MODELLBASIERTER EVOLUTIONÄRER FUNKTIONSTEST

vorgelegt von
Dipl.-Ing. Felix Lindlar
aus Berlin

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuss:
Prof. Dr. Odej Kao (Vorsitzender)
Prof. Dr.-Ing. Stefan Jähnichen (Gutachter)
Prof. Dr.-Ing. Armin Zimmermann (Gutachter)

Tag der wissenschaftlichen Aussprache:
07. Mai 2012

Berlin 2012
D 83

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik
Fachgebiet Softwaretechnik
Ernst-Reuter-Platz 7
D-10587 Berlin

Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet System- und Software-Engineering
Ehrenbergstr. 29
D-98693 Ilmenau

Felix Lindlar: *Modellbasierter evolutionärer Funktionstest*

DANKSAGUNG

Viele Menschen haben mir durch ihre Zeit und Fähigkeiten geholfen, diese Arbeit fertigzustellen. An vorderster Stelle möchte ich mich bei Prof. Stefan Jähnichen für seine Unterstützung bedanken. Durch wertvolle Vorschläge, lehrreiche Kritik und durch seine motivierende Art, hat er maßgeblich zum erfolgreichen Abschluss beigetragen. Bedanken möchte ich mich auch bei Prof. Armin Zimmermann für seine Unterstützung und seine guten Ratschläge.

Dank gilt meinen Kollegen Andreas Windisch, Benjamin Wilmes und Lars Kristian Klauske für anregende Diskussionen und die Korrekturlesung einzelner Kapitel.

Ferner möchte ich mich für die Unterstützung seitens der Daimler AG bedanken. Hervorheben möchte ich Dirk Johanson, Christian Scheidler, Michael Weber und Dr. Klaus Grimm.

Bedanken möchte ich mich ebenfalls bei meinem guten Freund Julian Röder für zahlreiche Hinweise und eine umfassenden Korrekturlesung der Arbeit.

Besonderer Dank gilt meiner Freundin Katja für wiederholte Korrekturlesungen der Arbeit. Darüber hinaus möchte ich ihr dafür danken, dass sie mich in allen Belangen des Lebens unterstützt und dadurch entscheidend zum erfolgreichen Abschluss der Promotion beigetragen hat.

Schließlich möchte ich mich bei meinem Vater Josef bedanken. Seine fürsorgliche und großherzige Art hat mir eine glückliche Kindheit und einen guten Start ins Leben ermöglicht. Er steht mir zudem immer als Vorbild und mit gutem Rat zur Seite.

Felix Lindlar
Berlin, 07. Mai 2012

KURZFASSUNG

Die Erforschung effizienterer und effektiverer Testansätze für die Entwicklung von Software eingebetteter Systeme wurde in den vergangenen Jahren durch den stetig zunehmenden Kostenanteil der Qualitätssicherung beflügelt. Ein vielversprechender Ansatz für die Automatisierung einer Vielzahl von Testarten ist das evolutionäre Testen. Dabei wird das Problem der Ermittlung von Testfällen in ein Optimierungsproblem transformiert, welches mit evolutionären Algorithmen gelöst wird. Der Schwerpunkt dieser Arbeit liegt auf dem evolutionären Funktionstest (EFT). Im Gegensatz zu Verfahren zur formalen Verifikation (scheitern aufgrund der kombinatorischen Zustandsexplosion), kann der EFT auch auf sehr komplexe Testobjekte angewendet werden.

Aufgrund der Zustandsbehaftung von Software eingebetteter Systeme, ist die Erzeugung komplexer Testdatensequenzen für die Testausführung unabdingbar. Zudem müssen physikalische Grenzen und Zusammenhänge berücksichtigt werden. Für einen effektiven Testansatz muss darüber hinaus Reaktivität zwischen Testobjekt und Teststeuerung ermöglicht werden, denn nur dann können Testdaten auch während der Ausführung entsprechend dem Verhalten des Testobjekts angepasst werden.

