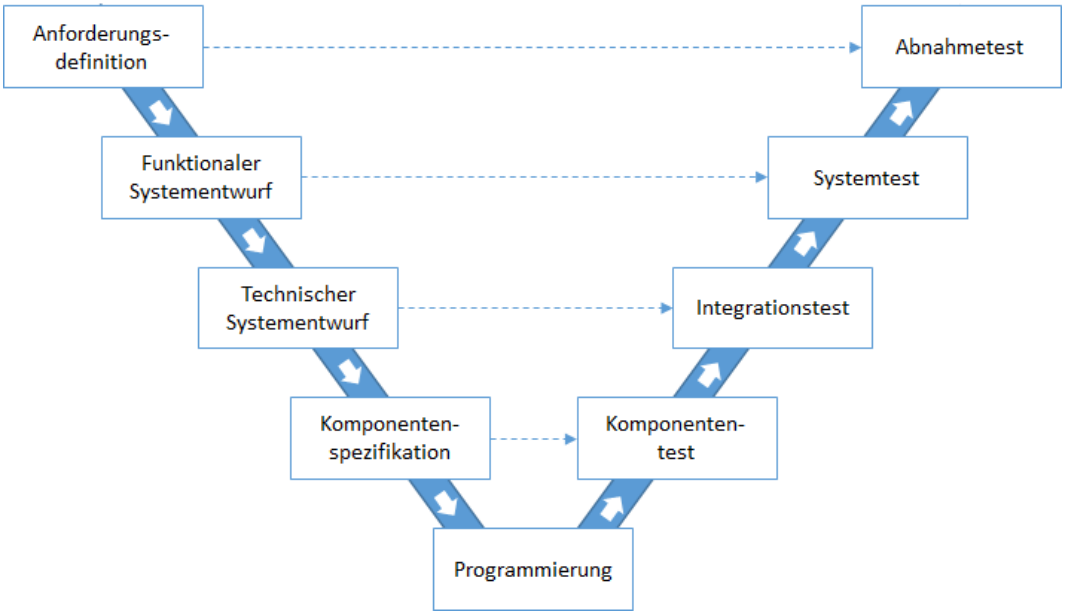


Abbildung 1.2.: *Allgemeines V-Modell nach [Spi06]*

Fachliche Experten übergeben einen Anforderungskatalog und erhalten anschließend ein realisiertes System. Bei der Abnahme müssen die fachlichen Experten entscheiden, ob das implementierte System den spezifizierten Anforderungen entspricht. Da diese Überprüfungen die Benutzung durch reguläre Benutzer repräsentieren, besteht bei der Abnahme kein Einblick in den technischen Aufbau des Systems. Unter Aufsicht und Anleitung der fachlichen Experten wird das System manuell betrieben. Allerdings sind ausschließlich manuelle Testverfahren bereits heute aufwendig und fehleranfällig (siehe Abschnitt 2.2). Inhalte werden umfangreicher und durch zunehmende Bedingungsabhängigkeiten steigen Komplexität und Dynamik der Benutzerschnittstellen an. Es ist davon auszugehen, dass manuelle Tests in den kommenden Jahren nicht mehr ausreichen, um die erwünschte Testabdeckung zu erreichen. Aus diesem Grund ist die Testautomatisierung für graphische Benutzerschnittstellen ein aktives Forschungsfeld.

Testautomatisierung auf der Ebene der Abnahmetests wird als modellbasierte Testautomatisierung (MBT) bezeichnet. Der, von den fachlichen Experten erstellte Anforderungskatalog entspricht einer abstrahierten Soll-Beschreibung und damit einem Modell der Wirklichkeit. Mit Hilfe von Algorithmen werden sowohl die zum Test notwendigen Bediensequenzen als auch die für die Verifikation erforderlichen Soll-Daten automatisch abgeleitet und anschließend gegen das zu testende System, dem so genannten System

Under Test (SUT), zur Ausführung gebracht. Für andere Modalitäten die beispielsweise einen Touch-Screen oder eine Maus verwenden, existieren bereits zahlreiche modellbasierte Ansätze für die Testautomatisierung (siehe Abschnitt 2.3). Direktionale Bedienkonzepte werden in der Literatur aktuell allerdings noch stark vernachlässigt. Dabei zeigen sich bei der Testautomatisierung dieses Bedienkonzepts im Detail grundlegende Unterschiede zu anderen Bedienkonzepten (siehe Abschnitt 2.3). In der praktischen Anwendung zeigt sich, dass die auf Basis des Modells generierten Testfälle sehr anfällig für Abweichungen zwischen erwartetem und tatsächlichem Verhalten sind. Dies lässt sich anhand der in Abbildung 1.1 gezeigten Oberfläche nachvollziehen. Angenommen, eine direktionale Oberfläche stellt die gezeigten fünf Menüpunkte bereit. Ein Test sieht vor, den dritten Menüpunkt «Business&Industry» zu aktivieren und anschließend mit der Bedienung fortzufahren. Falls sich dieser Menüpunkt allerdings fälschlicherweise aufgrund einer alphabetischen Sortierung an der ersten statt an der dritten Stelle befindet, kann der Test nicht fortgeführt werden ohne die ursprüngliche Bediensequenz anzupassen. Ein menschlicher Tester ist in der Lage, auf diesen Fehler zu reagieren und selbstständig von der Vorgabe des Testfalls abzuweichen. Diese Intelligenz besitzt ein Automat nicht. Ein verhältnismäßig geringfügiger Fehler verursacht den Abbruch der automatischen Durchführung des Testfalls. Lässt sich eine Bediensequenz nicht exakt auf die vorausberechnete Weise durchführen, schlägt der Testfall als Ganzes fehl. Diese Abweichungen sind während der Softwareentwicklung nicht zu vernachlässigen und treten sogar in späten Phasen der Entwicklung häufig auf. Neben tatsächlichen Fehlern im SUT gibt es weitere mögliche Ursachen wie unter anderem Änderungen im Anforderungskatalog, nicht absprachegemäße Umsetzungsreihenfolge oder Fehler während der Testdurchführung.

Unabhängig von der Fehlerursache verhindern die durch den Abbruch des Tests entstehenden Verzögerungen aktuell den praktischen Einsatz der modellbasierten Testautomatisierung. Die generierten Testfälle können erst dann weiter durchgeführt werden, wenn die Ursache für den Abbruch behoben ist. Als Folge bleiben die Fehler, die erst durch noch nicht durchführbare Anteile der Testfälle aufgedeckt werden, lange unerkannt. Dieses Problem wird durch die Übergabe zwischen Auftraggeber und Auftragnehmer und die dadurch erforderliche Abstimmung weiter verschärft. Nachdem ein Fehler durch die fachlichen Experten gefunden und dokumentiert wurde, beginnt der Fehlerbehebungsprozess. Es wird überprüft, ob ein Fehler des Prüflings vorliegt oder ob die Ursache

für die Abweichung unkritisch ist. Anschließend muss die Verantwortung für den Fehler geklärt werden. Gegebenenfalls ist die Abweichung auf eine unklare Spezifikation oder auf eine nicht dokumentierte mündliche Absprache zurückzuführen. Aufgrund der notwendigen Ressourcenplanung sind in einem Entwicklungsprozess Zeitpläne für die Durchführung der Tests und der Behebung der Fehler festgelegt. Dies kann unter Umständen dazu führen, dass aufgrund der Zeitplanung nicht genug Entwicklungszyklen vorgesehen sind, um alle Tests erfolgreich durchführen zu können. Diese Problematik wird aktuell in der Literatur vernachlässigt. Es bleibt zu untersuchen, welche Fehler in den genannten Oberflächen vorkommen und welche den Abbruch von Tests verursachen. Für die Akzeptanz der Modellbasierten Testautomatisierung (MBT) ist es unerlässlich, die Robustheit des Verfahrens gegenüber Fehlersymptomen zu erhöhen.

Die vorliegende Arbeit untersucht die Verwendung der modellbasierten Testautomatisierung von direktionalen Oberflächen eingebetteter Systeme im Rahmen des Forschungsprojektes automotiveHMI[1]. Ziel der Arbeit ist es, die Fehleranfälligkeit der modellbasierten Testautomatisierung zu reduzieren. Der Aufbau der Arbeit ist in Abbildung 1.3 skizziert. Zentraler Beitrag ist die Analyse von Fehlerberichten, um die Symptome zu klassifizieren, die bei der Entwicklung direktionaler Oberflächen zu beobachten sind. Diese Fehlersymptomanalyse fließt anschließend in alle Schritte der modellbasierten Testautomatisierung ein. Bei der Definition des Testfokus können mit Hilfe der Taxonomie Testziele definiert werden, indem Fehlersymptomklassen explizit ein- bzw. ausgeschlossen werden. Durch die klare Trennung kann abgeleitet werden, welche Informationen durch das Testmodell beschrieben werden müssen. Dies wiederum ist ausschlaggebend dafür, welches Beschreibungsmittel für die Modellierung geeignet ist und welche Bediensequenzen und Verifikationsinformationen abgeleitet werden müssen, um den Testfokus abzudecken.

Bei der Ausführung der generierten Testfälle fließen die Ergebnisse der Fehlersymptomanalyse an zwei Stellen ein. Erstens erlaubt die Taxonomie eine Abschätzung, welche Fehlersymptome die Durchführung der Testfälle beeinträchtigt. Zweitens dient die Taxonomie als Basis für Maßnahmen, wie Unterbrechungen der Testausführung verhindert werden können. Schließlich unterstützt die Fehlersymptomanalyse die Evaluation der entwickelten Lösung. Um die Wirkungsweise überprüfen zu können, muss ein Prüfling mit Fehlern versehen werden. Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt,

[1] www.automotivehmi.org

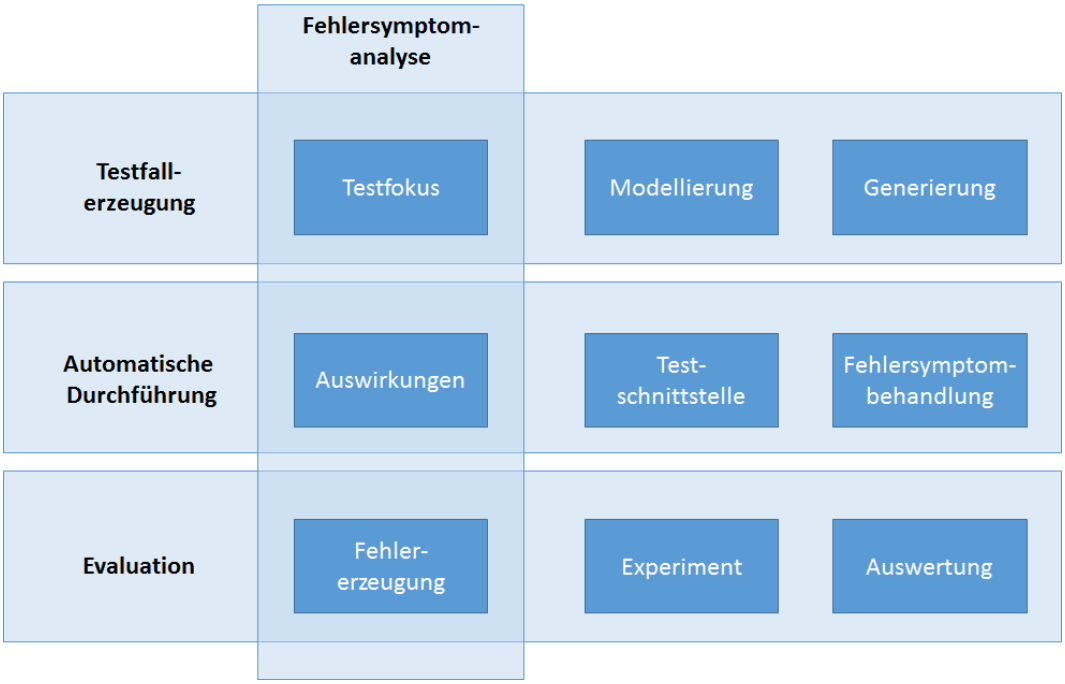


Abbildung 1.3.: *Aufbau und Beitrag der vorliegenden Arbeit*

um einen Prüfling auf eine Art zu verfälschen, so dass sich dessen Fehlerbild an die, durch die Taxonomie erfasste Verteilung annähert.

Die konkreten Beiträge der Arbeit sind demnach wie folgt:

- Fehlersymptom-Analyse
  - Taxonomie von Fehlersymptomen, die während der Entwicklung direktionaler Oberflächen beobachtet wurden
  - Diskussion der Auswirkungen der Fehlersymptomklassen während der Testdurchführung
- Modellierung & Generierung
  - Definition eines Testfokus, der durch eine modellbasierte Testautomatisierung abgedeckt werden kann
  - Definition der Informationen und Steuerparameter, die für die Ableitung entsprechender Testfälle erforderlich sind

- Automatische Durchführung
  - Entwicklung eines Verfahrens, um während der Durchführung Abweichungen automatisch einer Fehlersymptomklasse zuzuordnen
  - Entwicklung eines Verfahrens, um Fehlersymptome während der Durchführung zu umgehen
- Evaluation
  - Entwicklung einer exemplarischen Bedienoberfläche
  - Entwicklung eines Verfahrens, um das Fehlerbild einer Software der durch die Taxonomie vorgegebenen Verteilung anzunähern
  - Entwicklung einer Software, um die generierten Testfälle ausführen und Fehlersymptome umgehen zu können

Nach der Diskussion der Grundlagen und dem aktuellen Stand der Forschung in Kapitel 2 folgt der Aufbau der Arbeit den Phasen der modellbasierten Testautomatisierung. In Kapitel 3 werden die Voraussetzung und Möglichkeiten der modellbasierten Testautomatisierung direktionaler Oberflächen untersucht. Dies beinhaltet die Ergebnisse der Fehlersymptomanalyse, die Ableitung eines geeigneten Testfokus sowie die Modellierung der testrelevanten Informationen und Generierung der Testfälle. In Kapitel 4 werden die Aspekte diskutiert, die für die Durchführung der Testfälle entscheidend sind. Dies beinhaltet die Auswirkungen der Fehlersymptomklassen auf den Testverlauf, die automatische Einordnung auftretender Abweichungen anhand der Taxonomie sowie die Vorstellung eines Verfahrens, um Fehlersymptome während der Durchführung zu umgehen. In Kapitel 5 wird die Wirksamkeit des vorgestellten Verfahrens durch ein Experiment überprüft. Im Rahmen dieser Untersuchung wird ein Verfahren vorgestellt, um die Implementierung einer exemplarischen Bedienoberfläche auf eine Weise mit Fehlern zu versehen, um deren Fehlerbild der durch die Taxonomie vorgegebenen Verteilung anzunähern. Die Arbeit schließt mit einer Zusammenfassung und einem Überblick möglicher Ansatzpunkte zukünftiger Arbeiten in Kapitel 6.

2

Grundlagen

In diesem Kapitel werden die Grundlagen der modellbasierten Testautomatisierung (MBT) graphischer Benutzerschnittstellen erläutert und die aktuellen Forschungsansätze zusammengefasst. Zu Beginn werden Einsatzzweck, Anwendungskontext und Grundlagen graphischer Benutzerschnittstellen vorgestellt. Ziel ist es, herauszuarbeiten, welcher Ansatzpunkt sich für den Einsatz der Testautomatisierung eignet und inwiefern sich die vorliegende Arbeit von anderen Ansätzen unterscheidet. Da nicht die Benutzeroberfläche als solche sondern der Prozess der Entwicklung untersucht wird, ist für die Testautomatisierung entscheidend, welches Vorgehensmodell dem Entwicklungsprozess zugrunde liegt. Das bei der Entwicklung eingebetteter Systeme häufig eingesetzte Vorgehen wird vorgestellt und ein für das modellbasierte Vorgehen geeigneter Testansatz im Detail diskutiert. Anschließend werden die bestehenden Arbeiten zur Modellbasierte Testautomatisierung (MBT) vorgestellt und deren Eignung auf das ausgewählte Szenario erörtert. Schließlich wird herausgearbeitet, welche Probleme sich bei der Anwendung bestehender Arbeiten ergeben und ein Lösungsvorschlag vorgestellt.

2.1. Mensch-Maschine-Schnittstelle

In diesem Abschnitt wird skizziert, welche Computersysteme adressiert werden und abgeleitet, welche Bedienkonzepte in deren Anwendungskontexten relevant sind. Der Abschnitt schließt mit der Beschreibung einer konkreten Ausprägung einer Bedienoberfläche, die in dieser Arbeit als Diskussionsgrundlage dient.

2.1.1. Eingebettete Systeme

Gestalt und Anwendungsgebiete informationsverarbeitender Geräte haben sich im Laufe der Zeit stark verändert. Während anfangs große Mainframe-Rechner notwendig waren, erlaubte die Technik nach und nach die Entwicklung kompakterer Geräte. Nach einer weiten Verbreitung von Personal Computern, die an einem Arbeitsplatz eingesetzt werden konnten, sind heute kleine Computer überall in unserem Alltag zu finden [Mar08]. Eine Art der Geräte, die sich immer weiter verbreitet, sind so genannte *eingebettete Systeme*. Laut [Mar08] ist für diese Systeme charakteristisch, mit der physikalischen Umwelt über Sensoren und Aktuatoren verbunden zu sein. Damit stehen Software- oder Hardware-Systeme in enger, wechselseitiger Abhängigkeit zueinander [Rom05].

Für den Benutzer sind eingebettete Systeme selbst nicht offensichtlich, da diese häufig in ein größeres Produkt integriert sind [Mar08]. Dabei ist die Informationsverarbeitung nicht der vordergründige Verwendungszweck des Gesamtsystems. Als typische Anwendungsgebiete nennen sowohl [Mar08] als auch [Rom05] unter anderem den Automobilbereich, die Telekommunikation, Haushaltsgeräte, Unterhaltungselektronik oder medizinische Systeme. Software übernimmt dabei die Steuerung und Überwachung der Hardware-Komponenten. Ein Beispiel aus dem Alltag ist das in Abbildung 2.1 dargestellte Kochfeld. Während für die Temperaturregelung der Herdplatten früher die Verwendung von physischen Drehregler selbstverständlich schienen, wird dies in dem vorliegenden Beispiel ebenso durch Software realisierte.

Für den Anwender ist die *Mensch Maschine Schnittstelle* (im Englischen Human-Machine Interface, HMI) das zentrale Element des Gesamtsystems. Diese Schnittstelle stellt dem Benutzer die Mittel zur Verfügung, den Zustand des Systems einzusehen und zu verändern [ISO10]. Wie das Beispiel des Kochfeldes in Abbildung 2.1 zeigt, kann häufig erst durch die Einflussnahme des Menschen der Verwendungszweck des Systems erfüllt

Abbildung 2.1.: *Siemens Kochfeld EF601HN17 [Sie15]*

werden. Ohne die Möglichkeit, die Temperatur der Herdplatten über die Bedienoberfläche zu regulieren, wäre das Kochfeld nutzlos.

Für die Interaktion können alle Sinne des Benutzers angesprochen werden. Die Mittel, mit denen ein System dem Benutzer Informationen präsentiert, wird als *Ausgabemodalität* bezeichnet. Beispiele sind die graphische Ausgabe auf einem Bildschirm, die auditive Ausgabe durch Sprachausgabe, oder die haptische Ausgabe einer Braille-Zeile. Durch die sogenannten *Eingabemodalitäten* werden die Möglichkeiten des Benutzers zur Einflussnahme auf das System zusammengefasst. Beispiele sind die haptische Eingabe durch Maus, Tastatur oder Touchscreen sowie die auditive Eingabe mit Hilfe einer Sprachbedienung. Die Möglichkeiten, Ein- und Ausgabemodalitäten zu einem interaktiven System zusammenzustellen, sind vielfältig. In einem Touchscreen können Ein- und Ausgabemodalitäten derart kombiniert werden, dass es für den Anwender wirkt, als könnten die Elemente auf dem Bildschirm direkt bedient werden. Aus der technischen Perspektive müssen beide Modalitäten dennoch separat voneinander betrachtet werden. Ein System kann Interaktion durch mehrere Eingabemodalitäten erlauben. Viele mobile Geräte bieten heute Touch-Bedienung, Sprachkommandos oder eine Kombination der Modalitäten an. In manchen Fällen passt sich das System darauf an und verwendet dieselbe Modalität zur Ausgabe.

Die Ausgestaltung der Oberflächen eingebetteter Systeme ist vielfältig. Wie viele andere Funktionen wird auch die Oberfläche zunehmend in Software realisiert [Mar08, Hof08]. Neben dem Kochfeld in Abbildung 2.1 ist ein weiteres Beispiel dieses Trends die Instrumententafel von Automobilen (siehe Abbildung 2.2). Üblicherweise werden hier Informationen wie die Geschwindigkeit des Fahrzeugs, Drehzahl des Motors oder die Temperatur des Kühlwassers angezeigt. Traditionell werden dafür Rundelemente verwen-

(a) *S-Klasse (2013) [Dai13]* **(b)** *Audi TT (2014) [Aud14]*

Abbildung 2.2.: *Instrumententafel von Automobilen*

det. Bis vor wenigen Jahren war diese Funktion durch Hardware Komponenten realisiert. Mit der S-Klasse von Mercedes-Benz (Abbildung 2.2a) und dem Audi TT (Abbildung 2.2b) wurden in den vergangenen Jahren Automobile vorgestellt, die ausschließlich einen Bildschirm zur Anzeige verwenden. Während Mercedes die klassischen Elemente nachahmt, setzt Audi auf eine Neugestaltung der Oberfläche. In der Navigationsansicht nimmt eine Karte den Hauptteil der Anzeige ein. Auch hier zeigt sich die universelle Verwendung digitaler Bedienoberflächen; die herkömmlichen Informationen über den Zustand des Fahrzeugs treten in den Hintergrund. Durch die Realisierung mittels Software ist das Anwendungsgebiet der Benutzerschnittstelle nicht mehr durch einmalige Gestaltung und Montage festgesetzt, sondern kann vielseitig verändert und bei Bedarf nachträglich angepasst werden. Aufgrund der wechselseitigen Abhängigkeit mit angeschlossenen Regelsystemen weisen Benutzerschnittstellen eingebetteter Systeme allerdings häufig eine hohe Komplexität aus [Rom05]. Eingebetteten Systemen wird in der Lehre und der öffentlichen Diskussion häufig nicht die notwendige Aufmerksamkeit gewidmet [Mar08]. Dies ist unter anderem auf die aufwändige Ausstattung und die hohe Komplexität des Themas zurückzuführen. Die Auswirkungen dieses Trends wird in den folgenden Abschnitten anhand einer Ausprägung einer Bedienoberfläche vorgestellt.

2.1.2. Direktionale Bedienung

Eine Ausprägung graphisch-haptischer Benutzeroberflächen eingebetteter Systeme, die u.a. bei Fotokameras, einigen IVI Systemen oder aktuellen Smart TVs Anwendung findet, wird in dieser Arbeit als *direktionales Bedienkonzept* bezeichnet. Das grundle-

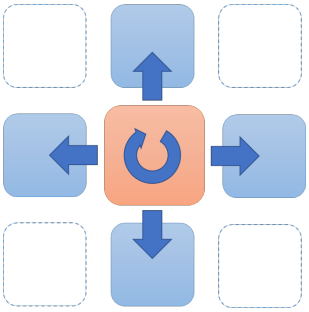


Abbildung 2.3.: *Prinzip des direktionalen Bedienkonzepts*

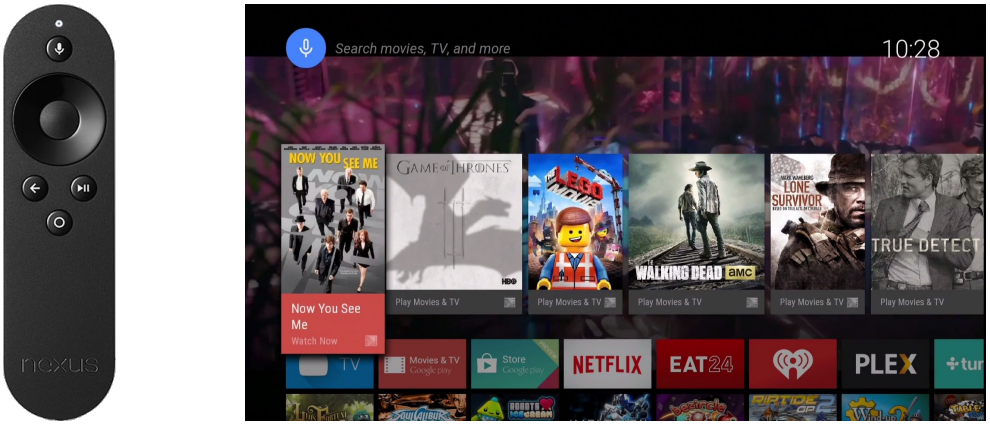
gende Prinzip wird in Abbildung 2.3 gezeigt. Das Computersystem bietet dem Benutzer alle verfügbaren Optionen als auswählbare Entitäten an. In der Abbildung sind diese Entitäten als Quadrate dargestellt. Ein *Fokuszeiger* markiert das aktuell ausgewählte Kommando und kann durch den Benutzer mit Hilfe von Richtungskommandos (links, rechts, hoch, runter) verschoben werden. Ist die gewünschte Option ausgewählt, kann der Benutzer diese durch eine weitere Bedienaktion aktivieren. Die Eingabe kann durch Directional Pads (DPAD), Drehräder, die von Blackberry bekannten optischen Trackpads, oder Wischgesten realisiert werden. Je nach Ausprägung von Oberfläche und Eingabegerät können die Bedienkommandos *diskret* oder *konsekutiv* durchgeführt werden. Auf diese Weise wird festgelegt, ob pro Bedienaktion nur ein einzelnes oder mehrere aufeinanderfolgende Bedienkommandos vom System ausgewertet werden.

Die Vorteile des direktionalen Bedienkonzeptes sind die Robustheit und Vielseitigkeit, die sich aus den reduzierten Interaktionsmöglichkeiten ergeben. Im Gegensatz zu beispielsweise einem Touchscreen können Ein- und Ausgabegeräte voneinander abgesetzt werden. Zudem ermöglicht die robuste Bedienung, dass die Interaktionen einfach zu erlernen und durchzuführen sind. Auch bei sehr wenig physischer Bewegung des Benutzers ist eine präzise Eingabe möglich. Die klar voneinander getrennten Bedienschritte erlauben eine wenig störanfällige Interaktion mit dem System. Dadurch kann die Bedienung trotz eingeschränktem Bewegungsfreiraum durchgeführt werden, wie es unter anderem durch das Tragen von Handschuhen oder bei der Bedienung in der Bewegung ergeben kann. Des Weiteren ermöglicht die reduzierte Eingabe die Konstruktion von Eingabegeräten, die robust gegenüber Stürzen oder Umwelteinflüssen sind. Die reduzierte Darstellung ermöglicht die Verwendung des direktionalen Bedienkonzeptes auch auf kleinen Displays, da lediglich das aktuell aktive Element dargestellt werden

muss. Die Nachteile des direktionalen Bedienkonzepts liegen in der geringen Benutzerfreundlichkeit. Die Auswahl einer Option aus einer großen Anzahl an Alternativen erfordert insbesondere bei einer strikt diskreten Ausprägung eine größere Anzahl an Bedienschritten, als bei einer Bedienung durch einen Touchscreen oder eine Maus. Ziel einer direktionalen Bedienung ist die Zweckmäßigkeit der Interaktion; die Freude bei der Benutzung ist zweitrangig.

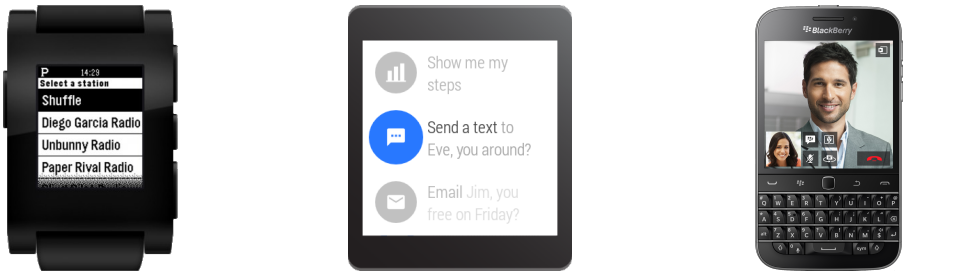
Im Folgenden werden einige Beispiele aktueller Systeme aufgeführt, die das direktionale Bedienkonzept anwenden. Abbildung 2.4 zeigt das Betriebssystems Android TV für den Google Nexus Player, mit dem Inhalte aus dem Internet geladen und auf dem Fernsehbildschirm angezeigt werden können. Inhalte sind in erster Linie Filme, Folgen von TV-Serien sowie Spiele. Bedient wird das Gerät mit Hilfe einer Fernbedienung. Diese bietet dem Benutzer mit dem Steuerkreuz und dem Auswahl-Button die essentiellen Kommandos eines direktionalen Bedienkonzeptes. Weitere Beispiele sind Wearables oder Smartphones. Die Pebble Smart Watch (siehe Abbildung 2.5a) verwendet abgesetzte Tasten, die abhängig vom Benutzungskontext belegt werden. In dem gezeigten Beispiel dienen die Tasten der Iteration verfügbarer Musiktitel bzw. dem Abspielen der aktuellen Auswahl. Android Wear (siehe Abbildung 2.5b) ist ein von Google angebotenes Betriebssystem für so genannte Wearable Devices. Durch Wischgesten verändert der Benutzer die Auswahl, die er anschließend per Touch aktivieren kann. Auch die von Blackberry verwendeten optischen Trackpads (siehe in der Mitte von Abbildung 2.5c) ermöglichen es dem Benutzer durch Wischgesten über einen optischen Sensor, die Position eines Fokuszeigers auf dem Bildschirm zu verändern.

Ein Einsatzgebiet direktionaler Benutzerschnittstellen, das keine graphische Ausgabe verwendet, ist die Funktionalität zur Barrierefreiheit Google TalkBack, die die ansonsten hauptsächlich auf graphische Ausgabe basierte Bedienung von Android ergänzt [Goo14c]. TalkBack ermöglicht Sehbehinderten, Inhalte auf dem Bildschirm abzufragen und zu bedienen. Nach Aktivierung von TalkBack werden alle Elemente auf dem Bildschirm durch einen Fokuszeiger selektierbar. Bei Selektion eines Elements wird dessen Inhalt durch Sprachausgabe verbalisiert. Die Selektion kann unter anderem durch die Wischgeste, einem so genannten Swipe, verändert werden. Ein Swipe nach rechts oder unten selektiert das nächste Element, ein Swipe nach links oder oben selektiert das vorangegangene Element. Buttons werden durch einen anschließenden Doppeltab aktiviert. Um TalkBack für eine App zu ermöglichen muss bei der Entwicklung die Fokus



(a) **(b)**

Abbildung 2.4.: *Google Android TV [Goo14b]*



(a) *Pebble[Peb14]* **(b)** *Android Wear [Goo15]* **(c)** *Blackberry [Bla15]*

Abbildung 2.5.: *Beispiele direktionaler Bedienoberflächen*

Navigation berücksichtigt werden [Goo14c]. So wird manuell festgelegt, welchen Pfad der Fokus durch die Applikation nehmen soll und welcher Text bei der Selektion des Elements ausgegeben wird.

Die Oberflächen von In-Vehicle Infotainment (IVI) Systemen, die durch ein so genanntes Zentrales Bedienelement bedient werden (siehe Abbildung 2.6) sind ein Beispiel für den umfangreichen Einsatz des direktionalen Bedienkonzepts. IVI Systeme bieten den Insassen eines Autos Zugriff auf Infotainment Funktionalität wie Radio, Fernsehen oder Musik. Die Ausgabe dieser Systeme erfolgt üblicherweise graphisch auf einem Display in der Mittelkonsole des Fahrzeugs. Vor allem die Hersteller hochpreisiger Automobile setzen auf das direktionale Bedienkonzept. Das ZBE (siehe Abbildung 2.6a) besteht aus einem Steuerrad, das gedreht, geneigt und eingedrückt werden kann, um die jeweiligen Bedienkommandos des direktionalen Bedienkonzepts auszulösen. Zusätzlich werden dem Benutzer Kurzwahltasten angeboten, um Anwendungen wie CD, Telefon oder Navigation direkt zu starten. Die angebotenen Optionen, die aktuelle Position des Fokuszeigers und die Infotainment-Inhalte werden auf einem Bildschirm angezeigt (siehe Abbildung 2.6b). Im Rahmen dieser Arbeit war ein weitreichender Einblick in die Entwicklung des IVI Systems bei Mercedes-Benz möglich. Dies beinhaltet den ständigen Zugriff auf verschiedene Versionen des Systems, direkten Kontakt zu Entwicklern und dem damit verbundenen Einblick in die Entwicklungshistorie. Die Benutzeroberflächen hängen stark von diversen Bedingungen, wie dem Modell oder der Ausstattung des Fahrzeugs, den angeschlossenen Geräten und sogar der aktuellen Geschwindigkeit des Fahrzeugs ab und sind daher eine sehr komplexe Anwendung des direktionaler Bedienkonzeptes. Aufgrund der hohen Bedingungsabhängigkeit und des Einblicks des Autors in den Entwicklungsprozess dienen IVI Systeme in dieser Arbeit als Diskussionsgrundlage und werden im Folgenden detailliert vorgestellt.

Als Hauptargument für das direktionale Konzept wird eine im Vergleich zur Touch-Bedienung geringere Ablenkung genannt [Dor11]. Bei der Bedienung eines IVI Systems handelt es sich in der Regel um ein Dual- bzw. Multi-Task-Szenario. Das bedeutet, dass zwei bzw. mehrere Aufgaben gleichzeitig ausgeführt werden. Dabei ist die primäre Aufgabe, das Fahrzeug zu steuern. Die Bedienung des IVI Systems ist der Fahraufgabe als sekundärer Task untergeordnet. Die Ablenkung muss bei der gesamten Interaktion, also sowohl bei der Ein- als auch bei der Ausgabe berücksichtigt werden. Die graphische Ausgabe sollte so angebracht werden, dass der Fahrer den Inhalt erfassen kann,

(a) **(b)**

Abbildung 2.6.: *BMW In-Vehicle Infotainment [BMW15]*

ohne die Fahraufgabe zu vernachlässigen. Die Driver Focus-Telematics Working Group der Alliance of Automotive Manufactures hat die wichtigsten Prinzipien, Kriterien und Verifikationsmethoden für die Ablenkung von IVI Systemen zusammengefasst und in Sektionen wie Präsentationsprinzipien, Interaktionsprinzipien und Verhaltensrichtlinien unterschieden [Dri06].

In heutigen Fahrzeugen befindet sich häufig ein großer Monitor in der Mittelkonsole, das *Zentrale Informationsdisplay*. Informationen, die für die Fahraufgabe relevant sind, werden in der Instrumententafel dargestellt oder durch ein Head-Up-Display an die Frontscheibe projiziert. Das Eingabegerät sollte idealerweise so angebracht werden, dass der Fahrer seine Position während der Bedienung kaum verändern muss. Kombiniert man diese Interaktionen in einem Gerät, wie dies bei einem Touch-Screen der Fall ist, muss ein Kompromiss zwischen Eingabe und Ausgabe gefunden werden. Sind Eingabe und Ausgabe getrennt, kann die Ablenkung beider Aspekte gesondert optimiert werden.

Die dargestellten Inhalte typischer IVI Systeme sind breit gefächert. Eine ausführliche Auflistung unterschiedlicher Funktionen ist in [BS11] gegeben. Beispiele sind die Konfiguration des Fahrzeugs wie Innenlicht und Sitzeinstellung, Navigation, das Abspielen von Musik- oder Videoinhalten, Telekommunikation wie Telefonie und Internet. Der durch IVI Systeme angebotene Inhalt beginnt sich zu wandeln. In den vergangenen Jahren haben Automobilhersteller die installierten Anwendungen um Angebote erweitert, die durch das IVI System aus dem Internet abgerufen werden können. Dadurch werden beispielsweise Zugriffe auf Facebook, Wetterdaten oder Nachrichten umgesetzt. Außerdem haben zuletzt Hersteller von Endbenutzergeräten IVI Systeme als möglichen Markt für sich entdeckt. Apple bietet mit CarPlay [App15] und Google über Android Auto [Goo14a] an,

die jeweiligen Smartphones über die im Auto verbaute Hardware bedienen zu können. Beide Systeme unterstützen auch die direktionale Bedienung per ZBE. Durch diese externen Geräte kann der Anwender die Funktionalität des IVI Systems durch Installation zusätzlicher Software erweitern.

Die Funktionen eines IVI Systems sind in hohem Maße abhängig vom Kontext der Benutzung. Unter dem Begriff *Anwendungskontext* (synonym: *Benutzungskontext*) werden in dieser Arbeit die Einflussfaktoren zusammengefasst, die die Mensch-Maschine-Interaktion beeinflussen. Dies sind unter anderem die aktuell ausgewählte Applikation, vorgenommene Einstellungen oder angeschlossene Systeme. Über die Fahrzeugausstattung wird beispielsweise bestimmt, ob Sitze montiert sind, die über das IVI System eingestellt werden können. Eine weitere Abhängigkeit sind die grundsätzlich anschließbaren bzw. aktuell verfügbaren Medien. Also ob beispielsweise ein Anschluss eines USB Mediums vorgesehen ist, bzw. ob und in welcher Form Audio oder Video Inhalte abgespielt werden können. Dabei spielt eine entscheidende Rolle, welche dieser Medien, und damit welche Inhalte, aktuell verfügbar sind. Zudem sind IVI Systeme in der Regel für unterschiedliche Märkte, wie den europäischen, US-amerikanischen oder japanischen Markt vorgesehen. Dies beeinflusst beispielsweise die Adresseingabe beim Starten der Zielführung oder die Verfügbarkeit unterschiedlicher Radio-Tuner und Wellenbänder. Darüber hinaus kann der Benutzer in der Regel unter diversen Sprachen wählen. Abhängig vom Anwendungskontext werden Optionen ein- oder ausgeblendet. Es ist eine komplexe Logik notwendig, um den Benutzungskontext korrekt auszulesen und die Benutzeroberfläche entsprechend anzupassen. Dadurch ergibt sich eine erhebliche Komplexität unterschiedlicher Abhängigkeiten. Trends wie die Anpassbarkeit der Oberfläche durch den Benutzer [MP15, WS13, MS11] oder automatische Adaption der Oberfläche aufgrund des Bedienkontexts [MP15, HBR+14, HSW+13] werden diese Komplexität voraussichtlich verstärken.

Aus dem Einsatzgebiet eingebetteter Systeme ergeben sich hohe Qualitätsanforderungen. Durch die wechselseitige Abhängigkeit mit angeschlossenen Regelkreisen bestehen bei eingebetteten Systemen gesteigerte Anforderungen an Verlässlichkeit, Zuverlässigkeit, Verfügbarkeit und Betriebssicherheit. Charakteristisch für eingebettete Systeme sind lange Produktlebenszyklen. Am Beispiel des Automobils ist von einer Betriebs- und Servicephase von 7 bis 15 Jahren auszugehen [Rom05]. In dieser Zeit ist es selten möglich, Updates einzuspielen. Aufgrund der Vielzahl an Herausforderungen

stellen IVI Systeme einen idealen Untersuchungsgegenstand für die modellbasierte Testautomatisierung dar.

2.1.3. Zusammenfassung

In diesem Abschnitt wurde die Alltäglichkeit informationsverarbeitender Systemen thematisiert. Es wurde herausgestellt, dass viele dieser Systeme in einen technischen Kontext eingebunden und damit den eingebetteten Systemen zuzuordnen sind. Die starke wechselseitige Abhängigkeit mit angeschlossenen Regelkreisen zeigt sich unter anderem bei der Entwicklung der Bedienoberfläche, mit Hilfe derer der Zustand des Gesamtsystems angezeigt und verändert werden kann. Aufgrund aktueller Trends wird die Komplexität dieser Systeme weiter zunehmen. Als ein verbreitetes Bedienkonzept eingebetteter Systeme wurden direktionale Oberflächen vorgestellt. Der Benutzer steuert durch Richtungskommandos eine Eingabemarke auf dem Bildschirm. IVI Systeme wurden als geeigneter Untersuchungsgegenstand eingeführt und dienen in dieser Arbeit als Diskussionsgrundlage. Im Folgenden wird auf den Prozess eingegangen, der der Entwicklung zugrunde liegt. Besonderes Augenmerk liegt auf der Sicherung der Softwarequalität.

2.2. Entwicklung & Test

Im vorangegangenen Abschnitt wurden Benutzerschnittstellen als zentraler Bestandteil des täglichen Lebens vorgestellt. Ein Großteil der Oberflächen dient der Überwachung und Kontrolle elektronischer Regelkreise und wird als eingebettetes Systeme bezeichnet. Ein immer wiederkehrendes Bedienkonzept dieser Oberflächen setzt auf direktionale Benutzereingaben. Oberflächen mit diesem Bedienkonzept wurden als Untersuchungsgegenstand dieser Arbeit ausgewählt. In diesem Abschnitt wird darauf eingegangen welche Schritte unternommen werden können, um die Qualität der Oberflächen sicherzustellen. Zu diesem Zweck werden Vorgehensmodelle für die Entwicklung eingebetteter Systeme vorgestellt und daraus resultierende Ansätze für Qualitätssicherungsmaßnahmen abgeleitet. Schließlich wurden die Eignung für Testautomatisierung erörtert und ein Ansatzpunkt im Detail diskutiert.

Software Engineering für eingebettete Systeme liegt zwischen der Entwicklung klassischer Software und der Systementwicklung [Rom05]. Verwendet werden Lösungen aus beiden Disziplinen und durch eigene Methoden ergänzt. Entwickler unterschiedlicher Fachrichtungen aus Hard- und Software entwickeln gemeinsam ein Gesamtsystem. Daraus ergibt sich eine Herausforderung an den Entwicklungsprozess, verschiedenartigen Anforderungen und Vorgehensweisen gerecht zu werden. Die Entwicklung folgt in den Grundzügen dem so genannten *Wasserfallmodell* [Roy70]. Da die Entwicklungsentscheidungen übergeordneter Systeme den Entwurf von Subsystemen maßgeblich beeinflussen, können untergeordnete Systeme erst dann entwickelt werden, wenn alle Anforderungen umschließender Systeme bekannt sind.

Abbildung 2.7 zeigt dieses Vorgehen anhand einer Version des V-Modells für eingebettete Systeme nach [Rom05]. Zu Beginn der Entwicklung wird die Problembeschreibung für das Gesamtsystem festgesetzt. Während des Arbeitsschritts «Anforderungsdefinition» wird das Problem in einzelne fachliche Anforderungen aufgeschlüsselt. Anschließend werden Subsysteme skizziert, mit Hilfe derer die Anforderungen an das Gesamtsystem erfüllt werden können. In diesem Beispiel werden Hardware und Software als mögliche Subsysteme genannt. Für jedes dieser Subsysteme wird erneut eine Anforderungsanalyse durchgeführt und ein Konzept abgeleitet. Die Anforderungen des jeweils übergeordneten Systems dienen dabei als Problembeschreibung. Das Konzept kann die Entwicklung weiterer ungeordneter Systeme vorsehen. Dieses Verfahren wird solange wiederholt, bis die Subsysteme ausreichend eindeutig beschrieben sind und realisiert werden können.

Aus der Vielschichtigkeit ergeben sich spezifische Anforderungen an den Entwicklungsprozess. Ein eingebettetes System ist häufig das Ergebnis einer Zusammenarbeit mehrerer Firmen [Rom05]. Ist beispielsweise die Software nur ein Teil eines Systems, dann ist die Organisation gegebenenfalls nicht auf deren Entwicklung ausgerichtet. Hier kann es sinnvoll sein, die Realisierung in Zusammenarbeit mit einem externen Partner durchzuführen. Dadurch ergibt sich eine, häufig sogar organisatorische, Trennung in *Auftraggeber* und *Auftragnehmer*. Diese Trennung lässt sich gut anhand des V-Modells in Abbildung 2.7 veranschaulichen, da die Arbeitsteilung üblicherweise ab einer Abstraktionsebene der Konzeption erfolgt. Der Auftragnehmer ist für die Konzeption des Gesamtsystems verantwortlich und gibt vor, welche grundsätzliche Funktionalität durch das System bereitgestellt werden soll. Der Auftraggeber übernimmt die ersten Iterationen

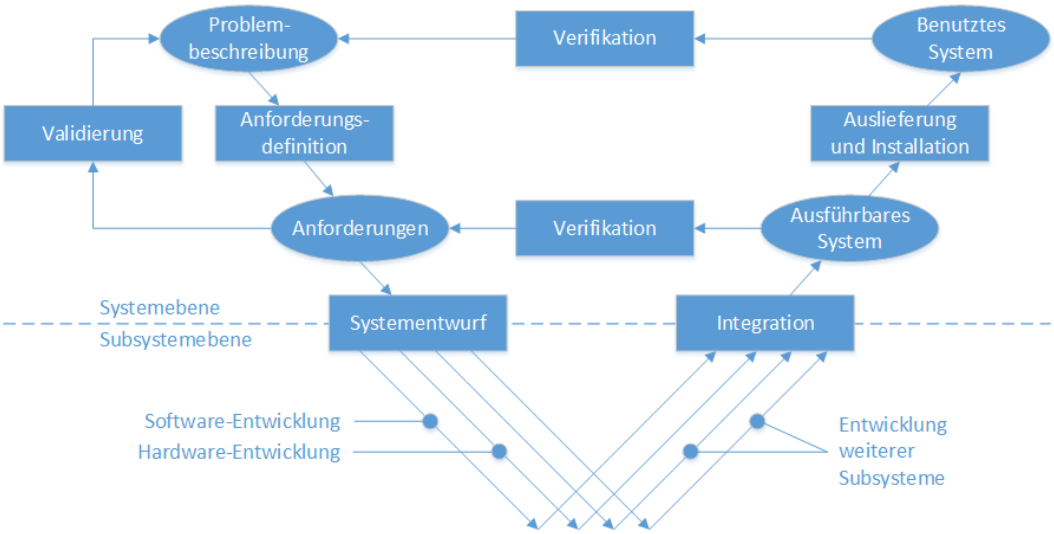


Abbildung 2.7.: *V-Modell der Entwicklung eingebetteter Systeme [Rom05]*

der Konzeption bis zu einem Punkt, an dem er nicht mehr über das notwendige Fachwissen verfügt. Ab dieser Abstraktionsebene wird ein Auftragnehmer mit der weiteren Konzeption und der Realisierung des Subsystems beauftragt. Der Auftraggeber vermittelt dem Auftragnehmer die Anforderungen an das Subsystem. Dies kann durch eine schriftliche Dokumentation der Anforderung in einer so genannten *Spezifikation* erfolgen. Der Auftragnehmer interpretiert diese Anforderungen und realisiert das Subsystem. Entscheidend ist, dass der Auftragnehmer dem Auftraggeber nach der Realisierung ein Subsystem übergibt, das dieser in das zwischenzeitlich fertiggestellte Gesamtsystem integrieren kann. Das Subsystem im Kontext dieser Arbeit ist die Benutzeroberfläche eines eingebetteten Systems. Während die Hardware und Hardwaresteuerung vom Auftraggeber verantwortet werden, wird die technische Realisierung der Benutzeroberfläche an einen Auftragnehmer übergeben. Als Grundlage für die Anforderungen dient die fachliche Funktionalität der Oberfläche inklusive der Schnittstellenbeschreibungen zur Anbindung in das Gesamtsystem. Der Auftragnehmer liefert anschließend eine Software, mit deren Hilfe die Schnittstellen bedient und dadurch das Gesamtsystem gemäß der fachlichen Anforderungen überwacht und manipuliert werden kann.

Durch diese Arbeitsteilung entsteht ein Bruch. Es besteht die Gefahr, dass wesentliche Informationen verloren gehen oder die Spezifikation unvollständig oder widersprüchlich ist. Daher liegt es im Interesse des Auftraggebers, sicherzustellen, dass das zugelieferte Subsystem die spezifizierten Anforderungen erfüllt. Diese Ebene wird in der Literatur als

Abnahmetest bezeichnet [Spi06, Rom05]. An diesem Punkt ist davon auszugehen, dass die Tests einzelner Softwarekomponenten bereits von den Entwicklern durchgeführt wurden und die Software technisch weitgehend stabil ist. Für den Auftraggeber stellt sich nun die Frage, ob die Software den fachlichen Anforderung entspricht. Im folgenden Abschnitt wird auf die Voraussetzungen und Möglichkeiten der Tests auf dieser Abstraktionsebene näher eingegangen.

2.2.1. Funktionstests

Im vorangegangenen Abschnitt wurde das Modell eines möglichen Entwicklungsprozesses vorgestellt. Es wurde hervorgehoben, dass die Teststufe des Abnahmetests eine mögliche Fehlerquelle birgt, da die Testverantwortung auf dieser Abstraktionsebene von Entwicklern zu den Spezifikateuren übergeht. Im Folgenden wird der *spezifikationsbasierte Funktionstest* der Benutzeroberfläche als ein Aspekt des Abnahmetests näher betrachtet und anhand der Literatur charakterisiert. Der spezifikations-basierte Funktionstest wird von einem Auftraggeber durchgeführt, wenn die Implementierung des zu testenden Systems vorliegt, die durch einen Auftragnehmer anhand einer Spezifikation angefertigt wurde. Durch den Test wird sichergestellt, dass die Benutzeroberfläche die gesamte, in der Spezifikation beschriebene Funktionalität bereitstellt.

Ein Test ist grundsätzlich ein Abgleich zwischen einer Referenz, dem so genannten *Soll-Zustand*, die durch eine Spezifikation vorgegeben ist und der tatsächlichen Implementierung, die im Folgenden als *Ist-Zustand* bezeichnet wird. Auf dieser Teststufe ist nicht davon auszugehen, dass während des Tests Zugriff auf den Programmcode besteht. Die Tests werden aus der Perspektive des Endnutzers durchgeführt. Daher besteht ein Testfall aus einer Reihe regulärer Bedienaktionen. Ausgehend von der Spezifikation werden die Bediensequenzen abgeleitet, die erforderlich sind, die benötigten Systemzustände herzustellen. Auf Basis der Spezifikation muss dann entschieden werden, ob sich das System erwartungskonform verhält. Ein Testfall besteht also aus *Aktionen* und *Verifikationen*.

Bei der Ausführung von Testfällen werden Abweichungen vom erwarteten Systemzustand festgestellt. Bei der Diskussion des Testverfahrens muss zwischen *Fehlerursachen* (eng. fault) und *Fehlersymptomen* (engl. failure) unterschieden werden (siehe Abbildung 2.8) [IEE09]. Die Fehlerursache ist eine fehlerhafte Code-Zeile oder ein fehlerhafter

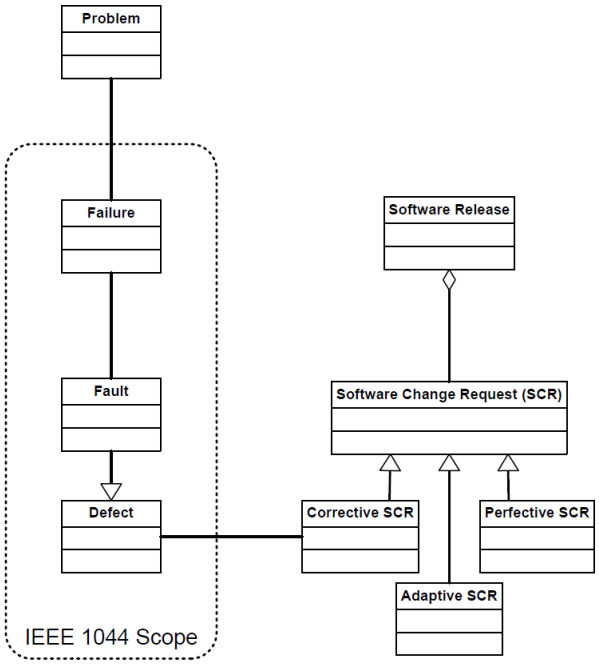


Abbildung 2.8.: *Unterscheidung von Fehlersymptom und Fehlerursache [IEE09]*

Sensor, das Fehlersymptom ist als Folge an der Oberfläche zu beobachten. Ein Beispiel ist die Implementierung eines Buttons, dessen Verfügbarkeit von einer Reihe von Werten abhängt, die aus Sensoren ausgelesen werden. Während des Tests wird als Fehlersymptom festgestellt, dass der Button nicht sichtbar ist. Eine mögliche Fehlerursache könnte sein, dass der Button nicht im Programmcode instantiiert wurde. Der Zusammenhang zwischen Fehlersymptomen und Fehlerursachen muss nicht eindeutig sein. Einem Fehlersymptom liegt mindestens eine Ursache zugrunde. Allerdings kann eine Fehlerursache zu mehreren Fehlersymptomen führen; ein Fehlersymptom kann die Folge mehrerer Ursachen sein. Dies kann erneut am Beispiel der Verfügbarkeit eines Buttons verdeutlicht werden. Das Fehlersymptom «Button wird nicht angezeigt» könnte neben der bereits erwähnten Ursache «Button wurde nicht instantiiert» auch durch die Bedingungsauswertung oder durch die Ermittlung der aktuellen Bedingungsbelegung bedingt sein. Der spezifikations-basierte Funktionstest ist demnach darauf beschränkt, Fehlersymptome festzustellen. Die Ermittlung und die Behebung der Ursachen müssen dem Test nachfolgen.

Einige eingebettete Systeme sind häufig als sicherheitskritisch einzustufen, so dass Fehler teilweise schwerwiegende Auswirkungen haben können [Mar08, Rom05]. Um die Qualitätssicherung anzuleiten, wurden daher diverse Standards und Normen entwickelt. Bereits 1987 wurde von der International Organization for Standardization die ISO 9000 Normreihe veröffentlich, um Begriffe und Anforderungen an Qualitätsmanagementsysteme zusammenzufassen [ENI05]. Ziel der Normreihe ist es, die Produktqualität durch kontinuierliche Verbesserung von Prozessen und Strukturen stetig zu steigern. ISO 9001 umfasst die konkreten Anforderungen an ein Qualitätsmanagementsystem und bezieht sich dabei u.a. auf die Verantwortung der Leitung, das Management von Ressourcen sowie die Produktrealisierung. Seit der Veröffentlichung der ISO 9000 Normreihe wurden weitere Standards entwickelt, um auf die spezifischen Produkte oder Produktgruppen einzugehen [ISO08, IEC08, ISO11, CMM10]. Da die Anwendbarkeit der vorliegenden Arbeit nicht auf eine spezifische Domäne festgelegt wird, steht nicht fest, welche Normen berücksichtigt werden müssen oder sollen. Die Erfüllung geltender Normen kann erst bei der Anwendung des Verfahrens geprüft und sichergestellt werden. Die Erfüllung von Standards steht demnach ausdrücklich nicht im Fokus dieser Arbeit.

Der spezifikations-basierte Funktionstest wird nun anhand der Literatur charakterisiert. Dadurch soll verdeutlicht werden, welche Ziele durch das Testverfahren erfüllt werden können. In der Literatur werden Testverfahren daran unterschieden, ob sie ein lauffähiges System voraussetzen oder nicht ausführbare Artefakte überprüfen [Ale12, Boa10]. Wird das Verhalten eines zu testenden Systems, dem so genannten *Prüfling*, während dessen Benutzung bewertet, wird dies als *dynamisches Testverfahren* bezeichnet. Werden im Gegensatz dazu Software Artefakte untersucht, ohne diese auszuführen, spricht man von einem *statischen Testverfahren*. Da bei Funktionstests der Benutzeroberfläche eine durch den Auftragnehmer gelieferte Software durch den Auftraggeber zum Zweck der Überprüfung ausgeführt wird, handelt es sich um eine dynamische Vorgehensweise.

Ein weiteres Unterscheidungsmerkmal ist der Gegenstand der Überprüfung [Ale12, Boa10, Rom05]. Wird ein Prüfling anhand spezifizierter Anforderungen überprüft, deren Korrektheit und Vollständigkeit vorausgesetzt wird, spricht man von *Verifikation*. Im Gegensatz dazu werden Maßnahmen zur Überprüfung, ob die Anforderungen ein System beschreiben, das für den beabsichtigten Gebrauch geeignet ist, als *Validation* bezeichnet. Diese Unterscheidung kann anhand des V-Modells in Abbildung 2.7 nachvollzogen werden. Eine Validation ist die Überprüfung eines Konzepts auf Basis vorangegangener

Abstraktionsebenen. Eine Verifikation gleicht die Implementierung mit den Vorgaben auf gleicher Abstraktionsebene ab. Da beim Funktionstest nicht der Inhalt der Spezifikation selbst sondern die darauf basierende Implementierung untersucht wird, handelt es sich bei dem Verfahren um eine Verifikation.

Des Weiteren werden Testverfahren anhand der adressierten Fehlerursachen unterschieden [Ale12, Boa10, Rom05]. Steht der Nachweis im Vordergrund, dass die Funktionen gemäß der Spezifikationen zur Verfügung stehen, spricht man von einem *Positivtest*. Die Testfälle werden anhand einer funktionalen Spezifikation hergeleitet und ausgewählt. Im Gegensatz dazu prüft ein *Negativtest* den Prüfling auf Korrektheit in den Fällen, in denen das System nicht erwartungskonform bedient wird. Ziel ist die Provokation von Ausnahmebehandlungen und damit die bewusste Eingabe unvollständiger oder falscher Werte. Während bei Positivtests sichergestellt werden soll, dass ein System die geforderte Funktionalität bereitstellt ist das Ziel des Negativtests, lückenhafte Programmierung aufzudecken [Boa10]. Bei dem spezifikations-basierten Funktionstest handelt es sich demnach um einen Positivtest.

Testverfahren für Software unterscheiden sich aufgrund des Einblicks in den Programmcode während des Testens [Ale12, Boa10, Hof08]. Stehen die internen Strukturen oder Komponenten der Programmierung auf dem Prüfstand spricht man von einem *White-Box Test*. Fließt das Wissen über die interne Programmstruktur in die Erzeugung funktionaler Testfälle ein, wird dies als *Gray-Box Test* bezeichnet. Steht bei den funktionalen Tests kein Wissen über die Programmstruktur zur Verfügung oder wird dieses wissentlich ignoriert, spricht man von einem *Black-Box Test*. In dem Kontext dieser Arbeit wird davon ausgegangen, dass der Auftraggeber bei der Durchführung des spezifikations-basierten Funktionstests keinen Einblick in die Struktur der Software hat. Damit ist das Verfahren als Black-Box Test einzuordnen.

Nach der Charakterisierung des spezifikations-basierten Funktionstests anhand der Literatur wird im Folgenden die Einordnung in den Entwicklungsprozess diskutiert. Eine direkte Folge aus dem umrissenen Entwicklungsprozess eingebetteter Systeme sind strenge Zeitvorgaben. Die Entwicklung der Subsysteme kann erst begonnen werden, wenn die Konzeption des Gesamtsystems abgeschlossen ist. Die finale Integration kann erst vorgenommen werden, wenn alle notwendigen Subsysteme vorliegen. Um die Entwicklung des Gesamtsystems bis zum festgesetzten Termin der Serienfertigung beenden zu können, müssen allen Entwicklungsphasen feste Zeitfenster zugewiesen werden.

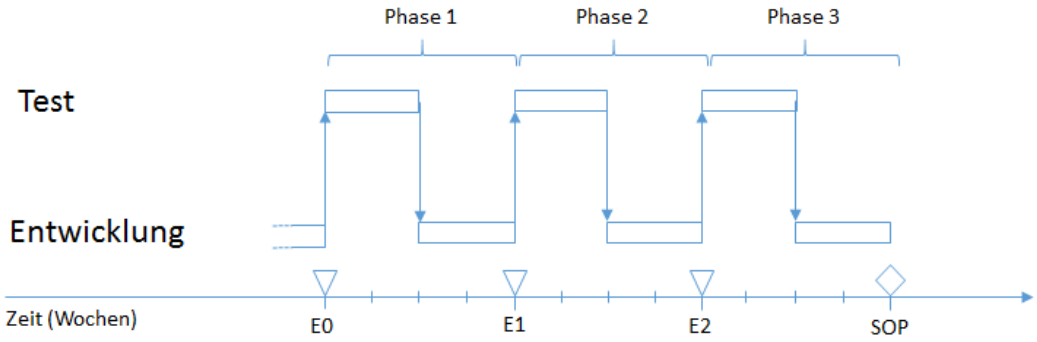


Abbildung 2.9.: *Beispielhafter Zeitplan der Softwareentwicklung eingebetteter Systeme*

Diesen Zeitfenstern sind Ressourcen wie Personal und Rechenkapazität zugewiesen. Abbildung 2.9 illustriert anhand eines Beispiels die Auswirkungen der Zeitvorgaben auf den Testprozess. Es wird angenommen, dass die Testphase drei Monate vor dem Start der Serienproduktion (start of production, SOP) begonnen werden kann. Es wird die Dauer von zwei Wochen für einen vollständigen Testdurchlauf inklusive Dokumentation der Fehler angenommen. Anschließend bleiben den Entwicklern weitere zwei Wochen Zeit, die gefundenen Fehler zu reproduzieren, zu beheben und durch Entwicklertests die Stabilität der Software sicherzustellen. Der Softwarestand zu Beginn der Testphase wird in diesem Beispiel als E-0 bezeichnet. Im darauf folgenden Entwicklungsstand E-1 wurden die im ersten Testlauf gefundenen Fehler behoben. In diesem Szenario sind demnach drei Testdurchläufe möglich. Fehler, die in dieser Zeit nicht gefunden werden können, gehen in das Produkt ein und stellen damit ein Risiko dar. Es wird deutlich, wie entscheidend sich die zeitlichen Vorgaben auf die Produktqualität auswirken. Dies zeigt die Notwendigkeit effizienter Testverfahren.

In diesem Szenario sind auch weitere Aspekte der Projektplanung zu berücksichtigen, die das Zeitfenster für Softwaretests weiter einschränken. In der Praxis ist nicht selten davon auszugehen, dass alle Ressourcen exklusiv für die Entwicklung eines Systems zur Verfügung stehen. So sind beispielsweise die Spezifikateure neben dem Test des vorliegenden Systems bereits mit der Konzeption eines Nachfolgesystems beschäftigt. Auch die Entwickler sind in der Testphase selten ausschließlich für ein System zuständig. In diesen Fällen ist eine Ressourcenplanung erforderlich. Es muss festgesetzt werden, welche Ressource zu welchem Zeitpunkt für die Testphase benötigt wird.

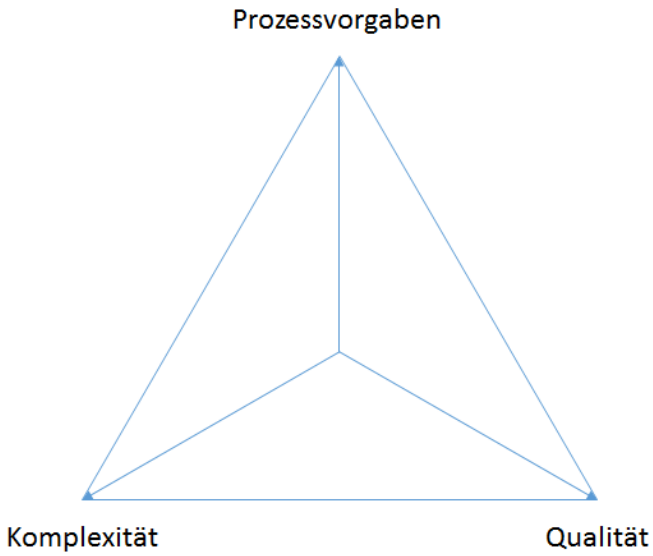


Abbildung 2.10.: *Spannungsfeld dieser Arbeit*

Daraus ergibt sich ein Spannungsfeld aus Qualitätsanspruch, inhärenter Komplexität und der Notwendigkeit fester Zeitplanung (siehe Abbildung 2.10). Als ein realistisches Szenario ist zwangsläufig anzunehmen, dass ein Funktionstest der Oberfläche nicht bis zur Fehlerfreiheit durchgeführt werden kann. Wahrscheinlicher ist, dass durch den, für die Testphase festgesetzten Zeitraum die Anzahl der Testdurchläufe beschränkt wird. Trotzdem ist aufgrund des in Abschnitt 2.1 beschriebenen Nutzungskontextes eingebetteter Systeme ein besonderes Maß an Qualität erforderlich. Dies wird durch die inhärente Komplexität eingebetteter Systeme erschwert, die in absehbarer Zeit durch die in Abschnitt 2.1 vorgestellten Trends voraussichtlich weiter ansteigen wird. Dieses Spannungsfeld macht den Test von Oberflächen eingebetteter Systeme zu einem notwendigen Forschungsfokus. Die Testautomatisierung verspricht eine Lösungfür dieses Spannungsfeldes.

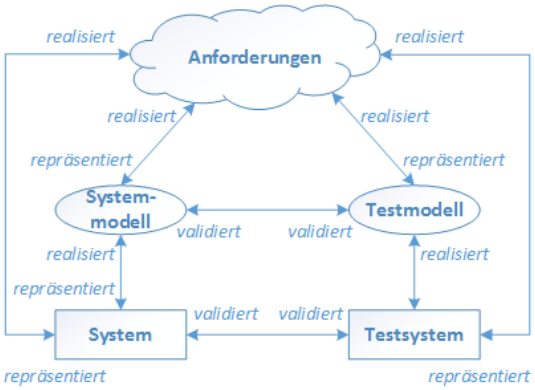
2.2.2. Zusammenfassung

In diesem Abschnitt wurde der Entwicklungsprozess von Bedienoberflächen eingebetteter Systeme beleuchtet. Anhand des V-Modells wurde die Entwicklung des Gesamtsystems durch iterative Unterteilung in Subsysteme verdeutlicht. In dieser Arbeit wird davon ausgegangen, dass die Entwicklung einiger Subsysteme, wie der Benutzeroberfläche,

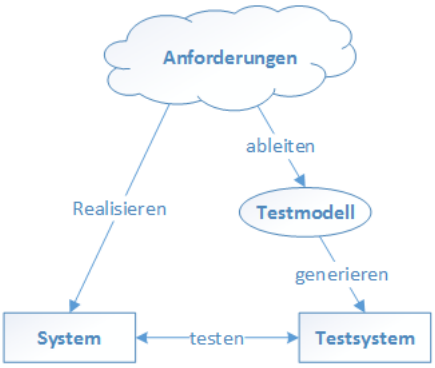
an einen externen Auftragnehmer übergeben wird. Durch einen Auftraggeber werden die fachlichen Anforderungen an das System zusammengestellt; ein Auftragnehmer übernimmt die technische Realisierung. Diese Realisierung beinhaltet eine eventuelle technische Feinspezifikation, die Implementierung durch Programmcode, gegebenenfalls eine Integration von Software und/oder Hardware Modulen und die Durchführung von Entwicklertests. Nach der Realisierung des Systems geht die Verantwortung für das Subsystem wieder an den Auftraggeber über. Bei dieser Übergabe muss sichergestellt werden, dass das Subsystem die gestellten fachlichen Anforderungen erfüllt. Anhand der fachlichen Anforderungen werden Testfälle abgeleitet, um sicherzustellen, dass Funktionen erwartungskonform aufgerufen werden können. Aufgrund der erforderlichen Planung des Gesamtprojektes und der Trennung in Auftraggeber und Auftragnehmer ergeben sich strikte Vorgaben an Dauer und Anzahl möglicher Testläufe. Das Spannungsdreieck aus Komplexität, Prozessvorgaben und Qualitätsanforderungen eingebetteter Systeme stellt eine besondere Herausforderung dar. Daher wurde der spezifikations-basierte Funktionstest als geeigneter Ansatzpunkt für eine Testautomatisierung ermittelt.

2.3. Modellbasierte Testautomatisierung

Im vorangegangenen wurde der Funktionstest der Benutzeroberfläche als geeigneter Ansatzpunkt für eine Testautomatisierung identifiziert. Die Oberfläche eines integrierten Systems wird auf Basis der fachlichen Anforderungen überprüft. Damit ist dieser Ansatz als *modellbasierter Test* zu bewerten [RBGW12, Sch07]. Bei einem modellbasierten Test werden aus einer abstrakten Beschreibung Testfälle abgeleitet, um sie gegen einen Prüfling zur Ausführung zu bringen. Ein Modell ist eine abstrakte Beschreibung der Realität, die auf die für den Einsatz relevanten Informationen reduziert ist. Damit bilden die fachlichen Anforderungen, die zum Zweck der Implementierung und des Testens eines Computersystems erstellt wurden, ein Modell. Im folgenden Abschnitt werden die aktuellen Forschungsarbeiten im Bereich der modellbasierten Testautomatisierung graphischer Benutzeroberflächen vorgestellt. Zu Beginn wird das Grundprinzip der modellbasierten Vorgehensweise skizziert und aufgezeigt, welche Voraussetzungen bei deren Anwendung erfüllt sein müssen. Anschließend werden die aktuellen Ansätze zur Automatisierung zusammengefasst und anhand der in den Abschnitten 2.1 und 2.2



(a) *Ansatzpunkte der modellbasierten Testautomatisierung*



(b) *Modellbasierter Ansatz dieser Arbeit*

Abbildung 2.11.: *Vorgehensmodelle der modellbasierten Testautomatisierung [Sch07]*

festgesetzten Vorgaben, die sich aus dem gewählten Entwicklungsprozess ergeben, diskutiert.

Abbildung 2.11a zeigt diverse Ansatzpunkte für modellbasierte Testverfahren [Sch07]. Die Anforderungen bilden die Basis sowohl der Tests als auch der Implementierung des Systems. Die Darstellung als Wolke symbolisiert, dass die Anforderungen nicht in einem spezifischen Format vorliegen. Basierend auf den Anforderungen können ein Systemmodell und ein Testmodell abgeleitet werden, um die Inhalte der Anforderungen aus den Perspektiven Implementierung und Test zu repräsentieren. Eine gegenseitige Validierung kann sicherstellen, dass beide Modelle inhaltlich deckungsgleich sind. Anhand des Systemmodells wird anschließend das System umgesetzt, das Testmodell stellt die Basis des Testsystems dar. Wiederum kann eine Validierung durchgeführt werden, um sicherzustellen, dass System und Testsystem übereinstimmen. Die Doppelpfeile zwischen Modellen und Systemen zeigen, dass die Validierung in beide Richtungen durchgeführt wird. Eventuelle Abweichungen zeigen nur an, dass sich deren Realisierung unterscheidet und nicht, welche der Artefakte fehlerhaft sind. Erst durch eine anschließende Untersuchung kann die Ursache ermittelt werden.

Da die Erstellung und Validierung allerdings aufwendig sind und die Redundanz in der Praxis schwer zu vermitteln ist, werden durch [RBGW12] sechs Alternativen abgeleitet, indem System und Testmodell jeweils weggelassen, zusammengefasst oder voneinander abgleitet werden. Beispielsweise kann das Systemmodell sowohl für die Erstellung

des Systems als auch zur Generierung der Testfälle verwendet werden. In einer anderen Variante wird, ausgehend von einem Testmodell, ein Systemmodell abgeleitet. Die Anforderungen als Basis sowie die gegenseitige Validierung von Testsystem und System bleiben bestehen. Ausgehend von diesen sechs Alternativen zeigt Abbildung 2.11b das Verfahren, dass dieser Arbeit zugrunde gelegt wird. Unabhängig von dem Testprozess wird das System basierend auf den Anforderungen umgesetzt. Parallel dazu wird ein Testmodell abgeleitet, mit dessen Hilfe anschließend das Testsystem generiert werden kann. Die Auswahl dieser Variante hat zwei Gründe. Erstens ist bislang nicht davon auszugehen, dass die Spezifikation in einem ausreichend formalen Format erstellt wird, so dass eine direkte Ableitung der Testfälle ermöglicht. Dies liegt daran, dass die Konzeption einer Benutzeroberfläche stark iterativ durchgeführt wird. In der Regel werden viele Alternativen evaluiert und bis zuletzt an dem Gesamtkonzept gefeilt. Von Beginn an jede Alternative formal zu dokumentieren und anschließend zu verwerfen ist ein nicht praktikabler Aufwand. Der zweite Grund für ein dediziertes Testmodell ist, dass für die Ableitung von Testfällen Informationen erforderlich sind, die für die Implementierung nicht oder nur in einer stark abstrahierten Form notwendig sind. Dies sind beispielsweise konkrete Informationen über die erreichbaren Radiosender oder verfügbaren Medien. Für die Realisierung ist lediglich notwendig, entsprechende Schnittstellen und Datenformate vorzusehen. Um einen Testfall zu erstellen, der überprüft, ob in einem Testszenario alle erforderlichen Einträge zugänglich sind, müssen diese Informationen vorliegen. Bei der Erstellung des Testmodells werden alle testrelevanten Informationen der Spezifikation in ein formales Format zusammengefasst und um die spezifischen Informationen des Testszenarios ergänzt. Ein Generator leitet aus dem Modell Testfälle ab.

Abbildung 2.12 zeigt die Schritte der modellbasierten Testautomatisierung im Kontext dieser Arbeit. Die fachlichen Anforderungen werden in Form einer sogenannten *HMI Spezifikation* dokumentiert. Das zu testende System wird in dieser Arbeit als *Prüfling* oder *System Under Test* (SUT) bezeichnet. Der Pfeil von der Spezifikation zum Prüfling symbolisiert die Realisierung des Computersystems durch den Auftragnehmer auf Basis der Spezifikation und unabhängig vom Testverfahren. Die gestrichelten Rechtecke zeigen die Schritte, die von jedem Testverfahren explizit oder implizit durchgeführt werden müssen (nach [Spi06]). Ausgehend von der Spezifikation wird abgeleitet, auf welche Weise der Prüfling während des Tests stimuliert werden muss und wie dessen Reaktion überprüft werden kann. Diese Bedienaktionen und Verifikationsinformationen werden

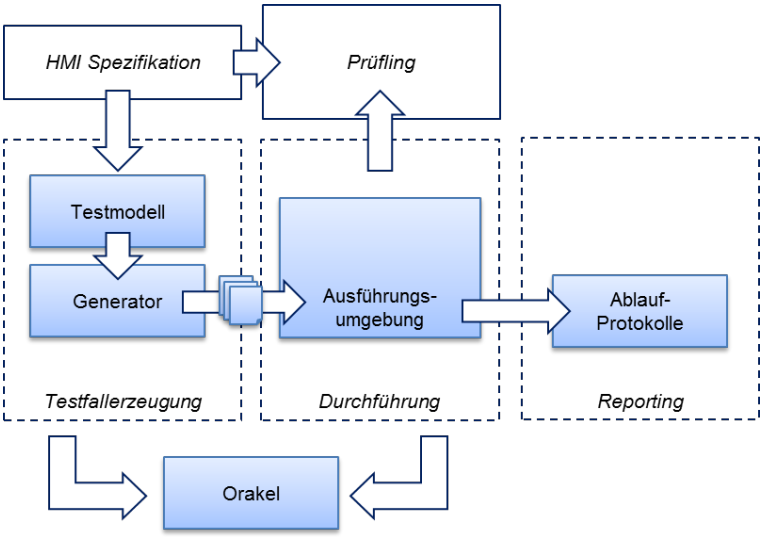


Abbildung 2.12.: *Vorgehensweise der Testautomatisierung in dieser Arbeit*

entweder in Form von Testfällen festgehalten oder unmittelbar gegen den Prüfling ausgeführt. Das Ergebnis der Durchführung wird schließlich in Form von Berichten dokumentiert.

Die ausgefüllten Rechtecke geben an, wie die Testschritte durch die modellbasierte Testautomatisierung realisiert werden. Im Zuge der Testfallerzeugung muss ein Modell erstellt und durch einen Generator ausgewertet werden. Die Durchführung der Testfälle gegen den Prüfling erfolgt durch eine Software, die im Folgenden als *Ausführungsumgebung* oder *Testharnisch* bezeichnet wird [RBGW12]. Die Ergebnisse der Testfallduchführung werden durch Ablaufprotokolle dokumentiert. Im folgenden Abschnitt werden die Teilschritte im Detail beschrieben und jeweils publizierte Lösungsansätze vorgestellt. Da die Einbindung der Testautomatisierung in den Entwicklungsprozess nicht anhand einzelner Teillösungen diskutiert werden kann, werden anschließend die in der Literatur vorgeschlagenen MBT-Rahmenwerke vorgestellt und deren Eignung im Kontext dieser Arbeit erörtert.

2.3.1. Testfallerzeugung

Die modellbasierte Testautomatisierung (MBT) beginnt mit der Modellierung der testrelevanten Informationen und der automatischen Ableitung der Testfälle. Die Testfälle bein-

halten alle notwendigen Stimuli, um die für den Test erforderlichen Inhalte aufzurufen. Einen umfangreichen Überblick zur modellbasierten Testautomatisierung graphischer Benutzeroberflächen bietet [BNGM13]. Bei der Literaturrecherche wird deutlich, dass die meisten Arbeiten Bedienkonzepte wie die Maussteuerung bei Desktop-Applikationen oder Touch-Bedienung mobiler Geräte adressieren. Nur wenige Arbeiten thematisieren den Test direktionaler Oberflächen [Dua12, DHNH11, AS09]. [Dua12] beispielsweise untersucht ebenfalls die Bedienoberflächen von IVI Systemen, setzt dabei allerdings voraus, dass die Spezifikation bereits ausreichend formal beschrieben wurde, um als Basis für eine Generierung zu dienen.

Viele Ansätze verwenden ereignis- und zustands-basierte Modellierung. Für eine Diskussion der beiden Modellarten wird auf [ZH00] verwiesen. Bei der ereignis-basierten Modellierung werden die Benutzeraktionen miteinander verknüpft, die der Benutzer nacheinander durchführen kann. Das Ergebnis ist ein Netz aller möglichen Bedienaktionen. Die Testfälle entsprechen den Pfaden durch den Graphen. Beispiele für die Ereignis-basierte Modellierung sind die Event Flow Graphen (EFG) [Mem01] und die zeitgleich veröffentlichten Event Sequence Charts [Bel01]. Bei der zustandsbasierten Modellierung liegt der Fokus auf den Auswirkungen der Benutzerinteraktion auf den Zustand des Gesamtsystems. Bereits in frühen Ansätzen wurde deutlich, dass die inhärente Komplexität zu einer großen Anzahl an Zuständen führt [SS97]. Daher wurde in späteren Arbeiten versucht, der Zustandsexplosion entgegenzuwirken, indem der Zustandsraum anhand von Aufgaben in mehrere Zustandsmaschinen untergliedert wird [WA00, WAA01, PTFV05]. Analog der ereignis-basierten Modellierung entsprechen die Aufgaben den Bedienaktionen, die der Benutzer in dem aktuellen Zustand des Systems durchführen kann. Eine ähnliche Entwicklung ist bei der Verwendung von Petri Netzen zu beobachten. Während in frühen Arbeiten noch flache Netze verwendet werden [PB95], wird später eine hierarchische Gliederung vorgeschlagen [REG07].

Auch die Diagramme der *Unified Modeling Language* (UML) wurden zur Testautomatisierung verwendet. [PFV07] verwenden Use-Case- und Aktiviäts-Diagramme zur Modellierung der Bedienaktionen. Die Struktur der Oberfläche wird mit Hilfe von Klassen- und Zustands-Diagrammen beschrieben. Später experimentiert die Gruppe mit der Modellierung mit Hilfe von Concur Task Trees [SCP08, Pat02, PMM97]. Da die Unified Modeling Language (UML) nicht für die Ableitung von Testfällen konzipiert wurde, schlagen [BDG+04] mit dem UML 2.0 Testing Profile (U2TP) eine Erweiterung vor. U2TP

erlaubt die explizite Modellierung testrelevanter Informationen wie Testdaten und Zeitvorgaben und erleichtert die Anbindung an die Programmiersprache *Testing and Test Control Notation* (TTCN-3) [ZDSD05]. [BSBV13] verwendet U2TP beispielsweise für Tests von Internetdiensten. [KG96] verwenden User Behavior Models zur Generierung von Testfällen, um die Bedienvorgänge unterschiedlich erfahrener Benutzer zu simulieren. Basierend auf der Bedienung eines Experten werden zusätzliche Bedienschritte hinzugefügt oder andere Pfade gewählt, um die Bedienung eines Anfängers abzuleiten. [SBK05] schlagen die Entwicklung eines Prototypen vor, der sowohl die Konzepte der Oberfläche erlebbar macht, als auch die Ableitung von Testfällen ermöglichen soll.

2.3.2. Durchführung

Bei der modellbasierten Vorgehensweise liegen explizit dokumentierte Testfälle vor, die anschließend sowohl manuell als auch automatisch durchgeführt werden können [RBGW12]. Unter der Annahme, dass die automatische Durchführung weniger Aufwand als die manuelle verursacht, ist laut [RBGW12] die Automatisierung vorzuziehen, um manuelle Testressourcen für die nicht automatisierbaren Anteile freizulegen. Eine automatische Ausführungsumgebung ist in drei Software Artefakte aufgeteilt (siehe Abbildung 2.13). Zuerst werden eingehende Testfälle von einem *Testrahmen* interpretiert und anschließend gemäß der, im Testfall hinterlegten Anweisungen mit Hilfe eines *Testtreibers* gegen den Prüfling zur Ausführung gebracht. Die Ausgabe des Systemzustands gleicht der Testrahmen anschließend mit den Vorgaben einer Referenz, dem so genannten *Orakel*, ab und gibt die Ergebnisse in Form von Reports aus. Bei der manuellen Durchführung obliegt es dem Tester, die Systemreaktion zu bewerten und gegebenenfalls als fehlerhaft einzuordnen. Bei der Testautomatisierung ist dafür eine Komponente vorgesehen, die als Orakel bezeichnet wird [HMSY13, XM07, MSP01]. Die notwendigen Referenzinformationen werden entweder in die Testfälle eingebettet oder separat erzeugt.

Für die Automatisierung von Oberflächentests werden diverse Testrahmen bereits als Produkte angeboten. Die Testfälle werden unabhängig von der Technologie des Prüflings erstellt. Eine gängige Form, bei diesen Produkten Testfälle zu erstellen ist der Capture&Replay Mechanismus [Fro15, Ran15, Ale12]. Dabei zeichnet der Testrahmen die manuelle Bedienung des Prüflings auf. Mit Hilfe von Verifikationspunkten kann der

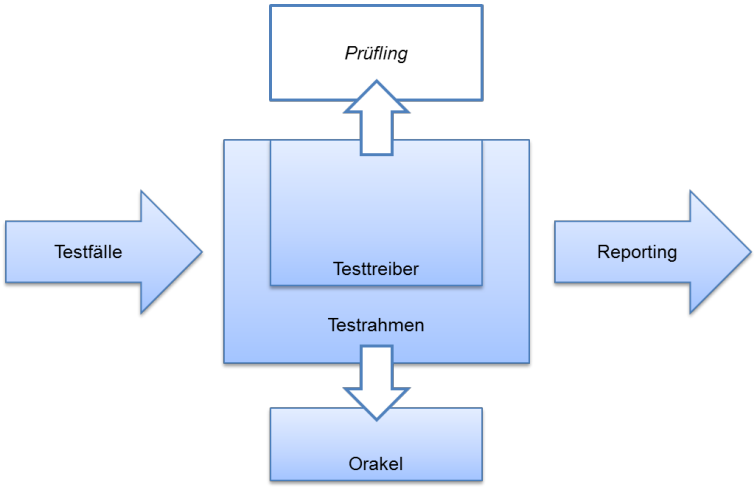


Abbildung 2.13.: *Aufteilung der Ausführungsumgebung in Testrahmen und Testtreiber*

Testingenieur zu jeder Zeit eine Überprüfung vorsehen und dabei entweder auf die aktuell angezeigten Werte des Prüflings verweisen oder Referenzwerte aus einer anderen Quelle beziehen. Zusätzlich können die Testfälle durch ein Datenformat, wie Code oder Skripte, importiert werden. Auf diese Weise könnten auch modellbasiert generierte Testfälle zur Ausführung gebracht werden.

Für die Kommunikation mit dem Prüfling müssen zwei Aspekte in Betracht gezogen werden: die Stimulation und die Verifikation. Eine bewährte Vorgehensweise ist die Implementierung einer spezifischen Schnittstelle, einer so genannten *Designed for Testability* (Designed for Testability (DFT)) Schnittstelle, um die Kommunikation in beide Richtungen zu ermöglichen [Fro15, Ran15, AO08, Bin94]. Durch vordefinierte Kommandos kann der Prüfling stimuliert und dessen aktueller Zustand auf Basis von Daten ausgelesen werden. Auf eine spezifische Code Schnittstelle setzen unter anderem die Produkte Squish [Fro15], MonkeyTalk [Clo15] oder Ranorex [Ran15]. Durch die Trennung in Testrahmen und Testtreiber ist es diesen Programmen möglich, mehrere Technologien wie Qt, .NET oder Java zu bedienen. Andere Produkte wie eggPlant [Tes15], Sikuli [Sik15] und RoutineBot [Rou15] werben durch Technologieunabhängigkeit, indem sie Bildverarbeitung bei der Verifikation des Prüflings verwenden.

Durch den allgemeinen Verwendungszweck ist die Konfigurierbarkeit dieser Produkte stark eingegrenzt. Dies zeigt sich bei der Differenzierung der Fehlersymptome. So sind die Produkte darauf beschränkt, eine Abweichung zwischen Referenzdaten und Prüfling

zu identifizieren, ohne das spezifische Fehlersymptom einzugrenzen. Die Abweichungen müssen anschließend manuell bearbeitet werden, um sprechende Fehlerberichte zu erstellen. Zudem sind die Möglichkeiten beschränkt, im Fall einer Abweichung zu reagieren. Aufgrund fehlender Informationen bleibt lediglich, den Test abzubrechen oder wie vorgesehen weiter durchzuführen. Gerade die Robustheit gegenüber Fehlersymptomen ist das zentrale Anliegen dieser Arbeit. Daher muss von der Verwendung von bestehenden Testrahmen abgesehen werden.

2.3.3. Reporting

Die Ergebnisse der Testfälle werden in Form von Ablaufprotokollen dokumentiert. Bei der Durchführung sollte darauf geachtet werden, dass bei Abweichungen Rückschlüsse auf die betroffenen Modellteile möglich sind, um etwaige Fehler effizient finden und beheben zu können [RBGW12]. Zudem kann durch eine große Anzahl generierter Testfälle eine nachgelagerte manuelle Analyse der Fehlerberichte erschwert werden. [RBGW12] schlägt zwei Strategien vor, um die manuelle Analyse der Ablaufprotokolle zu unterstützen. Erstens kann die automatisch erzeugte Ablaufdokumentation mit Hilfe einer Ähnlichkeitsbestimmung zusammengefasst und somit deren Umfang reduziert werden. Zweitens rät [RBGW12] dazu, die Anzahl der erzeugten Testfälle auf Redundanzen zu prüfen, um die wiederholte Prüfung der Anwendung ohne zusätzlichen Erkenntnisgewinn zu vermeiden.

2.3.4. Bestehende Rahmenwerke

In den vorangegangenen Abschnitten wurde beschrieben, welches Entwicklungsszenario dieser Arbeit zugrunde liegt, aus welchen Teilschritten die modellbasierte Testautomatisierung besteht und welche Teillösungen in der Literatur angeboten werden. Allerdings kann die Einordnung in den Entwicklungsprozess nur dann beurteilt werden, wenn die Testautomatisierung ganzheitlich betrachtet wird. Daher wurden in einigen Forschungsgruppen Teillösungen zu *Rahmenwerken* zusammengefasst. Im Folgenden werden die wichtigsten Rahmenwerke vorgestellt und deren Anwendbarkeit auf das vorgestellte Entwicklungsszenario diskutiert.

Ein seit 1999 aktiver Forscher im Bereich der modellbasierten Testautomatisierung graphischer Benutzeroberflächen ist Atif Memon [z.B. AFT+14, MPM13, Mem03, Mem01, MPS99]. Die Beiträge seiner Gruppe werden seit 2007 zum Testframework GUITAR[1] zusammengefasst [NRBM13, Mem07, XM07], das aus seinem Projekt DART (Daily Automated Regression Tester) hervorgegangen ist [MBHN03]. Bisher steht der Test von Java Anwendungen im Fokus. In aktuellen Arbeiten werden die Ergebnisse auf Anwendungen mobiler Geräte übertragen [AFT+14]. Abbildung 2.14 zeigt die Bestandteile des Rahmenwerks. Rechtecke symbolisieren Artefakte, die während des Prozesses entstehen. Die entwickelten Werkzeuge werden als Ovale dargestellt. Zu Beginn des Prozesses wird eine vorliegende Java Anwendung mit Hilfe des *GUI Ripper* analysiert [MBN03]. Durch Reverse Engineering wird aus einer lauffähigen Anwendung automatisch ein *GUI Tree* erstellt, der den Aufbau der Oberfläche beschreibt. Als Testmodell werden im GUITAR Framework sogenannte *Event Flow Graphen* (EFG) verwendet [MSP01].

Abbildung 2.15 zeigt anhand eines Beispiels den Aufbau der Event-Flow Graphen. Jedes Element, mit dem der Benutzer interagieren kann, wird in dem Graphen durch einen Knoten repräsentiert. Die Kanten zeigen, welche Interaktionen aufeinander folgen können. Der Testfallgenerator ist eine anwendungsspezifische Implementierung basierend auf den Funktionen des GUITAR Frameworks. In diesem Beispiel traversiert der Generator den Graphen und bis zu einer festgelegten Anzahl an Testschritten. Eventuell notwendige Texteingaben müssen von Testern in einer Konfigurationsdatei vorgegeben werden. Auch der Startpunkt wird manuell gesetzt und kann entscheidend dafür sein, ob alle Bereiche der Software erreicht werden können. Als Beispiel wird angegeben, dass in einem Texteditor die Kopierfunktion nur dann getestet werden kann, wenn das Programm nicht mit einem leeren Textdokument gestartet wurde.

Der *Replayer* führt die generierten Testfälle gegen den Prüfling aus und greift dabei auf die Verbindung aus EFG und GUI Tree zurück, um die Elemente auf dem Bildschirm zu identifizieren, die aktiviert werden müssen. Ein Orakel muss bei der Anwendung von GUITAR selbst implementiert werden. Eine Basisfunktionalität wie die Entdeckung eines Programmabsturzes, das Werfen von Java Exceptions oder ein allgemeiner Mechanismus zum Abgleich von Daten werden zur Verfügung gestellt. Die Ergebnisse des Tests werden in einem Ablaufprotokoll dokumentiert, das mit Hilfe von Skripten analysiert wird.