Das Ergebnis dieser Forschungsarbeit sind zwei Verfahren für den automatisierten Funktionstest von komplexer Software eingebetteter Systeme. Die Verfahren erlauben die Spezifikation realistischer Testdatensequenzen mit hybriden Automaten sowie mit einer hierarchischen Auszeichnungssprache. Durch die Verwendung evolutionärer Algorithmen werden die Testdatensequenzen variiert. Das modellbasierte Verfahren unterstützt die dynamische Anpassung der Testdatensequenzen während der Testausführung mit Zwischenergebnissen aus dem bisherigen Testverlauf. Die Bewertung der Testläufe erfolgt durch automatisch instrumentierte Testauswerteskripte. Weiterhin wird eine Testumgebung vorgestellt, die eine plattformübergreifende Testausführung ermöglicht. Um die Verfahren im Hinblick auf ihre Anwendbarkeit in der Industrie zu validieren, wurden umfassende Fallstudien mit komplexen Softwaresystemen aus der Serienentwicklung durchgeführt. Dabei gewonnene Resultate lassen auf eine hohe Effektivität und Anwendbarkeit der Verfahren schließen.

ABSTRACT

In recent years, the increasing cost proportion of quality assurance has stimulated research into more efficient and effective testing approaches for embedded software development. A promising approach for automating various test types is evolutionary testing. With evolutionary testing the problem of selecting suitable test cases is transformed into an optimization problem, which is then solved by evolutionary algorithms. The main focus of this work is on evolutionary functional testing (EFT). In contrast to formal verification approaches, which fail due to the problem of state explosion, EFT can be used for highly complex test objects.

Large numbers of internal states in embedded software require testing with complex test data sequences in order to perform tests effectively. In addition, when testing embedded software, physical boundaries and relationships between test data sequences have to be considered. Furthermore, test approaches should allow for reactivity between test object and test control. Only then, can test data be adapted during test execution in correspondence to the behaviour of the test object.

The results of this research are two approaches for automating functional testing of complex embedded software. The approaches facilitate specifying realistic test data sequences with hybrid automata and with a hierarchical mark-up language. Test data sequences are varied by using evolutionary algorithms. The model-based approach allows for dynamically adapting test data sequences during test execution by using test data achieved in the preceding test process. Test runs are evaluated using automatically instrumented assessment scripts. Furthermore, a test environment is presented, enabling cross-platform test execution. In order to evaluate the approaches with respect to usability in an industrial setting, thorough case studies using complex embedded software from automotive serial production have been conducted. Results indicate high effectiveness and usability of the approaches.

VERÖFFENTLICHUNGEN

Die in dieser Dissertation vorgestellte Arbeit entstand zwischen Januar 2008 und Dezember 2011 an der Technischen Universität Berlin (Daimler Center for Automotive Information Technology Innovations). Einige Ideen und Darstellungen sind bereits in den im Folgenden aufgelisteten vorherigen Veröffentlichungen des Autors erschienen:

JOURNALBEITRÄGE

- T. Vos, **F. Lindlar**, B. Wilmes, A. Windisch, A. Baars, P. Kruse, H. Gross und J. Wegener: *Evolutionary Functional Black-Box Testing in an Industrial Setting*, Software Quality Journal (SQJO), Februar 2012.

KONFERENZBEITRÄGE

- **F. Lindlar** und A. Windisch *A Search-Based Approach to Functional Hardware-in-the-Loop Testing*, Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010), September 2010, Benevento, Italien.
- **F. Lindlar**, A. Windisch und J. Wegener: *Integrating Model-Based Testing with Evolutionary Functional Testing*, Proceedings of the 3rd International Workshop on Search-Based Software Testing in conjunction with the 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST 2010), April 2010, Paris, Frankreich.
- **F. Lindlar**, A. Marrero: *Using evolutionary algorithms to select parameters from equivalence classes*, Dagstuhl Seminar Proceedings: Evolutionary Test Generation, 2009, Dagstuhl, Deutschland.
- B. Wilmes, **F. Lindlar**, A. Windisch: *Suchbasierter Test für den industriellen Einsatz*, Tagungsband des 4. Symposiums „Testen im System- und Software-Life-Cycle“, Technische Akademie Esslingen, November 2011, Ostfildern, Deutschland.