[1] http://sourceforge.net/projects/guitar/, zuletzt aufgerufen am 24.07.2015

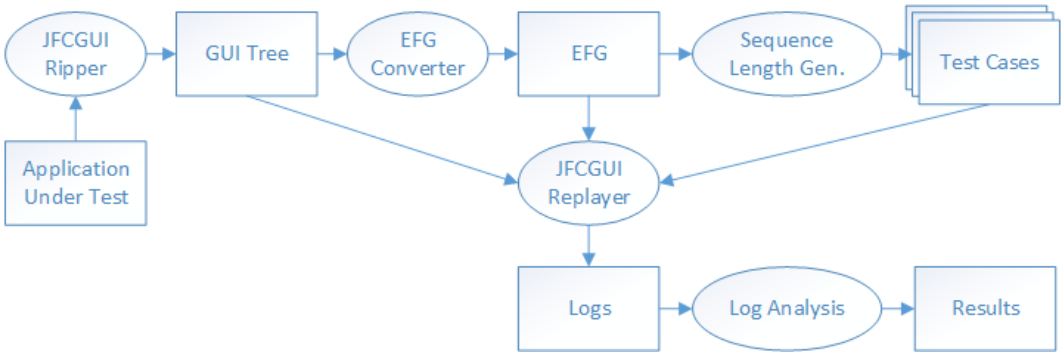
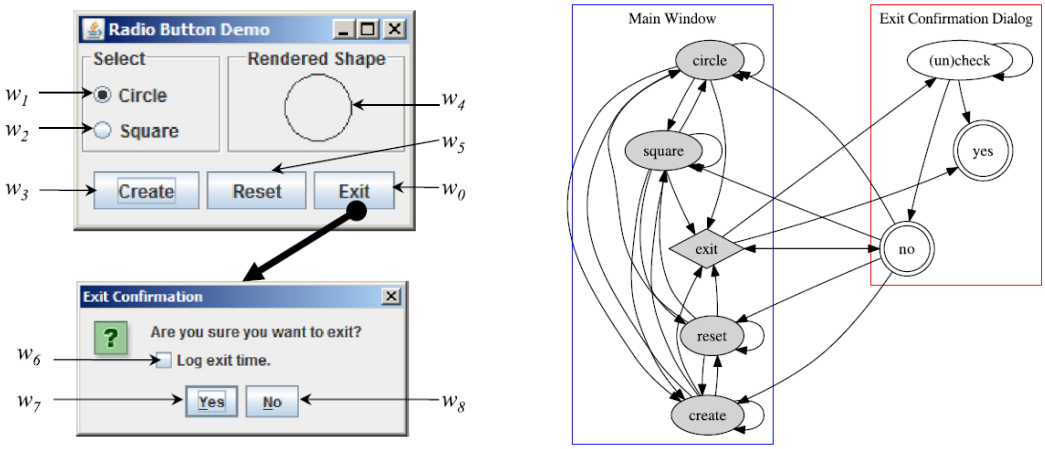


Abbildung 2.14.: *Komponenten des GUITAR Rahmenwerks für Java Foundation Classes [NRBM13]*



(a) *Anwendungsbeispiel* **(b)** *Event-Flow-Graph*

Abbildung 2.15.: *Beispielhafte Anwendung des GUITAR Rahmenwerks [NRBM13]*

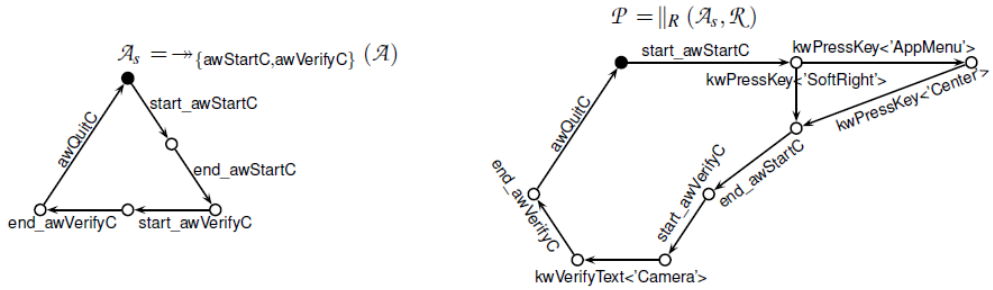
Um alle Werkzeuge des GUITAR Rahmenwerks nutzen zu können, ist eine ausführbare Datei als Prototyp notwendig. Dies ist im Kontext dieser Arbeit nicht möglich, da ein bereits voll integriertes System geprüft werden soll. Aber auch die einzelnen Komponenten erfüllen nicht die Anforderungen des zugrundeliegenden Entwicklungskontextes (siehe 2.2). Die Möglichkeit, die Testmodelle automatisch durch Reverse Engineering herzuleiten, entfällt. Eine manuelle Erstellung eines EFG ist nicht praktikabel, wie bereits an dem einfachen Beispiel aus Abbildung 2.15 zu erkennen ist. Jedes interaktive Element explizit zu erstellen und mit anderen in Verbindung zu setzen ist aufwendig und fehleranfällig. In Anbetracht der Tatsache, dass Änderungen der Spezifikation überführt werden müssen, stellt den Entwickler des Testmodells vor erhebliche Herausforderungen.

Auch das verwendete Testmodell ist für die Anwendung auf die Oberflächen eingebetteter Systeme ungeeignet. So ist in den Event-Flow-Graphen keine Behandlung von Bedingungen vorgesehen. Dies zeigt sich in der Anforderung, dass der Startpunkt des Tests bewusst durch Tester ausgewählt werden muss, um zu gewährleisten, dass alle erforderlichen Bereiche der Oberfläche berücksichtigt werden. Die Verwaltung der Bedingungen stellt eine der zentralen Herausforderungen der Oberflächen eingebetteter Systeme dar und sollte im Zuge einer Testautomatisierung von dem Testverfahren unterstützt werden.

Eine Strategie, wie während des Testens konstruktiv auf Fehler reagiert werden kann, fehlt gänzlich. Dies kann dadurch bedingt sein, dass GUITAR bislang in erster Linie auf produktive Programme angewendet wurde, um zu demonstrieren, dass durch das modellbasierte Verfahren auch Fehler gefunden werden, die durch den herkömmlichen Test unerkannt blieben. Dementsprechend gering ist die Anzahl Fehlerberichte. Bei dem Einsatz bei Prüflingen, die nicht im Vorfeld getestet wurden, ist eine größere Anzahl an Fehlern zu erwarten.

TEMA[2] ist ein Testframework, das aus verschiedenen wissenschaftlichen Arbeiten, auch in Zusammenarbeit mit Nokia, zur Testautomatisierung mobiler Anwendungen entstanden ist [JTK11, Jää11, KMPK05]. Das Testmodell basiert auf Labeled Transition Systemen und besteht aus zwei Komponenten (siehe Abbildung 2.16). In der *Action Machine* werden abstrakte Aktionen modelliert, die für die Bedienung der Anwendung erforderlich sind. Ausgehend von einem definierten Startpunkt zeigen eine Reihe von Transitionen welche Bedienschritte aufeinander folgen sollen. Die Transitionen sind mit

[2] http://tema.cs.tut.fi, zuletzt aufgerufen am 06.04.2015



(a) *Action machine* **(b)** *Refinement machine*

Abbildung 2.16.: *Beispielhafte Anwendung des TEMA Testframeworks [KMPK05]*

Aktionswörtern wie «awStartC», «awVerifyC» und «awQuitC» beschriftet. Diese Aktionswörter dienen zur Aktivierung von Transitionen der *Refinement Machine*. Hier werden die Aktionswörter in eine Reihe konkreter Interaktionen, den so genannten Schlüsselwörtern (Keywords) aufgeschlüsselt, die von der Ausführungsumgebung interpretiert werden können. In der abgebildeten Refinement Machine führt das Aktionswort awStartC dazu, dass die Taste «SoftRight» und zusätzlich aufeinanderfolgend die Tasten «AppMenu» und «Center» gedrückt werden. Um zusätzliche Nebenläufigkeit zu ermöglichen können Aktionsworte als Ganzes oder in den Bestandteilen «start» und «end» verwendet werden. Die Gültigkeit der Schlüsselwörter kann in der Refinement Machine dadurch eingeschränkt werden, dass die letzte Transition eines Aktionswortes nicht zu dem allgemeinen Startpunkt zurückgeführt wird.

Der Fokus von TEMA liegt stark auf der Nebenläufigkeit von Bedienaktionen. Wenn also für die Interaktionen mit dem System mehrere Aktionen gleichzeitig durchgeführt werden müssen. Diese Nebenläufigkeit ist in dem Szenario der vorliegenden Arbeit nicht erforderlich. Es ist notwendig, die sequentielle Reihung direktionaler Bedienaktionen zu bestimmen. Die Anwendung von TEMA auf direktionale Bedienkonzepte führt zu einer Anzahl von Aktionen und entsprechenden Schlüsselwörtern, die von einem Entwickler nur schwer verwaltet werden kann. Während die Kapselung von Aktionen bei Bedienkonzepten mit parallelen Interaktionen zu einer Verringerung der Komplexität führt, hat es bei rein sequentiell direktionalen Bedienkonzepten den gegenteiligen Effekt. Zudem fehlt auch bei TEMA die Modellierung der Bedingungsabhängigkeit der Oberfläche. Wie bei GUITAR wird die Reaktion auf Fehler nicht behandelt. Bei TEMA könnte dies durch den Fokus auf einzelne Applikationen mobiler Geräte begründet sein. Im Vergleich zu

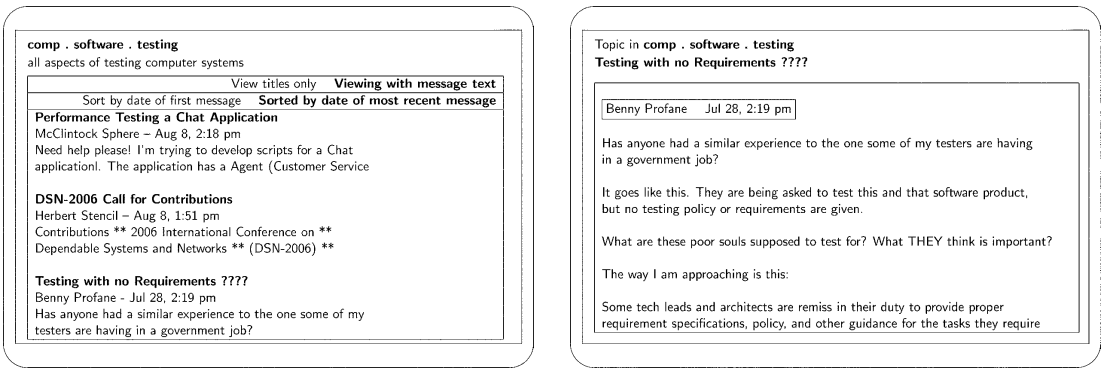
den Oberflächen von IVI Systemen ist der angebotene Funktionsumfang und dadurch die Anzahl der möglichen Fehler geringer.

Mit dem Testframework NModel[3] stellt Microsoft Research Werkzeuge zur Modellbasierten Testautomatisierung bereit. Die Ergebnisse der Arbeit wurden in einem Buch zusammengefasst [JVCS08]. Neben der Beschreibung der Vorgehensweise wird das Verfahren exemplarisch auf die Bedienoberfläche angewendet, die zur Überwachung von Sensoren dient. Damit ist die Auswahl der Prüflinge ähnlich zu denen, die in dieser Arbeit untersucht werden. Darüber hinaus wurde NModel auch auf die Bedienoberfläche eines mobilen Gerätes angewendet [CLS+09]. Im Folgenden wird die Funktionsweise von NModel anhand des Beispiels der Bedienoberfläche eines Diskussionsforums beschrieben (siehe Abbildung 2.17). Die Oberfläche bietet eine Übersicht aller verfügbaren Themen (Abbildung 2.17a), deren Detailgrad und Sortierung durch den Benutzer geändert werden können. Abbildung 2.17b zeigt die Detailansicht der Beiträge zu einem Thema.

Die bei NModel verwendeten Modelle werden als *Model Program* bezeichnet, das eine Finite State Machine (FSM) mit Hilfe der Programmiersprache C# beschreibt. Durch die Werkzeuge «Model Program Viewer» und «Model Program to Dot» kann das Modell visualisiert und analysiert werden. Abbildung 2.18 zeigt die graphische Darstellung FSM des Model Program. Als Startzustand ist die Anzeige der Topics inklusive Text mit der Sortierung ByMostRecent angegeben. Von hier aus kann der Benutzer den Detailgrad der Nachrichten reduzieren (Transition `ShowTitles()`) oder ein Thema zur Detailansicht auswählen (Transition `selectMessages()`). Mit Hilfe von NModel kann die Bedienung des Systems simuliert werden, um Abläufe nachvollziehen und fehlerhafte Modellierung zu erkennen. Zur Ableitung der Testfälle wird das Modell durch das Werkzeug *Offline Test Generator* traversiert. Die Generierung wird abgebrochen, sobald alle Zustände und Transitionen des Modells abgedeckt sind. Die Testfälle werden mit Hilfe des enthaltenen Testrahmens *Conformance Tester* ausgeführt.

Die Modellierung durch eine Kombination aus Programmcode und Zustandsmodellierung ist vielversprechend. Auf diese Weise ist die flexible Modellierung dynamischer und reaktiver Systeme möglich. Dies liegt insbesondere an der Berücksichtigung externer und interner Bedingungen. Allerdings erweist sich die Modellierung als unübersichtlich. Auch die Bereitstellung der Werkzeuge zur Visualisierung ist unzureichend. Wie bereits

[3] https://nmodel.codeplex.com/, zuletzt aufgerufen am 24.07.2015



(a) *Übersicht aller verfügbaren Themen*

(b) *Detailansicht der Beiträge zu einem Thema*

Abbildung 2.17.: *Bedienoberfläche eines Diskussionsforums zur exemplarischen Anwendung des NModel Frameworks [JVCS08]*

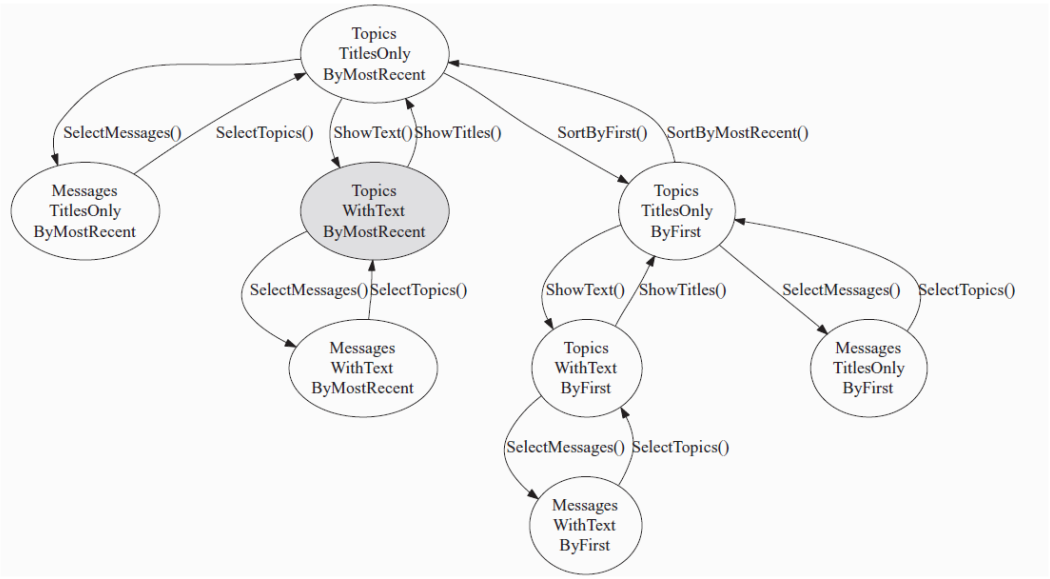


Abbildung 2.18.: *Beispiel einer visualisierten Finite State Machine (FSM) aus [JVCS08]*

in dem begrenzten Beispiel ersichtlich wird, sind die so erstellten Graphen aufgrund ihres Detailgrades für Oberflächen mit großem Funktionsumfang ungeeignet.

Auch die Mechanismen zur Testfallgenerierung ist für die Anwendung im Kontext dieser Arbeit ungenügend. Die schlichte Traversierung von Modellen skaliert unzureichend. Dies zeigt sich sobald Bereiche der Oberfläche erst dann erreichbar sind, nachdem eine Einstellungsänderung vorgenommen wurde. Der Algorithmus ist nicht in der Lage, die Abhängigkeit zu erkennen und die Einstellungsänderung in den Testfall aufzunehmen. Testfälle für eingeschränkt zugänglichen Bereiche der Oberfläche werden erst erreicht, nachdem zufällig bereits die notwendige Einstellung traversiert wurde. Damit führt das Abbruchkriterium der Zustandsüberdeckung zu einer großen Anzahl an Testfällen die ohne Erkenntnisgewinn verworfen werden müssen. Um die Anforderungen des Szenarios dieser Arbeit gerecht zu werden, ist das Abdeckungskriterium nicht ausreichend. Zudem auch bei diesem Framework ist die Reaktion von Fehlern nicht berücksichtigt worden.

2.3.5. Zusammenfassung & Diskussion

In diesem Abschnitt wurden die Absätze der modellbasierten Automatisierung spezifikations-basierter Funktionstests vorgestellt und deren Anwendbarkeit im Kontext dieser Arbeit abgewogen. Ziel der MBT ist es, die Testschritte der Testfallerzeugung, der Durchführung und der Berichterstellung möglichst unabhängig von Menschen auszuführen. Ausgangspunkt ist eine Beschreibung des Soll-Zustands, aus der ein Generator ableiten kann, wie das das zu testende System, der so genannte Prüfling, überprüft werden kann und welche Bedienschritte dafür notwendig sind. In vorangegangenen Abschnitt wurden aktuelle Forschungsansätze vorgestellt. Viele dieser entwickelten Ansätze beschränken sich allerdings auf einzelne Aspekte der Testautomatisierung. Die meisten Arbeiten untersuchen die Modellierung testrelevanter Inhalte und führen lediglich eine exemplarische Generierung durch. Es wurde deutlich, dass die Einsatztauglichkeit eines Ansatzes zur MBT in dem zugrundeliegenden Entwicklungsszenario nur dann ermittelt werden kann, wenn die einzelnen Schritte zu einem Rahmenwerk verbunden und damit abgestimmte Konzepte für Modellerstellung, Generierung, Ausführung und Auswertung von Tests angeboten werden.

Bei der Recherche der in der Literatur vorgestellten Rahmenwerke zeigte sich, dass diese für die Automatisierung der Tests von direktionalen Oberflächen eingebetteter

Systeme nicht geeignet sind. Auch bei den Rahmenwerken wurde die Einbettung des Testverfahrens in den Entwicklungsprozess und die daraus entstehenden Vorgaben nicht ausreichend berücksichtigt. Dies gilt vor allem bei der Entwicklung der Bedienoberflächen eingebetteter Systeme. In dem beschriebenen Entwicklungsszenario ist es notwendig, die Testfälle aus den manuell erstellten Inhalten der Spezifikation abzuleiten, um die Korrektheit einer vorliegenden Oberfläche überprüfen zu können. Damit sind die Rahmenwerke untauglich, die eine Ableitung des Testmodells aus einer bestehenden Software vorsehen. Allerdings sind auch die Ansätze, die von einer manuellen Erstellung des Testmodells ausgehen, für die Anwendung in dem bestehenden Szenario ungeeignet.

Das folgenreichste Versäumnis bisheriger Arbeiten ist die mangelnde Untersuchung der Vorgehensweise im Falle eines Fehlers. Die Ansätze beschäftigen sich in der Hauptsache mit der Erstellung der Modelle, der Ableitung von Testfällen und nur rudimentär mit deren Ausführung. Dies liegt darin begründet, dass diese Ansätze Bedienkonzepte voraussetzen, die weniger anfällig für die Auswirkungen von Fehlern sind. Bei Bedienkonzepten mit Touch oder Mausbedienung kann direkt mit Widgets auf der Oberfläche interagiert werden. Bei direktionalen Oberflächen hingegen müssen die Benutzeraktionen bewusster eingesetzt werden. Der Zustand des Systems und die Struktur der Oberfläche bestimmen, in welche Richtung der Positionsanzeiger bewegt werden muss. Befindet sich das System aufgrund eines Fehlers in einem undefinierten Zustand, ist die vorausberechnete Bediensequenz nicht mehr gültig und muss abgeändert werden. Dies ist für eine Ausführungsumgebung nicht ersichtlich, für die Fortführung der Testfälle allerdings unbedingt erforderlich.

Des Weiteren fehlt bei den aktuellen Ansätzen eine klare Aussage, welche Fehlersymptome beobachtet werden können, inwiefern sich diese auf die weitere Durchführung auswirken und welche Symptome durch das modellbasierte Verfahren gezielt überprüft werden sollen. Auch wenn viele Arbeiten mit oberflächlichen Beschreibungen der Fehler beginnen, fehlen bislang belastbare Aussagen. Dies zeigt sich vor allem darin, dass als Abdeckungskriterien ausschließlich Modelleigenschaften angegeben werden, ohne diese mit dem Prüfling in Beziehung zu setzen. Dies ist zwingend erforderlich, um systematisch verschiedene Testverfahren zu kombinieren.

Diese Mängel werden durch die vorliegende Arbeit behoben. Als Grundlage für die Anwendung der modellbasierten Testautomatisierung auf direktionale Oberflächen ein-

gebetteter Systeme dient eine Fehlersymptomtaxonomie, die eine Abgrenzung unterschiedlicher Symptome ermöglicht und Fehlerschwerpunkte offenlegt. Diese Taxonomie fließt in die Eingrenzung des Testfokus und Auswahl eines geeigneten Modells ein. Mit Hilfe der Taxonomie können Fehlersymptome während der Testfalldurchführung klassifiziert und bei der weiteren Ausführung berücksichtigt werden. Schließlich dient die Taxonomie als Basis einer wirklichkeitsgetreuen Evaluationsumgebung, um die Wirksamkeit des Verfahrens zu überprüfen.

3

Testautomatisierung

Wie im vorangegangenen Kapitel dargelegt, wird die modellbasierte Testautomatisierung (MBT) direktionaler Oberflächen in der Literatur bislang vernachlässigt. Dadurch lassen aktuelle Ansätze insbesondere die Betrachtung charakteristischer Fehlersymptome und deren Auswirkungen auf den Testverlauf vermissen. Im Folgenden wird ein Ansatz zur modellbasierten Testfallerzeugung direktionaler Oberflächen vorgeschlagen, um diese Lücke zu schließen. Der Modellierungsansatz wird in Kapitel 5 exemplarisch angewendet.

Die Voraussetzung für Automatisierung von Prozessen ist es, die wesentlichen Einflussfaktoren parametrisieren und messen zu können. Bei der Automatisierung eines Testverfahrens muss definiert werden, welche Fehlersymptome durch das Verfahren aufgedeckt werden sollen. Nur so kann sichergestellt, dass alle Vorkommen dieser Fehlersymptome erkannt werden und Fehler nicht vorwiegend zufällig gefunden werden. Erst wenn klargestellt ist, welches Testziel durch die MBT abgedeckt ist, kann bestimmt werden, welche weiteren Testverfahren hinzugezogen werden müssen, um auch die verbleibenden Symptome abzuprüfen. Bei der Testmodellierung muss also sichergestellt werden, dass diejenigen Eigenschaften des Zielsystems beibehalten werden, die für die Ableitung und die Durchführung der Testfälle gegen den implementierten Prüfling

notwendig sind. Der Bezug zwischen Testmodell und den fokussierten Fehlersymptomen des SUT ist also entscheidend für die Effektivität des modellbasierten Verfahrens.

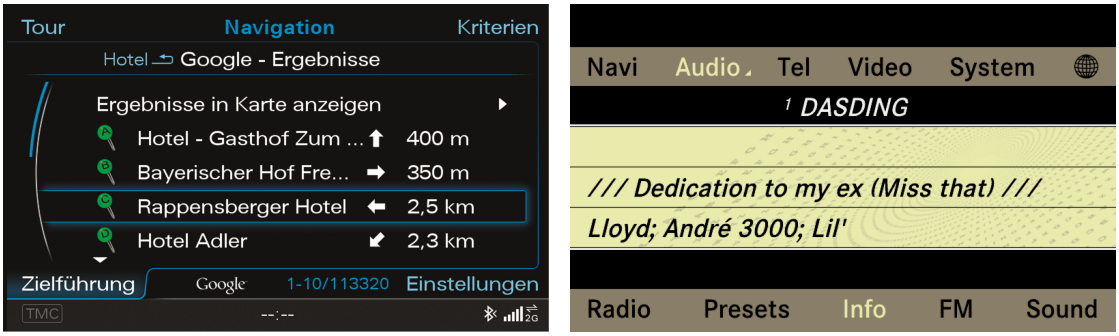
Um festzulegen, welche Fehler durch die MBT abgedeckt werden sollen, muss bekannt sein, welche Fehlersymptome in der Domäne vorkommen. Erst durch die Definition des Testfokus können diejenigen Eigenschaften des System Under Test (SUT) bestimmt werden, die bei der Testmodellierung berücksichtigt werden müssen. Dies wiederum stellt die Grundlage für die Auswahl eines geeigneten Beschreibungsmittels dar. Auf Basis des Modells können anschließend Parameter bestimmt werden, um die Generierung anleiten zu können, mit dem verwendeten Modell die vorgegebenen Testziele zu erreichen.

3.1. Fehlersymptomanalyse

Um die MBT zielgerichtet einsetzen zu können, ist es notwendig, sich einen Überblick der Fehlersymptome der Domäne zu verschaffen. Ein Teilziel dieser Arbeit ist, das Fehlerbild direktionaler Oberflächen realitätsnah abzubilden, um Fehlerschwerpunkte aufzuzeigen und zu quantifizieren. Basierend auf dieser Übersicht können die Fehlersymptome als Testziele zusammengestellt werden, die sich für den Einsatz der modellbasierten Testautomatisierung (MBT) eignen. Im Rahmen dieser Arbeit war es möglich, Fehlerberichte zu untersuchen, die während der Entwicklung eingebetteter Systemen mit direktionalen Oberflächen dokumentiert wurden [MKZD12, MKHZ13]. Die Firmen Audi AG[1], Robert Bosch GmbH[2] und der Geschäftsbereich Mercedes-Benz Cars der Daimler AG[3] stellten die Fehlerdatenbanken zur Verfügung, die während des spezifikations-basierten Funktionstests aktueller IVI Systeme durch das Testpersonal erstellt wurden.

Durch die unterschiedlichen Quellen repräsentieren diese Berichte eine große Bandbreite an Entwicklungskontexten. Dadurch wird vermieden, dass die erhobenen Daten ausschließlich in einem spezifischen Kontext gültig sind. Der wichtigste Unterschied der Datenquellen sind die getesteten IVI Systeme. Bei Audi ist dies das Audi MMI (siehe Abbildung 3.1a) bei Mercedes-Benz das Mercedes COMAND (siehe Abbildung 3.1b). Beide Systeme bieten dem Benutzer die typischen IVI Inhalte wie Navigation, Radio und Systemeinstellungen an. Wie bei direktionalen Oberflächen üblich, werden

[1] http://www.audi.de, zuletzt aufgerufen am 24.07.2015
[2] http://www.bosch.de, zuletzt aufgerufen am 24.07.2015
[3] http://www.daimler.com, zuletzt aufgerufen am 24.07.2015



(a) *Audi Multi Media Interface (MMI)*

(b) *Mercedes-Benz Cockpit Management And Data System (COMAND)*

Abbildung 3.1.: *Beispiele aktueller IVI Systeme*

Informationen und interaktive Elemente in Form von Listen präsentiert. Während bei dem MMI neben dem Zentralen Bedienelement (ZBE) vermehrt separate Kurzwahltasten eingesetzt werden, sind bei dem COMAND System alle Optionen direkt in der Bedienoberfläche aufgelistet. Ein Beispiel für die unterschiedlichen Konzepte ist der Wechsel von Applikationen. Bei dem MMI steht für jede Applikation eine Kurzwahltaste zur Verfügung, bei COMAND werden die Applikationen in einem global verfügbaren Hauptmenü ausgewählt. Die Datenquelle der Fehlerberichte wird ergänzt durch Projekte, in denen die Robert Bosch GmbH als Zulieferer fungierte.

Durch die unterschiedlichen Herangehensweisen der Firmen repräsentieren die Berichte zudem jeweils andere Phasen des Entwicklungsprozesses. Bei Audi wird die Konzeption und Entwicklung in einem Konzern durchgeführt. Die Audi AG erstellt die Konzepte des IVI, das Tochterunternehmen e.solutions[4], ein Joint Venture mit dem Werkzeughersteller Elektrobit[5], übernimmt die Implementierung. Basierend auf dem integrierten System prüft die Audi AG, ob die fachlichen Anforderungen erfüllt wurden. Konzeption, Implementierung und Test werden also im selben Unternehmen, aber in unterschiedlichen Organisationseinheiten durchgeführt. Dadurch besteht die Möglichkeit eines direkten Erfahrungsaustauschs zwischen Spezifikateuren und Software Entwicklern.

Bei Mercedes-Benz Cars sind die Entwicklungsphasen organisatorisch vollkommen getrennt. Die Konzeption erfolgt innerhalb des Unternehmens und wird in Form einer

[4] http://www.esolutions.de, zuletzt aufgerufen am 24.07.2015
[5] http://www.elektrobit.com, zuletzt aufgerufen am 24.07.2015

umfangreichen Spezifikation beschrieben. Die Implementierung wird von einem externen Unternehmen durchgeführt, der für jede Telematik-Generation durch eine Ausschreibung neu ermittelt wird. Der abschließend vom Auftraggeber durchgeführte Abnahmetest stellt sicher, dass die Implementierung der Spezifikation entspricht. Bei der Testphase kann demnach ausschließlich auf die Expertise der Spezifikateure und nicht auf das Fachwissen über die Software-Architektur zurückgegriffen werden. Die von Mercedes-Benz durchgeführte Entwicklung fokussiert sich auf die obere Hälfte des V-Modells.

Im Gegensatz dazu positioniert sich die Robert Bosch GmbH als Zulieferer bei der IVI Entwicklung hauptsächlich in der unteren Hälfte des V-Modells. Basierend auf den Spezifikationen der Auftraggeber werden entweder Software- oder Hardware-Komponenten realisiert. Dadurch stehen bei den Tests der Bedienoberfläche zwar die Kenntnisse der internen Struktur der Implementierung, aber kein fachliches Domänenwissen, das der Konzeption zugrunde liegt, zur Verfügung.

Darüber hinaus bestehen Unterschiede durch unterschiedliche strategische Entscheidungen der Firmen, die nicht im Einzelnen bekannt sind. Diese Unterschiede ergeben sich beispielsweise aus verschiedenen Teststrategien und Testschwerpunkten, unterschiedlichem Testpersonal oder den eingesetzten Testtechnologien. Aufgrund natürlicher Divergenz ist davon auszugehen, dass auch durch diese Faktoren die allgemeine Aussagekraft der Daten verstärkt wird.

In einer manuell durchgeführten Analyse wurden Berichte, die ähnliche Fehlersymptome beschreiben, zu Fehlersymptomklassen zusammengefasst. Das Ergebnis ist ein hierarchisches Klassifikationsmodell, eine so genannte *Fehlersymptomtaxonomie*. In dieser Analyse wurden ausschließlich die Berichte berücksichtigt, die sowohl von Testern als auch Entwicklern als tatsächliche Fehler anerkannt wurden. Berichte, die sich als Fehlinterpretationen der Tester oder Missverständnisse erwiesen haben, wurden verworfen. Ein Drittel dieser Berichte wurde manuell analysiert, um die obersten Hierarchieebenen und damit die Grundlage der Taxonomie zu erarbeiten. Die entstandene Struktur wurde mit Hilfe der verbleibenden Daten überprüft bzw. korrigiert. Während der Klassifikation wurden drei Gütekriterien abgeleitet, um den Erstellungsprozess der Taxonomie zu lenken.

1. Eine flache Taxonomie, wie sie in ähnlichen Arbeiten verwendet werden [LLS10, BC06, Chi99, BV90], sollte bei dieser Studie vermieden werden, da sich diese

Struktur als unübersichtlich herausgestellt hat. Stattdessen sollte eine hierarchische Struktur verwendet werden. Dadurch ist eine nachträgliche Detaillierung der Taxonomie möglich, ohne höhere Ebenen anzupassen. Auch wenn sich die Taxonomien auf unteren Hierarchieebenen verändern, können die Anteile der weiterhin übereinstimmenden, übergeordneten Fehlersymptomklassen für Vergleiche herangezogen werden. Dadurch können beispielsweise Fehlerbilder verschiedener Versionen oder Produkte gegenüberstellt werden. [Hol13] zeigt eine Anpassung der Taxonomie auf die Fehlersymptome einer Applikation auf einem mobilen Endgerät.

2. Auf jeder Hierarchieebene sollten zwischen mindestens zwei und maximal fünf Kategorien unterschieden werden. Dabei sollten die Blattkategorien maximal 10% der Gesamtfehlerzahl enthalten. Dieses Kriterium vermeidet die Erstellung von allgemeinen Kategorien ohne Aussagekraft. Während der Erstellung der Taxonomie hat sich herausgestellt, dass ab 10% eine Aufteilung der Kategorie angemessen ist.
3. Durch eine neutrale Kategorie *To Be Categorized* (TBC) wird vermieden, dass Fehlerberichte aus Ermangelung einer geeigneten Kategorie und damit ohne fachliche Begründung klassifiziert werden. Gemäß der zweiten Anforderung ist auch diese Kategorie auf 10% der Gesamtfehlerzahl beschränkt. Damit deckt die Taxonomie mindestens 90% der untersuchten Fehlerberichte ab.

Tabelle 3.1 zeigt sowohl die hierarchische Klassifikation als auch die Anteile der einzelnen Klassen an Fehlerberichten. Auf oberster Ebene wird neben der bereits beschriebenen Kategorie TBC zwischen den Fehlersymptomklassen *Logik*, *Inhalt* und *Design* unterschieden. Fehler in der Logik beschreiben fehlerhafte Reaktionen des Systems auf entweder eine Aktion des Benutzers oder Änderungen des Kontextes. Die Kategorie Logik wird in die Unterkategorien *Widgets*, *Übergänge*, *PopUps* und *Struktur* aufgegliedert. Die Unterkategorie Widgets fasst Berichte zusammen, die den modularen Bestandteilen der Oberfläche, wie Buttons und Listen, zuzuschreiben sind. Dies schließt explizit auch das Fokusverhalten mit ein. Durch die Kategorie PopUps werden Fehlerberichte zusammengefasst, die das fehlerhafte Erscheinen bzw. Ausbleiben, sowie die Priorisierung von proaktiven Systemmeldungen beschreiben. Die Kategorie Struktur repräsentiert den Anteil der Berichte, die eine fehlerhafte Zusammenstellung oder einen fehlerhaften Aufbau der aktuell auf dem Bildschirm angezeigten Komponenten beschreiben. Diese Kategorie beinhaltet Berichte über das Ausgrauverhalten (Kategorie *GrayOut*), Auswahl

Tabelle 3.1.: *Fehlersymptomtaxonomie*

Level 1	Level 2	Level 3	Level 4	Anteil in %
Logik (61,5%)	Übergänge (17,9%)	Existenz		8,7
		Fehlerhaft		9,2
	PopUps (11,7%)	Existenz		7,3
		Priorität		4,4
	Struktur (13,8%)	Komposition		5,4
		Optionen		5,4
		GrayOut		3,0
	Widgets (18,1%)	Spezifika (14,7%)	Existenz	6,0
			Fehlerhaft	8,7
		Fokus		3,4
Inhalt (25,1%)	Text (15,1%)	Statisch		5,9
		Dynamisch		9,2
	Animationen			1,8
	Symbole			8,2
Design				5,8
TBC				7,6
				100

der korrekten Widget-Klassen (Kategorie *Komposition*) oder die Auflistung von Einträgen einem Menü (Kategorie *Optionen Angebot*).

Die Kategorie Inhalt fasst Fehlerberichte zusammen, die sich auf die angezeigten Informationen beziehen. Dies sind in den untersuchten Fällen *Text*, *Symbole* und *Animationen*. Aufgrund der Größe musste die Unterkategorie Text zusätzlich aufgeteilt werden. Bei der Analyse der Daten hat sich die Unterteilung angeboten, zu welchem Zeitpunkt die vollständige Beschriftung vorliegt. Sind die Texte bereits bei der Implementierung des Systems vollständig bekannt, wie beispielsweise die Beschriftung der Buttons im Hauptmenü des Systems, werden die Berichte der Unterkategorie *Statisch* zugeordnet. Wird die Beschriftung aus Informationen zusammengesetzt, die erst während der Laufzeit ermittelt werden, wie beispielsweise die Auflistung von Liedern, Künstlern und Alben verfügbarer Musikdateien, werden die beschriebenen Fehlersymptome als *Dynamisch* eingeordnet. Zur Kategorie Design werden Fehlerberichte gezählt, die sich ausschließlich auf die Darstellung beziehen. Dies ist beispielsweise eine fehlerhafte Farbe eines Buttons oder dessen Positionierung auf dem Bildschirm. Aufgrund der geringen Anzahl Design-bezogener Fehlerberichte war eine Aufspaltung dieser Kategorie nicht notwendig.

3.2. Testfokus

Basierend auf der Fehlersymptomtaxonomie kann nun bestimmt werden, für welche Fehlersymptome eine Überprüfung durch eine modellbasierte Testautomatisierung geeignet ist. In dieser Arbeit wird dabei die folgende Terminologie verwendet. Ein *Testziel* entspricht einer Fehlersymptomklasse, die durch das Testverfahren überprüft werden soll. Testziel kann beispielsweise sein, alle Texte des SUT oder die spezifischen Eigenschaften der Widgets zu überprüfen. Der *Testfokus* fasst schließlich alle Testziele zusammen, die durch die Testverfahren überprüft werden sollen. Bei der Bestimmung der fokussierten Testziele muss abgewogen werden, für welche Fehlersymptome sich das Mittel einer Testautomatisierung eignet. Das ausschlaggebende Kriterium ist, ob die Fehlersymptome automatisch provoziert und zuverlässig erkannt werden können. Im Folgenden wird diskutiert, inwieweit dies für die beschriebenen Fehlersymptomklassen zutrifft.

Design & Animationen

Um eine automatische Verifikation im Rahmen der MBT anzuwenden, müssen die Ergebnisse des Verfahrens zuverlässig reproduziert werden können. Für die Fehlersymptomklassen Design und Animationen ist dies allerdings noch ein aktuelles Forschungsfeld [Hei14, AZJH11, CYM10]. Bei einer Verifikation wird die Anzeige auf dem Bildschirm mit Hilfe von Algorithmen zur Bildverarbeitung analysiert. Die Herausforderung ist, dass Farben und Positionen je nach Darstellungstechnologie variieren oder nur annäherungsweise bestimmt werden können. Eine Übereinstimmung zwischen den Vorgaben des Modells und den erkannten Werten ist zudem häufig nicht zwingend erforderlich, da im praktischen Einsatz auch ähnliche Werte toleriert werden können. [Hei14] schlägt daher vor, mit Hilfe von Bayschen Netzen die Wahrscheinlichkeit zu ermitteln, dass die erkannten Werte den Anforderungen genügen. Da Verfahren zur Verifikation von Design und Animationen allerdings bisher noch nicht mit einem ausreichenden Reifegrad umgesetzt wurden, ist von einem Einsatz zu diesem Zeitpunkt noch abzusehen. Zudem fallen beide Kategorien wegen des geringen Anteils von zusammen 7,6% der Fehlerberichte kaum ins Gewicht.

Widgets

Einen großen Anteil der Logik-Fehler nehmen die spezifischen Eigenschaften von Widgets ein (14,7%). In einem automatischen Test müssten die als testrelevant gekennzeichneten Eigenschaften der Widgets aufgerufen und verifiziert werden. Da allerdings bei dem hier zugrunde liegenden Szenario die interne Struktur der Implementierung unbekannt ist, kann nicht die in heutigen Ansätzen sehr wahrscheinlich eingesetzte objektorientierte Struktur genutzt werden. Die Feststellung, dass eine Ausprägung des Widgets fehlerfrei ist, lässt sich demnach nicht auf ähnliche Widgets übertragen. Das hat zu Folge, dass jede Ausprägung des Widgets untersucht werden muss. Für Tests dieser Art empfiehlt sich also ein Verfahren, das sich an der Architektur des Prüflings orientiert, wie beispielsweise Unit Tests. Basierend auf den abstrakten Beschreibungen der Widgets können gezielt alle Varianten der Klasse reproduziert und überprüft werden.

PopUps

Eine andere Fehlerklasse mit einem großen Anteil nehmen die Fehlerberichte ein, die sich auf fehlerhafte proaktive Systemmeldungen, den so genannten PopUps, beziehen. PopUps können als Fenster realisiert sein, die die weiteren Inhalte der Oberfläche überlagern und dadurch den regulären Bedienablauf unterbrechen. Diese PopUps sind bei eingebetteten Systemen häufig mit den Zuständen angeschlossener Regelkreise verknüpft. Ein Beispiel für in PopUp ist die Meldung, dass die Temperatur einer Komponente einen kritischen Stand erreicht. Um diese Meldung zu provozieren, muss dieser Zustand real angelegt oder simuliert werden. Beides ist in dem Kontext dieser Arbeit nicht praktikabel.

Struktur & Übergänge

Bei eingebetteten Systemen ist die Bedingungsabhängigkeit von Optionen eine zentrale Herausforderung (siehe Abschnitt 2.1). Dies zeigt sich vor allem darin, dass Bereiche der Oberfläche nur in bestimmten Ausstattungsvarianten zur Verfügung stehen oder durch die Komponenten der Oberfläche Zustände des Gesamtsystems angezeigt werden. Durch die generierten Testfälle soll gezeigt werden, dass genau die Zustände und Optionen, die unter den aktuell geltenden externen Bedingungen verfügbar sein sollen, tatsächlich erreicht werden können. Alle Zustände und Optionen, die unter der aktuellen Bedingungsbelegung nicht verfügbar sein sollen, müssen gemäß der Vorgaben ausgeblendet oder ausgegraut sein. Die dafür notwendigen internen Bedingungsbelegungen des Systems sollen dabei während des Testfalls selbstständig hergestellt werden. Die Manipulation externer Bedingungen ist im Kontext dieser Arbeit nicht praktikabel.

Anhand der Fehlersymptomtaxonomie setzt sich damit der Testfokus dieses Ansatzes aus den Logik-Kategorien Übergänge (17,9%) und Struktur (13,8%) zusammen und deckt damit 31,7% der untersuchten Fehlerberichte ab. Dieser Testfokus wird im Folgenden als *Menüstruktur* bezeichnet.

3.3. Testfallgenerierung

Im vorangegangenen Abschnitt wurde mit Hilfe der Fehlersymptomtaxonomie der Testfokus «Menüstruktur» definiert. In diesem Abschnitt wird nun darauf eingegangen, wie das modellbasierte Verfahren eingesetzt werden kann, um diesen Testfokus abzudecken. Der Abschnitt beginnt mit einer Diskussion, welche Informationen für die Testautomatisierung relevant sind. Darauf aufbauend wird abgeleitet, welche Modellarten geeignet sind, diese Informationen zu beschreiben. Abschließend wird diskutiert, welche Schritte notwendig sind, aus dem Modell Testfälle automatisch abzuleiten.

3.3.1. Testrelevante Informationen

Zu Beginn der Modellierung muss festgestellt werden, welche Eigenschaften des SUT für die Ableitung und Durchführung von Testfällen notwendig sind. Diese Eigenschaften werden im Folgenden als *testrelevante Informationen* bezeichnet. Entscheidend für die Relevanz ist die Art und Weise, wie während des Tests auf den Prüfling eingewirkt werden kann. Dies beinhaltet, auf welche Weise er stimuliert werden muss, und wie die Reaktion des Systems verifiziert werden kann. In dem hier untersuchten Kontext wird von einem Black-Box-Test ausgegangen (siehe Abschnitt 2.2). Das bedeutet, dass kein direkter Einfluss auf Komponenten des Prüflings besteht. Die Stimulation erfolgt ausschließlich über die vorgesehenen Bedienaktionen eines direktionalen Bedienkonzeptes. Damit ist dieses Testverfahren dem Keyword Driven Testing zuzuordnen [RBGW12]. Es werden Schlüsselwörter definiert, die von einer Ausführungsumgebung interpretiert werden können, um das SUT zu stimulieren. Bei direktionalen Oberflächen sind dies zumindest die Kommandos `UP`, `DOWN`, `LEFT`, `RIGHT`, `PRESS` sowie ggfls. zusätzliche Tasten mit gesonderter Funktion, die global verfügbar sind.

Stimulation

Um den festgesetzten Testfokus abzudecken, ist es notwendig, die Bedingungen des Systems gezielt zu manipulieren, um alle notwendigen Zustände herzustellen. Bei der Modellierung muss definiert werden, welche Bedingungen aus der Perspektive des Benutzers existieren, und wie diese verändert werden können. Diese Bedingungen müssen

nicht denen entsprechen, die bei der technischen Umsetzung des Systems verwendet wurden. Lediglich deren Auswirkungen müssen in den Testsituationen identisch ausfallen. Eine vollständige Referenzimplementierung ist demnach nicht erforderlich. Es ist ausreichend, dass sich Prüfling und Modell ausschließlich bezüglich der Testfälle analog verhalten. Durch diese Reduktion kann Aufwand und Fehleranfälligkeit des Modells reduziert werden.

Aus der Sicht des Benutzers werden Bedingungen ausschließlich durch Widgets verändert. In dem Testmodell ist demnach zu hinterlegen, welche Widgets durch welche Aktionen welche Bedingungen des Gesamtsystems verändern. Dies beinhaltet Bedingungen, die sich ausschließlich auf die Anzeige auf dem Bildschirm bzw. die angebotenen Optionen auswirken. Diese Bedingungen werden im Folgenden als *interne Bedingungen* aus Sicht der Benutzeroberfläche bezeichnet. Beispiele für interne Bedingungen sind das aktuell geöffnete Menü oder die aktive Applikation. Die Benutzeroberfläche eingebetteter Systeme dient dem lesenden und schreibenden Zugriff auf den Zustand eines übergeordneten Gesamtsystems (siehe Abschnitt 2.1). Diese Bedingungen werden im Folgenden als *externe Bedingungen* aus der Perspektive der Oberfläche bezeichnet. Beispiele sind die angeschlossenen Systeme oder verfügbaren Medien. Das modellbasierte Verfahren muss in der Lage sein, die exakte Folge von Bedienaktionen abzuleiten, die für die Bedingungsmanipulation notwendig sind. Bei direktionalen Oberflächen bedeutet dies im Hauptteil der Fälle die Auswahl von Optionen durch die Iteration von Listen.

Verifikation

Neben der Generierung sind diejenigen Informationen testrelevant, die zur Verifikation des Systems benötigt werden. Wie in Abschnitt 2.3 beschrieben, steht für die Verifikation zur Laufzeit ein im Vorfeld definierter Einblick in das System zur Verfügung. Dieser Einblick wird als DFT-Schnittstelle bezeichnet (siehe Abschnitt 2.3). Die für die Verifikation notwendigen Referenzinformationen sollen in die Testfälle eingebettet werden. Bei diesem Vorgehen müssen in dem Testmodell die Eigenschaften des Prüflings hinterlegt werden, die auch über die Code-Schnittstelle ausgegeben werden können. Diese Eigenschaften und deren Ausprägungen müssen zumindest zu dem Zeitpunkt im Verlauf eines Testfalls verfügbar sein, in denen eine Verifikation des Systems vorgesehen ist. In dem untersuchten Kontext ist durch die Ausgabe der Struktur der aktuellen Anzeige realisiert

(siehe Abschnitt 2.3), da diese Informationen sowohl über eine dedizierte Software Schnittstelle oder durch die Auswertung der Anzeige durch Bildverarbeitungsalgorithmen bezogen werden können. Um das volle Potential dieser Verifikation nutzen zu können, muss das Testmodell gewährleisten, dass die zu überprüfenden Eigenschaften aller auf dem Bildschirm angezeigter Widgets im Falle einer Verifikation ausgegeben werden können.

3.3.2. Modellierung

Im vorangegangenen Abschnitt wurde festgestellt, welche Eigenschaften des SUT mit dem Testziel «Menüstruktur» als testrelevant einzustufen sind. Im Folgenden wird diskutiert, welche Beschreibungsformen sich für die Modellierung dieser Informationen eignen. Aufgrund der Charakteristika graphischer Benutzeroberflächen ergeben sich die folgenden Anforderungen an ein Testmodell [MKH13]:

1. Die Bedienoberfläche eingebetteter Systemen ist reaktiv. Das bedeutet, dass das System eingehende Bedienaktionen oder Zustandsänderungen auswertet und als Reaktion den eigenen Zustand anpasst. Diese Zustände werden häufig als Konstellation von internen und externen Bedingungen realisiert. Es ist daher ein notwendiges Kriterium an die Beschreibungsform eines Testmodells, Bedingungen und deren Auswirkungen auf das System modellieren zu können.
2. Sowohl das Modell als auch die generierten Testfälle sollten von Testingenieuren nachvollzogen werden können, um eine manuelle Qualitätssicherung zu ermöglichen. Daher sollte das Modell dem Ersteller eine Übersicht über die Struktur der Oberfläche und dem Verlauf der Testfälle bieten können. Dies bezieht sich in erster Linie auf die Verknüpfung bzw. den Ablauf von Screens.
3. Graphische Benutzeroberflächen bestehen aus elementaren Bestandteilen, den so genannten Widgets (siehe Abschnitt 2.2). Diese Widgets sind einmal beschrieben und werden an verschiedenen Stellen der Oberfläche wiederverwendet. Um die Fehleranfälligkeit der Testmodelle zu reduzieren, sollte das Testmodell eine modulare Modellierung erlauben.

Zustandsautomaten eignen sich zur Modellierung reaktiver Systeme und werden bereits bei Ansätzen zur MBT verwendet (siehe Abschnitt 2.3). Der Fokus der Zustandsautoma-

ten liegt auf der Modellierung von Bedingungen sowie der Auswertung von Ereignissen und erfüllt damit die erste der gestellten Anforderungen. Durch die Anwendung von so genannten *View States*, die innerhalb eines Zustandsautomaten eine konkrete graphische Ausgabe repräsentieren (siehe Abschnitt 2.2), ist eine übersichtliche Darstellung der Struktur bei der Modellierung gegeben. Der Ablauf der Testfälle kann durch die Angabe der aktivierten View States und Transitionen gut nachvollzogen werden. Allerdings unterstützen Zustandsautomaten nur unzureichend die modulare Struktur der Oberflächen. Dies lässt sich am Beispiel eines Menüs nachvollziehen.

Ein Menü ist eine Ansammlung von Einträgen, die der Reihe nach iteriert werden können. Die Verfügbarkeit der Einträge kann von dem Zustand des Gesamtsystems abhängen und muss bei jeder Zustandsänderung neu ausgewertet werden. Um dem Benutzer die Auswahl des gewünschten Eintrags zu erleichtern, ist bei direktionalen Oberflächen häufig vorgesehen, dass beim Öffnen eines Menüs der mittlere der verfügbaren Einträge vorausgewählt ist. Die Ermittlung des Fokus hängt also von der Verfügbarkeit jedes Eintrags des Menüs ab. Die Modellierung dieser Auswertung ist mit den Mitteln eines Zustandsautomaten aufwendig und fehleranfällig. Die Komplexität wird zudem weiter gesteigert, wenn man zusätzlich die in aller Regel iterative Entwicklung des Modells in berücksichtigt. Dies hat zur Folge, dass während der Modellierung noch nicht alle Einträge der Menüs vorliegen. Während der Modellierung muss also die Bedingungsauswertung bei jeder Änderung manuell angepasst werden.

Die Komplexität der Bedingungsauswertung kann durch Parametrisierung bzw. Objektorientierung abgebildet werden. Daher sieht der Ansatz dieser Arbeit vor, dem Zustandsautomaten eine Programmierung der Widgets hinzuzufügen. Die internen Bedingungen, die Screens und deren Übergänge, werden durch Zustände und Transitionen in einem Zustandsautomaten repräsentiert. Das Verhalten innerhalb der Screens und den darin enthaltenen Widgets wird mit Hilfe einer objektorientierten Programmiersprache beschrieben. Durch Klassen werden die grundsätzlichen Eigenschaften und Methoden der Widgets definiert. Innerhalb des Modells werden Objekte instantiiert und dadurch mit Daten versehen. Die Schnittstelle zwischen Zustandsautomaten und OO-Programmierung sind View States, die Zustände mit Screen Objekten in Beziehung setzen. Mit Hilfe des Zustandsautomaten wird ermittelt, welcher View State aktuell angezeigt wird und damit die Bedienaktionen empfangen soll. Der Startzustand entspricht der bei Systemstart gültigen Bedingungsbelegung.

Die Bedienaktionen und damit die Navigation innerhalb eines Screen Objekts wird durch den Programmcode ausgewertet. Durch die Interaktion mit Elementen auf dem Bildschirm werden entweder weitere Elemente ausgeblendet bzw. angezeigt, Bedingungen geändert oder eine Transition im Zustandsautomaten ausgelöst. Externe Bedingungen sind als globale Variablen realisiert, die sowohl von Elementen des Zustandsautomaten als auch Code-Objekten zugegriffen und verändert werden können. Das Modell besteht demnach aus einer Widget-Klassen-Bibliothek, einer Methode zur Initialisierung der Screen- und Widget-Objekte und einem Zustandsautomaten, der die instantiierten Screens in Beziehung zueinander setzt.

3.3.3. Referenzdaten

Bei diesem Verfahren handelt sich um einen Offline-Test (siehe Abschnitt 2.3). Das bedeutet, dass die Testfälle vorgeneriert und anschließend gegen ein SUT ausgeführt werden. Bei manuell durchgeführten Tests entscheidet der Mensch, ob die Reaktion des Systems auf seine Eingaben korrekt ist. Bei der automatisierten Durchführung soll diese Entscheidung von einer Software getroffen werden können, die als *Testrahmen* bezeichnet wird (siehe Abschnitt 2.3). In der Literatur werden Testfallgenerierung und Verifikation häufig separat betrachtet. Das Testmodell dient ausschließlich der Ableitung von Bediensequenzen. Die Beurteilung der Systemreaktion wird durch eine Software durchgeführt, die als *Orakel* bezeichnet wird (siehe Abschnitt 2.3). Dabei handelt es sich um eine Referenzimplementierung, die auf Abruf den zu diesem Zeitpunkt der Bedienung vorhergesehenen Systemzustand ausgeben kann. Eine separate Referenzimplementierung ist allerdings eine weitere Fehlerquelle und bedeutet zusätzlichen Aufwand.

Zudem sind die Informationen, die für die Generierung von Testfällen verwendet werden zumindest teilweise deckungsgleich mit denen, die für die Referenzdaten notwendig sind. Ziel ist daher, bereits das Testmodell als Referenzimplementierung zu verwenden. Für die Iteration eines Menüs müssen alle verfügbaren Einträge zum Zeitpunkt der Generierung bekannt sein. Diese Information kann während der Durchführung zusätzlich zur Verifikation des Systemzustands verwendet werden. Hierbei handelt es sich um einen Ausschnitt des tatsächlichen Systems. So werden bspw. Layout-Informationen nur in dem Maße berücksichtigt, zu dem diese für die Verifikation notwendig sind. Wird

Bildverarbeitung zur Verifikation des Systems eingesetzt, muss angegeben werden, in welchem Bildschirmbereich sich das Label befindet.

Um das Testmodell als Referenz verwenden zu können, muss die notwendige Informationsbasis geschaffen werden. Die Referenzinformationen und der aktuelle Zustand des Prüflings müssen in einer Form vorliegen, die den Vergleich mit Hilfe von Algorithmen ermöglichen. Die vorgestellte Vorgehensweise, Testfälle und Referenzdaten modellbasiert zu generieren, ermöglicht es, zu jedem Zeitpunkt der Bedienung den Systemzustand und die Datenbelegung der auf dem Bildschirm angezeigten Widgets ausgeben zu lassen. Das Testmodell erfüllt in diesem Fall den Zweck der Referenzsimulation. Die modellierten Elemente nehmen im Laufe der Testfallgenerierung genau die Eigenschaften an, die auch das Zielsystem erfüllen soll. Screen- und Widget- Objekte verfügen über eine Selbstauskunftsfähigkeit, um deren Repräsentation ausgeben zu können. Container-Elemente geben dabei rekursiv die Repräsentation enthaltener Elemente aus. Auf diese Weise wird der Aufbau der Oberfläche, der zu diesem Zeitpunkt der Bedienung auf dem Bildschirm zu sehen ist, in einem hierarchischen Format abgebildet. Bei der Modellierung wird das Modell mit Ereignissen versehen, die eine Ausgabe von Verifikationsinformationen auslösen. Detailreichtum und Frequenz können durch den Modellierer vorgegeben werden. So kann beispielsweise der Aufruf eines neuen View States während der Generierung für eine vollständige Referenzbeschreibung des neu anzuzeigenden Inhalts genutzt werden.

3.3.4. Generierung

Es liegen alle Informationen vor, die nun von einem Generierungsalgorithmus ausgewertet werden können. Dem Generierungsalgorithmus müssen Anreize geschaffen werden, Modellteile aufzurufen. Diese Anreize werden als *Steuerparameter* bezeichnet und setzen das Modell und die festgesetzten Testziele in Beziehung (siehe Abschnitt 2.3). Diese Steuerparameter beziehen sich bei den in diesem Verfahren verwendeten Modellen sowohl auf Elemente des Zustandsautomaten als auch auf Attribute im Programmcode. Um das Testziel «Menüstruktur» zu erreichen muss innerhalb der Zustandsmaschine eine Zustandsüberdeckung erreicht werden. Jeder modellierte View State muss mindestens einmal betreten werden. Dadurch wird gewährleistet, dass jeder modellierte Screen aufgerufen wird. Weiterhin muss für dieses Testziel sichergestellt werden, dass beim

Aufruf des Screens jede Bedingungskonstellation vorliegt, die notwendig ist, dass jeder Listeneintrag auf dem Bildschirm erreichbar ist. Da Einträge und Bedingungen durch Programmcode modelliert sind, beziehen sich die Steuerparameter auf Codefragmente. In den Klassen der Widgets, die von diesem Testziel betroffen sind, sollte daher eine Methode vorgesehen werden, die aufgerufen wird, sobald das Testziel erreicht wurde. Dies ist beispielsweise bei der positiven Beurteilung der Verfügbarkeit oder beim Aufruf des Eintrags gegeben. Das Testziel «Übergänge» entspricht einer Abdeckung aller modellierten Transitionen.

Der Generierungsalgorithmus leitet alle Pfade durch das Testmodell ab, die notwendig sind, um die durch die Steuerparameter vorgegebenen Modellfragmente zu erreichen. Im Verlauf eines Testfalls werden die notwendigen Einstellungen selbsttätig hergestellt und die zu überprüfenden Inhalte aufgerufen. Wenn es beispielsweise notwendig ist, eine Einstellung in einem anderen Bereich der Oberfläche einzurichten, soll dies ebenfalls Bestandteil des Testfalls sein. Andere Ansätze sehen vor, Testfälle aufeinander aufbauen zu lassen. Dabei ist der Systemzustand, mit dem ein Testfall das System hinterlässt, die Voraussetzung für den Beginn des darauf folgenden Testfalls. Falls allerdings der erste Testfall zu einem Fehler führt, wovon vor allem bei frühen Testphasen ausgegangen werden muss, hat dies zur Folge, dass der darauf aufbauende Testfall nicht gestartet werden kann. Außerdem können die Testergebnisse nur durch einen vollständigen Testdurchlauf reproduziert werden. Aus diesem Grund beginnt in diesem Verfahren jeder Testfall an dem Zustand des Systems unmittelbar nach dem Neustart. Dadurch können Testfälle unabhängig voneinander durchgeführt werden.

Das Ergebnis der Generierung wird als *Testsuite* bezeichnet. Eine Testsuite enthält alle Testfälle, die notwendig sind, um die vorgegebenen Testziele zu erreichen. Die Testfälle entsprechen einer Kommunikationssequenz zwischen einem «Tester» und dem SUT. Das SUT informiert den Tester über den Systemzustand, der Tester führt die Bedienaktion durch. Testfälle beginnen und enden immer mit der Ausgabe des Systemzustands, um den Test an einem definierten Zustand zu beginnen und zu verlassen. Es folgen die Bedienaktionen, die notwendig sind, um die vorgegebenen Testziele zu erfüllen. Diese Bedienaktionen sind immer gepaart mit einer Verifikation des Systemzustands um sicherzugehen, dass die Auswirkungen der Bedienaktion erwartungskonform sind.

Bei einer Testsuite kann eine prozentuale *Abdeckung* des Testfokus angegeben werden. Die Abdeckung beschreibt, welcher Anteil aller durch die Steuerparameter angegebenen

Modellfragmente durch die Testfälle erreicht werden kann. Bei der Generierung wird unterschieden, ob Bedingungen, die für die Erreichbarkeit von Modellteilen notwendig sind, durch Elemente im Modell verändert werden können, oder ob deren Belegung über den Verlauf des Tests gleich bleibt. Dies hat Auswirkungen auf die Abdeckung, da gegebenenfalls Teile des Testziels unter der Belegung der konstanten Bedingungen nicht erreicht werden können.

3.4. Zusammenfassung

In diesem Abschnitt wurde ein Verfahren zur modellbasierten Testautomatisierung direktionaler Oberflächen vorgeschlagen. Besonderes Augenmerk lag dabei auf den charakteristischen Fehlersymptomen und deren Auswirkungen auf Modellierung und Generierung. Mit Hilfe von Fehlerberichten aus aktuellen Entwicklungsprojekten wurde eine Fehlersymptomtaxonomie entwickelt, um das Fehlerbild direktionaler Oberflächen eingebetteter Systeme realitätsnah abzubilden. Basierend auf der Taxonomie wurden Fehlersymptomklassen als Testziele ausgewählt, die im Zuge einer Testautomatisierung provoziert und überprüft werden können. Die Testziele «Übergänge» und «Struktur» wurden zu dem Testfokus «Menüstruktur» zusammengefasst. Anschließend wurde ermittelt, welche Informationen für die Generierung und Durchführung entsprechender Testfälle notwendig sind und eine Kombination aus Zustandsautomaten und Programmcode als geeignetes Beschreibungsmittel dieser Informationen identifiziert. Der Modellierungsansatz wird in Kapitel 5 exemplarisch angewendet. Im folgenden Kapitel wird beschrieben, wie dieses Modell dazu verwendet werden kann, um einen Abbruch der Durchführung aufgrund eines fehlerhaften SUT zu vermeiden.

4

Automatische Durchführung

Wie in Abschnitt 2.2 beschrieben, gefährden die Auswirkungen der Fehlersymptome bei direktionalen Oberflächen die Praxistauglichkeit der modellbasierten Testautomatisierung. Die Tatsache, dass die Testdurchführung bereits bei einfachen Fehlern abgebrochen werden muss, kann in dem Entwicklungsszenario dieser Arbeit dazu führen, dass die Überprüfung von Teilen der Oberfläche aufgeschoben werden muss oder aufgrund des festgesetzten Zeitrahmens nicht beendet werden kann. Dies führt dazu, dass die Funktionalität des Gesamtsystems nicht sichergestellt ist. Wie in Abschnitt 2.3 aufgezeigt, wird dieser Umstand von den aktuellen Forschungsansätzen vernachlässigt. Dabei ist durch den modellbasierten Ansatz eine Möglichkeit gegeben, dass die Testautomatisierung selbstständig zumindest auf einige der möglichen Fehlersymptome reagieren kann. Diese Fähigkeit wird im Folgenden als *differenzierte Fehlerbehandlung* zusammengefasst. Im Laufe des Kapitels wird anhand der Fehlersymptomtaxonomie (siehe Abschnitt 3.2) aufgeschlüsselt, welche Auswirkungen die einzelnen Fehlersymptome verursachen und ein Verfahren abgeleitet, wie der Abbruch der Durchführung mit Hilfe des Testmodells vermieden werden kann. Das Kapitel schließt mit einer Diskussion der Faktoren, die die Wirksamkeit der differenzierten Fehlerbehandlung bestimmen.

4.1. Problembeschreibung & Lösungsansatz

Wie in Abschnitt 2.3 beschrieben, beschäftigen sich die aktuellen Arbeiten mit der Ausführung der generierten Testfälle nur in der Nebensache. Testgüteuntersuchungen beschränken sich auf den Nachweis, ob ein Verfahren in der Lage ist, Fehler aufzudecken. Die Reaktion des Testverfahrens spielt in diesem Rahmen eine untergeordnete Rolle. Für den praktischen Einsatz bei direktionalen Oberflächen ist die Robustheit gegenüber Fehlern allerdings ein entscheidender Faktor (siehe Abschnitt 2.3). Ein Testverfahren, das nicht darauf ausgelegt ist Fehler nach der Identifikation umgehen zu können, kann nicht effizient betrieben werden. Aufgrund der begrenzten Eingabeoptionen ist nicht in jeder Situation automatisch zu ermitteln, welche Bedienaktionen in einem Fehlerfall die weitere Durchführung ermöglichen. Daher wird in dieser Arbeit der Fehlerfall genauer untersucht. Sobald eine Abweichung registriert wird, bleiben zwei Möglichkeiten: (a) die Durchführung wird abgebrochen oder (b) trotz Abweichung weiter wie vorgesehen durchgeführt.

In den bisherigen Ansätzen muss diese Frage pauschal beantwortet werden. Dabei können beide Optionen schwerwiegende Folgen haben. Bislang wird bevorzugt, die Testdurchführung abzubrechen sobald ein Fehler gefunden wurde, da sich das System Under Test (SUT) in einem nicht definierten Zustand befinden kann. Alle weiteren Ergebnisse sind nur unter Vorbehalt gültig, da sie gegebenenfalls auf den bereits bekannten Fehler zurückzuführen sind. Nur durch einen Abbruch können Folgefehler vermieden werden, die wiederum durch Testpersonal überprüft und von tatsächlichen Fehlern unterschieden werden müssen. Für die praktische Anwendung führt ein Abbruch bei jeder Abweichung zu einer Erhöhung des Aufwands und gefährdet damit die Praxistauglichkeit des Testverfahrens. Dieses Vorgehen ist in dem in Abschnitt 2.2 beschriebenen Prozess nicht praktikabel, da nach einer Testphase einige Zeit für die Behebung der Fehler vergeht. Dies kann unter Umständen dazu führen, dass die vorgesehene Anzahl der Testzyklen nicht ausreicht, um alle generierten Testfälle durchführen zu können.

Auch die weitere Durchführung ist nicht in allen Fällen zielführend. Einige Fehlersymptome beeinträchtigen die Durchführung und verhindern den vorgesehenen Verlauf. Abbildung 4.1 zeigt die Auswirkungen eines Fokusfehlers am Beispiel des Mercedes-Benz IVI NTG 4.5 High. Laut des Testfalls soll, ausgehend vom Hauptinhalt, die Option «Navi» aktiviert werden. Dazu wird durch die Bedienaktion `UP` die darüber liegende Hauptmenü-

(a) *Vorgesehener Verlauf des Testfalls*

(b) *Auswirkung eines fehlerhaften Fokusverlaufs*

(c) *Anpassung des Testfalls*

Abbildung 4.1.: *Beispiel für die Auswirkung eines Fokus Fehlers*

zeile fokussiert. Gemäß Spezifikation ist vorgesehen, dass der Fokus in diesem Fall auf die Option «Audio» übergeht. Anschließend wird durch ein `LEFT` die Option «Navi» fokussiert, die durch ein `PRESS` aktiviert werden soll. Für den Fall allerdings, dass beim Betreten der Hauptmenüzeile statt des Eintrags «Audio» der Eintrag «Tel» fokussiert wird, schlägt der Testfall fehl. Wenn die vorgesehene Bediensequenz beibehalten wird, kann der Eintrag «Navi» nicht erreicht werden. Um auch den Inhalt der Navi Applikation testen zu können, muss der Testfall angepasst werden. Nach der Registrierung des Fokusfehlers muss geprüft werden, ob die Erreichung des Testziels gefährdet ist. Da der «Navi» Button angezeigt wird und die Fokusverwaltung bis auf die Vergabe des initialen Fokus erwartungskonform funktioniert, ist eine Fortführung des Testfalls möglich, sobald ein weiterer Bedienschritt `LEFT` nach dem Betreten der Hauptmenüzeile eingefügt wird.

Tabelle 4.1 gibt Aufschluss darüber, welche Fehlersymptome zu einem Abbruch der Durchführung führen. Die Markierung «x» bedeutet, dass das Fehlersymptom auf jeden Fall die Fortführung eines der generierten Testfälle verhindert. Die Angabe «(x)» zeigt an, dass ein Abbruch von weiteren Faktoren wie der Natur des Fehlers oder der verwendeten Testschnittstelle abhängt. Fehlersymptome der Logik Kategorie sind die Hauptursache für Testabbrüche. Fehlerhafte oder fehlende Übergänge führen dazu, dass Bereiche der Oberfläche nicht, wie vorgesehen, von dem Testfall erreicht werden können. Um den Fehler zu umgehen müsste ein alternativer Aufruf dieses Bereiches ermittelt werden. Auch PopUps, die unbeabsichtigt oder mit der falschen Priorität angezeigt werden führen dazu, dass sich die folgenden Bedienaktionen nicht wie vorgesehen auf des SUT auswirken. Diese Meldungen müssten vor der Fortführung des Testfalls bestätigt werden oder durch Deaktivierung des ursächlichen Stimulus ausblendet werden. Falls ein PopUp ausbleibt, müssen die Bedienaktionen ignoriert werden, die von der eingeblendeten

Tabelle 4.1.: *Auswirkungen der Fehlersymptomklassen auf die automatische Testdurchführung*

Level 1	Level 2	Level 3	Level 4	Abbruch
Logik	Übergänge	Existenz		x
		Fehlerhaft		x
	PopUps	Existenz		x
		Priorität		x
	Struktur	Komposition		x
		Optionen		x
		GrayOut		x
	Widgets	Spezifika	Existenz	(x)
			Fehlerhaft	(x)
		Fokus		x
Inhalt	Text	Statisch		(x)
		Dynamisch		(x)
	Animationen			(x)
	Symbole			(x)
Design				(x)

Systemmeldung ausgehen. Schwerwiegend für die weitere Durchführung ist auch eine fehlerhafte Struktur der Anzeige. Werden die falschen Widgets zusammengestellt, wirken die Bedienaktionen nicht auf die vorgesehene Art und Weise. Insbesondere ein fehlerhaftes Angebot an Optionen bzw. das fehlerhafte Ausgrauen von Listeneinträgen führt dazu, dass die weitere Durchführung der vorgenerierten Testfälle verhindert wird. Um Fehler dieser Art zu umgehen, müssten Richtungskommandos hinzugefügt oder weggelassen werden, um dennoch die Option zu erreichen, die zur Fortführung des Testfalls notwendig ist. Auch ein fehlerhafter Fokusverlauf führt zu einer unvorhergesehenen Auswirkung der Bedienaktionen auf das SUT. Um eine Fortführung zu ermöglichen, müsste ein Muster des fehlerhaften Fokusverlaufs erkannt werden. Die Auswirkungen der Fehler von spezifischen Eigenschaften von Widgets sind situationsabhängig und können daher nicht pauschal beurteilt werden.

Entscheidend für die Schwere der Auswirkung ist, neben dem Typ des Fehlersymptoms, der Zeitpunkt des Auftretens während des Testfallverlaufs. Schlägt bereits die erste Verifikation eines Testfalls fehl, bleibt ein großer Teil des Testfalls unerreichbar. Handelt es sich um die letzte Verifikation des Testfalls, sind die Auswirkungen weniger gravierend. Außerdem ist entscheidend, ob der Fehler einen zentralen Mechanismus der Oberfläche betrifft. Die Auswirkungen sind schwerwiegender, falls eine große Anzahl an Testfällen von dem Fehler beeinträchtigt wird. Ist beispielsweise das Hauptmenü des Systems nicht verfügbar, können wesentliche Zustandsänderungen nicht provoziert werden.

Aus diesen Beobachtungen lassen sich die folgenden beiden Schlüsse ziehen. Es wurde gezeigt, dass ein großer Anteil der Fehlersymptome dazu führt, dass die automatische Durchführung unterbrochen werden muss. Mit den Logik-Kategorien Übergänge, Pop-Ups, Struktur und Fokus führen, gemäß der Verteilung der Taxonomie (siehe Abschnitt 3.2), 46,8% der Fehlersymptome auf jeden Fall zu einem Abbruch. Bei den verbleibenden Kategorien unterscheiden sich die Auswirkungen situationsabhängig. Je nach Fehlersymptom und Ausführungstechnologie können allerdings auch diese Symptome die weitere Durchführung verhindern. Abhängig davon, ob zentrale Elemente der Oberfläche betroffen sind, kann ein Fehler im schlimmsten Fall die Durchführung der vollständigen Testsuite beeinträchtigen.

Die zweite Aussage bezieht sich auf die Fehlerbehandlung. Aufgrund der direktionalen Oberfläche ist die Testausführung nicht in der Lage, das Fehlersymptom auf Basis pauschaler Angaben zu umgehen. Die Struktur der Oberfläche und die daraus resultierende

Handlungsanweisung sind für einen Algorithmus nicht ersichtlich. Daher wird in dieser Arbeit untersucht, wie der modellbasierte Ansatz bei der Fehlerbehandlung genutzt werden kann. Durch die, von diesem Ansatz generierten detaillierten Referenzinformationen ist es möglich, die Fehlersymptome einzugrenzen. In einem zweiten Schritt sollen die Informationen des Modells genutzt werden, um die Testfälle in einigen Fehlerfällen auf die tatsächliche Situation anzupassen

4.2. Grundkonzept

Diese Arbeit sieht vor, den in Abschnitt 2.3 vorgestellten Prozess der modellbasierten Testautomatisierung um zwei Schritte zu erweitern (siehe Abbildung 4.2). Nach der Identifikation einer Abweichung sollen die vorliegenden Informationen dazu verwendet werden, das Fehlersymptom anhand der Fehlersymptomtaxonomie zu klassifizieren (Schritt 1). Dadurch kann ermittelt werden, ob (a) der Testfall wie vorgesehen weiter durchgeführt werden kann, ob (b) der Testfall abgeändert werden muss, oder (c) eine weitere Durchführung unter keinen Umständen möglich ist. Falls eine Abänderung des Testfalls in Betracht kommt, kann mit Hilfe einer Schnittstelle zum Testfallgenerator eine alternative Bediensequenz errechnet werden (Schritt 2). Dafür wird das Testmodell gemäß der Informationen der Testschnittstelle automatisch angepasst. Im den folgenden Abschnitten 4.2.1 und 4.2.2 werden die Schritte detailliert beschrieben.

4.2.1. Klassifikation

Bei automatisierten Tests wird häufig jede Form der Abweichung zwischen Soll- und Ist-Daten pauschal als Abweichung registriert und in einem Protokoll vermerkt. Dieses Protokoll muss anschließend von Fachleuten analysiert werden. Dadurch kann ein Aufwand entstehen, der mit dem des manuellen Testens vergleichbar ist, weswegen Verfahren gefordert werden, die Auswertung zu unterstützen (siehe Abschnitt 2.2). Dieser Aufwand kann durch den, in dieser Arbeit vorgeschlagenen Verfahren durch umfangreiche Verifikation des Prüflings reduziert werden. Aufgrund der umfangreichen Verifikationsinformationen ist es möglich, das Fehlersymptom weiter einzugrenzen und anhand der in Abschnitt 3.2 vorgestellten Fehlersymptomtaxonomie zu klassifizieren.

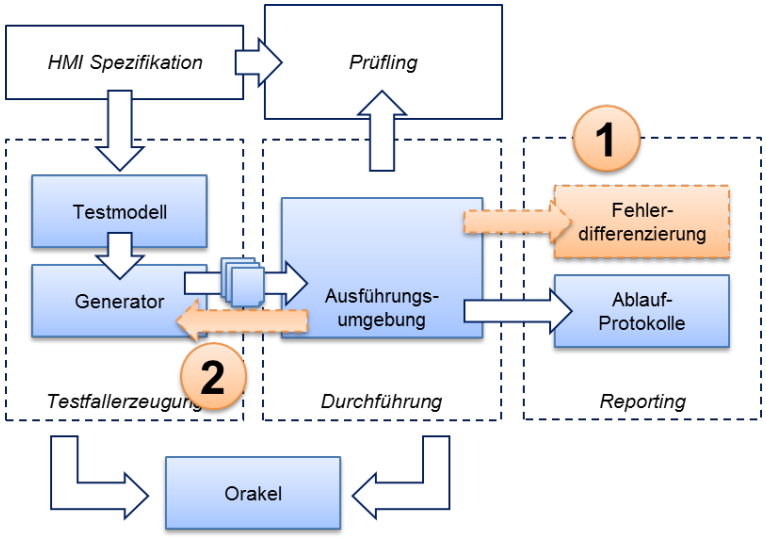


Abbildung 4.2.: *Erweiterung der modellbasierten Testautomatisierung und eine differenzierte Fehlerbehandlung*

Für diese Überprüfung sind zwei Teilschritt erforderlich. In einem ersten Schritt muss eine Beziehung zwischen den Elementen der Referenzdaten und den Ist-Daten der Testschnittstelle hergestellt werden. Abhängig davon, wie stark die Eigenschaften des Elements durch das Fehlersymptom beeinträchtigt wurden, ist die Zuordnung unzuverlässig. Hier muss ein Kompromiss gefunden werden zwischen Fehleranfälligkeit und Robustheit. Einerseits soll die Zuordnung auch im Falle einer Beeinträchtigung möglich sein. Andererseits sollte eine fehlerhafte Zuordnung vermieden werden. Im Fehlerfall ist entscheidend, wie viele Veränderungen diese automatische Überprüfung bewältigen kann.

Ist die Beziehung zwischen den Elementen sichergestellt müssen in einem zweiten Schritt eventuelle Abweichungen festgestellt werden. Dazu wird eine Reihe von aufeinander aufbauenden Überprüfungen der Ist-Informationen durchgeführt. Wie in Abschnitt 3.3 beschrieben, liegen die Referenzdaten in einer hierarchischen Struktur vor. Eine vertikale Überprüfung, stellt sicher, dass die hierarchische Struktur übereinstimmt. Eine anschließende horizontale Überprüfung gewährleistet, dass die Eigenschaften der einzelnen Elemente übereinstimmen. Bei jeder Orientierung muss festgestellt werden, ob auf der Oberfläche des SUT bezüglich der Referenz zusätzliche, fehlende oder falsche Elemente oder Eigenschaften angezeigt werden.

Aus jeder einzelnen dieser Überprüfungen können Fehlerberichte generiert und gemäß der Fehlersymptomtaxonomie klassifiziert werden. Ein großer Vorteil dieses Verfahrens ist, dass Fehlerberichte erstellt werden können, die sowohl von Menschen als auch Maschinen interpretiert werden können. Dadurch ist es auch der Maschine möglich, gleiche Fehlerberichte in Zusammenhang zu stellen, anstatt, wie bisher, jede Abweichung separat auszugeben. Fehler, die das exakt gleiche Symptom im exakt gleichen Zustand mit den exakt gleichen beteiligen Widgets beschreiben, werden als *Ausprägung* (engl. *Occurrence*) des gleichen Fehlerberichts betrachtet. Dadurch kann die Anzahl der Fehlerberichte auf die tatsächliche Anzahl separater Berichte reduziert werden. Diese Überprüfung kann mit sehr viel Aufwand getrieben werden, da eine verallgemeinerbare Logik notwendig ist. Werden zu viele Einzelfälle berücksichtigt, wird die Überprüfung komplex und damit fehleranfällig und weniger wiederverwendbar. Bei der Realisierung ist also abzuwägen, inwiefern die manuelle Überprüfung und die automatisierte Auswertung kombiniert werden sollten.

4.2.2. Partielle Neugenerierung

Nachdem das Fehlersymptom erkannt wurde, können dessen Auswirkungen gezielt adressiert werden. Ziel ist es, den Testfall weiter durchzuführen. Aufgrund des hohen Detaillierungsgrads der differenzierten Fehlerbehandlung kann flexibel auf die Fehlersituation reagiert werden. Mit dem Testmodell liegen alle Informationen vor, die notwendig sind, um die Fehlerbehandlung durchzuführen. Kern des Verfahrens ist eine partielle Neugenerierung des Testfalls auf Basis der Ist-Informationen des fehlerhaften SUT. Die betroffenen Teile des Testfalls werden durch die Bedienaktionen und Verifikationen ersetzt, die eine Fortführung des Tests ermöglichen. Die Vorgehensweise ist analog der ursprünglichen Testfallgenerierung. Grundlage der Generierung ist allerdings nicht der ursprünglich spezifizierte Soll-Zustand sondern der tatsächliche Ist-Zustand des SUT. Ermittelt werden die notwendigen Schritte, um das System wieder in einen definierten Zustand zu versetzen. Die partielle Neuberechnung wird in dieser Arbeit als *Rerouting* bezeichnet.

Das Rerouting kann je nach Fehlersymptom in verschiedenen Ausprägungen durchgeführt werden. Tabelle 4.2 gibt Aufschluss darüber, welche Fehlersymptome durch eine partielle Neugenerierung umgangen werden können. Die Markierung «SC» bedeutet,

Tabelle 4.2.: *Einschätzung der Wirksamkeit der differenzierten Fehlerbehandlung auf Basis der Fehlersymptomklassen*

Level 1	Level 2	Level 3	Level 4	Rerouting
Logik	Übergänge	Existenz		SC
		Fehlerhaft		SC
	PopUps	Existenz		Screen
		Priorität		Screen
	Struktur	Komposition		Screen
		Optionen		Screen
		GrayOut		Screen
	Widgets	Spezifika	Existenz	-
			Fehlerhaft	-
		Fokus		-
Inhalt	Text	Statisch		Screen
		Dynamisch		Screen
	Animationen			Screen
	Symbole			Screen
Design				Screen

dass für die Neugenerierung das gesamte Modell, inkl. Zustandsmaschine und Widget-Programmierung, verwendet werden muss. «Screen» bedeutet, dass die Generierung voraussichtlich auf die Widget-Programmierung beschränkt werden kann. «-» bedeutet, dass der Testfall auch mit einer Neuberechnung nicht weiter fortgeführt werden kann. Handelt es sich bei dem vorliegenden Fehler um eine fehlerhafte Struktur, also beispielsweise um fälschlicherweise angezeigte oder ausgeblendete Elemente auf dem aktuellen Screen, reicht in der Regel eine Anpassung des Testfalls, um den folgenden Screen aufzurufen. Durch das Rerouting werden Bedienaktionen hinzugefügt oder ausgelassen, um das Element auf dem Bildschirm zu erreichen, das für den Aufruf des nächsten Screens benötigt wird. Die Neugenerierung kann auf den Code-Anteil des Testmodells beschränkt werden. Zu beachten ist, dass die partielle Neugenerierung nur dann erfolgreich sein kann, wenn nicht die Option betroffen ist, die für die Fortführung des Testfalls benötigt wird.

Ein Beispiel für einen fehlenden Übergang ist der Test der Audio Applikation des NTG 4.5 High IVI Systems von Mercedes-Benz. Der generierte Testfall sieht vor, die Applikation durch eine Option auf dem Bildschirm aufzurufen. Wenn diese Funktion nicht korrekt implementiert wurde oder der Eintrag nicht verfügbar ist, schlagen alle Testfälle fehl, die diesen Aufruf der Audio Applikation voraussetzen. Basierend auf dem Testmodell kann ein Generierungsalgorithmus ermitteln, ob weitere Möglichkeiten vorgesehen sind, diese Voraussetzung zu erfüllen. So kann beispielsweise für den Aufruf der Audio Applikation eine Kurzwahltaste vorgesehen sein. Durch die partielle Neugenerierung wird dieser zusätzliche Weg erkannt und der Testfall temporär um die Aktivierung der Kurzwahltaste ergänzt. Für eine Neugenerierung müssen neben dem Code-Anteil zusätzlich die State Charts des Testmodells einbezogen werden.

4.3. Testschnittstelle

Modellierung und Generierung der Testfälle ist hauptsächlich abhängig vom Bedien- und Anzeigekonzept des SUT. Die technische Umsetzung spielt eine untergeordnete Rolle. Dies entspricht der Abstraktionsschicht des Abnahmetests (siehe Abschnitt 2.2). Die Ausführung der generierten Testfälle ist abhängig von der Technologie, die zur Realisierung des SUT verwendet wurde. Das Testfallformat verbindet Generierung und Ausführung. Die Testfälle liegen nun in abstrakter Form vor und müssen in ein Format

überführt werden, dass von der Ausführungsumgebung interpretiert werden kann. Es ist nicht Ziel dieser Arbeit ein allgemein anwendbares Testfallformat anzubieten. Von Bedeutung ist lediglich, dass die Testfälle inklusive der Verifikationsinformationen in ein Format gefasst werden müssen.

Wesentlicher Faktor ist die Definition der Testschnittstelle (siehe Abschnitt 2.3). Die Schnittstelle zum SUT dient zwei Zwecken. Zum einen wird auf diese Weise die Stimulation des Prüflings durchgeführt. Bei Blackbox-Testverfahren entspricht diese Stimulation den möglichen Benutzeraktionen. Zum anderen dient die Testschnittstelle dem Auslesen des Systemzustands des SUT zur Verifikation. Mögliche Ansatzpunkt einer Testschnittstelle ist ein Zugriff auf die Software in Form einer Code-Schnittstelle oder die Auswertung der Hardware-Ausgaben durch Bildverarbeitung und BUS Manipulation. Die Testfallinformationen, die durch das hier vorgestellte Verfahren erzeugt werden, liegen in einer ausreichend abstrahierten Form vor, so dass es auf beide Ausführungstechnologien angewendet werden kann. Für die automatische Durchführung der Testfälle müssen die abstrahierten Daten der Testfallgenerierung mit der Ausgabe des SUT zusammengeführt werden.

Dafür muss eine Instanz gegeben sein, die in der Lage ist, die Testfälle zu interpretieren und das SUT entsprechend zu stimulieren bzw. zu verifizieren. Diese Instanz wird als Testrahmen bezeichnet (siehe Abschnitt 2.3). Da die Grundmenge an Bedienaktionen direktionaler Oberflächen sich auf Richtungs- und Kurzwahltasten beschränkt, ist die Zuordnung der Stimulationsinformationen durch definierte Stichwörter verhältnismäßig einfach. Bei der Verifikation muss eine Verbindung zwischen Elementen des SUT und den Referenzinformationen hergestellt werden, um einen Abgleich durchzuführen. Dafür ist eine klare Definition der Datenstruktur z.B. in Form eines Schemas notwendig. Der Testrahmen vergleicht die Ausgabe der Testschnittstelle mit den Referenzdaten und dokumentiert eventuelle Abweichungen in Form von Fehlerberichten.

4.4. Zusammenfassung & Diskussion

In diesem Kapitel wurden die Auswirkungen der Fehlersymptomklassen auf die automatische Testfalldurchführung diskutiert und ein Verfahren zur automatischen Fehlerbehandlung vorgeschlagen. Der aktuell in der Literatur präferierte Abbruch der Testdurchführung

ist in dem hier zugrunde liegenden Entwicklungsszenario nicht praktikabel. Um die Praxistauglichkeit des modellbasierten Verfahrens zu erhöhen, müssen Mechanismen eingerichtet werden, die eine Fehlerbehandlung zur Laufzeit ermöglichen. Mir der differenzierten Fehlerbehandlung (DFB) wurde ein Ansatz vorgestellt, um diese bestehende Einschränkungen der modellbasierten Testautomatisierung auszugleichen. Grundlage des Konzepts ist die Annahme, dass das Modell die notwendigen Informationen bereitstellt, um auf eine Reihe von Fehlersymptomen reagieren zu können. Basierend auf dem fehlerhaften Zustand wird der betroffene Anteil des Testfalls neu generiert und so eine Fortführung ermöglicht.

Dafür muss der Prozess der modellbasierten Testautomatisierung (MBT) lokal automatisiert nachvollzogen werden (siehe Kapitel 3). Die betroffenen Elemente müssen im Testmodell identifiziert und gemäß des vorliegenden Fehlersymptoms abgeändert werden. Anschließend muss das Testziel des betroffenen Testfalls bestimmt und durch Steuerparameter ausgedrückt werden. Gegebenenfalls muss auch der Startzustand der Generierung verändert werden, falls Teile des Testfalls bereits erfolgreich ausgeführt wurden und dieser Fortschritt bei der partiellen Neugenerierung nicht verloren gehen soll. Entweder werden für die Generierung gezielt Teile des Modells oder das gesamte Modell verwendet. Modus und Umfang der partiellen Neugenerierung kann davon abhängig sein, wie aufwendig die Testfallgenerierung als Ganzes oder in Form einer Teilgenerierung ist. Die pauschale Form der Fehlerbehandlung ist eine Anpassung des Modells und eine erneute Generierung der vollständigen Testsuite. Dieses Vorgehen bei jedem Fehlersymptom zu wiederholen kann unter Umständen nicht praktikabel sein. Mit dem vorgeschlagenen Verfahren zur Testfallgenerierung kann der Umfang einer Neugenerierung eingeschränkt werden. Durch die Abstraktionsebene «Screen» können Teile der Strukturfehler ausgeglichen werden. Sollen fehlerhafte Übergänge behandelt werden, wird der eine Zustand als Startzustand und der Zielzustand als Ziel der Neugenerierung angegeben. Das Konzept einer differenzierten Fehlerbehandlung setzt demnach nicht das in Kapitel 3 vorgestellte Verfahren voraus. Dadurch, dass die Abstraktionsebenen an die Fehlersymptomtaxonomie angelehnt sind, wird die Neugenerierung allerdings stark erleichtert.