- A. Baars, P. Kruse, **F. Lindlar**, T. Vos, J. Wegener und A. Windisch: *Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool*, Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST2010), April 2010, Paris, Frankreich.
- A. Windisch, **F. Lindlar**, S. Topuz und S. Wappler: *Evolutionary Functional Testing of Continuous Control Systems*, Proceedings of the 11th ACM Genetic and Evolutionary Computation Conference (GECCO 2009), Juli 2009, Montréal, Kanada.

DOKTORANDENSYMPOSIEN

- **F. Lindlar**: *Search-Based Functional Testing of Embedded Software Systems*, International Doctoral Symposium in conjunction with the 2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009), April 2009, Denver, USA.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

1 EINLEITUNG

„Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen."

— E. W. Dijkstra (1930 - 2002)

1.1 HINTERGRUND UND MOTIVATION

Der Einfluss eingebetteter Systeme nimmt weltweit rapide zu. In vielen produzierenden Industriezweigen liegt der Wertschöpfungsanteil dieser Systeme am Gesamtprodukt bereits bei bis zu 80% – Tendenz steigend. Laut einer Studie wird das Marktvolumen für eingebettete Systeme allein in Deutschland im Jahr 2011 bei knapp 20 Mrd. Euro liegen [10]. Die Automobilindustrie ist besonders durch die technologischen Möglichkeiten der stetigen Weiterentwicklung eingebetteter Systeme geprägt [50]. In diesem Industriefeld werden eingebettete Systeme auf Grund des Aufgabenschwerpunkts der Steuerung und Regelung von Fahrzeugfunktionen grundsätzlich als elektronische Steuergeräte bezeichnet. Im Jahr 1967 wurde von der Firma Bosch eine elektronisch gesteuerte Benzineinspritzung entwickelt. Im Jahr 2011 wird bereits der Großteil aller Funktionen in einem modernen Kraftfahrzeug von einem Netzwerk mit bis zu 80 Steuergeräten realisiert [11, 99]. Fahrerassistenzsysteme, die für mehr Komfort und Sicherheit beim Fahren sorgen, sowie Maßnahmen zur Verringerung der Umweltbelastung, sind einige der Kernfunktionalitäten, die heutzutage serienmäßig als Verbund von Steuergeräten, Sensoren und Aktoren im Fahrzeug verbaut werden.

Die zunehmende Komplexität der Software für Steuergeräte und der zunehmende Grad ihrer Vernetzung stellen allerdings immer

höhere Ansprüche an den Entwicklungsprozess und insbesondere an die Qualitätssicherung. Eine Fehlfunktion bei einem Fahrerassistenzsystem kann unter Umständen zu hohen Kosten durch eine Rückrufaktion oder sogar zu einer Gefahr für die Insassen und Passanten werden. Obwohl in der Vergangenheit die Ursachen für Rückrufaktionen im Automobilumfeld zumeist auf mechanische oder Hardware-Probleme zurückzuführen waren, gab es auch vermehrt Fälle, bei denen die Ursache in fehlerhafter Software lag. Toyota musste zum Beispiel im Jahr 2010 die Steuerungssoftware des Antiblockiersystems (ABS) von über 52.000 Fahrzeugen allein in Europa ersetzen [113]. BMW musste im Jahr 2008 20.000 Fahrzeuge in die Werkstatt beordern, um die Motorsteuerungssoftware zu aktualisieren [31]. Aktuell wird bei der Entwicklung sicherheitsrelevanter Systeme bereits 50±20% der Entwicklungskosten für das Testen ausgegeben. Dennoch lässt sich nicht ausschließen, dass Fehler übersehen werden [8, 37]. Die berühmte Aussage von Dijkstra bleibt zumindest für derart komplexe Systeme weiterhin bestehen [32].