Durch eine automatische Fehlersymptombehandlung kann nicht jeder Abbruch der Testdurchführung vermieden werden. Steht kein Auslöser für eine benötigte Funktion oder Zustandsänderung zur Verfügung, kann auch mit Hilfe der Differenzierte Fehlerbehand-

lung (DFB) keine Fortführung ermöglicht werden. Auch können die Auswirkungen eines Abbruchs nicht pauschal beurteilt werden. Betrifft der Fehler ein zentrales Element der Oberfläche, das ein großer Anteil der Testfälle bereits zu einem frühen Zeitpunkt während der Durchführung verwenden, ist die Auswirkung des Fehlersymptoms gravierend. Handelt es sich um ein Element, dass lediglich von wenigen Testfällen zu einem späten Zeitpunkt der Durchführung verwendet wird, sind die Auswirkungen zu vernachlässigen. Aufgrund der theoretischen Analyse mit Hilfe der Fehlersymptomtaxonomie ergibt sich, dass dennoch ein großer Anteil an Testfällen, die bisher aufgrund der Notwendigkeit pauschaler Heuristiken abgebrochen werden mussten, durch den Einsatz einer DFB fortgeführt werden können. Es ist davon auszugehen, dass jede Form der DFB die Praxistauglichkeit der MBT erhöht. In Kapitel 5 wird im Rahmen eines Experiments die Wirkungsweise der DFB einer herkömmlichen Durchführung gegenübergestellt.

5

Evaluation

Im Folgenden wird das in Kapitel 3 beschriebene Verfahren zur modellbasierten Testautomatisierung (MBT) und die in Kapitel 4 beschriebene Differenzierte Fehlerbehandlung (DFB) exemplarisch angewandt. Ziel ist es, die theoretischen Annahmen aus Abschnitt 4.4 zur Wirkungsweise der DFB durch eine praktische Anwendung zu überprüfen. Zu Beginn des Kapitels wird die Vorgehensweise der Evaluation anhand der Literatur abgeleitet. Daraufhin wird beschrieben, wie ein geeigneter Prüfling entwickelt und auf Basis der Fehlersymptomtaxonomie mit Fehlern versehen wird. Anschließend werden durch die Anwendung des vorgestellten Verfahrens Testfälle abgeleitet, die mit Hilfe eines im Rahmen der Arbeit entwickelten Testrahmens gegen den Prüfling zur Ausführung gebracht werden. Das Kapitel schließt mit der Vorstellung und Diskussion der erhobenen Ergebnisse.

5.1. Vorgehensweise

Bei der Evaluation wird die Wirkung des Einsatzes des vorgeschlagenen Testverfahrens unter kontrollierten Bedingungen untersucht. Bei Untersuchungsmethoden wird

grundsätzlich unterschieden, inwieweit die Untersuchungsumgebung beeinflusst werden kann [Bor06]. Ist die Möglichkeit gegeben, sämtliche Einflussfaktoren zu messen und zu verändern spricht man von einer *Laboruntersuchung*. Erforscht man ein Phänomen in seinem natürlichen Umfeld ohne direkten Einfluss nehmen zu können, spricht man von einer *Felduntersuchung*.

Die Aussagekraft der Ergebnisse beider Herangehensweisen unterscheidet sich. Der Vorteil einer Laboruntersuchung besteht darin, dass Effekte direkt mit Einflussfaktoren in Verbindung gebracht werden können. Dies wird als hohe *interne Validität* bezeichnet. Bei Felduntersuchung können Wirkungsketten gegebenenfalls nur vermutet werden. Die interne Validität ist demnach gering. Allerdings kann bei einer Laboruntersuchung nicht gewährleistet werden, dass die künstliche Nachbildung der Einflussfaktoren vollständig der natürlichen Umgebung entspricht. Dies gefährdet die so genannte *externe Validität*. Bei einer Felduntersuchung ist dies gegeben. Die externe Validität ist demnach hoch.

Die Auswahl der Vorgehensweise hängt von der beabsichtigten Gesamtaussage ab [Bor06]. Gilt ein Effekt als quasi erwiesen, wird die Durchführung einer Felduntersuchung vorgeschlagen. Ist ein Phänomen noch nicht ausreichend dokumentiert, sollte eine Laboruntersuchung durchgeführt werden. Da es sich bei der differenzierten Fehlerbehandlung um ein neu entwickeltes Verfahren handelt, dessen Wirkungsweise bisher noch nicht erforscht wurde, ist demnach eine Laboruntersuchung erforderlich.

Des Weiteren werden Forschungsmethoden darin unterschieden, ob eine vermutete Wirkung überprüft werden soll (*explanativ*) oder aufgrund von Ergebnissen eine Aussage abgeleitet werden soll (*explorativ*) [WRH$^+$12]. Da die vorliegende Untersuchung auf die Beurteilung abzielt, ob ein vorgeschlagenes Verfahren in vorgegeben Punkten eine Verbesserung darstellt, handelt es sich um eine explanatives Vorgehen.

Die methodisch beste Möglichkeit eines Kausalzusammenhangs einer explanativen Laboruntersuchung ist das *kontrollierte Experiment* [Bor06]. Eine Umgebung, in denen Einflussfaktoren gemessen und bewusst manipuliert werden können, ermöglicht es, das zu untersuchende Phänomen isoliert zu betrachten. Die erwarteten Ursachen für einen Effekt werden als *unabhängige Variablen*, und Faktoren, deren Einfluss ausgeschlossen werden soll, als *konstante Parameter* bezeichnet. Die Ergebnisse werden in quantifizierbarer Form als so genannte *abhängigen Variablen* erfasst. Durch eine systematische Manipulation der unabhängigen Variablen unter Beibehaltung der konstanten Parame-

ter kann eine Veränderung der abhängigen Variablen eindeutig auf den Einfluss der unabhängigen Variablen zurückgeführt werden.

5.1.1. Vergleich der Testverfahren

Um untersuchen zu können, ob das vorgeschlagene Verfahren eine Verbesserung gegenüber dem aktuellen Stand darstellt, ist es notwendig, ein Referenzverfahren zu definieren. Wie in Abschnitt 2.3 beschrieben, wurde der modellbasierte Test direktionaler Bedienoberflächen bislang in der Forschung vernachlässigt. Aus diesem Grund konnte sich noch kein Standard etablieren, der in einer Evaluation als Referenz dienen kann. Daher wird in diesem Experiment ein Referenzverfahren aus den Arbeiten verwandter Themengebiete extrapoliert.

Basierend auf den Ergebnissen der Literaturrecherche ist davon auszugehen, dass Testfälle auch durch andere Verfahren abgeleitet oder manuell geschrieben werden können, die denen in Abschnitt 3.3.4 beschriebenen Vorgaben entsprechen. Der grundlegende Unterschied dieses neuen Verfahrens ist die Möglichkeit einer differenzierten Fehlerbehandlung. Insbesondere die partielle Neugenerierung von Testfällen auf Basis eines fehlerhaften Prüflings ist ein Alleinstellungsmerkmal. Aus diesem Grund besteht der wesentliche Anteil dieses Experiments in der Gegenüberstellung der automatischen Durchführung ohne DFB, im Folgenden als *Referenzverfahren* bezeichnet, bzw. mit DFB, im Folgenden *Verfahrensvorschlag* genannt.

Als konstante Parameter werden bei diesem Experiment die Testfallgenerierung, das Testfallformat und die automatische Systemverifikation vorausgesetzt. Daher verwenden sowohl das Referenzverfahren als auch der Verfahrensvorschlag dieselben generierten Testfälle und Verifikationsinformationen. Für die Stimulation und die Verifikation des Prüflings werden dieselben Algorithmen verwendet. Ebenfalls konstant sind die zu testenden Prüflinge bzw. deren Fehlerbilder. Daher werden sowohl das Referenzverfahren als auch der Verfahrensvorschlag gegen dieselben Prüflinge zur Ausführung gebracht.

Die unabhängige Variable ist die Reaktion der Ausführungsumgebung im Falle einer Abweichung. Beim Referenzverfahren wird die Testfalldurchführung, wie in der Literatur präferiert (siehe Kapitel 2.3), bei einer fehlgeschlagenen Systemverifikation gestoppt, da nicht mehr gewährleistet werden kann, dass sich das System in einem definierten Zustand befindet. Beim Verfahrensvorschlag hingegen wird bei einer Abweichung

die differenzierte Fehlerbehandlung angewendet. Schlägt eine Screen-Verifikation fehl, wird ein Rerouting angestoßen (siehe Abschnitt 3). Das bedeutet, dass auf Basis des tatsächlichen Zustands des Prüflings ermittelt wird, ob bzw. wie der Testfall trotz Fehlersymptomen weiter durchgeführt werden kann.

5.1.2. Kennzahlen & Hypothesen

Zentraler Bestandteil eines Experiments ist die Formulierung von *Hypothesen* [Ant14]. Unter Hypothesen versteht man gerichtete Aussagen, die anhand von Beobachtungen eindeutig bestätigt oder zurückgewiesen werden können [Bor06]. Bei dieser Untersuchung bezieht sich die zu prüfende Aussage auf den Vergleich des vorgeschlagenen Verfahrens und des Standes der Forschung. Aktuell verhindern die Auswirkungen vieler Fehler, dass Bereiche von Oberflächen durch den Test erreicht werden können. Eine direkte Folge davon ist, dass Fehler in diesen Bereichen erst zu einem späten Zeitpunkt während der Testphase gefunden werden können. Diese Einschränkung ist in dem zugrunde gelegten Entwicklungsszenario nicht praktikabel, da die Testphase unter Umständen nicht bis zur vollständigen Fehlerfreiheit ausgedehnt werden kann. Demnach ist die Effizienz des Testverfahrens entscheidend. Dies zeigt sich durch die Anzahl der bis zu einem Zeitpunkt gefunden Fehler, dem so genannten *Fehlerbehebungsverlauf*. Diese Aussage kann in der folgenden Hypothese zusammengefasst werden:

Hypothese 1. *Durch Anwendung der differenzierten Fehlerbehandlung steigt die Anzahl der gefundenen Fehler schneller.*

Zur Bestätigung oder Ablehnung der Hypothese werden die behobenen Fehler aufeinanderfolgender Testdurchläufe aufaddiert, wodurch sich eine Dichtefunktion ergibt. Diese Kennzahl wurde bereits von [BBG12] verwendet, um die Güte zweier Testverfahren zu vergleichen. Bei dem Experiment der vorliegenden Arbeit muss die Steigung des Fehlerbehebunsverlaufs durch Anwendung der DFB größer sein als beim aktuellen Stand der Forschung.

Neben der Entdeckung von Fehlern ist ein weiteres Ziel eines Testverfahrens, die Verlässlichkeit des Prüflings sicherzustellen. Dies zeigt sich durch den Nachweis, dass eine Funktionalität während des Tests wie vorgesehen ausgeführt werden kann. Erneut ist es in dem zugrunde liegenden Entwicklungsszenario notwendig, dass die Verlässlichkeit

möglichst vieler Funktion bereits zu einem möglichst frühen Zeitpunkt in der Testphase nachgewiesen werden kann. In dem Kontext dieser Arbeit entspricht der Nachweis der Verlässlichkeit einer fehlerfrei durchgeführten Verifikation. Dadurch wird gezeigt, dass eine Funktion aufgerufen und ohne Abweichung ausgeführt werden kann. Dies wird im Folgenden als *Testabdeckung* bezeichnet. Das Ziel des vorgestellten Ansatzes kann durch die folgende zweite Hypothese beschrieben werden:

Hypothese 2. *Durch Anwendung der differenzierten Fehlerbehandlung steigt die Anzahl der fehlerfrei durchgeführten Verifikationen schneller.*

Für jeden Testlauf wird die Anzahl der durchgeführten Verifikationen erfasst. Aufgrund des Vorgehens wird durch die Behebung der Fehler die Anzahl der Verifikationen, die keine Abweichung registrieren, mit jedem Testlauf zunehmen. [BBG12] verwendete die Anzahl ausgeführter Testereignisse, um den Testfortschritt zu veranschaulichen. Durch die zusätzliche Einschränkung auf fehlerfreie Verifikation, wird durch dieses Maß die durch den Test gestiegene Verlässlichkeit verdeutlicht. Bei der Anwendung der DFB muss diese Kurve stärker ansteigen als bei dem Referenzverfahren, um die Hypothese zu bestätigen.

Wie in Abschnitt 5.6.1 aufgeführt, werden die Hypothesen dieses Experiments qualitativ untersucht. Um die Beurteilung zu erleichtern, wurden neben den unmittelbar für die Hypothesen relevanten Kennzahlen weitere Daten erhoben. In diesem Experiment wird zusätzlich die Anzahl der pro Testlauf erstellten Fehlerberichte dokumentiert. Inwiefern die Anzahl an Berichten den Entwicklungsprozess beeinflusst bleibt zu untersuchen. Ein möglicher positiver Effekt wäre, dass durch eine große Anzahl an Fehlerberichtinstanzen Fehlersymptome gegebenenfalls auf unterschiedliche Weise reproduziert werden können. Dies kann weiteren Aufschluss über die zugrundeliegende Ursache geben. Außerdem kann auf diese Weise die Behebung eines Fehlers durch zusätzliche Tests sichergestellt werden. Allerdings stellt sich die Frage, ob diese Vorteile im Rahmen des Entwicklungsprozesses tatsächlich realisiert werden können. Es besteht die Gefahr, dass eine große Anzahl an Berichten nicht effektiv verwaltet werden kann. Vor allem, wenn dem Entwickler kein MBT Verfahren zur Verfügung steht. Schließlich werden in diesem Experiment die Anzahl der durchgeführten partiellen Neuberechnungen erfasst, um zu verdeutlichen, wie häufig durch die DFB eine Fortführung des Testlaufs ermöglicht wurde. Die Dokumentation der Dauer der Testläufe zeigt die Auswirkungen der DFB auf die Ressourcenbelegung.

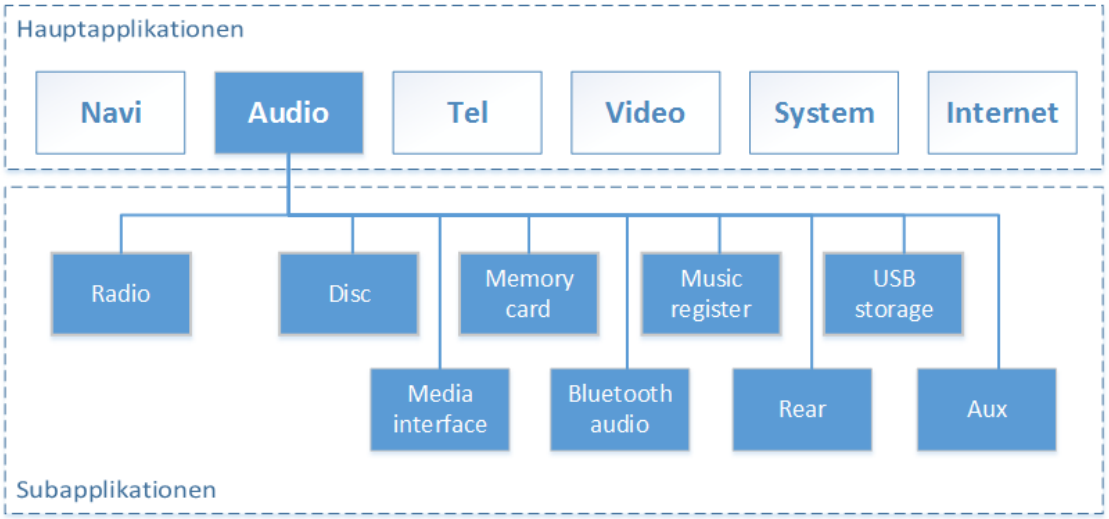
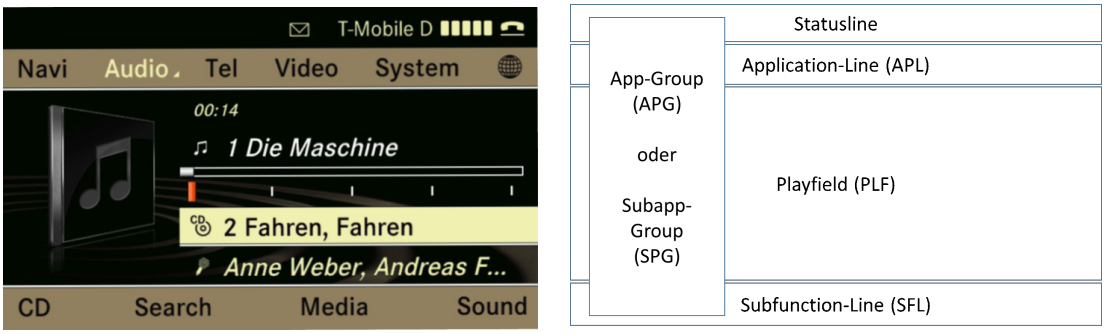


Abbildung 5.1.: *Der Funktionsumfang der NTG 4.5 High*

5.1.3. Untersuchungsgegenstand

In diesem Abschnitt wird beschrieben, welcher Untersuchungsgegenstand in diesem Experiment verwendet wird. Benötigt wird die Implementierung einer Oberfläche, die das Grundprinzip einer direktionalen Bedienoberfläche repräsentiert. Die Software muss geeignet aufgebaut sein, um die Schritte der MBT, wie sie in Kapitel 3 beschrieben sind, exemplarisch durchführen zu können. Wie in Abschnitt 2.1.2 beschrieben, entsprechen einige IVI Systeme komplexen Ausprägungen direktionaler Oberflächen. Insbesondere in den Premium Varianten werden aus Gründen der Ablenkung abgesetzte Bedieneinheiten einem Touchscreen vorgezogen. Um das Experiment realitätsnah zu gestalten, basiert das untersuchte SUT auf einem tatsächlichen Produkt eines solchen IVI System.

Als Vorlage dient ein aktuelles Mercedes-Benz IVI System; die so genannte Neue Telematik-Generation (NTG) 4.5 High. Der Namenszusatz *High* bedeutet, dass, im Gegensatz zur sogenannten *Entry* Variante, der volle Funktionsumfang dieser Telematik-Generation zur Verfügung steht. Der tatsächlich verfügbare Funktionsumfang hängt allerdings, wie bei IVI Systemen üblich, von dem Kontext des Systems im Einsatz ab. Dies beinhaltet die Ausstattung des Fahrzeugs oder der konfigurierte Markt. Zwar ist der gesamte Funktionsumfang in der Software implementiert, aufgrund der externen Bedingungen des Gesamtsystems in der Oberfläche ausgeblendet.

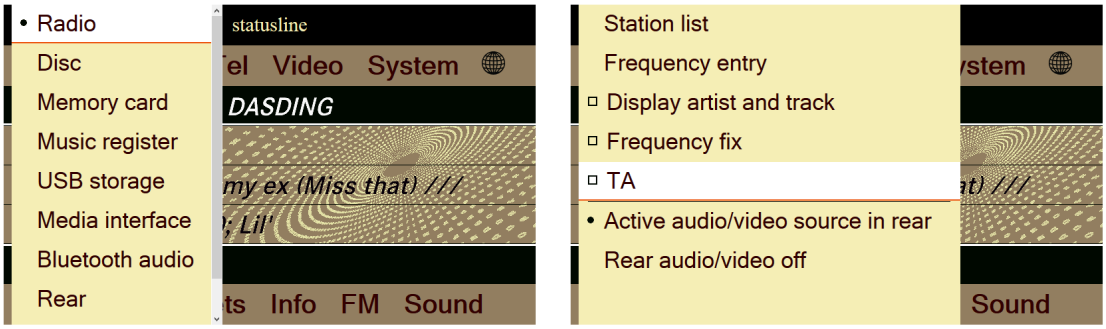


(a) *Originale Darstellung* **(b)** *Schematische Darstellung*

Abbildung 5.2.: *Bedienoberfläche der NTG 4.5 High*

Abbildung 5.1 zeigt den Funktionsumfang des Prüflings. Das Originalsystem sieht eine Unterteilung in *Hauptapplikationen* und *Subapplikationen* vor. Im Original sind die Hauptapplikationen Navi, Audio, Tel, Video System und Internet sowie deren jeweilige Subapplikationen verfügbar. Dieses Experiment wurde auf die Inhalte der Audio-Hauptapplikation begrenzt. Dies beinhaltet alle Funktionen, die zur Bedienung eines Medienplayers sowie des Radios notwendig sind. Zur Auswahl stehen die Medien MP3 CD, Musik CD, Audio DVD, Memory Card (bespielt mit MP3 Dateien), Music Register (entspricht einem internem Speichermedium), USB (bespielt mit MP3 Dateien) und Media Interface (USB Anbindung, in diesem Fall ohne angeschlossenes Gerät) sowie die Wellenbänder FM, MW, SW, LW. Durch diese Belegung ist ein Großteil der im Original verfügbaren Funktionen abgedeckt. Die Ermittlung der exakten Abdeckung hat keine allgemeine Aussagekraft und ist daher in diesem Experiment nicht erforderlich.

Die Oberfläche des Prüflings ist vertikal in vier Bereiche aufgeteilt (siehe Abbildung 5.2). Am oberen Rand des Bildschirms befindet sich die *Statusline*. Im Originalsystem werden dort Systeminformationen wie die Uhrzeit oder der Verbindungsstatus des Telefons angezeigt. Da diese nicht im Fokus des Tests stehen, wurden diese Informationen für das Experiment durch einen statischen Platzhalter ersetzt. Unterhalb der Statuszeile befindet sich das Hauptmenü, die so genannte Application-Line (APL). Hier werden alle verfügbaren Hauptapplikationen angezeigt. Durch die Aktivierung der Optionen wird der Inhalt der entsprechenden Hauptapplikation aufgerufen. Wird dieser Inhalt bereits angezeigt, erscheint stattdessen eine Liste der verfügbaren Unterapplikationen als PopUp Menü, das als App-Group (APG) bezeichnet wird. Bei der Audio-Hauptapplikation sind dies die Radio-Subapplikation sowie die Subapplikationen der verfügbaren Medien (siehe Abbil-



(a) *Beispiel einer APG* **(b)** *Beispiel einer SPG*

Abbildung 5.3.: *Beispiele für PopUp Menüs der NTG 4.5 High*

dung 5.3a). Bei Systemstart wird die Radio Subapplikation der Audio-Hauptapplikation angezeigt. Das Wellenband FM ist vorausgewählt.

Den Hauptteil des Bildschirms nimmt das so genannte Playfield (PLF) ein. Hier werden die Inhalte der Unterapplikationen angezeigt. In Abbildung 5.2a sind dies Informationen zum aktuell abgespielten Musiktitel. Die Inhalte des PLF sind für den Testfokus dieser Arbeit nicht relevant, sondern werden gemäß des in Abschnitt 2.2 vorgestellten Testprozesses durch ein separates Verfahren überprüft. Am unteren Bildschirmrand befinden sich in der so genannten Subfunction-Line (SFL) die kontextabhängigen Optionen. Hier kann der Benutzer beispielsweise das Wellenband wechseln oder die Sound Einstellungen ändern.

Die Auswahl einer Option in der SFL verursacht entweder einen Bildschirmwechsel oder ruft ein PopUp Menü mit untergeordneten Optionen, die so genannte SPG, auf. Abbildung 5.3b zeigt die SPG, die nach der Aktivierung der SFL-Option «Radio» in der Radio-Subapplikation mit Wellenband FM angezeigt wird. Die dort aufgelisteten Optionen verursachen entweder einen Bildschirmwechsel(z.B. Station list: direkte Eingabe der Radio Frequenz), eine Änderung des Systemzustands (z.B. TA: die De-/Aktivierung der Traffic Announcements) oder rufen eine weitere SPG-Ebene auf. Die Liste der Optionen und deren Belegung sind abhängig vom Zustand des Gesamtsystems.

Wie bei vergleichbaren IVI Systemen handelt es sich bei der NTG 4.5 High um eine direktionale Oberfläche, die durch das Zentrale Bedienelement (ZBE) bedient wird (siehe Abbildung 5.4). Das ZBE, ist ein Drehrad, mit dem der Benutzer den Fokus auf dem Bildschirm steuert. Der Benutzer kann das ZBE drehen, in acht Richtungen neigen und

Abbildung 5.4.: *ZBE der S-Klasse (Baureihe 222) aus dem Jahr 2013 [Dai13]*

drücken. Für den Test der Menüstruktur ist bei diesem Experiment nur entscheidend, dass durch das ZBE die Bedienaktionen `UP`, `DOWN`, `LEFT`, `RIGHT` und `PRESS` ausgelöst werden können. Die ersten vier Aktionen verändern die Position der Eingabemarke, durch die Aktion `PRESS` wird die aktuell fokussierte Option aktiviert.

Um derartige Systeme zu testen, müssen ausgehend von einem definierten Startpunkt, alle Optionen aufgerufen werden. Die Herstellung etwaiger Vorbedingungen muss im Testfall enthalten sein. Sollen also beispielsweise die Unteroptionen der Player-Subapplikation mit dem Medium Musik CD getestet werden, muss zuerst die Player-Subapplikation durch Aufruf der Audio APG geändert werden. Anschließend ist es notwendig, aus der SFL heraus alle Ebenen aller verfügbaren SPGs aufzurufen. Die Verifikation des Systemzustands erfolgt jeweils nach jedem Bedienschritt über die auf dem Bildschirm angezeigten Elemente. Dabei wird überprüft, ob die angelegte Bedingungskonstellation korrekt ausgewertet wurde und ob der Fokus den spezifizierten Verlauf nimmt. Insbesondere ist bei der Verifikation auf die Elemente zu achten, die gegebenenfalls ausgeblendet oder deaktiviert werden sollen.

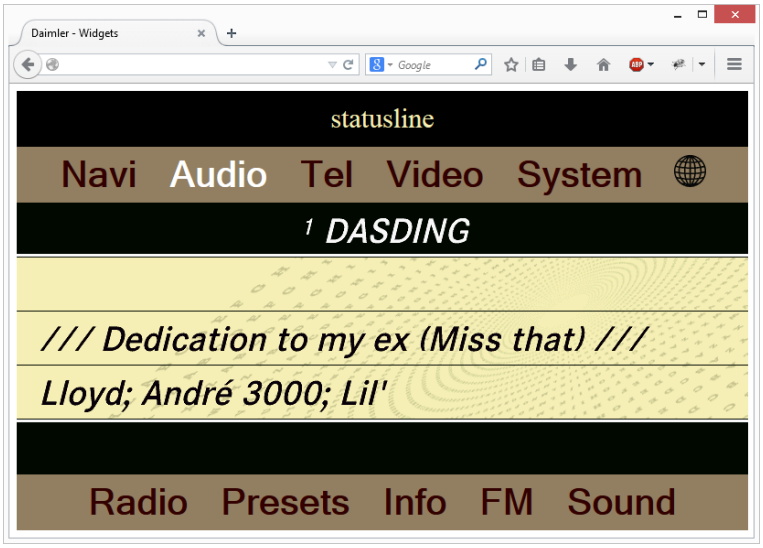


Abbildung 5.5.: *Darstellung des Prüflings in einem Browser*

5.2. Entwicklung des Prüflings

Für dieses Experiment ist es notwendig, einen Prüfling zu implementieren, um den Verfahrensvorschlag anwenden zu können. Der Funktionsumfang entspricht der in Abschnitt 5.1.3 beschriebenen NTG 4.5 High. Der Prüfling ist in JavaScript (JS) und Hypertext Markup Language (XHTML) implementiert. Die Darstellung wird per Cascading Style Sheet (CSS) definiert. Durch die in den JS Dateien beschriebenen Logik wird das Document Object Model (DOM) der XHTML Datei derart manipuliert, dass alle sichtbaren Widgets hierarchisch strukturiert in den `body` Knoten eingetragen werden. Beim Öffnen der XHTML Datei wird in einem Browser die Oberfläche des Prüflings angezeigt (siehe Abbildung 5.5). Die Implementierung ist für den Browser Mozilla Firefox [Moz15] (Version 18) optimiert.

Ein Beispiel der XHTML Datei ist in Listing 5.1 dargestellt. Im `header` Knoten werden der ausgelagerte JS Programmcode geladen, der gemäß Model View Controller (MVC) gegliedert ist: Im *Controller-Anteil* (`widget.js`) wird die Verhaltenslogik der Widgets beschrieben und ist mit den Möglichkeiten von Javascript objektorientiert aufgebaut. Die Klassenstruktur befindet sich im Anhang. Der *View-Anteil* (`domBuilder.js`) erstellt bei Bedarf das DOM auf Basis des Inhalts, der durch den *Model-Anteil* (`init.js`) initialisiert und vorgehalten wird. Beim Öffnen der XHTML Datei wird das System in

einem Browser initialisiert. Anschließend kann der Prüfling mit Hilfe der Pfeiltasten und Enter analog zu den ZBE Eingaben bedient werden.

Listing 5.1: Ausschnitt des HTML DOM des Prüflings

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.
      org/TR/xhtml1/DTD/xhtml1-strict.dtd">
2 <html lang="en" xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
3   <head>
4     <title>Daimler - Widgets</title>
5     <link rel="stylesheet" type="text/css" href="css/style.css"/>
6
7     <script type="text/javascript" src="js/init.js"/>
8     <script type="text/javascript" src="js/widgets.js"/>
9     <script type="text/javascript" src="js/domBuilder.js"/>
10
11  </head>
12  <body>
13    ...
14  </body>
15 </html>
```

Listing 5.2 zeigt einen Ausschnitt des `body` Knotens nach Systemstart des Prüflings. Wie beim Originalsystem werden nach Systemstart die Inhalte der Radio Applikation mit dem Wellenband FM angezeigt. Der `screen` Knoten des DOM beinhaltet eine Reihe von `menu` Tags, die den Menüzeilen der NTG 4.5 entsprechen. Das XHTML beschreibt ausschließlich Aufbau und Inhalt der Anzeige, analog der Referenz Informationen aus Listing 5.5. Da die Darstellung in einer separaten CSS Datei definiert wird, beinhalten die XHTML Elemente zusätzlich zu den, durch die Testschnittstelle vorgegebenen Informationen, Attribute wie `id` oder `class`. Anhand des `id` Attributs wird das Element einem Layout zugeordnet, so dass das Element beispielsweise korrekt positioniert und gegebenenfalls eingefärbt wird. Mögliche Belegungen sind `status`, `apl`, `plf`, `sfl` und `popup-1` (entspr. Statusline, Application-Line, Playfield und den PopUp Menüs App-Group und Subapp-Group). Das `class` Attribut gibt an, ob das Element aktuell fokussiert ist. Alle verbleibenden Informationen werden auch für die Verifikation des Systemzustands verwendet.

Listing 5.2: Beispiel: Ausschnitt des HTML DOM des Prüfligs

```
<body>
  <screen id="screen" widgetid="audio\_radio\_fm">
    <menu id="status" orientation="horizontal" entryStrategy="first"
        looped="false" selectable="false" widgetid="status">
      <entry id="statusline" selectable="true" widgettype="entry" widgetid
          ="statusline">
        <label class="status" widgetid="statusline_label">statusline</
            label>
      </entry>
    </menu>
    <menu id="apl" orientation="horizontal" entryStrategy="second" looped=
        "false" selectable="true" widgetid="apl" class="highlight">
      ...
      <button id="apl_audio" selectable="true" widgettype="button" class="
          active" widgetid="apl_audio">
        <label widgetid="apl_audio_label">Audio</label>
      </button>
      ...
    </menu>
    ...
  </screen>
</body>
```

5.3. Fehlererzeugung

Um die Evaluation durchzuführen, muss der Prüfling kontrolliert mit einer Reihe an Fehlersymptomen versehen werden, ohne dabei voreingenommen die Fehlersymptome zu bevorzugen oder zu vernachlässigen, die durch die DFB behandelt werden können. Eine gebräuchliche Methode, um Programmcode zu verfälschen wird als *Mutationsanalyse* bezeichnet [Ale12, Rom05, ADH+89]. Auf Basis fester Regeln, den so genannten *Mutationsoperatoren*, können Code-Fragmente gezielt abgeändert werden. Bisherige Arbeiten verwenden bei der Mutation häufig einen einzelnen Mutationsoperator, um eine einzelne Codezeile zu verfälschen. Das Ergebnis wird als First Order Mutant (FOM) bezeichnet. Die Beschränkung auf FOM wird durch die «*coupling effect hypothesis*» begründet, die besagt, dass komplexe Fehler an einfache gekoppelt sind [BMV01, Off92]. Dadurch sei gewährleistet, dass ein Test, der alle einfachen Fehler findet, auch einen großen Teil aller komplexen Fehler entdeckt. Diese Hypothese wird so ausgelegt, dass es ausreicht, anstelle von komplexen Fehlern einfache Fehler in ein System einzubringen, um gleichwertige Ergebnisse bei der Überprüfung der Testverfahren zu erzielen.

Der Grund dieser Annahmen ist, dass die Mutationsanalyse in den bisherigen Ansätzen in erster Linie zur Ermittlung der Testgüte verwendet wird. Im Vordergrund steht dabei, ob ein Testverfahren prinzipiell in der Lage ist, den erzeugten Fehler zu identifizieren. Das Ziel dieser Evaluation ist allerdings nicht auf die Ermittlung der Testgüte beschränkt. Untersucht wird nicht nur, ob die erzeugten Fehler durch das Testverfahren gefunden werden können, sondern wie effektiv die Fehlersymptome umgangen werden können. Dadurch soll die Testabdeckung eines Testlaufs erhöht und gegebenenfalls weitere Fehler aufgedeckt werden können. Eine einzelne Mutation ist dafür nicht ausreichend. Es ist notwendig, mehrere Mutationsoperatoren anzuwenden. Auf diese Weise veränderte Software wird als Higher Order Mutant (HOM) bezeichnet. Bei der Erzeugung eines HOM stellt sich die Frage, welche Mutationsoperatoren kombiniert werden sollen. Es ist anzustreben, dass durch die Mutationen des Prüflings die Fehler nachgestellt werden, die auch bei der regulären Entwicklung der Software entstanden wären.

Der zweite Kritikpunkt aktueller Arbeiten ist die fehlende Unterscheidung zwischen Fehlersymptom und Fehlerursache (siehe Abschnitt 2.2). Bei Tests können ausschließlich die Auswirkungen der Fehler an der Oberfläche und nicht deren Ursache im Programmcode entdeckt werden. Dennoch wird bei den Ansätzen davon ausgegangen, dass durch die Anwendung der Mutationsanalyse Fehler entstehen, die denen eines reellen Entwicklungsszenarios entsprechen. Dies gilt allerdings ausschließlich bei der Entwicklung unter den exakt gleichen Voraussetzungen, wie der Technologie und Software-Architektur. Gemäß [MRT09] gilt ein Fehlerbild allerdings erst dann als wirklichkeitsgetreu, wenn es basierend auf einer Taxonomie erstellt wurde. Eine verallgemeinerbare Aussage kann also nur dadurch erreicht werden, wenn die Mutationsanalyse nicht auf Fehlerursachen sondern auf den Fehlersymptomen basiert.

Daher wurde im Rahmen einer Masterthesis, die durch den Autor der vorliegenden Arbeit betreut wurde, eine Methode entwickelt, um das Fehlersymptombild eines HOM der Taxonomie (siehe Abschnitt 3.2) anzunähern [Mar14]. Die Methode sieht zwei Phasen vor: basierend auf der Literatur werden in einer *Analysephase* Mutationsoperatoren zusammengestellt, die auf die vorliegende Software angewendet werden können. Anschließend werden die Code-Stellen identifiziert, die für die Anwendung der Operatoren geeignet sind und ermittelt, welches Fehlerbild aus der Anwendung des Operators auf eine spezifische Code-Stelle resultiert. Die Anwendung eines Mutationsoperators auf diese Code-Stellen ergibt jeweils einen FOM.

Das Ergebnis dieser Analysephase ist eine Zusammenstellung aller möglichen Mutationen und deren Auswirkung auf den Prüfling. Diese Analyse dient ebenfalls dazu, die Mutationen, deren Auswirkungen nicht durch das vorgestellte Testverfahren erkannt werden können, von der weiteren Betrachtung auszuschließen. Dies gilt insbesondere für Mutationen, die die Ausführung des Programms verhindern. Gemäß des zugrundeliegenden Entwicklungsszenarios ist davon auszugehen, dass die Ausführbarkeit der Software durch vorgelagerte Tests sichergestellt wurde. Der Effekt eines Operators ist stark von der Implementierung abhängig, die beispielsweise aufgrund der zu implementierenden Inhalte oder des Programmierstils des Entwicklers variieren kann. Da Programmieraufgaben in aller Regel auf vielfältige Weise bearbeitet werden können, ergeben sich unterschiedliche Angriffspunkte für die Abänderung des Codes. Die Ergebnisse dieser Analysephase gelten demnach ausschließlich für den hier verwendeten Prüfling und haben keine allgemeine Aussagekraft.

In der anschließenden *Konstruktionsphase* werden die FOMs iterativ kombiniert und das neu entstandene Fehlerbild überprüft. Es hat sich herausgestellt, dass das Fehlerbild eines höherwertigen Mutanten nicht zwangsläufig den Teilergebnissen der kombinierten Mutanten entsprechen muss. Dieses Phänomen ist auf Seiteneffekte zurückzuführen. Wie in der Analysephase ist auch während der Konstruktion sicherzustellen, dass der entstandene Prüfling weiterhin ausführbar ist. Andernfalls wird die Mutation verworfen. Für den nächsten Schritt der Kombination wird jeweils der Mutant ausgewählt, der gemäß der Analyseergebnisse am ehesten dafür geeignet ist, das Fehlerbild des Mutanten an die, durch die Fehlertaxonomie vorgegebene Verteilung anzunähern. Das Ergebnis dieser Konstruktionsphase ist eine Reihe an Prüflingen, deren Fehlerbild der Taxonomie angenähert ist. Die Übereinstimmung nimmt mit der Zahl an Kontruktionsiterationen zu. Bei diesem Experiment wurden 10 Mutanten mit jeweils 9 Fehlern versehen.

5.4. Modellbasierte Testfallgenerierung

Aufgrund der Komplexität von Modellierungswerkzeugen und Testfallgeneratoren wird im Rahmen dieser Arbeit auf ein bestehendes Produkt zurückgegriffen. Bei einer Marktrecherche wurden unter anderem die Generatoren Rhapsody ATG [Rha15], MBT Suite [sep15], Conformiq [Con15] und Test Designer [sma15] berücksichtigt. Ziel war es, ein

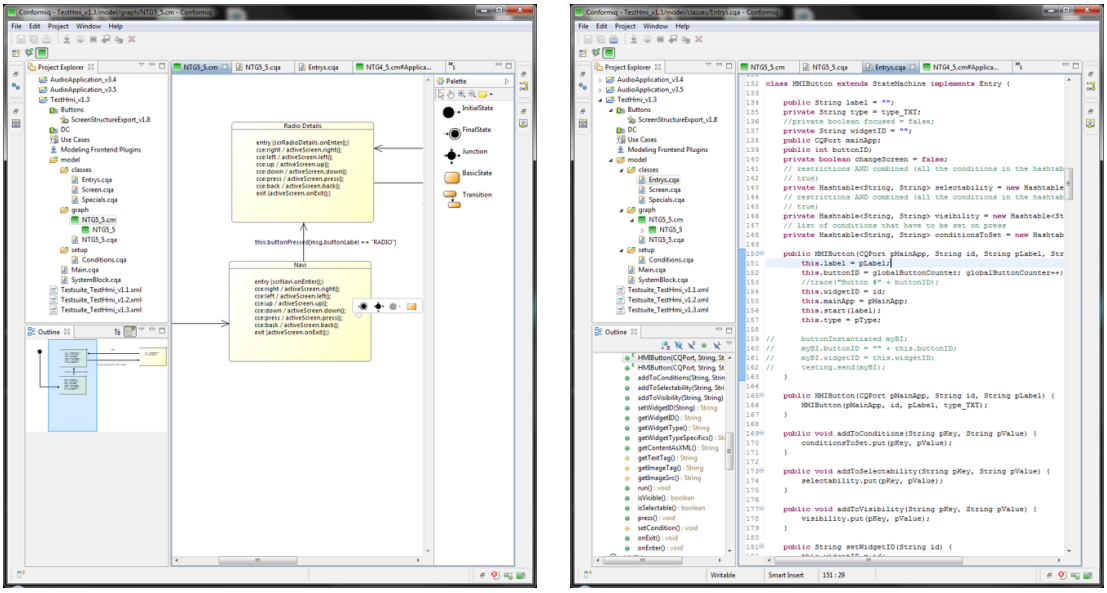
Produkt auszuwählen, dass die Kombination aus Zustandsautomaten und Programmcode (siehe Kapitel 3) als Eingabe akzeptiert sowie Modelle mit der erforderlichen Größe und Komplexität verarbeiten kann.

Basierend auf der Marktrecherche wurde das Werkzeug Conformiq [Con15] für Modellierung und Generierung ausgewählt. Conformiq verwendet Satisfiability Solving für die Herleitung der Testfälle und ist damit in der Lage, Modelle der erforderlichen Größe zu verarbeiten. Andere Generierungsalgorithmen, die auf rekursiver Traversierung basieren, wie im Konkurrenzprodukt MBTSuite, sind bereits bei Modellen mit verhältnismäßig wenigen Elementen überfordert. Die Algorithmen benötigen vergleichsweise lange für die Generierung und erreichen dennoch eine unzureichende Modellabdeckung.

5.4.1. Modellierung

Sowohl die Programmierung als auch die State Chart Modellierung wurden mit Hilfe der Integrated Development Environments (IDE) «Conformiq Designer» in der Version 4.5.1 durchgeführt. Die Oberfläche der IDE ist in Abbildung 5.6 dargestellt. Für die graphische Modellierung stehen die simplen Elemente *InitialState*, *FinalState*, *BasicState*, *Junction* und *Transition* zur Verfügung. Wie durch die UML vorgesehen, können für BasicStates Exit- und Entry-Aktionen sowie interne Transitionen festgesetzt werden. Im Conformiq Designer erfolgt dies durch eine textuelle Beschriftung, die einem spezifischen Schema folgt. BasicStates können zu komplexen Zuständen erweitert werden. Für Transitionen werden weitere Angaben gemäß dem `Event[Guard]/Action` Muster hinzugefügt. Events geben an, welche Ereignisse die Transition auslösen. Guards schränken Bedingungen der Transition ein. Actions geben an, welche Funktionen mit der Auslösung der Transition zusätzlich aufgerufen werden.

Die Programmierung erfolgt über die proprietäre Programmiersprache Qtronic Modeling Language (QML), einem Java Derivat. Um den QML Programmcode bei der Testfallgenerierung zu berücksichtigen und um die Entwicklung von Testmodellen zu erleichtern, wurde der Funktionsumfang im Vergleich zu Java abgeändert. So sind beispielsweise keine anonymen inneren Klassen möglich. Auch Enumerationen sind nicht vorgesehen und wurden in diesem Experiment durch statische Strings ersetzt, um automatische Überprüfungen auf Konsistenz zu ermöglichen. Im Gegensatz zu Java ist es allerdings möglich,



(a) *Modellierung* **(b)** *Programmierung*

Abbildung 5.6.: *Conformiq Designer IDE*

global verfügbare Klassen und Methoden zu definieren. Eine detaillierte Beschreibung der Umfänge von QML sind der Dokumentation zu entnehmen.

Im Rahmen dieser Arbeit wurden die Ausdrucksmittel des Produktes für die Erstellung eines Testmodells angewendet, das den in Kapitel 3 beschriebenen entspricht. Demnach werden Logik und Bedatung der elementaren Bausteine der Bedienoberfläche in Programmcode realisiert. Abbildung 5.7 zeigt einen Auszug der Klassenstruktur des Testmodells. Das Interface `Entry` ist dabei die Grundlage der auf dem Bildschirm sichtbaren Einträge. Es stellt sicher, dass jeder Eintrag anhand eines Strings eindeutig identifiziert werden kann (`widgetID`). Zudem müssen alle Klassen, die dieses Interface implementieren, eine Methode bereitstellen, um die eigene Repräsentation in Form einer XML Struktur auszugeben (`getContentAsXML`). Die Methoden `isVisible` bzw. `isSelectable` geben auf Nachfrage an, ob der Eintrag unter der aktuellen Bedingungsbelegung sichtbar ist bzw. fokussiert werden kann. Die Methoden `onEnter` bzw. `onExit` werden immer dann aufgerufen, wenn der Eintrag den Fokus erhält bzw. den Fokus verliert. Das Interface `Entry` wird von den Klassen `HMIEntry`, `HMIButton` und `HMIMenu` implementiert. Bei `HMIEntry` handelt es sich um die Standardimplementierung des Interfaces, die für Einträge verwendet wird, die ausschließlich der Anzeige

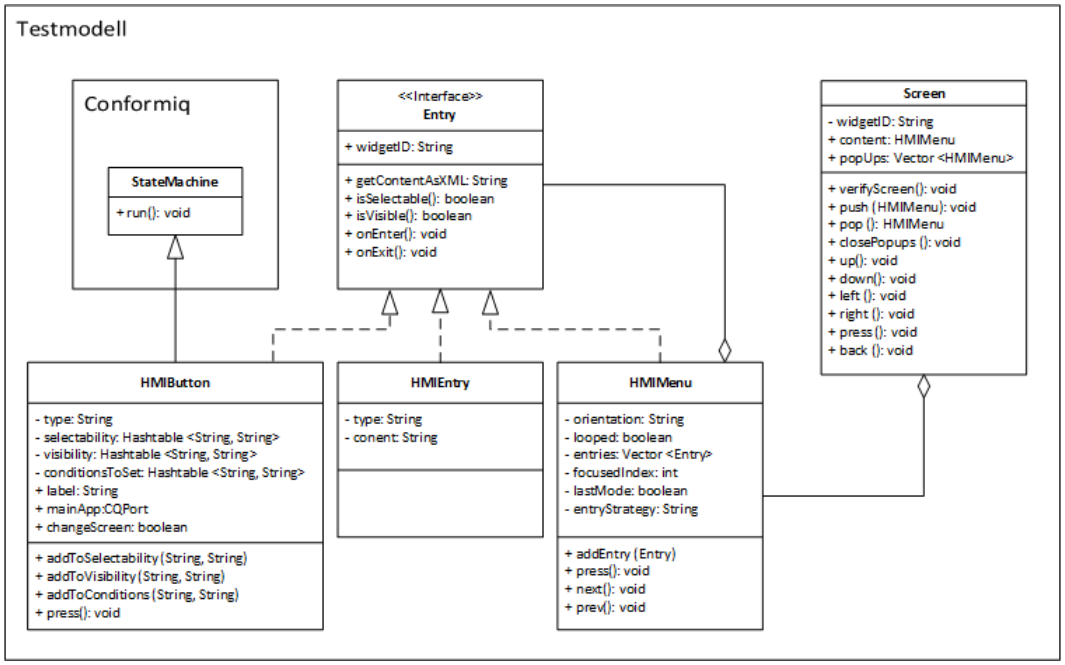


Abbildung 5.7.: *Strukturdiagramm der im Testmodell verwendeten Klassen*

dienen und nicht fokussiert werden können. Die `Screen` Klasse stellt die höchste Abstraktionsebene dar, die ausschließlich durch Programmcode modelliert ist. Hier werden die Widget-Objekte in der Struktur zusammengesetzt, wie sie auf dem Bildschirm zu sehen sind. Eine ausführliche Beschreibung der Klassenstruktur befindet sich im Anhang.

Der Ablauf der Screens wird schließlich im graphischen Anteil der `StateMachine` modelliert. Die unterste Zustandsebene verweist auf einen spezifischen Screen. Zustände mit zugeordnetem Screen Objekt werden im Folgenden als *View States* bezeichnet. Übergeordnete Zustandsebenen dienen der Bedingungsauswertung. Sowohl bei Systemstart als auch bei einem Kontextwechsel wird die Belegung der globalen Bedingungen mit Hilfe der Zustandsmaschine hierarchisch ausgewertet, um den aktuell gültigen View State zu ermitteln. Diese, im Folgenden als *Entscheidungsbaum* bezeichnete Modellierung widerspricht der ursprünglichen Semantik einer Zustandsmaschine, wonach die Zustände selbst die Bedingungsbelegung repräsentieren. Die gewählte Form ist notwendig, da die Bedingungsbelegung jederzeit auch durch global verfügbare Sprungtasten geändert werden kann. So stehen dem Benutzer beispielsweise Tasten zur Anwahl der Hauptapplikationen zur Verfügung, die bei mehrmaliger Betätigung die jeweiligen Unter-

applikationen iterieren. Sobald entweder eine dieser Tasten oder ein Button-Objekt eine Bedingung ändern, die einen Screen Wechsel zur Folge hat, wird durch das Ereignis *changeScreen* eine Selbsttransition auf oberster Ebene der Zustandsmaschine ausgelöst, die eine erneute Auswertung des Entscheidungsbaumes anstößt. Beim Betreten eines View States wird das damit verknüpfte Screen als aktiv registriert. In der Folge werden alle Benutzeraktionen von diesem Screen verarbeitet.

5.4.2. Testfallgenerierung

Für die Testfallgenerierung wird bei Conformiq das Modell per Hypertext Transfer Protocol (HTTP) an einen Computation Server transferiert. Für die Ableitung der Testfälle wird das Modell mit so genannten *Checkpoints* versehen, um den Generierungsalgorithmus zu steuern. Als Checkpoints können sowohl Stellen im Programmcode als auch Elemente des graphischen Modells verwendet werden. Beispiele für Checkpoints im Programmcode sind Methodenaufrufe, die alternativen Zweige einer IF-Anweisung oder die Grenzwerte von Variablenbelegungen. Zusätzlich besteht die Möglichkeit, Code-Stellen als so genannte *ad-hoc Requirements* zu markieren. Als Steuerparameter der Zustandsmaschine steht beispielsweise die Zustands-, Übergangs- oder Pfadabdeckung zur Verfügung. Eine vollständige Auflistung der Steuerparameter ist der Produktbeschreibung zu entnehmen. Eine Konstellation aus Checkpoints wird im Folgenden als *Design Configuration* bezeichnet.

Listing 5.3: Abdeckungsanforderung «Buttons» in der Button Klasse

```
1  public void onEnter()
2  {
3    ...
4
5    cq_begin_region(this.widgetID + " " + this.buttonID);
6      requirement "Buttons";
7    cq_end_region();
8
9    ...
10 }
```

Bei Testfallgenerierung müssen diese Steuerparameter in geeigneter Weise zusammengestellt werden, um den Testfokus abzubilden. Im Fall des in Abschnitt 3.2 beschriebenen

Testfokus «Menüstruktur» bedeutet dies, alle Instanzen der Button Klasse zu fokussieren. In der gewählten Struktur wird dies durch ein ad-hoc Requirement in der `onEnter` Methode der Klasse erreicht (siehe Listing 5.3). Der Aufruf der Methoden `cq_begin_region` bzw. `cq_end_region` stellt sicher, dass der umschlossene Code für jede Kombination des Parameters (`this.widgetID + " " + this.buttonID`) aufgerufen wird. Dies entspricht in dieser Struktur jeder Instanz der Klasse Button. Das Generat entspricht einer Kommunikation zwischen Tester und SUT und wird im Conformiq Designer als Message Sequence Chart dargestellt. Die ausgetauschten Nachrichten entsprechen den im `System Block` definierten und aus Code sowie Zustandsautomat verwendeten Ereignissen. In diesem Fall sind dies Benutzeraktionen und Verifikationsinformationen.

Für jede Design Configuration wird eine Reihe an Testfällen generiert, um die angegeben Checkpoints abzudecken. Die Gesamtheit dieser Testfälle wird als *Testsuite* bezeichnet. Ein Testfall beginnt immer mit einer oder einer Reihe von Verifikationen, um sicherzustellen, dass sich der Prüfling beim Start des Tests in einem definierten Zustand befindet. Im aufgeführten Beispiel werden daher sowohl eine Fokus- als auch eine Screen-Verifikation durchgeführt. Anschließend folgt eine Reihe von Stimulationen, die jeweils erneut durch Verifikationen abgeschlossen werden, um die Systemreaktion zu überprüfen. Jeder Testfall beginnt am definierten Systemstart. Dies erlaubt es, die Reihenfolge bei der Durchführung zu verändern oder Testfälle separat von anderen auszuführen.

5.4.3. Testfallexport

Für den Testfallexport sieht Conformiq die Implementierung eines so genannten *Scripting Backends* vor. Es handelt sich um eine in Java geschriebene Klasse, die das Interface `ScriptBackend` der Conformiq Application Programming Interface (API) implementiert. Die Klasse wird als ausführbares Java Archive (JAR) exportiert und in den Conformiq Designer eingebunden. Im Lieferumfang der Software sind bereits einige Scripting Backends für gebräuchliche Formate wie bspw. TTCN-3 [Eur15] oder Cucumber [Cuc15] enthalten. Aus Gründen der Flexibilität wird in dieser Arbeit allerdings ein behelfsmäßiges Testfallformat verwendet. Es ist nicht Ziel dieser Arbeit ein neues, allgemein anwendbares Testfallformat anzubieten. Alle Informationen, die notwendig sind, die Testschritte und die

System Verifikation durchzuführen, werden in einem Extensible Markup Language (XML) Format gespeichert.

Listing 5.4: Beispiel eines Testfalls

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <TestSuite exportDate="13_06_01_10_16" modelName="AudioApplication_v3.4"
      version="">
3   <TestCase name="Test Case 7">
4     <Verification type="screen">
5       ...
6     </Verification>
7     <Verification contentid="audio_radio_fm_plf_entry" contenttype="img"
          type="focus">img/audio_radio_fm_plf_entry.png</Verification>
8     <Command type="cce:up"/>
9     <Verification contentid="apl_audio" contenttype="txt" type="focus">
          Audio</Verification>
10    <Command type="cce:left"/>
11    <Verification contentid="apl_navi" contenttype="txt" type="focus">Navi
          </Verification>
12  </TestCase>
13  ...
14 </TestSuite>
```

Listing 5.4 zeigt einen Ausschnitt der generierten Testsuite. Die Struktur folgt dem in Abschnitt 3.3.4 beschrieben Schema. Die Art der Verifikation wird im XML durch das Attribut `type` vorgegeben. Beispiele für die Soll-Daten einer Fokus-Verifikation befinden sich in Listing 5.4 u.a. in den Zeilen 7 und 9. Das Attribut `contenttype` gibt an, ob sich die Verifikation auf einen Text oder ein Symbol bezieht. Die `contentid` ermöglicht eine eindeutige Zuordnung zwischen Soll- und Ist-Daten. Listing 5.5 zeigt einen Ausschnitt einer Screen-Verifikation, wie er innerhalb des `verification` Knotens in Listing 5.4 (Zeile 7) eingebettet sein könnte. Hierbei handelt es sich um eine GUI-Map des aktuell auf dem Bildschirm dargestellten Inhalts. Wie aus dem Attribut `widgetid` des `screen` Knotens ersichtlich ist, wird zu diesem Zeitpunkt der Inhalt der Radio Applikation mit dem Wellenband FM angezeigt. Gemäß der darauf folgenden Fokus-Verifikation befindet sich der Cursor auf dem Playfield. In diesem Testfall soll nun durch die Benutzeraktionen `UP` und `LEFT` der Fokus erst auf den Eintrag «Audio» und anschließend auf den Eintrag «Navi» bewegt werden.

Listing 5.5: Beispiel einer Screen Verifikation

```
<screen widgetid="audio_radio_fm">
  <menu entryStrategy="first" looped="false" orientation="horizontal"
      selectable="false" widgetid="status">
    <entry selectable="true" widgetid="statusline" widgettype="entry">
      <label widgetid="statusline_label">statusline</label>
    </entry>
  </menu>
  <menu entryStrategy="second" looped="false" orientation="horizontal"
      selectable="true" widgetid="apl">
    <button selectable="true" widgetid="apl_navi" widgettype="button">
      <label widgetid="apl_navi_label">Navi</label>
    </button>
    ...
  </menu>
  ...
</screen>
```

5.5. Automatische Durchführung

In diesem Abschnitt werden die Aspekte der automatischen Durchführung im Rahmen des Experiments beschrieben. Dazu gehören die Realisierung des Testharnischs, die Erstellung der Fehlerberichte und die Durchführung der DFB. Wie in Abschnitt 2.3 beschrieben, wird die Software, mit Hilfe derer die Testfälle zur Ausführung gebracht werden, als Testharnisch bezeichnet. Die Durchführung beinhaltet die Stimulation und Verifikation des Prüflings gemäß der Vorgaben des Testfalls. Ein Testharnisch besteht aus zwei Komponenten: der Testrahmen übernimmt Ablaufsteuerung und Testauswertung, der Testtreiber übernimmt die Stimulation des Prüflings sowie die Anbindung an die Testschnittstelle. Als Testtreiber für den Prüfling wird die frei verfügbare Software Selenium [Sel15] verwendet.

Zur Stimulation werden die Pfeiltastenbefehle Links, Rechts, Oben, Unten und Enter an das Browserfenster gesendet. Da der Testrahmen in diesem Experiment die differenzierte Fehlerbehandlung initiiert, war es notwendig, diesen Teil des Testharnischs selbst zu implementieren. Die Kernaufgaben des Testrahmens sind Import und Interpretation der generierten Testfälle. Beim Start der Testdurchführung veranlasst der Testrahmen den Start des Prüflings durch den Testtreiber. Anschließend werden die Testfälle bzw. die darin beschriebenen Testschritte iteriert und die entsprechenden Anweisungen an den Testtreiber weitergeleitet. Der aktuelle Status der Durchführung wird an der Oberfläche

angezeigt. Ist im Testfall eine Verifikation vorgesehen, führt der Testrahmen den Abgleich zwischen den im Testfall hinterlegten Soll-Daten und den durch die Testschnittstelle ausgegebenen Ist-Daten durch. Aufgrund der Isomorphie von Ist- und Soll-Daten beschränkt sich der Abgleich auf einen Baumvergleich. Die Soll-Daten werden im Folgenden als Kontrollbaum, die Ist-Daten als Testbaum bezeichnet.

5.5.1. Verifikation des Prüflings

Wie durch die generierten Testfälle vorgesehen, erfolgt die Verifikation entweder auf Basis des aktuell fokussierten Elements oder aller auf dem Bildschirm angezeigten Widgets (Siehe Abschnitt 3.3.4). Durch die Fokus-Verifikation wird die korrekte Cursor-Bewegung innerhalb eines Screens sichergestellt. Dafür wird im DOM des Prüflings das Element gesucht, dessen Attribut `class` auf `highlight` gesetzt ist. Die Verifikation erfolgt anhand des Inhalts dieses Elements. In diesem Experiment können fokussierte Widgets einen Text und/oder ein Symbol beinhalten, wobei zur Verifikation der Text höher priorisiert wird. Sollte ein Widget sowohl einen Text als auch ein Symbol beinhalten, wird nur der Text überprüft.

Um eine Screen-Verifikation durchzuführen, wird der Teilbaum unterhalb des HTML Body Tags des DOM ausgelesen und mit der im Testfall eingebetteten Referenzstruktur verglichen. Um die Elemente aus Ist- und Soll-Daten vergleichen zu können, wurden Widgets sowohl bei der Testmodellierung als auch bei der Implementierung des Prüflings mit der gleichen Identifikationsbezeichnung, der so genannten `widgetID`, versehen. Auf diese Weise können äquivalente Elemente einfach identifiziert werden. Der Baumvergleich wird in diesem Experiment mit Hilfe des Testframeworks <XMLUnit>[1] in einer dreistufigen Überprüfung durchgeführt. Eine mehrstufige Überprüfung wird in der Literatur als *Oracle Pipeline* bezeichnet [RMD10].

In einem ersten Schritt wird sichergestellt, dass für jedes Widget in beiden Bäumen ein entsprechendes Element innerhalb des jeweils anderen Baumes existiert. Kann für ein Element in einem der Bäume kein entsprechendes Gegenstück gefunden werden, handelt es sich gemäß der Fehlersymptomtaxonomie um einen Fehler der *Struktur* (siehe Abschnitt 3.2). Elemente ohne Äquivalent werden aus den entsprechenden Bäumen entfernt. Dadurch wird vermieden, dass aufgrund des erkannten Fehlers die fehlerhafte

[1] xmlunit.sourceforge.net/

Indizierung der nachfolgenden Einträge ebenfalls als Fehler registriert wird. Nachdem sichergestellt wurde, dass für jedes Element der Bäume ein Äquivalent existiert, kann die Reihenfolge der Elemente abgeglichen werden. In einer dritten Phase werden die Eigenschaften der Elemente überprüft: jedes Element des Testbaums muss mindestens die gleichen Attribute und deren Belegung aufweisen wie das Äquivalent im Kontrollbaum. Für die HTML Darstellung sind bei den Elementen des Testbaums weitere Attribute wie bspw. id für das CSS Layouting notwendig.

Die Testausführungsumgebung muss ermöglichen, den in Kapitel 3 beschriebenen Testprozess durchzuführen. Demnach wird eine Software iterativ bis zur definierten Fehlerfreiheit getestet. In einem Testlauf werden die generierten Testfälle zur Ausführung gebracht und Abweichungen zwischen Soll- und Ist-Daten festgestellt, die in einer nachfolgenden Entwicklungsphase behoben werden. Von der Software entsteht auf diese Weise nach jeder Testphase ein neuer Entwicklungsstand. Testablauf und Testfortschritt werden mit Hilfe von Testplänen und Runs verwaltet. In einem Testplan werden alle Informationen eines Tests zusammengefasst. Bestimmendes Element eines Testplans ist die Testsuite, die auf einen Prüfling angewendet wird. Die Verwaltung der iterativen Fehlerbehebung erfolgt bei der hier implementierten Testausführungsumgebung durch sogenannte Runs. Wird ein neuer Entwicklungsstand hinzugefügt, wird automatisch ein neuer Run angelegt. Auf diese Weise können die Testfälle beliebig häufig gegen die aktuelle Version des Prüflings ausgeführt werden, ohne die Ergebnisse eines früheren Runs zu verfälschen.

5.5.2. Differenzierte Fehlerbehandlung

Im Falle einer Abweichung werden mit Hilfe einer *Report-Factory* entsprechende Fehlerberichte erstellt werden. Einzelne Teile der Verifikation sind Fehlersymptomkategorien zugeordnet. Tritt beispielsweise beim `looped` Attribut eines Menüs eine Abweichung auf, wird der Bericht als fehlerhaftes Fokusverhalten kategorisiert. Ist ein `selected` Attribut eines Buttons falsch gesetzt, wird der Bericht als fehlerhaftes Ausgrauverhalten eingeordnet. Liegt eine Abweichung bei einem `marked` Attribut vor, handelt es sich um fehlerhaftes Widget-Verhalten. Für jede Fehlersymptomkategorie ist ein Mustertext mit Platzhaltern hinterlegt. Diese Berichte geben Auskunft, welches Element auf welchem Screen auf welche Weise betroffen ist. Dadurch sind die Fehlerberichte auch für

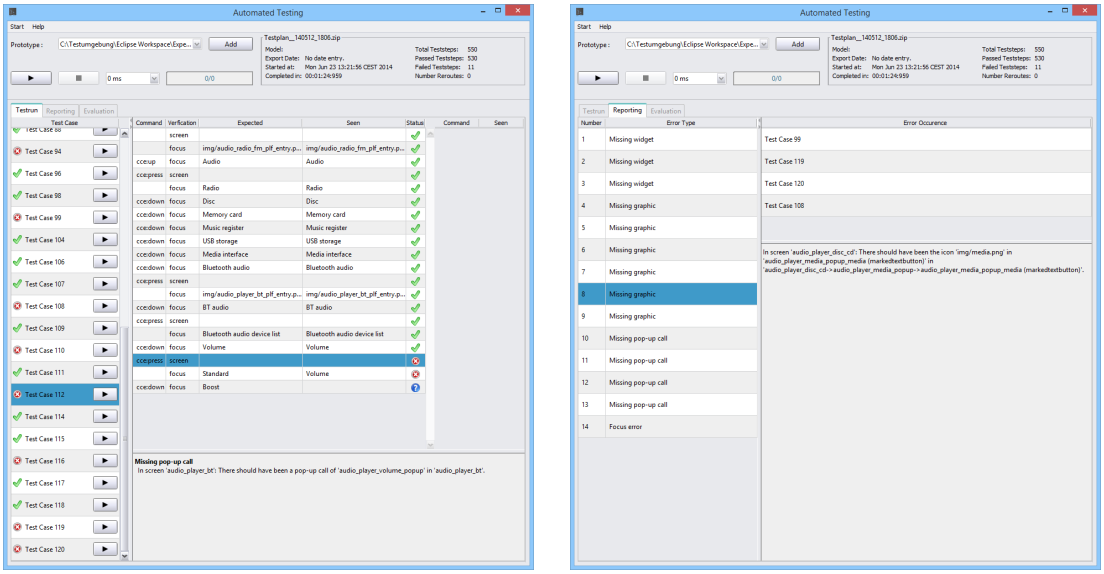
Menschen nachvollziehbar. Durch die umfassenden Informationen, die durch dieses Verfahren vorliegen, kann zudem die mehrfache Erstellung desselben Fehlerberichts vermieden werden. Dafür sieht die Datenhaltung der Reports die Nennung von Fehlerberichtsinstanzen, so genannten *Occurences* vor. Wird ein Fehlerbericht generiert, der exakt einem Bericht entspricht, der bereits registriert ist, wird kein identischer Bericht erzeugt, sondern dem bereits existierenden ein Occurence Eintrag hinzugefügt.

Da der Testfallgenerator eine partielle Neugenerierung nicht vorsieht, wird, abweichend von dem beschriebenen Verfahren, nicht das ursprüngliche Modell verwendet. Die Realisierung der entsprechenden Schnittstelle war aufgrund geschlossener Quellen nicht möglich. Daher war es notwendig, für die Berechnung des Rerouting eine separate Referenzimplementierung umzusetzen. Diese Software ist in der Lage, aufgrund der Screen-Struktur-Informationen und dem vorliegenden Testfall ein kleines Rerouting durchzuführen. Dafür identifiziert die Software innerhalb des Testfalls den Eintrag, der als nächstes aktiviert werden muss, um den Testfall fortzusetzen und sucht, falls vorhanden, den Eintrag in den Ist-Daten der Testschnittstelle. Anschließend errechnet der Algorithmus, basierend auf den Strukturinformationen die Bediensequenz, die den Fokus vom aktuell fokussierten Eintrag zum ermittelten Zieleintrag navigiert. In diesem Fall wäre es auch gegebenenfalls möglich gewesen, ein großes Rerouting durchzuführen und so den vollen Umfang der differenzierten Fehlerbehandlung zu überprüfen.

Vor dem Start des Tests kann in der Oberfläche des Testrahmens eingestellt werden, ob die DFB angewendet werden soll. Ist dies der Fall, wird bei einer fehlgeschlagenen Verifikation ein kleines Rerouting angestoßen (siehe Abschnitt 4.1). Dabei wird die Bediensequenz ermittelt, wie von dem aktuell fokussierten Eintrag zu dem Eintrag navigiert werden kann, der im Laufe des Testfalls als nächstes aktiviert werden muss. Anschließend wird der neu errechnete Pfad mit dem Testfall abgeglichen. Bei einer Übereinstimmung verursacht der registrierte Fehler keinen Abbruch. Der Test kann unverändert fortgeführt werden. Bei einer Abweichung wird die im Testfall vorgesehene Bediensequenz durch die neu ermittelte ersetzt.

5.5.3. Oberfläche des Testrahmens

Die Oberfläche des im Rahmen der Arbeit implementierten Testrahmens wird in Abbildung 5.8 dargestellt. Oben links befindet sich die Ablaufsteuerung. Durch das DropDown



(a) *Ablaufsteuerung und Statusübersicht*

(b) *Erzeugte Fehlerberichte*

Abbildung 5.8.: *Oberfläche des im Rahmen der Arbeit implementierten Testrahmens*

Menü «Prototype» kann aus den bisher angelegten Runs ausgewählt werden. Durch «Add» wird unter Angabe der neuen Version des SUT ein neuer Run hinzugefügt. Bei «Play» und «Stop» wird der aktuell ausgewählte Run gestartet, bzw. ein laufender Run gestoppt. Durch das DropDown-Menü rechts neben «Stop» kann eingestellt werden, ob bzw. wie stark die Durchführung verzögert werden soll. Dies ist hilfreich, um den Testablauf schrittweise nachvollziehen zu können. Der Fortschrittsbalken gibt an, welcher Anteil der Testschritte bereits durchgeführt wurde. Rechts oben werden die Testschritte des aktuell ausgewählten Testfalls angezeigt und weitere Detailinformationen des gewählten Testplans angezeigt. Dies sind bspw. der Name des Modells, das für die Generierung der Testsuite verwendet wurde, sowie Kennzahlen des letzten Testlaufs.

Den Großteil der Oberfläche nehmen drei Kartenreiter ein: Testfallübersicht, Fehlerberichte und die Ergebnisübersicht. Die Testfallübersicht (Abbildung 5.8a) teilt sich in drei Master-Detail Ansichten. Auf der linken Seite werden alle Testfälle der Testsuite aufgelistet. Die «Play» Funktion bei jedem der Testfälle ermöglicht eine separate Durchführung, um dessen Verlauf nachvollziehen oder überprüfen zu können. Bei Auswahl eines Testfalls werden rechts die vorgesehen Testschritte angezeigt. Jede Zeile der Testschritt-Übersicht repräsentiert eine Paarung aus einer Bedienaktion und der

dazugehörigen Verifikation. Wie in den Testfällen vorgesehen, werden bei einem Screen-Wechsel sowohl eine Screen- als auch eine Fokusverifikation durchgeführt. Jeder Testfall beginnt mit einer Verifikation um sicherzustellen, dass sich das System beim Start des Tests in einem definierten Zustand befindet. Während der Durchführung des Tests werden der aktuell bearbeitete Testfall und dessen aktuell bearbeiteter Testschritt blau hinterlegt. Nach der Bearbeitung wird das Ergebnis durch ein Symbol visualisiert. Ist der Status eines Schrittes unbekannt, wird dieser mit einem blauen Fragezeichen markiert. Dies ist die Standardmarkierung der Testschritte. Ist der Testschritt fehlgeschlagen bzw. erfolgreich, wird die Markierung in ein rot hinterlegtes Kreuz bzw. einen grünen Haken geändert. Unterhalb der Testschrittübersicht befindet sich die Ergebnisanzeige des aktuell ausgewählten Testschritts. Hier werden gegebenenfalls die erstellten Fehlerberichte angezeigt. Die Berichte werden gemäß ihrer Fehlerklasse zusammengefasst.

Der zweite Karteireiter ruft die Übersicht aller Fehlerberichte auf (siehe Abbildung 5.8b), um auf alle Fehlerberichte des aktuellen Runs zugreifen zu können. Auf der linken Seite werden alle Fehlerberichte aufgelistet. Bei Auswahl eines Berichts werden rechts weitere Details eingeblendet. Oben werden als «Error Occurences» die Testfälle ausgegeben, bei denen das Fehlersymptom beobachtet wurde. Darunter wird eine textuelle Zusammenfassung des Fehlerberichts angezeigt. Im dritten Kartenreiter wird tabellarisch zusammengefasst, welche Fehlersymptome in allen Runs des Testplans erkannt wurden.

Falls vorhanden, wird bei Start des Testharnischs der zuletzt durchgeführte Test geladen. Ist ein Testplan verfügbar, werden die von der Testsuite vorgegebenen Testfälle sowie die entsprechenden Testschritte und gegebenenfalls deren vorherige Ergebnisse angezeigt. Über das Menü «Datei» kann unter Angabe einer Testsuite und eines Prüflings ein neuer Testplan angelegt werden. In diesem Menü kann außerdem eingestellt werden, ob im Fehlerfall ein Rerouting durchgeführt werden soll. Diese Option ist standardmäßig aktiviert und muss geändert werden, falls der Referenzprozess nachgestellt werden soll. In Menü «Datei» kann zudem eingestellt werden, ob beim Start des Prüflings das Firebug Plugin[2] für den Mozilla Browser geladen werden soll. Mit Hilfe dieses Plugins kann während der Laufzeit das DOM des Prüflings und damit die Testschnittstelle manuell aufgerufen werden. Diese Funktion ist vor allem bei der Fehlerbehebung nützlich. Da das Plugin den Start des Prüflings stark verzögert, ist die Option standardmäßig deaktiviert.

[2] https://addons.mozilla.org/de/firefox/addon/firebug/

5.5.4. Versuchsablauf

In diesem Experiment wird der Abnahmetest des in Abschnitt 2.2 beschriebenen Entwicklungsprozesses exemplarisch nachvollzogen. Dafür wurde ein Prüfling in JS umgesetzt (siehe Abschnitt 5.2). Mit Hilfe der in Abschnitt 5.3 beschriebenen Methodik wurden zehn Mutanten der neunten Stufe erstellt. Die Mutationen wurden in der Reihenfolge, in der sie kombiniert wurden, fortlaufend nummeriert. Um die Behebung des Fehlers zu erleichtern, wurde bei jeder veränderten Code Stelle durch einen Kommentar angegeben, um welche Mutation es sich handelt und wie der Fehler behoben werden kann.

Diese Prüflinge werden nun mit Hilfe der MBT iterativ bis zur Fehlerfreiheit getestet. Die Fehlerfreiheit ist dann erreicht, sobald die generierten Testfälle vollständig und ohne eine Abweichung zwischen Soll- und Ist-Daten durchgeführt werden können. Für jeden der Prüflinge wird sowohl das Referenzverfahren als auch der Verfahrensvorschlag durchgeführt (siehe Abschnitt 5.1). Der erste Schritt bei beiden Verfahren ist es, unter Angabe derselben generierten Testsuite und der XHTML Datei jedes der Prüflinge Testpläne anzulegen. Mit Erstellung eines Testplans wird automatisch der erste Run angelegt. Anschließend wird der erste Lauf gestartet. Im Falle des Referenzverfahrens ohne Rerouting, im Falle des Verfahrensvorschlags mit Rerouting. In beiden Fällen werden nach Beendigung jedes Testlaufs diejenigen durch die Mutation veränderten Code-Fehler behoben, die für die Fehlerberichte des Testlaufs verantwortlich sind.

Um den Testprozess zu unterstützen, werden in der Oberfläche des Testharnischs diejenigen Testberichte farblich markiert, die bereits durch den vorangegangenen Testlauf erkannt wurden. So kann sichergestellt werden, dass genau die für die Fehlerberichte eines Testlaufs ursächlichen Code-Fehler behoben wurden. Die Reihenfolge der Fehlerbehebung ist unter Umständen ausschlaggebend für anderslautende Fehlerberichte. Um bei diesem Experiment reproduzierbare Ergebnisse zu gewährleisten, wurden die Fehlerberichte nach deren Auftreten im Verlauf der Testsuite sortiert und dann der Reihe nach bearbeitet. Ein Testlauf gilt dann als beendet, sobald kein Fehlerbericht mehr reproduziert werden kann. Dieses Verfahren wird für jeden Prüfling bis zur Fehlerfreiheit wiederholt.

5.6. Ergebnisse & Diskussion

Im Folgenden werden Ergebnisse des Experiments vorgestellt. Der Abschnitt beginnt mit einer Erörterung möglicher Einschränkungen der Allgemeingültigkeit und wie diese adressiert werden. Anschließend werden die in Abschnitt 5.1 hergeleiteten Kennzahlen vorgestellt und diskutiert.

5.6.1. Einschränkungen der Allgemeingültigkeit

Es stellt sich die Frage, wie die Allgemeingültigkeit einer Evaluation in diesem Rahmen einzuschätzen ist, da sich das Experiment auf einen spezifischen Anwendungsfall bezieht. Die untersuchten Funktionen sowie Bedien- und Anzeigekonzepte sind auf diejenigen begrenzt, die in dem Prüfling verwendet werden. Der Einfluss dieser Einschränkung auf den Testfokus «Menüstruktur», alle Optionen aufzurufen und dafür alle notwendigen Bedingungen automatisch anzulegen, ist allerdings gering. Die für direktionale Oberflächen maßgeblichen Konzepte sind in dem gewählten Prüfling umgesetzt. Die Interaktion erfolgt durch Benutzeraktionen, mit Hilfe derer die Position einer Eingabemarkierung innerhalb von Listen verändert werden und eine selektierte Option aktiviert werden kann. Dem vorgestellten Prüfling liegt ein Repertoire aktuell üblicher Widgets wie beispielsweise CheckBoxen und RadioGroups zugrunde. Die Möglichkeit, auch weitere Widget-Konzepte einzubeziehen, ist gegeben. Der dedizierte Test spezifischer Widgeteigenschaften steht allerdings nicht im Fokus dieser Arbeit.

Auch die für eingebettete Systeme als typisch angenommene wechselseitige Abhängigkeit angeschlossener Regelsysteme trifft in dem gewählten Szenario zu. Der Prüfling wurde der Oberfläche eines produktiven IVI Systems nachempfunden. Der ausgewählte Ausschnitt der Funktionalität des Originals dient zur Anzeige und Steuerung eines Medienplayers sowie eines Radios. Die Verbindung mit dem Gesamtsystem, wie die angeschlossenen Medien, wird durch Parameter simuliert, die Sichtbarkeit bzw. Verfügbarkeit der angebotenen Optionen beeinflussen.

Ein weiterer Einflussfaktor ist die Entwicklung des Prüflings. Sowohl die Implementierungstechnologie als auch die individuelle Vorgehensweise der Entwickler sind spezifisch und können die Allgemeingültigkeit der Aussagen gefährden. Aus diesem Grund wurden

in dieser Arbeit bewusst die Fehlersymptome und nicht deren Ursachen im Quellcode in den Fokus gesetzt. Die Symptomklassen und deren Auftrittswahrscheinlichkeit wurden im Rahmen einer Studie unabhängig von Umsetzungstechnologie und Prozessausprägung festgestellt (siehe Abschnitt 3.1). Es ist also davon auszugehen, dass diese Fehlersymptome grundsätzlich bei direktionalen Oberflächen eingebetteter Systeme dieser Art zu beobachten sind.

Schließlich schränkt die Anzahl der untersuchten Mutanten sowie die Anzahl der eingebrachten Fehler die Allgemeingültigkeit der Ergebnisse ein. Aufgrund des zeitlichen Aufwands ist die Stichprobe auf zehn Mutanten der neunten Stufe beschränkt. Um die Aussagekraft der Ergebnisse zu erhöhen, werden die erhobenen Kenndaten aus zwei Perspektiven betrachtet. In einem ersten Teil werden die Zahlen der Mutanten als Ganzes analysiert, indem das arithmetische Mittel der jeweiligen Kennzahlen gebildet wird. Dieser Abschnitt wird im Folgenden als *Pauschalbetrachtung* bezeichnet. Die gewonnenen Erkenntnisse sind allerdings als Tendenzen und nicht als verallgemeinerbar zu werten. Zusätzlich wird für jede Kennzahl eine Einzelfallbetrachtung durchgeführt. Anhand der Ergebnisse werden diejenigen Mutanten identifiziert, bei denen der Testverlauf durch die Anwendung der DFB in besonderem Maße auswirkt. Der Mutant, dessen Testverlauf die DFB besonders positiv beeinflusst, wird als *Bestfall* bezeichnet. Der Mutant, bei dem der Testverlauf sich im Sinne der Hypothesen verschlechtert, wird als *Schlechtfall* bezeichnet. Aufgrund der Kombination von Pauschal- und Einzelfallbetrachtung wird sowohl der allgemeine Trend als auch dessen Varianz berücksichtigt.

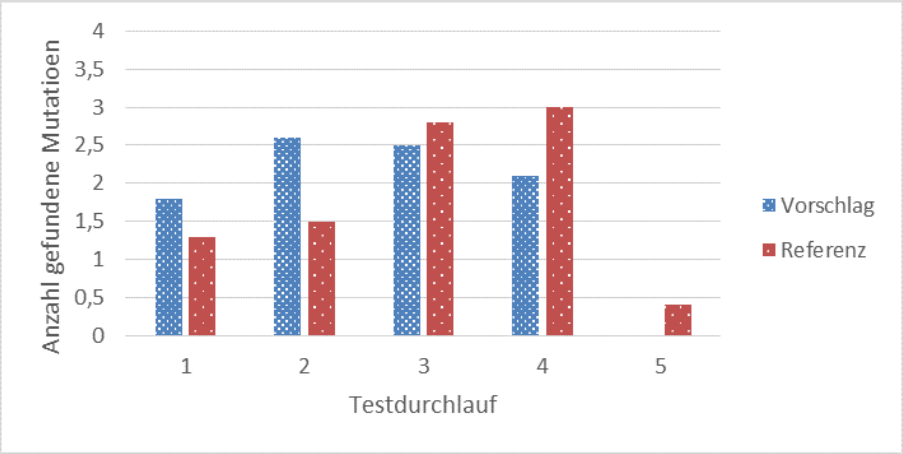
Im folgenden Abschnitt werden die Auswirkungen von Referenzverfahren und Verfahrensvorschlag auf die abhängigen Variablen vorgestellt und diskutiert. Während der Testdurchführung wurden die Anzahl der Fehlerberichte bzw. Fehlerberichtsinstanzen, die Dauer der Testdurchläufe, die Anzahl der Verifikationen pro Testlauf sowie die Anzahl der durchgeführten Reroutes im Fehlerfall automatisch erfasst. Der Tester dokumentiert zusätzlich für jeden Testlauf und jeden Mutanten, welche Mutationen in welchem Testlauf behoben wurden. Wie in Abschnitt 5.6.1 beschrieben, wird für jede erhobene Kennzahl sowohl eine Pauschalbetrachtung als auch eine Einzelfallbetrachtung durchgeführt. Für diese Einzelfallbetrachtung wurden die Mutanten mit den Nummern 315 als Bestfall und 319 als Schlechtfall ausgewählt.

5.6.2. Fehlerbehebungsverlauf

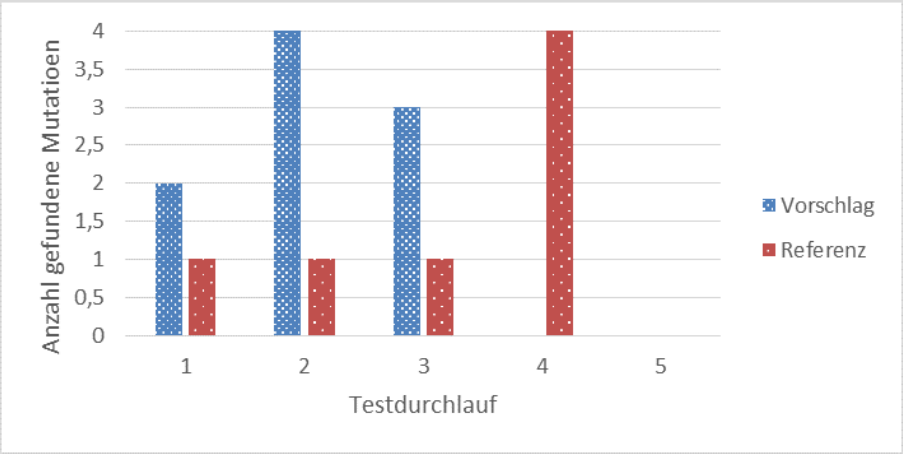
Die erste Hypothese des Experiments besagt, dass die Fehler in der Oberfläche durch die Anwendung der DFB schneller gefunden werden. Um dies beurteilen zu können, wird bei jedem Testlauf dokumentiert, welche Mutationen des Prüflings entdeckt wurden, woraus sich die Anzahl der pro Testdurchlauf erkannten Fehler ergibt. Hierbei handelt es sich um eine Verteilung der neun vorliegenden Mutationen auf die durchgeführten Testdurchläufe. Um die Hypothese zu prüfen wird die Anzahl der erkannten Fehler verglichen. Steigt die Fehlerverteilung schneller an, wird die Hypothese gestützt.

Abbildung 5.9a zeigt die durchschnittlich pro Testlauf gefundenen Mutationen. Bei der Pauschalbetrachung ist zu beobachten, dass mit Hilfe der DFB bereits in früheren Testläufen mehr Mutationen entdeckt werden als bei den Tests mit dem Referenzverfahren. In den ersten beiden Testläufen werden durchschnittlich 1,6 Fehler mehr Fehler aufgedeckt. Am größten ist der Unterschied im zweiten Durchlauf: Im Durchschnitt werden mit dem Verfahrensvorschlag 2,6 Fehler gefunden und damit 1,1 Fehler mehr als beim Referenzvorschlag (1,5 Fehler). Während beim Verfahrensvorschlag dieser zweite Durchgang die größte Anzahl an Fehlern aufdeckt, ist dies beim Referenzverfahren erst beim vierten Testlauf der Fall (3 Fehler). In keinem der Fälle wurden in einem Testlauf des Verfahrensvorschlags weniger Fehler aufgedeckt als in dem entsprechenden Testlauf des Referenzverfahrens.

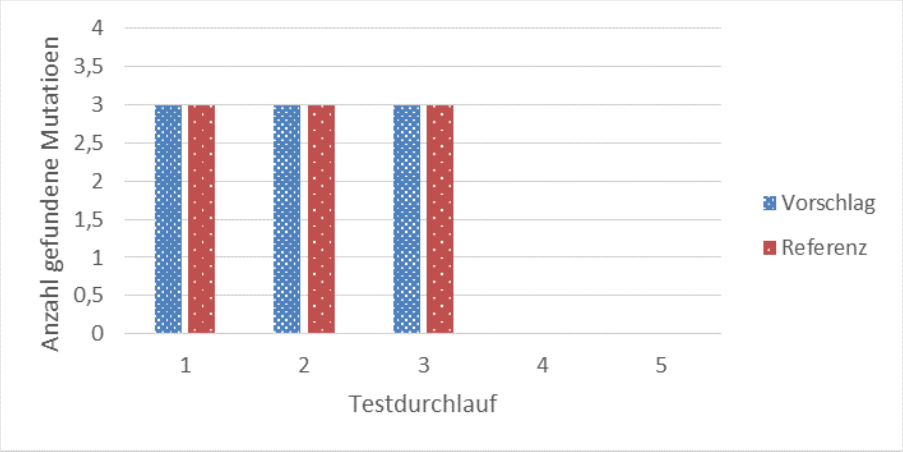
Auch die durchschnittliche Anzahl der Testdurchläufe bis zur Fehlerfreiheit kann tendenziell verringert werden. In drei der zehn Fälle wurde durch die Anwendung der differenzierten Fehlerbehandlung die definierte Fehlerfreiheit bereits einen Testlauf früher erreicht als beim Referenzverfahren. In fünf der zehn Fälle war zwar die gleiche Anzahl an Testdurchläufen notwendig, es wurden durch den Verfahrensvorschlag allerdings bereits in früheren Phasen mehr Fehler aufgedeckt. In zwei der zehn Fälle ergab sich durch Anwendung der DFB keine Veränderung. In keinem Fall waren durch die Anwendung der DFB mehr Testdurchläufe notwendig. Dieses Ergebnis wird durch die Einzellfallbetrachtung verdeutlicht. Im Bestfall (siehe Abbildung 5.9b) sind durch den Einsatz der DFB lediglich 4 anstatt 5 Testdurchläufe notwendig. Die meisten Fehler können durch den Verfahrensvorschlag bereits nach dem zweiten Durchlauf behoben werden. Im Referenzverfahren bleiben die meisten Fehler bis zum vierten Durchlauf unerkannt. Im Schlechtfall (siehe Abbildung 5.9c) ist die Anzahl der pro Durchlauf gefundenen Fehler



(a) *Pauschalbetrachtung der gefundenen Fehler pro Durchlauf*



(b) *Gefundene Fehler pro Durchlauf im Bestfall*



(c) *Gefundene Fehler pro Durchlauf im Schlechtfall*

Abbildung 5.9.: *Fehlerbehebungsverlauf*

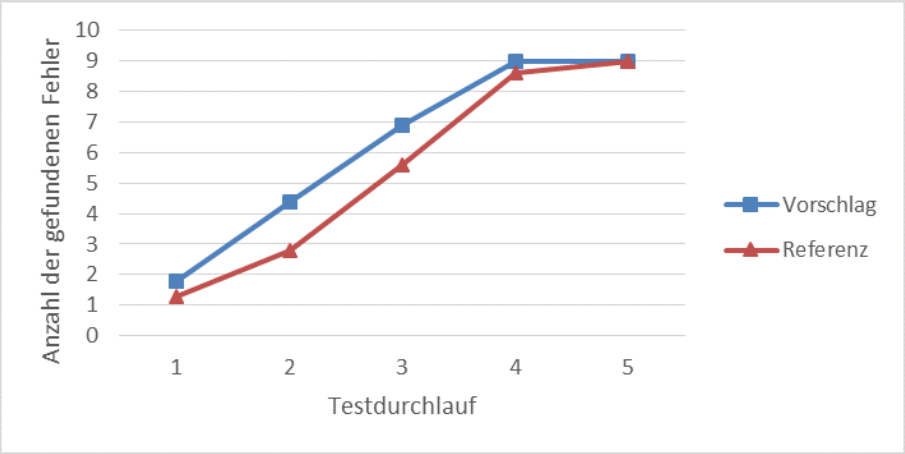
identisch. Die DFB führt bezüglich dieser Kriterien demnach situationsbezogen eine Verbesserung und, zumindest in diesem Experiment, nie eine Verschlechterung dar.

Auf Basis dieser Daten kann zusätzlich ein kumulativer Fehlerbehebungsverlauf ermittelt (siehe Abbildung 5.10). Hier wird illustriert, welcher Anteil der Fehler bis zu einem Testlauf durchschnittlich behoben wurde. Dies entspricht der Dichtefunktion der Fehlerverteilung. In der Pauschalbetrachtung (siehe Abbildung 5.10a) zeigt sich, dass die kumulative Fehlerbehebungskurve durch Anwendung der DFB schneller ansteigt. Bereits nach zwei Durchläufen konnten 4,4 Fehler behoben werden 1,6 Fehler mehr als ohne Modellunterstützung (2,8 Fehler). Dies entspricht einer Zunahme von 57%. Bliebe nur noch Zeit für beispielsweise drei Testphasen, könnten ohne DFB 5,6 und mit 6,9 Fehler gefunden werden. Mit Hilfe der DFB kann also der Anteil gefundener Fehler um 23% gesteigert werden.

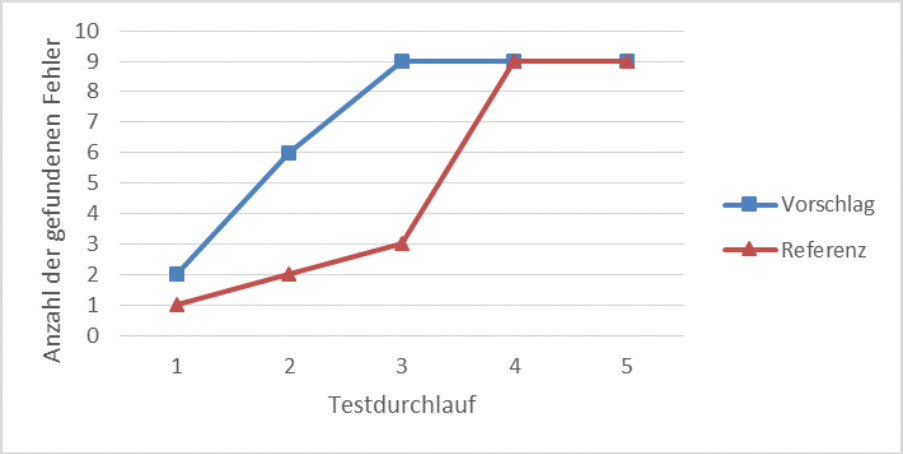
In der Einzelfallbetrachtung zeigt sich, dass sich die positiven Effekte der differenzierten Fehlerbehandlung situationsbedingt stark unterscheiden. Die Abbildungen 5.10b und 5.10c zeigen den Fehlerbehebungsverlauf mit und ohne Rerouting für den Best- und Schlechtfall. Es zeigt sich, dass im Bestfall durch die Anwendung der DFB ein vollständiger Testlauf weniger benötigt wird, um alle Fehler zu finden. Darüber hinaus werden in früheren Testphasen mehr Fehler gefunden. Die Kurve des Fehlerbehebungsverlaufs ist in Richtung der früheren Testläufe verschoben. Bliebe in der Testphase nur noch Zeit für drei Testdurchläufe, wäre das Referenzverfahren lediglich in der Lage, ein Drittel der Fehler aufzudecken. Durch Anwendung der DFB könnten alle Fehler gefunden und behoben werden. Im Schlechtfall hingegen ergibt sich durch die Anwendung des Verfahrensvorschlags keinerlei Abweichung vom Referenzverfahren. In den iterativen Testphasen werden die gleichen Mutationen entdeckt.

5.6.3. Testabdeckung

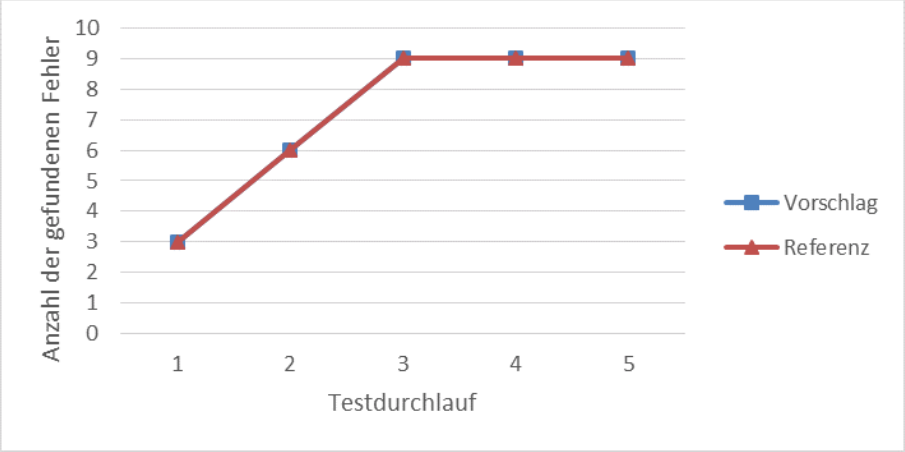
Die zweite Hypothese des Experiments bezieht sich auf die Anzahl der erfolgreich durchgeführten Verifikationen. Ziel des Testens ist es nicht nur, Fehler zu finden, sondern die Fehlerfreiheit bezüglich des Testziels sicherzustellen. Dies wird als Testabdeckung bezeichnet. Dafür ist es notwendig, alle Verifikationen durchzuführen, die einen Fehler entdecken könnten und dadurch gewährleisten, dass keine weiteren Fehler als die bereits bekannten vorliegen. Um den in Abschnitt 2.2 beschriebenen Entwicklungsprozess



(a) *Pauschalbetrachtung des Fehlerbehebungsverlaufs*



(b) *Fehlerbehebungsverlauf im Bestfall*



(c) *Fehlerbehebungsverlauf im Schlechtfall*

Abbildung 5.10.: *Fehlerbehebungsverlauf*

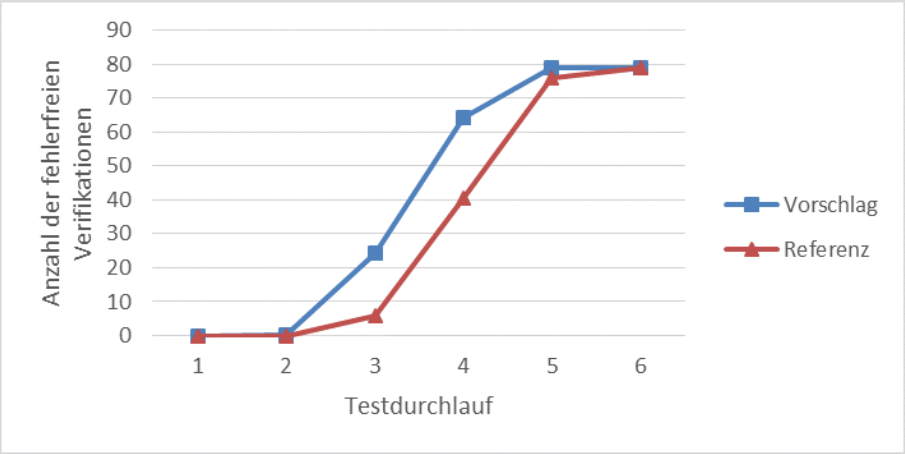
zu unterstützen, sollten diese Verifikationen in möglichst frühen Testläufen durchgeführt werden. Neben der eigentlichen Fehlerbehebung wird daher auch der Anteil der pro Testdurchlauf fehlerfreien Verifikationen erhoben (siehe Abbildung 5.11). In der Pauschalbetrachtung (Abbildung 5.11a) zeigt sich, dass auch die Anzahl der fehlerfreien Verifikationen durch die Anwendung der DFB schneller ansteigt. Bei der Durchführung des Verfahrensvorschlags wurden bereits nach dem vierten Testlauf durchschnittlich 64,3 Verifikationen ohne Fehler (entspr. 81%) durchgeführt. Beim Referenzverfahren wurden zu diesem Zeitpunkt erst 40,5 Verifikationen (entspr. 51%) fehlerfrei absolviert. Dies entspricht einer Zunahme von 59%.

Wiederum wird durch die Einzelfallbetrachtung deutlich, dass der Effekt der DFB stark durch die Fehlersituation bedingt ist. Im Bestfall unterscheidet sich die Anzahl der fehlerfreien Verifikationen zwischen Referenzverfahren und Verfahrensvorschlag erheblich (siehe Abbildung 5.11b). Durch Einsatz der DFB ist es möglich, im dritten Testlauf 60 (entspr. 76%) und bereits in Testlauf 4 alle 79 Verifikationen ohne Abweichung durchzuführen. Im Vergleich dazu kann ohne DFB auch im vierten Testlauf keine einzige Verifikation ohne Abweichung durchgeführt werden.

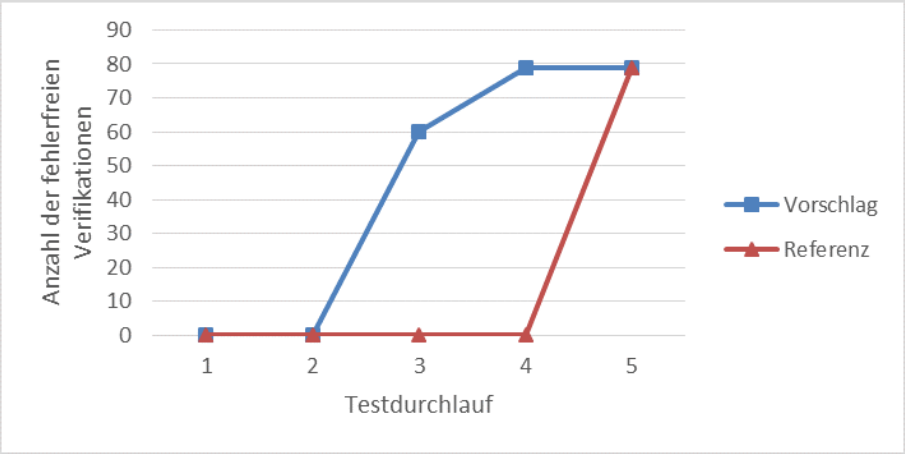
Die Betrachtung des Schlechtfalls zeigt, dass hier durch den Verfahrensvorschlag nur eine geringfügige Änderung erreicht wurde (siehe Abbildung 5.11c). Lediglich im dritten Testlauf weicht die Anzahl der fehlerfrei durchgeführten Verifikationen ab. Der Unterschied beträgt lediglich drei Verifikationen. Dieser Effekt lässt sich dadurch erklären, dass durch die Fortführung der Testfälle Bereiche der Oberfläche erreicht werden können, die nicht durch eine Mutation verfälscht werden. Durch den Verfahrensvorschlag ist es also möglich, diese Bereiche bereits in früheren Testphasen als fehlerfrei zu erkennen. Beim Referenzverfahren bleiben diese Bereiche der Oberfläche bis zur Behebung des Fehlers unerreichbar. Eventuelle Fehler könnten in dieser Zeit nicht erkannt werden.

5.6.4. Berichte & Berichtsinstanzen

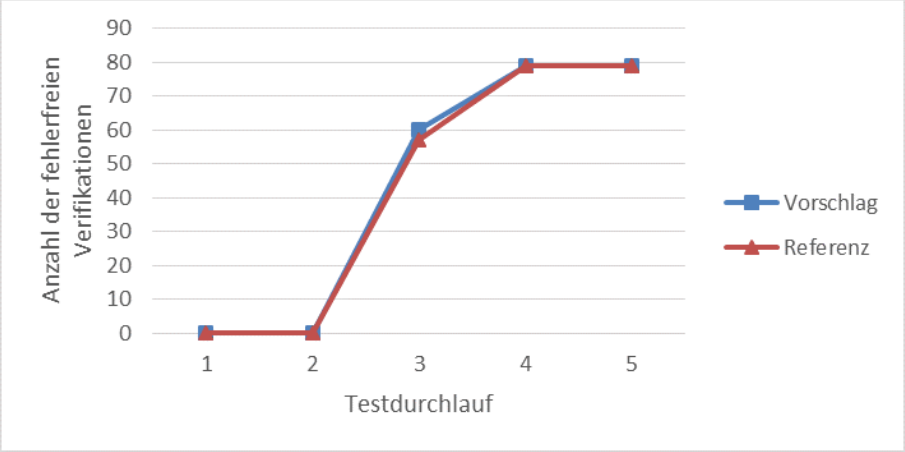
Neben den unmittelbar für die Hypothesen relevanten Kennzahlen wird in diesem Experiment zusätzlich die Anzahl der pro Testlauf erfassten Fehlerberichte bzw. deren Instanzen erfasst. Die möglichen Auswirkungen einer großen Anzahl an der Berichten wurden in Abschnitt 5.1) besprochen und wurden bei der Entwicklung des Testrahmens adressiert (siehe Abschnitt 5.5). Mit Hilfe einer farblichen Markierung war in



(a) *Pauschalbetrachtung der fehlerfreien Verifikationen*



(b) *Fehlerfreie Verifikationen im Bestfall*



(c) *Fehlerfreie Verifikationen im Schlechtfall*

Abbildung 5.11.: *Einzelfallbetrachtung des Fehlerbehebungsverlaufs*

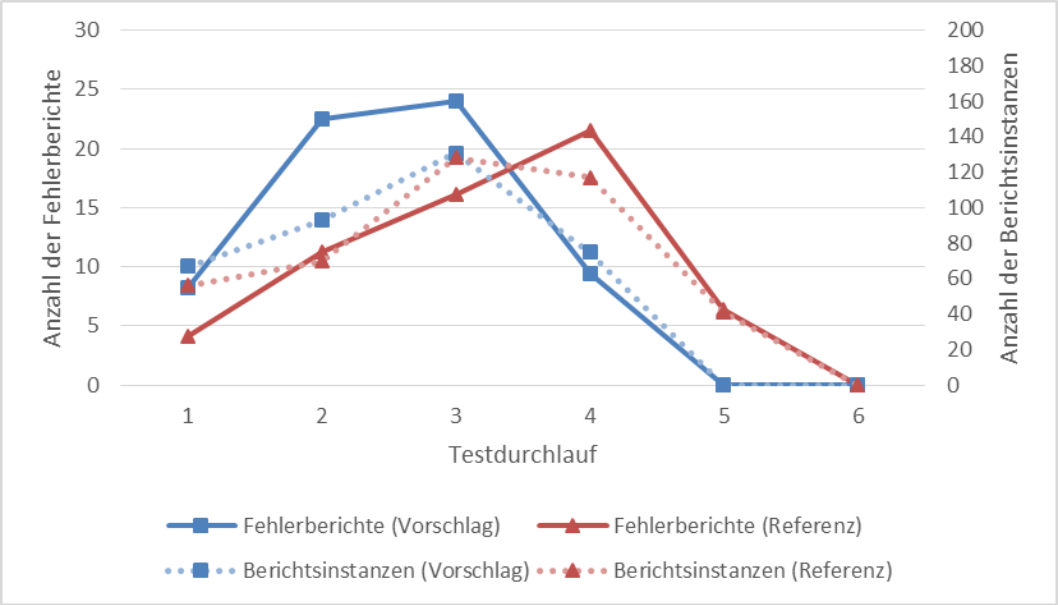


Abbildung 5.12.: *Anzahl der Fehlerberichte und Berichtsinstanzen pro Testlauf*

der Oberfläche des Testharnischs ersichtlich, welche Fehlerberichte bereits durch den vorangegangenen Testdurchlauf erstellt wurden und deshalb noch behoben werden müssen. Durch die automatische Durchführung konnten die entsprechenden Testfälle unmittelbar wiederholt werden und dadurch überprüft werden, ob die Fehlerursache weiterhin besteht. Die Erfahrung aus der Durchführung des Experiments zeigt, dass eine solche Hilfestellung die Fehlerbehebung stark vereinfacht. Ebenso bewährt haben sich die Gruppierung identischer Berichtsinstanzen sowie die automatische Klassifikation der Fehlerberichte.

Die Anzahl der Berichte und Berichtsinstanzen pro Durchlauf ist in Abbildung 5.12 dargestellt. Bereits die Gesamtzahl der erstellten Fehlerberichte unterscheidet sich. Durch die Anwendung der DFB werden über den gesamten Testprozess hinweg durchschnittlich 62,2 Berichte erstellt. Ohne DFB sind es lediglich 50,6 und damit 18,6% weniger. Bei der Gesamtanzahl der Berichtinstanzen verhält es sich invers. Hier werden ohne die Anwendung der DFB insgesamt 359,9 Einträge dokumentiert und damit mehr als mit dem Verfahrensvorschlag (350,5 Einträge). Außerdem zeigt sich, dass durch die Anwendung der DFB bereits in früheren Testdurchläufen eine größere Anzahl an Fehlerberichten erstellt wird. Während bei dem Referenzverfahren erst im vierten Testlauf die maximale

Anzahl an Berichten erreicht wird, ist dies bei Einsatz des Verfahrensvorschlags bereits im dritten Testlauf der Fall.

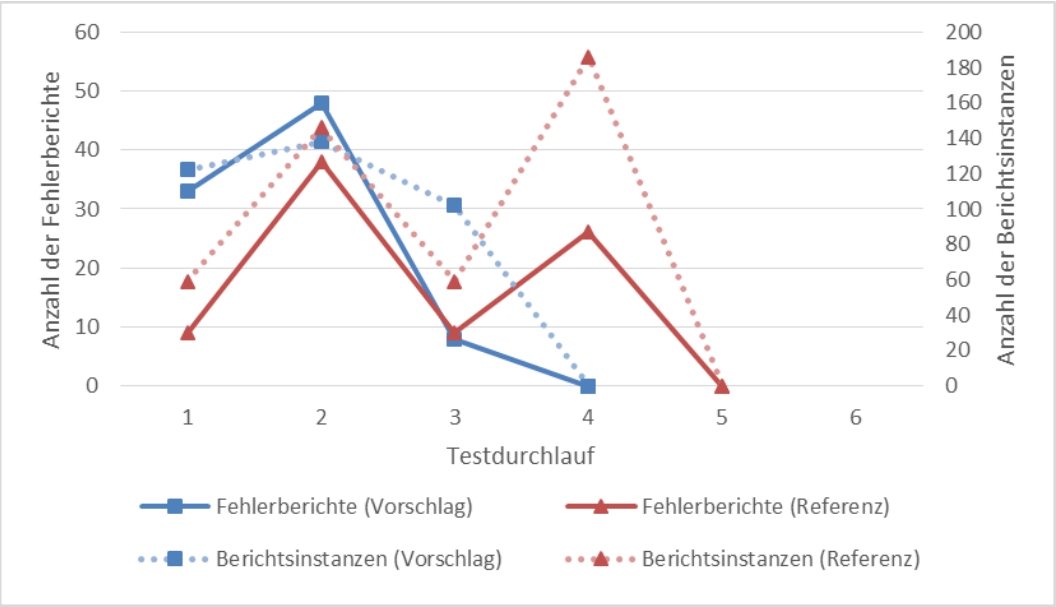
Abbildung 5.13 zeigt die Einzelfallbetrachtung der erstellten Fehlerberichte und Berichtsinstanzen. In beiden Fällen werden im Verfahrensvorschlag insgesamt mehr Fehlerberichte erstellt. Im Bestfall beträgt die Differenz sieben Berichte (Referenzverfahren: 82, Verfahrensvorschlag: 89), im Schlechtfall drei (Referenzverfahren: 27, Verfahrensvorschlag: 30). Allerdings zeigt sich, dass im Bestfall durch den Verfahrensvorschlag in den frühen Testphasen mehr Fehlerberichte erstellt werden konnten. Die letzten 26 der 82 Berichte (entspr. 32%) werden im Referenzverfahren erst im vierten Testlauf erstellt. Zu diesem Zeitpunkt liegen bereits alle Berichte des Verfahrensvorschlags vor. Im Gegensatz dazu ist die Anzahl der Berichte im Schlechtfall nahezu identisch. Die Kurven unterscheiden sich lediglich im zweiten Testlauf, in dem die zusätzlichen drei Berichte geschrieben werden. Auch die Fehlerberichtinstanzen weichen ausschließlich um drei zusätzliche Berichtsinstanzen im zweiten Durchlauf ab.

5.6.5. Reroutes

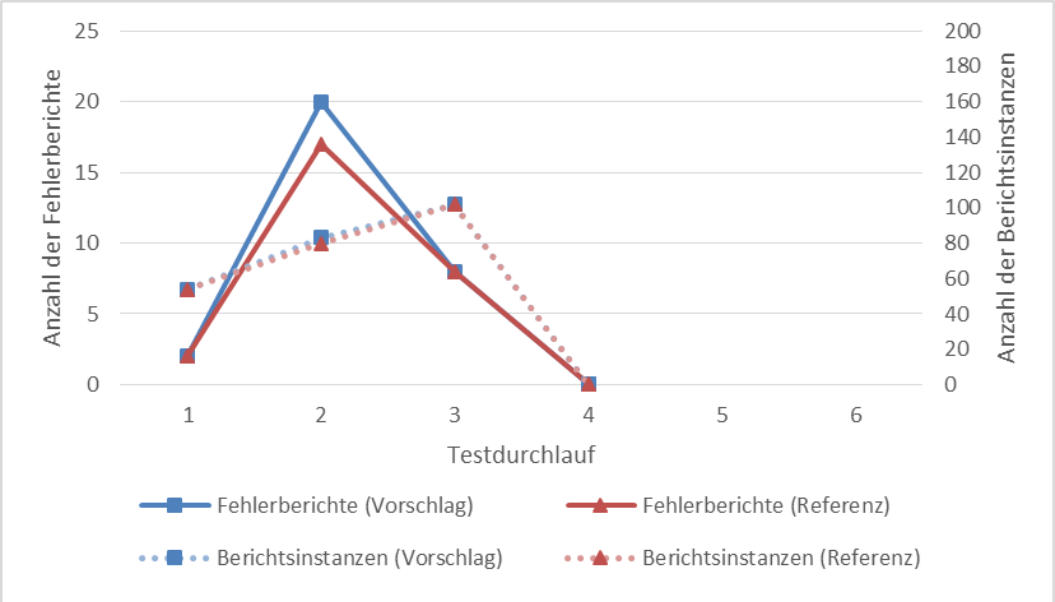
Bei der Testausführung wurde ebenfalls festgestellt, wie häufig die im Testfall vorgegebene Bediensequenz durch die partielle Neuberechnung abgeändert wurde. Dies zeigt, wie häufig ein Fehlersymptom mit Hilfe der DFB umgangen werden konnte. Die durchschnittliche Anzahl der pro Testlauf durchgeführten Reroutes ist in Abbildung 5.14 dargestellt. Hier wird deutlich, dass dies durchschnittlich im zweiten Testlauf am häufigsten zur Anwendung kam. Die Einzelfallbetrachtung zeigt bei der Anzahl der Reroutes deutliche Unterschiede. Im Bestfall wurden insgesamt 50 Reroutes durchgeführt. 25 bereits im ersten, weitere 22 im zweiten Testlauf (siehe Abbildung 5.14b). Im Gegensatz dazu werden im Schlechtfall insgesamt lediglich sechs Reroutes durchgeführt (siehe Abbildung 5.14c). Diese sind gleichmäßig aufgeteilt auf den zweiten und dritten Testlauf.

5.6.6. Laufzeit

Ein weiterer relevanter Aspekt bei der praktischen Anwendung eines Testverfahrens sind die Dauer der Testdurchführung bzw. die notwendigen Systemressourcen. Sollte die Durchführung unverhältnismäßig große Rechenkapazität benötigen, ist die Eignung

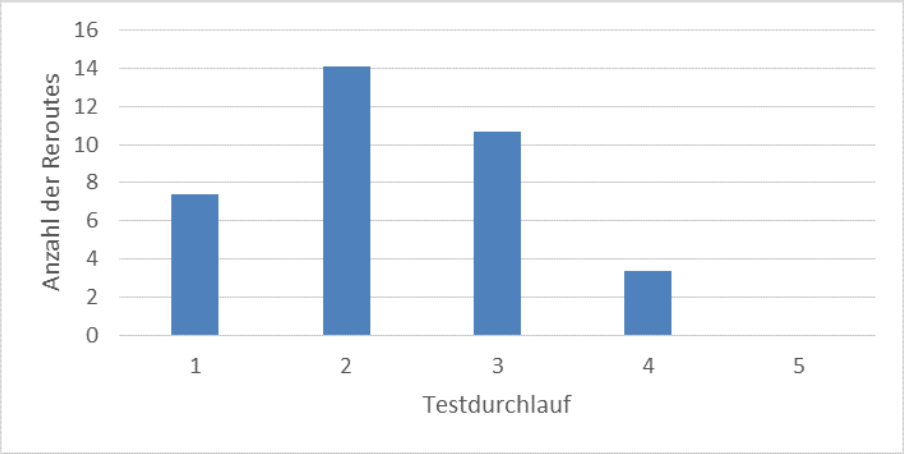


(a) *Fehlerberichte und Berichtsinstanzen im Bestfall*

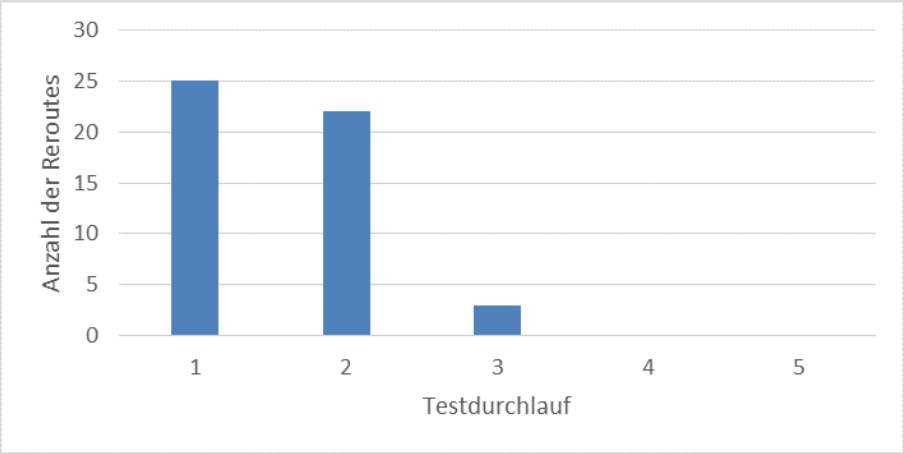


(b) *Fehlerberichte und Berichtsinstanzen im Schlechtfall*

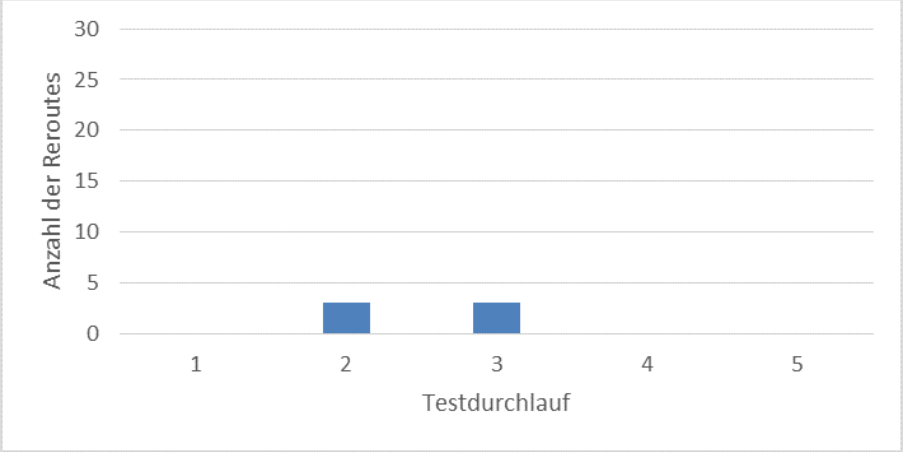
Abbildung 5.13.: *Fehlerberichte und Berichtsinstanzen*



(a) *Pauschalbetrachtung der Reroutes der DFB pro Testdurchlauf*

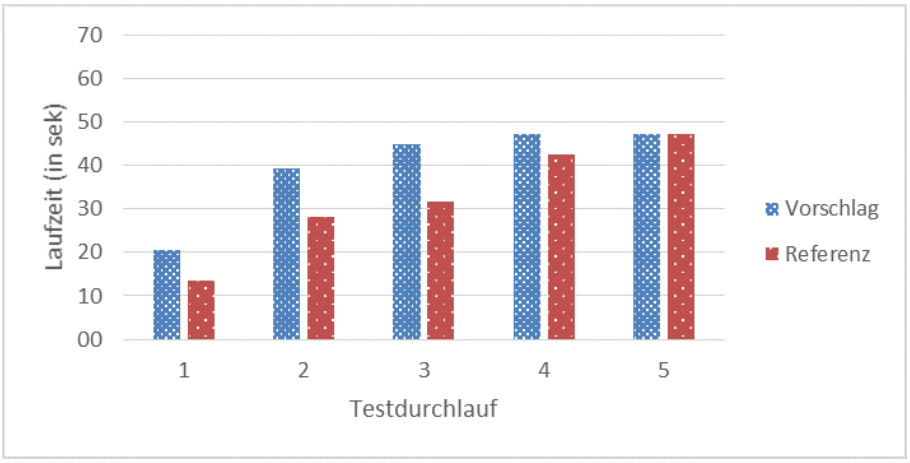


(b) *Reroutes der DFB pro Testdurchlauf im Bestfall*

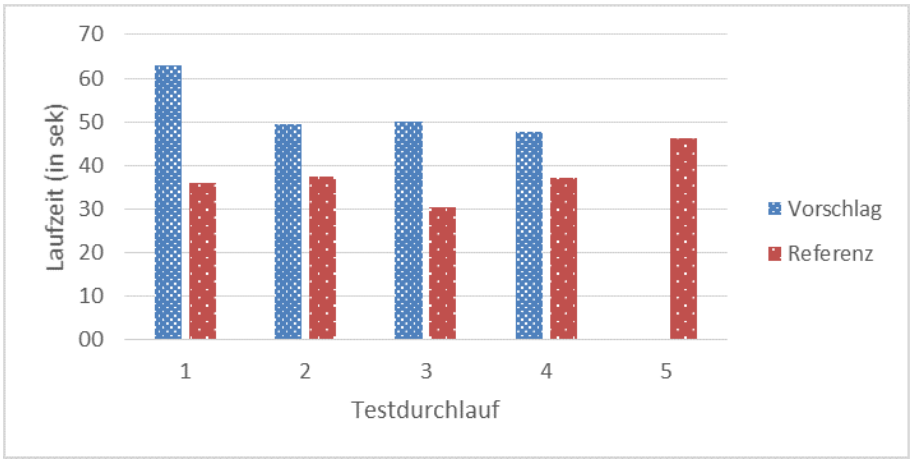


(c) *Reroutes der DFB pro Testdurchlauf im Schlechtfall*

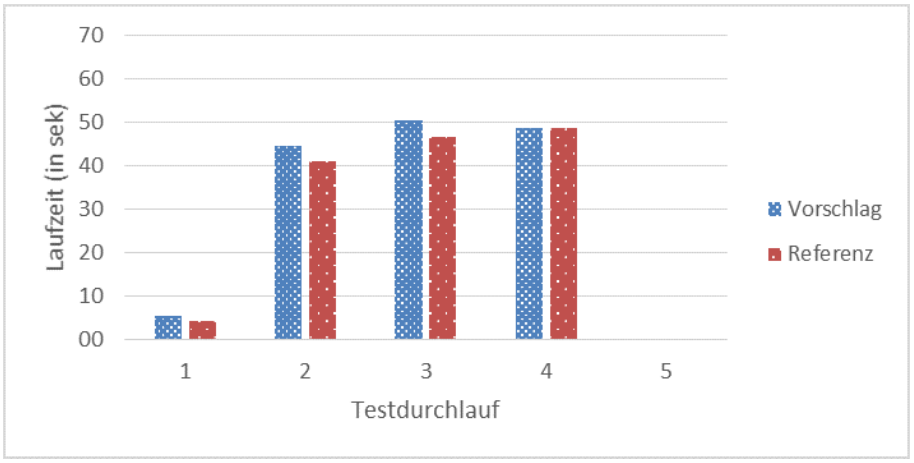
Abbildung 5.14.: *Anzahl der Reroutes pro Testlauf*



(a) *Pauschalbetrachtung der Testlaufzeit*



(b) *Testlaufzeit im Bestfall*



(c) *Testlaufzeit im Schlechtfall*

Abbildung 5.15.: *Testlaufzeit*

für einen praktischen Einsatz gegebenenfalls gefährdet. Allerdings ist die Verhältnismäßigkeit in einem Experiment nicht endgültig zu ermessen. Einflüsse wie der technische Fortschritt, die bereits verfügbaren Systemressourcen und die Effizienz der Implementierungen variieren kontextabhängig stark. Zudem fallen die Testlaufzeiten in dem angenommenen Entwicklungsprozess (siehe Abschnitt 2.2) kaum ins Gewicht. Bei einem deutlichen Unterschied müssten die Testphasen gegebenenfalls verlängert werden, um sicherzustellen, dass alle Testfälle durchgeführt werden können. Die Tests wurden auf einem Rechner mit einem Intel Core i5 Prozessor und 6 Gigabyte Hauptspeicher durchgeführt. Als Betriebssystem war Windows 7 mit Service Pack 1 installiert. Die Anwendung ist in Java (Version 6, Update 45) implementiert.

In der Pauschalbetrachung zeigt sich, dass die Gesamtlaufzeit jedes Tests durch den Verfahrensvorschlag verlängert wird. Während die reine Testdauer per Referenzvorschlag durchschnittlich 2min 35sec beträgt, benötigt der Test per Verfahrensvorschlag 3min 00sec. Zwar wird durch Anwendung der DFB in zwei Fällen ein sechster Durchgang überflüssig, Berechnung und Durchführung der partiellen Neugenerierung sind allerdings für die Differenz ausschlaggebend. Am größten ist der Unterschied im ersten Testlauf. Hier dauert der Verfahrensvorschlag 52% länger. Die Differenz nimmt im Zuge der Durchläufe tendenziell ab (Durchlauf 2: 38%, Durchlauf 3: 41%, Durchlauf 4: 10%). Der fünfte Durchlauf dauert in beiden Fällen gleich lang.

Die Detailbetrachtung bestätigt den Eindruck der Pauschalbetrachtung. Im Bestfall (Abbildung 5.15b) dauert der Test mit dem Referenzverfahren insgesamt 3min 07sec, mit dem Verfahrensvorschlag 3min 30sec. Die Testläufe eins bis vier dauern aufgrund der partiellen Neugenerierung jeweils länger. Die größte Differenz ist im ersten Testlauf zu beobachten. Hier dauert der Test mit dem Verfahrensvorschlag 75% länger als der Test mit dem Referenzverfahren. In diesem Testlauf wurden mit 25 Reroutes am häufigsten eine partielle Neuberechnung durchgeführt. Obwohl auf einen fünften Testlauf verzichtet werden kann, führt dies nicht zu einer Verringerung der Gesamtlaufzeit. Auch im Schlechtfall (Abbildung 5.15c) benötigt der Verfahrensvorschlag mehr Zeit. Der Test mit dem Referenzverfahren dauert insgesamt 2min 20sec, mit dem Verfahrensvorschlag 2min 29sec. Dieser Unterschied ist in den Testläufen zwei und drei zu beobachten, in denen tatsächlich Reroutes durchgeführt wurden. Die Differenz von 1sec im ersten Testlauf ist auf eine natürliche Varianz zurückzuführen.

5.7. Zusammenfassung

Das in dieser Arbeit vorgeschlagene MBT-Verfahren wurde im Rahmen eines Experiments auf ein produktives IVI System von Mercedes-Benz angewendet, um die Wirksamkeit der DFB zu beurteilen. Es wurden Hypothesen formuliert, um die Wirkungsweise des Verfahrens mit dem aktuellen Stand der Forschung vergleichen zu können. Durch die Anwendung der DFB sollte der Fehlerbehebungsverlauf in Richtung früherer Testläufe verschoben werden, um trotz zeitlicher Einschränkungen einen möglichst hohen Anteil der Fehler zu entdecken. Des Weiteren sollte die Verlässlichkeit des Systems durch Anwendung der DFB früher als bisher überprüft werden können. Als Maß werden die fehlerfrei durchgeführten Verifikationen gewertet.

Für das Experiment wurde ein Prüfling entwickelt, der eine umfassende Überprüfung durch ein automatisches Testverfahren ermöglicht. Dieser Prüfling wurde basierend auf der Fehlersymptomtaxonomie dieser Arbeit mit Fehlern versehen, um eine wirklichkeitsgetreue Fehlerverteilung abzubilden. Die Erstellung des Testmodells sowie die Testfallgenerierung wurden mit dem Werkzeug Conformiq durchgeführt. Es war möglich, die in Kapitel 3 vorgestellte Struktur umzusetzen und Testfälle abzuleiten, um den Testfokus «Menüstruktur» abzudecken. Ausgeführt wurden die Testfälle mit Hilfe eines im Rahmen der Arbeit entwickelten Testrahmens. Abweichungen werden anhand der Fehlersymptomtaxonomie klassifiziert und durch Fehlerberichte dokumentiert. Der Testrahmen ist in der Lage, eine partielle Neugenerierung durchzuführen.

Die Thesen des Experiments werden durch die Ergebnisse gestützt. Vor allem unter der Annahme, dass ein System nicht bis zur Fehlerfreiheit getestet werden kann, sondern nur ein im Vorfeld festgesetzter Zeitraum zur Verfügung steht, stellt der Verfahrensvorschlag eine Verbesserung gegenüber dem Referenzverfahren dar. Durch den Einsatz der DFB werden durchschnittlich mehr Mutationen pro Testlauf gefunden als beim Referenzverfahren. Durch die partielle Neugenerierung ist es offensichtlich möglich, Fehlersymptome zu übergehen. Dadurch können weitere Bereiche der Oberfläche erreicht und Fehler in früheren Testdurchläufen entdeckt werden. Dies führt dazu, dass im Mittel weniger Testläufe benötigt werden, da sich der Fehlerbehebungsverlauf in Richtung der früheren Durchläufe verschiebt. Wie in Abschnitt 4.4 ist die Wirkungsweise des Verfahrens stark situationsbedingt. Im Bestfall war zu beobachten, dass die DFB den Testverlauf erheblich beeinflussen konnte. Fehlersymptome, die umgangen werden konnten, betrafen zentrale

Element der Oberfläche. Im Schlechtfall hingegen verhinderten schwerwiegende Fehler die Durchführung. Nur in wenigen Fällen war eine partielle Neuberechnung möglich, ohne den Testverlauf maßgeblich zu beeinflussen.

Als ein weiterer Effekt des Verfahrensvorschlags werden in früheren Testläufen mehr Berichte geschrieben. Auch wenn der Effekt einer großen Anzahl an Berichten noch untersucht werden muss, ist dies ein weiteres Indiz dafür, dass die Fehler bereits in früheren Testphasen entdeckt und behoben werden können. Zudem steigt die Testabdeckung schneller an, da in früheren Phasen mehr Verifikationen fehlerfrei durchgeführt werden können. Lediglich die Dauer der Durchführung wurde durch den Verfahrensvorschlag verlängert.

Auch wenn die Hypothesen aufgrund der geltenden Einschränkungen (siehe Abschnitt 5.6.1) nicht statistisch beurteilt werden können, zeigen sowohl die Pauschalbetrachtung als auch die Betrachtung der Einzelfälle einen positiven Trend. Die Einzelfallbetrachung zeigt durchgängig, dass der Bestfall die Ergebnisse der Pauschalbetrachung bestätigt. Selbst im Schlechtfall stellt der Verfahrensvorschlag keinen Rückschritt im Vergleich zu dem Referenzverfahren dar.

6

Fazit & Ausblick

In dieser Arbeit wurde die modellbasierte Testautomatisierung von direktionalen Oberflächen eingebetteter Systeme diskutiert. Als zentrale Herausforderung wurde die Robustheit gegenüber Fehlersymptomen während des spezifikations-basierten Funktionstest identifiziert. Durch die Literaturrecherche wurde deutlich, dass die spezifischen Umstände beim Test direktionaler Oberflächen trotz ihrer Verbreitung bisher nur unzureichend untersucht wurden (siehe Kapitel 2). Außerdem beschränken sich viele Arbeiten auf die Modellierung und die exemplarische Generierung von Testfällen. Ausführung und Auswertung werden allerdings kaum behandelt. Die in der Literatur angebotenen Rahmenwerke, die den gesamten Prozess der Testautomatisierung betrachten, werden den Anforderungen des zugrundeliegenden Entwicklungsprozess nicht gerecht. Bei den aktuellen Verfahren verhindern Fehler häufig die Fortführung der Tests. Ein Abbruch des Tests wiederum gefährdet im engen Zeitkorsett des Entwicklungsprozesses eingebetteter Systeme die Erfüllung der erforderlichen Qualitätsanforderungen. Das modellbasierte Verfahren ist demnach aktuell nicht dafür ausgelegt, den eigentlich Zweck, Abweichungen aufzudecken, zu erfüllen.

Ein wesentlicher Beitrag dieser Arbeit ist die Analyse von Fehlerberichten (siehe Abschnitt 3.1). Erst durch die Erstellung einer Taxonomie der Fehlersymptome kann die

Testautomatisierung zielgerichtet eingesetzt, die automatische Durchführung unterstützt und deren Wirksamkeit überprüft werden. Die klare Abgrenzung der Testziele ermöglicht die sinnvolle Ergänzung verschiedener Testverfahren und zeigt die Stärken und Schwächen automatisierter Ansätze auf. Während beispielsweise Inhalte wie Texte einfach automatisch überprüft werden können, stellen Animationen eine größere Herausforderung dar. Nach einer detaillierten Betrachtung der einzelnen Symptomklassen wurde der Testfokus «Menüstruktur» als geeigneter Untersuchungsgegenstand der modellbasierten Testautomatisierung ausgewählt (siehe Abschnitt 3.2). Dieser Testfokus fasst die Symptome zusammen, die sich aus einer komplexen und dynamischen Menüführung ergeben. Durch die klaren Zielvorgaben können nun die Informationen ermittelt werden, die für die Ableitung der erforderlichen Testfälle notwendig sind (siehe Abschnitt 3.3). Dies umfasst sowohl die Ableitung der erforderlichen Bediensequenz, um das zu testende Verhalten aufzurufen, als auch die Soll-Informationen, die zur Verifikation notwendig sind. Aus einer Analyse der testrelevanten Informationen ergibt sich, dass eine Kombination aus Zustandsautomaten und Programmcode geeignet sind um die erforderlichen Testfälle abzuleiten. Dabei wird auf die bewährte Abstraktion von Screens und Widgets zurückgegriffen. Die Bedingungsabhängigkeit von Programmteilen kann so durch den Zustandsautomat beschrieben, die Komplexität der Bedingungsauswertung kann durch Code abgebildet werden.

Das Aufdecken von Fehlern und der dadurch erforderliche Abbruch der Testfalldurchführung gefährdet die Praxistauglichkeit der modellbasierten Testautomatisierung. Anhand der Fehlersymptomanalyse kann gezeigt werden, dass alle klassifizierten Fehlersymptome unter Umständen zu einem Abbruch führen können (siehe Kapitel 4). Das modellbasierte Vorgehen ermöglicht ein Verfahren, um Fehlersymptome während der Durchführung zu erkennen und zu umgehen, die differenzierte Fehlerbehandlung. Sobald erwartetes und tatsächliches Verhalten abweichen, wird das Symptom mit Hilfe der Fehlersymptomtaxonomie klassifiziert und durch einen Fehlerbericht dokumentiert. Basierend auf dem fehlerhaften Zustand des Prüflings wird anschließend ermittelt, ob bzw. mit Hilfe welcher eine Bediensequenz der Testfall trotz Fehler fortgeführt werden kann.

Die Wirksamkeit der differenzierten Fehlersymptombehandlung wird in einem Experiment untersucht. Basierend auf der Fehlersymptomanalyse wird eine direktionale Oberfläche mit Hilfe der Mutationsanalyse so verfälscht, dass sich deren Fehlerbild der

Fehlersymptomtaxonomie annähert. Auf diese Weise werden 10 verfälschte Versionen der Oberfläche erzeugt. Anschließend wird die modellbasierte Testautomatisierung jeweils einmal mit und einmal ohne differenzierte Fehlerbehandlung durchgeführt. Die Evaluation zeigt, dass die Robustheit der modellbasierten Testautomatisierung erhöht und Abbrüche der Testfalldurchführung vermieden werden konnten. Das Ziel dieser Arbeit wurde damit erreicht.

Die Herangehensweise dieser Arbeit, der modellbasierten Testautomatisierung eine Fehlersymptomanalyse voranzustellen, hat sich bewährt. Auf diese Weise wurde von Beginn an das Verständnis für die Problemstellung und die erforderlichen Schritte geschärft. Die Entscheidung für Modellierungs- und Generierungsverfahren konnte auf einer belastbaren Basis getroffen werden. Durch die Fehlerklassen konnten die Ursachen der Testabbrüche klar benannt und Gegenmaßnahmen ergriffen werden. Die Taxonomie diente als Referenz bei der Erstellung einer wirklichkeitsgetreuen Evaluationsumgebung. Die Fehlersymptomklassen und deren Verteilung sind eine Momentaufnahme und werden sich durch neue Technologien und Inhalte verändern. Durch die hierarchische Struktur ist allerdings die Grundvoraussetzung erfüllt, dass zumindest Teile der Taxonomie erhalten bleiben können. Durch die Regeln zur Erstellung der Taxonomie ist ein Rahmen gegeben, die Fehlersymptomklassen auf zukünftige Bedürfnisse anzupassen. Durch eine fortlaufende Anwendung und Veränderung der Taxonomie, könnte sogar der Wandel von Bedienoberflächen dokumentiert und nachvollzogen werden. Um Aufwand zu reduzieren, sollten die Fehlersymptome allerdings nicht, wie in dieser Arbeit, nachträglich eingeordnet, sondern von Beginn an im Testprozess verankert werden. Fehlerdatenbanken könnten bereits bei der Erfassung der Fehlerberichte entsprechende Kategorien anbieten und eventuell vorhandene Verfahren zur Testautomatisierung eine automatische Zuordnung von Abweichungen anbieten. Eine stetige Auswertung aktueller Fehlerbilder ermöglicht es, Testverfahren allgemein und die Testautomatisierung im Besonderen der tatsächlichen Fehlersituation anzupassen.

Mit dem Wandel der Fehlerbilder steht mit der differenzierten Fehlerbehandlung auch das Verfahren zur Umgehung der Symptome auf dem Prüfstand. Bei der zukünftigen Entwicklung von Testfallgeneratoren sollte die Möglichkeit einer partiellen Neugenerierung berücksichtigt werden. Dadurch könnte die Evaluation der Wirkungsweise um ein Rerouting auf Ebene der Zustandsmaschine erweitert und damit das volle Potential des Verfahrens ermittelt werden. Zudem kann in zukünftigen Arbeiten der Testfokus variiert

werden. Basierend auf Fehlersymptomanalyse zeichnet sich bereits jetzt Notwendigkeit ab, den Test proaktiver Systemmeldungen auf Existenz, Dauer und Priorisierung zu systematisieren. Hier zeigt sich deutlich die Abhängigkeit der Oberfläche von angeschlossenen Regelsystemen. Die klare aber häufig komplexe Abhängigkeit eignet sich für eine automatisierte Überprüfung. Während in dieser Arbeit der Aufruf der Funktionen und die Navigation innerhalb der bestehenden Oberfläche im Vordergrund standen könnte zukünftig das Zusammenspiel verschiedener Ein- und Ausgabemodalitäten an Bedeutung gewinnen. Es wird zu untersuchen sein, ob die Kombination aus Zustandsautomaten und Programmcode weiterhin für die Modellierung der testrelevanten Informationen geeignet ist.

Nicht nur die Inhalte oder Umsetzungstechnologien befinden sich im Wandel, auch der Entwicklungsprozess ist Gegenstand der Veränderung. Es bleibt abzuwarten, wie lange das V-Modell, das dieser Arbeit zugrunde liegt, in dieser Form bestand haben bzw. auf welche Weise sich der Ablauf der Entwicklung verändern wird. Trotzdem wird es eine Herausforderung bleiben, die Komplexität der Oberflächen eingebetteter Systeme zu beherrschen. Wie in Abschnitt 2.1 beschrieben, wird diese Komplexität zukünftig voraussichtlich weiter steigen. Eine klare und überprüfbare Beschreibung der Einflussfaktoren und deren Auswirkungen werden auch weiterhin erforderlich sein. Denkbar wäre, dass, anders als in dieser Arbeit angenommen, diese Beschreibung sowohl als Grundlage der Entwicklung als auch für die Testverfahren verwendet wird.

Unabhängig von Umsetzungstechnologie, Inhalt oder Entwicklungsprozess ist davon auszugehen, dass die Testphase auch zukünftig einen wesentlichen Teil der Entwicklung von Benutzerschnittstellen bilden und voraussichtlich an Bedeutung gewinnen wird. Dies könnte auch den Trend der Automatisierung weiter beschleunigen. Die vorliegende Arbeit trägt dazu bei, das Verständnis für die wesentlichen Schritte der Testautomatisierung zu vertiefen und ermöglicht dadurch, auf eine Veränderung der Rahmenbedingung bewusst und systematisch zu reagieren.

A

Testmodell

In diesem Anhang wird die Struktur des Testmodells in Ergänzung zu Abschnitt 5.4 beschrieben. Abbildung A.1 gibt einen Übersicht über die Klassenstruktur. Die Klasse `HMIButton` wird für alle Einträge verwendet, die fokussiert und aktiviert werden können. Eine Aktivierung hat gegebenenfalls eine Wechselwirkung mit der Zustandsmaschine, wie beispielsweise die Auslösung einer Transition zur Folge. Die Architektur des Testfallgenerators sieht vor, dass in diesen Fällen die Conformiq eigene Klasse `StateMachine` erweitert und die Methode `run` überschrieben werden muss. Diese Vererbung ist der Grund dafür, dass das Artefakt `Entry` nicht als Superklasse sondern als Interface und die Klasse `HMIEntry` als Standardimplementierung realisiert werden muss.

Das Attribut `type` gibt erneut an, ob es sich bei dem Inhalt um Text und/oder Bilder handelt. Die Hashtables `visibility` und `selectability` halten alle Bedingungen und deren Belegungen vor, unter denen der Eintrag sichtbar bzw. fokussierbar ist. Diese Bedingungen werden durch die Methoden `isVisible` bzw. `isSelectable` ausgewertet, die nur `true` zurückgeben, wenn alle Bedingungen und deren Belegungen der Hashtable aktuell zutreffen. Es handelt sich dabei demnach ausschließlich um eine UND Verknüpfung. Die Hashtable `conditionsToSet` gibt an, welche Bedingungen (key) bei der Aktivierung der Buttons in welcher Form (value) verändert werden müssen.

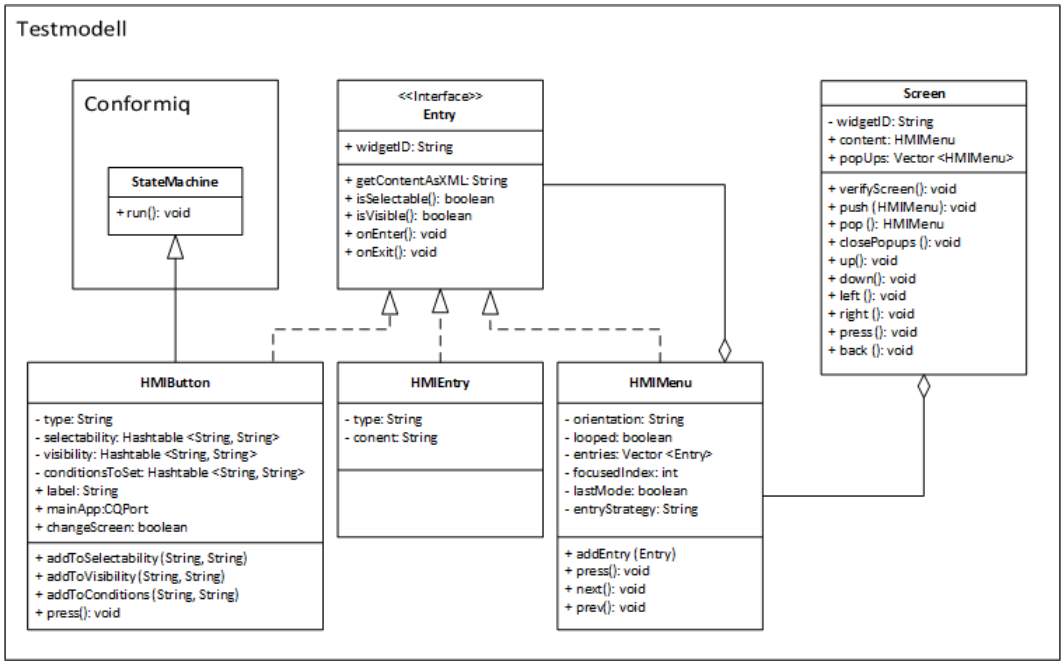


Abbildung A.1.: *Strukturdiagramm der im Testmodell verwendeten Klassen*

Die Referenz auf die Zustandsmaschine des Testmodells wird durch das Attribut `mainApp` vorgehalten. Der Booleanwert `changeScreen` gibt an, ob bei Aktivierung des Eintrags Bildschirm des SUT gewechselt wird. Neben Methoden für die Angaben zur Sichtbarkeit (`addToVisibility`), Verfügbarkeit (`addToSelectability`) oder zu den Auswirkungen auf das Gesamtsystem (`addToConditions`), bietet die Klasse an, den Eintrag zu aktivieren (`press`) und damit die gegebenenfalls verbundenen Effekte auszulösen. Unterklassen von `HMIButton` sind beispielsweise `HMICheckBox`, `HMIRadioButton` und `HMIActiveButton`. Auf die Darstellung der Unterklassen wurde der Übersichtlichkeit halber verzichtet.

Die Klasse `HMIMenu` ist die dritte direkte Implementierung des `Entry` Interfaces. Hierbei handelt es sich um einen Behälter für Klassen, die ebenfalls das Interface `Entry` implementieren. Damit kann ein `HMIMenu` Objekte der Klassen `HMIButton`, `HMIEntry` und erneut `HMIMenu` beinhalten. Die Verweise auf die entsprechenden Objekte werden durch den Vector<Entry> `entries` gehalten. Die zentrale Aufgabe dieser Klasse ist die Verwaltung des Fokus der zugeordneten Einträge. Sobald das Menü fokussiert wird, ermittelt die Methode `onEnter`, an welchen Eintrag die Fokussierung weitergeleitet werden muss. Dabei muss berücksichtigt werden, ob das Menü seit Systemstart bereits

betreten wurde und ob durch das Attribut `lastMode` angegeben ist, dass der zuletzt fokussierte Eintrag erneut fokussiert werden soll (Belegung: `true`). Falls das Menü noch nicht betreten wurde und/oder kein lastMode Verhalten vorliegt, wird die Eintrittsstrategie (`entryStrategy`) ausgewertet. Durch das Attribut wird angegeben, ob in diesen Fällen bspw. der erste Eintrag oder der mittlere der selektierbaren Einträge fokussiert werden soll. Diese Funktion soll den Benutzer bei direktionalen Bedienoberflächen dabei unterstützen, in einer großen Anzahl verfügbarer Einträge durch möglichst wenige Bedienschritte den gewünschten Eintrag fokussieren zu können.

Für die Iteration der Einträge des Menüs bietet diese Klasse die Methoden `next` und `prev` an, die nach erneuter Abfrage der Sichtbarkeit und Verfügbarkeit der enthaltenen Einträge den als nächstes zu fokussierenden Eintrag bestimmen. Das Attribut `orientation` gibt an, ob das Menü horizontal oder vertikal ausgerichtet ist und damit welche Benutzeraktion (Links und Rechts, Hoch und Runter) durch das Menü konsumiert werden. Durch das Attribut `looped` wird festgelegt, ob für das Menü am ersten und am letzten Eintrag ein fester Anschlag vorgesehen ist (Belegung: `false`), oder ob der Iteration der Einträge nach dem letzten Element wieder das erste, bzw. nach dem ersten Eintrag wieder der letzte fokussiert werden soll (Belegung: `true`). Die Methode der Klasse `press` leitet die Benutzerinteraktion an den aktuell fokussierten Eintrag weiter.

Die auf dem `Entry` Interface basierenden Klassen können für der Modellierung verschiedener direktionaler Oberflächen eingesetzt werden. Dies gilt insbesondere für die Weiterentwicklung eines bestehenden Systems, wie beispielsweise eines IVI Systemen. Die `Screen` Klasse stellt die höchste Abstraktionsebene dar, die ausschließlich durch Programmcode modelliert ist und für jede IVI Generation angepasst werden muss. Hier werden die Widget-Objekte in der Struktur zusammengesetzt, wie sie auf dem Bildschirm zu sehen sind. Die horizontalen Menüzeilen der NTG 4.5 High werden durch das `HMIMenu content` in einem Screen-Objekt repräsentiert. Der `Vector<HMIMenu> popUps` verwaltet eventuell geöffnete APG oder SPG. Mit den Methoden `up`, `down`, `left`, `right` und `press` wird das `Screen` Objekt darüber informiert, welche Aktion der Benutzer durchgeführt hat und leitet diesen Aufruf an das jeweils fokussierte Menü weiter. Eine weitere Funktion eines `Screen` Objekts ist es, beim Betreten des Screens die GUI Map als Verifikationsinformationen in den Testfall einzubetten. Mit Hilfe der Methode `verifyScreen` ist die `Screen` Klasse in der Lage, den aktuellen Zustand aller enthaltenen Widget Objekte in XML Form zu persistieren. Beim Aufruf der Methode wird

für jedes Menü in `content` die Methode `getContentAsXML` aufgerufen, die wiederum den Aufruf an die enthaltenen Elemente weiterleitet.

Listing A.1: Ausschnitt der NTG4_5 Klasse

```
1  system
2  {
3    Inbound cce : press, back, left, right, up, down;
4    Outbound verification : focusedEntry, screenChanged, popupAdded;
5  }
6
7  record screenChanged
8  {
9    String screenID;
10   String screenStructure;
11 }
```

Alle Ereignisse, die auf das SUT einwirken können, werden im so genannten `System Block` deklariert (siehe Listing A.1). Hier wird neben einem eindeutigen Namen angegeben, ob es sich um ein eingehendes, ausgehendes oder internes Ereignis handelt (siehe Zeilen 3+4). Anschließend werden die Parameter eines Ereignisses festgelegt. Beispielsweise wird bei einem `screenChanged` Ereignis die `screenID` und die Struktur des neues Screens übergeben. Diese hier deklarierten Ereignisse können sowohl im Code als auch in der Zustandsmaschine verwendet werden.

Listing A.2: Ausschnitt der NTG4_5 Klasse

```
1  Hashtable<String, String> conditions = new Hashtable<String, String>();
2
3  class NTG4_5 extends StateMachine
4  {
5    public Screen audioPlayerMR;
6
7    public NTG4_5()
8    {
9      //condition initialisation
10     this.initConditions();
11
12     //global Application Line
13     Vector<Entry> aplContent = new Vector<Entry>();
14     HMIButton new HMIButton(this, "apl_navi", "Navi");
```

```
15        aplContent.add(tmp);
16      HMIMenu apl = new HMIMenu("apl", aplContent);
17
18      //Player specific playfields
19      HMIMenu audio_player_mr_plf = new HMIMenu("audio_player_mr_plf");
20        audio_player_mr_plf.addEntry(new HMIEntry("audio_player_mr_plf_entry
              ", type_IMG, true));
21
22      //Player specific Subfunction Line
23      Vector<Entry> sflContent = new Vector<Entry>();
24
25      //Player screens
26      audioPlayerMR = new Screen("audio_player_mr",
27          status, apl, audio_player_mr_plf, audioPlayerSfl);
28    }
29
30    private void initConditions()
31    {
32      conditions.put(radioWaveband, radioWaveband_FM);
33      conditions.put(BTAudio, BTAudio_FS1);
34      conditions.put(CD, CD_CHANGER);
35    }
36  }
```

Mit der Programmierung sind wiederverwendbare Module beschrieben, die nun innerhalb der Zustandsmaschine instantiiert und in Zusammenhang gestellt werden. Das Objekt `NTG4_5` der Conformiq Klasse `StateMachine` ist in ein graphisches Modell und Programmcode aufgeteilt. Ein Ausschnitt des Programmcodes ist in Listing A.2 dargestellt. Die Systembedingungen werden durch die globale Variable `conditions` verwaltet, die außerhalb der Zustandsmaschine (Zeile 1) initialisiert und zu Beginn des Konstruktors (Zeile 10) durch die Methode `initConditions` gefüllt wird. Wiederum werden hier aus Ermangelung an Enumerationen global verfügbare, statische Strings verwendet. Hier wird unter anderem festgesetzt, dass das Wellenband der Radioapplikation auf FM voreingestellt ist (Zeile 29). Innerhalb der NTG4_5 Klasse werden die notwendigen Screens deklariert. In diesem Ausschnitt ist dies auf den Screen audioPlayerMR (entspricht dem Screen für die Music Register Subapplikation) beschränkt. In dem vollständigen Modell der Audioapplikation waren insgesamt 13 Screens notwendig. Innerhalb des Konstruktors werden die notwendigen Widgets instantiiert. In dem in Abschnitt 2.1 beschriebenen

Entwicklungsprozess den Angaben der Spezifikation gefolgt werden. Globale Elemente wie die APL und die Statuszeile werden in allen `Screen` Objekten wiederverwendet (Zeile 13-16). In diesem Ausschnitt wird der APL lediglich der Button «Navi» hinzugefügt. Elemente wie bspw. das PLF der Music Register Subapplikation (Zeile 19-20) können ausschließlich in diesem Kontext verwendet werden. Diese Objekte werden schließlich zu einem `Screen` Objekt zusammengeführt.

B

Evaluationsergebnisse

B.1. Fehlerhebenungsverlauf

Tabelle B.1.: *Fehlerbehebungsverlauf - Verfahrensvorschlag*

Mutation	313	315	318	319	320	322	325	327	328	332
1	4	2	3	2	4	4	2	3	4	4
2	2	2	1	2	2	4	1	4	3	1
3	4	3	3	3	4	4	3	4	4	3
4	4	3	3	3	4	1	0	1	1	2
5	4	2	2	0	2	2	1	2	2	1
6	1	1	1	1	1	4	1	4	4	3
7	2	3	3	2	2	3	3	4	3	3
8	4	2	1	3	4	2	2	3	2	3
9	3	1	2	1	3	3	2	3	3	2

Tabelle B.2.: *Fehlerbehebungsverlauf - Referenzverfahren*

Mutation	313	315	318	319	320	322	325	327	328	332
1	4	4	3	2	5	4	3	4	4	4
2	2	4	3	2	4	4	3	4	4	3
3	4	4	3	3	4	4	3	5	4	3
4	4	4	3	3	4	1	0	1	1	3
5	4	4	3	0	5	2	3	2	2	1
6	1	1	1	1	1	4	1	5	4	3
7	2	4	3	2	2	4	3	4	4	3
8	4	3	1	3	4	2	3	3	2	3
9	3	2	2	1	3	3	2	4	3	2

B.2. Fehlerberichte und Berichtsinstanzen

Tabelle B.3.: *Anzahl Fehlerberichte - Verfahrensvorschlag*

	313	315	318	319	320	322	325	327	328	332
1	2	33	5	2	2	2	19	2	2	13
2	2	48	47	20	20	13	47	9	6	13
3	45	8	10	8	12	48	11	9	50	39
4	11				16	12		24	10	2
5										
6										

Tabelle B.4.: *Anzahl Fehlerberichte - Referenzvorschlag*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	2	9	2	2	2	2	9	2	2	9
2	2	38	9	17	9	9	9	9	2	9
3	9	9	28	8	9	9	33	9	9	38
4	11	26			17	28		54	34	2
5					12			7		
6										

Tabelle B.5.: *Anzahl Berichtsinstanzen - Verfahrensvorschlag*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	54	122	74	54	54	54	79	54	54	71
2	71	138	135	83	92	71	130	62	74	71
3	130	102	106	102	68	135	149	62	137	318
4	109				118	108		154	106	4
5										
6										

Tabelle B.6.: *Berichtsinstanzen - Referenzverfahren*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	54	59	59	54	54	54	59	54	54	59
2	59	146	59	80	59	59	59	59	59	59
3	59	59	189	102	59	59	270	59	59	367
4	109	186			121	160		172	184	4
5					24			99		
6										

B.3. Fehlerfreie Verifikationen

Tabelle B.7.: *Anzahl fehlerfreier Verifikationen - Verfahrensvorschlag*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	0	163	55	0	0	0	179	0	0	153
2	141	163	169	260	141	141	188	136	41	159
3	172	339	323	339	150	163	287	136	163	213
4	323	367	367	367	223	310	367	327	314	351
5	367				367	367		367	367	367
6										

Tabelle B.8.: *Anzahl fehlerfreier Verifikationen - Referenzverfahren*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	0	136	50	0	0	0	136	0	0	136
2	136	136	136	251	136	136	136	136	50	136
3	136	136	172	327	136	136	154	136	136	172
4	311	172	367	367	202	292	367	251	284	351
5	367	367			343	367		327	367	367
6					367			367		

B.4. Reroutes

Tabelle B.9.: *Anzahl Reroutes*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	0	25	9	0	0	0	25	0	0	15
2	12	22	25	3	12	15	25	3	9	15
3	25	3	3	3	15	25	9	3	18	3
4	3				3	4		3	4	
Summe	40	50	37	6	30	44	59	9	31	33

B.5. Durchgeführte Testschritte

Tabelle B.10.: *Anzahl durchgeführter Testschritte - Verfahrensvorschlag*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	27	259	107	27	27	27	256	27	27	228
2	190	256	253	338	231	234	259	220	96	234
3	262	358	344	355	234	259	345	220	259	268
4	344	367	367	367	274	339	367	355	339	353
5	367				367	367		367	367	367
6										

Tabelle B.11.: *Anzahl durchgeführter Testschritte - Referenzverfahren*

	313	**315**	**318**	**319**	**320**	**322**	**325**	**327**	**328**	**332**
1	54	220	102	54	54	54	220	54	54	220
2	188	220	220	334	220	220	220	220	118	220
3	220	220	243	349	220	220	244	220	220	259
4	337	262	367	367	262	332	367	334	325	353
5	367	367			353	367		349	367	367
6					367			367		

B.6. Laufzeiten

Tabelle B.12.: *Laufzeiten - Verfahrensvorschlag*

	313	315	318	319	320	322	325	327	328	332
1	00:04,459	01:02,832	00:20,631	00:05,485	00:05,443	00:05,553	00:52,257	00:05,423	00:04,660	00:38,879
2	00:34,517	00:49,498	00:51,927	00:44,592	00:37,733	00:40,432	00:48,983	00:29,523	00:17,086	00:37,959
3	00:47,881	00:50,114	00:47,794	00:50,508	00:37,698	00:54,553	00:48,157	00:28,062	00:47,856	00:36,8
4	00:48,557	00:47,822	00:48,755	00:48,682	00:39,742	00:52,835	00:46,712	00:47,896	00:44,998	00:46,269
5	00:46,611				00:49,344	00:50,032		00:44,486	00:45,994	00:46,687
6										

Tabelle B.13.: *Laufzeiten - Referenzverfahren*

	313	315	318	319	320	322	325	327	328	332
1	00:05,595	00:35,962	00:13,073	00:04,234	00:05,182	00:05,227	00:27,634	00:05,219	00:05,315	00:28,218
2	00:29,523	00:37,334	00:26,427	00:41,040	00:27,3	00:27,754	00:26,483	00:27,577	00:13,246	00:26,199
3	00:29,597	00:30,440	00:32,898	00:46,665	00:27,822	00:27,971	00:32,776	00:27,523	00:26,859	00:35,455
4	00:49,268	00:37,088	00:32,625	00:48,505	00:34,006	00:45,064	00:49,594	00:41,836	00:42,514	00:45,087
5	00:50,804	00:46,250			00:46,481	00:47,042		00:46,105	00:46,308	00:47,091
6					00:46,654			00:47,090		

Abkürzungsverzeichnis

APG	App-Group
API	Application Programming Interface
APL	Application-Line
CSS	Cascading Style Sheet
DFB	Differenzierte Fehlerbehandlung
DFT	Designed for Testability
DOM	Document Object Model
DPAD	Directional Pad
EFG	Event Flow Graph
FOM	First Order Mutant
FSM	Finite State Machine
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol
HOM	Higher Order Mutant
IDE	Integrated Development Environments
IVI	In-Vehicle Infotainment
JS	JavaScript
MBT	Modellbasierte Testautomatisierung
MVC	Model View Controller
NTG	Neue Telematik-Generation
QML	Qtronic Modeling Language
PLF	Playfield
SFL	Subfunction-Line
SPG	Subapp-Group
SUT	System Under Test
UML	Unified Modeling Language
XHTML	Hypertext Markup Language

XML Extensible Markup Language

ZBE Zentrales Bedienelement

Literaturverzeichnis

[ADH+89] AGRAWAL, Hiralal ; DEMILLO, Richard A. ; HATHAWAY, Bob ; HSU, William ; HSU, Wynne ; KRAUSER, Edward W. ; MARTIN, Rhonda J. ; MATHUR, Aditya P. ; SPAFFORD, Eugene: Design of mutant operators for the C programming language. 1989. – Techreport

[AFT+14] AMALFITANO, Domenico ; FASOLINO, Anna R. ; TRAMONTANA, Porfirio ; TA, Bryan D. ; MEMON, Atif M.: MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps. In: IEEE Software (2014), Nr. 99, S. 1

[Ale12] ALEZ, Gaby (Hrsg.): Software Testing: Compatibility, Testing Methods, Non-Functional Testing, Automated Testing, Testing Tools, and More. Webster's Digital Services, 2012

[Ant14] ANTONY, Jiju: Design of experiments for engineers and scientists. Elsevier, 2014

[AO08] AMMANN, Paul ; OFFUTT, Jeff: Introduction to software testing. Cambridge University Press, 2008

[App15] APPLE: CarPlay. Version: 2015. https://www.apple.com/de/ios/carplay/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[AS09] ALT, Oliver ; SCHÜRR, Andy: Car Multimedia Systeme Modell-basiert testen mit SysML. Springer, 2009

[Aud14] AUDI AG: Audi TT - Instrumententafel. Version: 2014. https://www.audi-mediacenter.com/, Abruf: 24. Jul. 2015. – Abbildung

[AZJH11] ADAMOLI, Andrea ; ZAPARANUKS, Dmitrijs ; JOVIC, Milan ; HAUSWIRTH, Matthias: Automated GUI performance testing. In: Software Quality Journal 19 (2011), Nr. 4, S. 801–839

[BBG12] BELLI, Fevzi ; BEYAZIT, Mutlu ; GÜLER, Nevin: Event-Oriented, Model-Based GUI Testing and Reliability Assessment - Approach and Case Study. In: Advances in Computers 85 (2012), S. 277–326

[BC06] BØRRETZEN, J. ; CONRADI, R.: Results and experiences from an empirical study of fault reports in industrial projects. In: Product-Focused Software Process Improvement (2006), S. 389–394

[BDG+04] BAKER, Paul ; DAI, Zhen R. ; GRABOWSKI, Jens ; HAUGEN, Oystein ; LUCIO, Serge ; SAMUELSSON, Eric ; SCHIEFERDECKER, Ina ; WILLIAMS, Clay E.: The UML 2.0 testing profile. In: 8th Conference on Quality Engineering in Software Technology (QuEST), 2004, S. 181–189

[Bel01] BELLI, Fevzi: Finite state testing and analysis of graphical user interfaces. In: 12th International Symposium on Software Reliability Engineering (ISSRE) IEEE, 2001, S. 34–43

[Bin94] BINDER, Robert: Design for Testability in Object-oriented Systems. In: Commun. ACM 37 (1994), September, Nr. 9, S. 87–101

[Bla15] BLACKBERRY: BlackBerry Classic. Version: 2015. http://de.blackberry.com/content/dam/blackBerry/images/device/smartphone/blackberry-10/classic/structured/classic/de/desktop/black/specs-hero.png, Abruf: 24. Jul. 2015. – Abbildung

[BMV01] BARBOSA, Ellen F. ; MALDONADO, José C. ; VINCENZI, Auri Marcelo R.: Toward the determination of sufficient mutant operators for C. In: Software Testing, Verification and Reliability 11 (2001), Nr. 2, S. 113–136

[BMW15] BMW AG: BMW 7er Infotainment. Version: 2015. http://press.bmwgroup.com/, Abruf: 24. Jul. 2015. – Abbildung

[BNGM13] BANERJEE, Ishan ; NGUYEN, Bao ; GAROUSI, Vahid ; MEMON, Atif: Graphical user interface (GUI) testing: Systematic mapping and repository. In: Information and Software Technology 55 (2013), Nr. 10, S. 1679 – 1694

[Boa10] BOARD, German T.: ISTQB® GTB Standard Glossar der Testbegriffe: Deutsch, Englisch/Deutsch und Deutsch Englisch. 2010

[Bor06] BORTZ, Nicola Jürgen und D. Jürgen und Döring: Forschungsmethoden und evaluation: für Human-und Sozialwissenschaftler. Springer, 2006

[BS11] BRAESS, Hans-Hermann ; SEIFFERT, Ulrich: Vieweg Handbuch Kraftfahrzeugtechnik. Springer, 2011

[BSBV13] BAGNATO, Alessandra ; SADOVYKH, Andrey ; BROSSE, Etienne ; VOS, Tanja E.: The OMG UML Testing Profile in Use–An Industrial Case Study for the Future Internet Testing. In: 17th European Conference on Software Maintenance and Reengineering (CSMR) IEEE, 2013, S. 457–460

[BV90] BEIZER, Boris ; VINTER, Otto: Bug taxonomy and statistics. (1990)

[Chi99] CHILLAREGE, R.: Orthogonal defect classification. In: Handbook of Software Reliability Engineering (1999), S. 359–399

[Clo15] CLOUDMONKEY: MonkeyTalk. Version: 2015. http://www.cloudmonkeymobile.com/monkeytalk, Abruf: 24. Jul. 2015. – Produktbeschreibung

[CLS+09] CHINNAPONGSE, Vivien ; LEE, Insup ; SOKOLSKY, Oleg ; WANG, Shaohui ; JONES, Paul L.: Model-based testing of gui-driven applications. In: Software Technologies for Embedded and Ubiquitous Systems. Springer, 2009, S. 203–214

[CMM10] Carnegie Mellon University: CMMI Capability Maturity Model® Integration for Development, Version 1.3. November 2010

[Con15] CONFORMIQ: Conformiq Test Automation. Version: 2015. http://www.conformiq.com/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Cuc15] CUCUMBER: Cucumber. Version: 2015. http://cukes.info/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[CYM10] CHANG, Tsung-Hsiang ; YEH, Tom ; MILLER, Robert C.: GUI testing using computer vision. In: SIGCHI Conference on Human Factors in Computing Systems ACM, 2010, S. 1535–1544

[Dai13] DAIMLER AG: Mercedes-Benz S-Klasse - Instrumententafel. Version: 2013. http://media.daimler.com/, Abruf: 24. Jul. 2015. – Abbildung

[DHNH11] DUAN, Linshu ; HUSSMANN, Heinrich ; NIEDERKORN, Dieter ; HOFER, Alexander: Model-Based Testing for the Menu Behavior of Automotive Infotainment System HMIs. In: International Workshop on Model-based

Interactive Ubiquitous Systems (Modiquitous) at ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), 2011, S. 5–9

[Dor11] DORAU, Rainer: Emotionales Interaktionsdesign. In: Emotionales Interaktionsdesign: Gesten und Mimik interaktiver Systeme 1 (2011)

[Dri06] DRIVER FOCUS-TELEMATICS WORKING GROUP: Statement of principles, criteria and verification procedures on driver interactions with advanced in-vehicle information and communication systems. In: Alliance of Automotive Manufacturers (2006)

[Dua12] DUAN, Linshu: Model-based testing of automotive HMIs with consideration for product variability, Ludwig-Maximilians-Universität München, Diss., Juni 2012

[ENI05] International Organization for Standardization: EN ISO 9000:2005 - Quality management. 2005

[Eur15] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE: TTCN-3. Version: 2015. http://www.ttcn-3.org/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Fro15] FROGLOGIC: Squish. Version: 2015. http://www.froglogic.com/squish/gui-testing/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Goo14a] GOOGLE: Auto. Version: 2014. http://www.android.com/auto/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Goo14b] GOOGLE: Nexus - Fernbedienung & UI. Version: 2014. http://www.google.com/nexus/player/, Abruf: 24. Jul. 2015. – Abbildung

[Goo14c] GOOGLE: Talkback. Version: 2014. https://support.google.com/talkback/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Goo15] GOOGLE: Wear - UI. Version: 2015. https://developer.android.com/design/media/wear/selection_list.png, Abruf: 24. Jul. 2015. – Abbildung

[HBR+14] HONOLD, Frank ; BERCHER, Pascal ; RICHTER, Felix ; NOTHDURFT, Florian ; GEIER, Thomas ; BARTH, Roland ; HORNLE, Thilo ; SCHUSSEL, Felix ;

REUTER, Stephan ; RAU, Markus u. a.: Companion-technology: Towards user- and situation-adaptive functionality of technical systems. In: International Conference on Intelligent Environments (IE) IEEE, 2014, S. 378–381

[Hei14] HEILEMANN, Miriam: Automatisierte Verifikation des Designs von grafischen Benutzeroberflächen mit Bayes' schen Netzen-Konzeptualisierung und praktische Anwendung am Beispiel des Audi Infotainmentsystems MMI, Universität Ulm, Diss., 2014

[HMSY13] HARMAN, Mark ; MCMINN, Phil ; SHAHBAZ, Muzammil ; YOO, Shin: A Comprehensive Survey of Trends in Oracles for Software Testing. 2013. – Techreport

[Hof08] HOFFMANN, Dirk W.: Software-Qualität. Springer Science & Business, 2008

[Hol13] HOLL, Konstantin: An efficient quality assurance method for mobile business application development projects. In: Tagungsband des DASMA Software Metrik Kongresses (MetriKon), 2013, S. 301–306

[HSW+13] HONOLD, Frank ; SCHUSSEL, Felix ; WEBER, Matthias ; NOTHDURFT, Florian ; BERTRAND, Gregor ; MINKER, Wolfgang: Context models for adaptive dialogs and multimodal interaction. In: International Conference on Intelligent Environments (IE) IEEE, 2013, S. 57–64

[IEC08] International Electrotechnical Commission: IEC 61508:2008 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. 2008

[IEE09] Institute of Electrical and Electronics Engineers: IEEE 1044:2009 - Standard Classification for Software Anomalies

[ISO08] International Organization for Standardization: ISO IEC 12207:2008 - Systems and Software Engineering - Software Life Cycle Processes. 2008

[ISO10] International Organization for Standardization: ISO 9241-210:2010 Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems. 2010

[ISO11] International Organization for Standardization: ISO 26262:2011 - Road vehicles - Functional safety. 2011

[Jää11] JÄÄSKELÄINEN, Antti: Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications. Tampere, Finland, Tampere University of Technology, Diss., Januar 2011

[JTK11] JÄÄSKELÄINEN, Antti ; TAKALA, Tommi ; KATARA, Mika: Model-Based GUI Testing of Smartphone Applications: Case S60 and Linux. In: ZANDER, Justyna (Hrsg.) ; SCHIEFERDECKER, Ina (Hrsg.) ; MOSTERMAN, Pieter J. (Hrsg.): Model-Based Testing for Embedded Systems. Boca Raton, FL, USA : CRC Press, September 2011 (Computational Analysis, Synthesis, and Design of Dynamic Systems), Kapitel 18, S. 525–544

[JVCS08] JACKY, Jonathan ; VEANES, Margus ; CAMPBELL, Colin ; SCHULTE, Wolfram: Model-based software testing and analysis with Csharp. Bd. 1. Cambridge University Press Cambridge, 2008

[KG96] KASIK, David J. ; GEORGE, Harry G.: Toward automatic generation of novice user test scripts. In: SIGCHI Conference on Human Factors in Computing Systems ACM, 1996, S. 244–251

[KMPK05] KERVINEN, Antti ; MAUNUMAA, Mika ; PÄÄKKÖNEN, Tuula ; KATARA, Mika: Model-Based Testing Through a GUI. In: GRIESKAMP, Wolfgang (Hrsg.) ; WEISE, Carsten (Hrsg.): International Workshop on Formal Approaches to Testing of Software (FATES) Bd. 3997. Berlin, Heidelberg : Springer, Juli 2005 (Lecture Notes in Computer Science), S. 16–31

[LLS10] LI, N. ; LI, Z. ; SUN, X.: Classification of Software Defect Detected by Black-box Testing: An Empirical Study. In: World Congress on Software Engineering (WCSE) Bd. 2 IEEE, 2010, S. 234–240

[Mar08] MARWEDEL, Peter: Eingebettete Systeme. Korrigierter Nachdr. Berlin; Heidelberg : Springer, 2008. – XVIII, 264 S.

[Mar14] MARTIN, Leonhard J.: Methodik zur Erzeugung eines wirklichkeitsgetreuen Fehlerbildes in graphischen Benutzeroberflächen, Universität Ulm, Diplomarbeit, 2014

[MBHN03] MEMON, Atif ; BANERJEE, Ishan ; HASHMI, Nada ; NAGARAJAN, Adithya: DART: A Framework for Regression Testing Nightly/daily Builds of GUI Appli-

cations. In: International Conference on Software Maintenance (ICSM) IEEE, 2003, S. 410–419

[MBN03] MEMON, Atif M. ; BANERJEE, Ishan ; NAGARAJAN, Adithya: GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In: Working Conference on Reverse Engineering, 2003, S. 260–269

[Mem01] MEMON, Atif M.: A comprehensive framework for testing graphical user interfaces, University of Pittsburgh, Diss., 2001

[Mem03] MEMON, Atif M.: Advances in GUI Testing. In: ZELKOWITZ, Marvin V. (Hrsg.): Highly Dependable Software – Advances in Computers Bd. 58. Academic Press, 2003, S. 149–201

[Mem07] MEMON, Atif M.: An event-flow model of GUI-based applications for testing. In: Software Testing, Verification and Reliability 17 (2007), Nr. 3, S. 137–157

[MKH13] MAUSER, Daniel ; KLAUS, Alexander ; HOLL, Konstantin: Towards a GUI Test Model Using State Charts and Programming Code. In: Testing Software and Systems. Springer, 2013, S. 271–276

[MKHZ13] MAUSER, Daniel ; KLAUS, Alexander ; HOLL, Konstantin ; ZHANG, Ran: GUI Failures of In-Vehicle Infotainment: Analysis, Classification, Challenges, and Capabilities. In: International Journal On Advances in Software 6 (2013), Nr. 1 and 2, S. 142–154

[MKZD12] MAUSER, Daniel ; KLAUS, Alexander ; ZHANG, Ran ; DUAN, Linshu: GUI failure analysis and classification for the development of in-vehicle infotainment. In: International Conference on Advances in System Testing and Validation Lifecycle (VALID), 2012, S. 79–84

[Moz15] MOZILLA: Mozilla - Firefox. Version: 2015. `https://www.mozilla.org/de/firefox`, Abruf: 24. Jul. 2015. – Produktbeschreibung

[MP15] MOORE, Philip T. ; PHAM, Hai V.: Personalization and rule strategies in data-intensive intelligent context-aware systems. In: The Knowledge Engineering Review 30 (2015), 3, S. 140–156. – ISSN 1469–8005

[MPM13] MOREIRA, Rodrigo M. L. M. ; PAIVA, Ana C. R. ; MEMON, Atif: A Pattern-Based Approach for GUI Modeling and Testing. In: Proceedings of the 24th annual

International Symposium on Software Reliability Engineering (ISSRE 2013), IEEE Press, 2013, S. 288–297

[MPS99] MEMON, Atif M. ; POLLACK, Martha E. ; SOFFA, Mary L.: Using a goal-driven approach to generate test cases for GUIs. In: International conference on Software engineering (ICSE). Los Alamitos, CA, USA : IEEE Computer Society Press, 1999, S. 257–266

[MRT09] MARCHETTO, Alessandro ; RICCA, Filippo ; TONELLA, Paolo: An empirical validation of a web fault taxonomy and its usage for web testing. In: Journal of Web Engineering 8 (2009), Nr. 4, S. 316–345

[MS11] MARATHE, Sampada ; SUNDAR, S S.: What drives customization?: control or identity? In: SIGCHI Conference on Human Factors in Computing Systems ACM, 2011, S. 781–790

[MSP01] MEMON, Atif M. ; SOFFA, Mary L. ; POLLACK, Martha E.: Coverage criteria for GUI testing. In: European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. New York, NY, USA : ACM Press, 2001, S. 256–267

[NRBM13] NGUYEN, BaoN. ; ROBBINS, Bryan ; BANERJEE, Ishan ; MEMON, Atif: GUI-TAR: an innovative tool for automated testing of GUI-driven software. In: Automated Software Engineering (2013), S. 1–41

[Off92] OFFUTT, A J.: Investigations of the software testing coupling effect. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 1 (1992), Nr. 1, S. 5–20

[Pat02] PATERNÒ, Fabio: ConcurTaskTrees: an engineered approach to model-based design of interactive systems. In: The Handbook of Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates (2002), S. 483–500

[PB95] PALANQUE, Philippe A. ; BASTIDE, Remi: Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In: Interactive systems: Design, specification, and verification. Springer, 1995, S. 383–400

[Peb14] PEBBLE: Smartwatch UI. Version: 2014. http://cdn.androidcommunity.com/wp-content/uploads/2014/05/pebble-pandora-600x400.jpg, Abruf: 24. Jul. 2015. – Abbildung

[PFV07] PAIVA, Ana C. ; FARIA, Joao C. ; VIDAL, Raul F.: Towards the integration of visual and formal models for GUI testing. In: Electronic Notes in Theoretical Computer Science 190 (2007), Nr. 2, S. 99–111

[PMM97] PATERNÒ, Fabio ; MANCINI, Cristiano ; MENICONI, Silvia: ConcurTaskTrees: A diagrammatic notation for specifying task models. In: Human-Computer Interaction (INTERACT). Springer US, 1997, S. 362–369

[PTFV05] PAIVA, Ana C. ; TILLMANN, Nikolai ; FARIA, Joao C. ; VIDAL, Raul F.: Modeling and testing hierarchical GUIs. In: International Workshop on Abstract State Machines (ASM), 2005, S. 329–344

[Ran15] RANOREX: Testautomatisierung-Tools. Version: 2015. http://www.ranorex.com/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[RBGW12] ROSSNER, Thomas ; BRANDES, Christian ; GOETZ, Helmut ; WINTER, Mario: Basiswissen Modellbasierter Test. dpunkt. verlag, 2012

[REG07] REZA, Hassan ; ENDAPALLY, Sandeep ; GRANT, Emanuel: A model-based approach for testing gui using hierarchical predicate transition nets. In: International Conference on Information Technology (ITNG) IEEE, 2007, S. 366–370

[Rha15] RHAPSODY: Rhapsody ATG. Version: 2015. http://www.btc-es.de/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[RMD10] ROEST, Danny ; MESBAH, Ali ; DEURSEN, Arie van: Regression testing ajax applications: Coping with dynamism. In: International Conference on Software Testing, Verification and Validation (ICST) IEEE, 2010, S. 127–136

[Rom05] ROMBACH, Dieter (Hrsg.): Software Engineering eingebetteter Systeme: Grundlagen - Methodik - Anwendungen. 1. Aufl. München : Elsevier, Spektrum Akad. Verl., 2005. – XV, S. 585. – ISBN 3–8274–1533–0

[Rou15] ROUTINEBOT: RoutineBot. Version: 2015. http://www.routinebot.com, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Roy70] ROYCE, Winston W.: Managing the development of large software systems. In: IEEE WESCON Bd. 26, 1970, S. 328–388

[SBK05] STOLLE, Reinhard ; BENEDEK, Thomas ; KNUECHEL, Christian: Model-based Test Automation for Automotive Human Machine Interfaces. In: Technische Universität Berlin-Forschungsberichte der Fakultät IV, Bericht (2005)

[Sch07] SCHIEFERDECKER, Ina: Modellbasiertes testen. In: Journal OBJEKTspektrum, SIGSDATACOM 3 (2007), S. 39–45

[SCP08] SILVA, José L ; CAMPOS, José C. ; PAIVA, Ana C.: Model-based user interface testing with spec explorer and concurtasktrees. In: Electronic Notes in Theoretical Computer Science 208 (2008), S. 77–93

[Sel15] SELENIUM: Selenium. Version: 2015. http://www.seleniumhq.org, Abruf: 24. Jul. 2015. – Produktbeschreibung

[sep15] SEPP.MED: MBT Suite. Version: 2015. http://www.seppmed.de/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Sie15] SIEMENS AG: Siemens - Kochfeld - EF601HN17. Version: 2015. http://www.siemens-home.de/store/medias/sys_master/vib/hab/hef/9129724280862/MCSA00710198-2-EF601HN17-def.jpg, Abruf: 24. Jul. 2015. – Abbildung

[Sik15] SIKULI: Sikuli. Version: 2015. http://www.sikuli.org/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[sma15] SMARTTESTING: Test Designer. Version: 2015. http://www.smartesting.com/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[Spi06] SPILLNER, A.: Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum certified Tester ; advanced Level nach ISTQB-Standard. dpunkt-Verlag, 2006

[SS97] SHEHADY, Richard K. ; SIEWIOREK, Daniel P.: A method to automate user interface testing using variable finite state machines. In: International Symposium on Fault-Tolerant Computing (FTCS) IEEE, 1997, S. 80–88

[Tes15] TESTPLANT: eggPlant. Version: 2015. http://www.testplant.com/, Abruf: 24. Jul. 2015. – Produktbeschreibung

[WA00] WHITE, Lee ; ALMEZEN, Husain: Generating test cases for GUI responsibilities using complete interaction sequences. In: International Symposium on Software Reliability Engineering (ISSRE) IEEE, 2000, S. 110–121

[WAA01] WHITE, Lee ; ALMEZEN, Husain ; ALZEIDI, Nasser: User-based testing of GUI sequences and their interactions. In: International Symposium on Software Reliability Engineering (ISSRE) IEEE, 2001, S. 54–63

[WRH+12] WOHLIN, Claes ; RUNESON, Per ; HÖST, Martin ; OHLSSON, Magnus C. ; REGNELL, Björn ; WESSLÉN, Anders: Experimentation in software engineering. Springer, 2012

[WS13] WEISS, Michaela ; SCHWEIGGERT, Franz: Opportunities and Challenges of Software Customization. In: ACEEE International Journal of Infomation Technology (2013), S. 1–11

[XM07] XIE, Qing ; MEMON, Atif M.: Designing and comparing automated test oracles for GUI-based software applications. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 16 (2007), Nr. 1, S. 4

[ZDSD05] ZANDER, Justyna ; DAI, Zhen R. ; SCHIEFERDECKER, Ina ; DIN, George: From U2TP models to executable tests with TTCN-3-an approach to model driven testing. In: Testing of Communicating Systems. Springer, 2005, S. 289–303

[ZH00] ZHU, Hong ; HE, Xudong: A Theory of Testing High Level Petri Nets. In: International Conference on Software: Theory and Practice (IFIP), 2000, S. 443–450