Eine Steigerung der Qualität von Software eingebetteter Systeme kann bei gleichbleibendem Aufwand durch besser ausgebildete Tester erreicht werden. In der Vergangenheit war der Beruf des Testers allerdings eher unbeliebt. Testen wurde oft als langweilig und Tester als weniger wichtig als Softwareentwickler angesehen. Weiterhin stellten viele Universitäten ihr Lehrprogramm sehr langsam auf die erhöhte Nachfrage aus der Industrie ein [23]. Zusätzlich kann eine erhöhte Effizienz des Testprozesses durch verbesserte Testverfahren erreicht werden. Beim Testen von Steuergerätesoftware in der Automobilindustrie besteht der Großteil des Testaufwands im manuellen Design, der manuellen Auswahl und der manuellen Auswertung von Tests. Geeignete Tools für die Automatisierung dieser zeitaufwendigen Aufgaben können zu erheblichen Kosteneinsparungen beitragen. Zudem werden Fehler oft erst sehr spät im Entwicklungszyklus entdeckt. Obwohl beispielsweise der Fahrversuch zu den teuersten Qualitätssicherungsmaßnahmen zählt, wird er weiterhin intensiv eingesetzt – auch um Spezifikations- und Implementierungsfehler aufzudecken [24]. Es gilt als allgemein anerkannt, dass das frühe Aufdecken von Fehlern im Entwicklungszyklus große Einsparpotentiale bietet (vergleiche Spillner und Linz [103]).

Beim Testen von Software eingebetteter Systeme werden verschiedene Testarten angewendet. Zu den gängigsten zählen der Strukturtest (Überdeckung von Programmcode), der nicht-funktionale Test (zum Beispiel für die Absicherung der maximalen Ausführungszeit einer Funktion) und der Funktionstest (Überprüfung funktionaler Anforderungen). In dieser Arbeit werden Verbesserungen für den Funktionstest vorgestellt. Funktionstests werden bei der Softwareentwicklung für Steuergeräte während des gesamten Qualitätssicherungszyklus durchgeführt. Angefangen mit Funktionstests einzelner Softwaremodule, einer Integration dieser Softwaremodule, werden Funktionstests auch realen Steuergeräten unter Echtzeitbedingungen und bei Fahrversuchen durchgeführt. In dieser Arbeit werden ausschließlich reine Black-Box-Testverfahren betrachtet – die zugrundeliegende Struktur des jeweiligen Testobjekts wird nicht berücksichtigt. Testfälle werden hierbei von den funktionalen Anforderungen abgeleitet und dynamisch auf verschiedenen Testplattformen ausgeführt. Ein *Testfall* enthält Stimuli für die dynamische Ausführung eines Testobjekts und das erwartete Verhalten. Der Begriff *Testplattform* wird als eine Ausführungsplattform aufgefasst, die die notwendige Infrastruktur für die Durchführung von Tests bereitstellt.

Ein Trend bei der Entwicklung von Software eingebetteter Systeme ist die modellbasierte Entwicklung (MBE) [13]. Im Unterschied zu herkömmlichen Entwicklungstechniken wird bei der MBE das gewünschte Systemverhalten in Form abstrakter Modelle spezifiziert [104, 111]. Mit Hilfe von Codegeneratoren wird automatisch Programmcode erzeugt und auf ein Steuergerät geladen. Zu den Vorteilen der MBE zählen ein höherer Abstraktionsgrad sowie die Möglichkeit der Einschränkung von Freiheitsgraden im Vergleich zu Programmiersprachen. Dadurch kann insbesondere bei der (verteilten) Entwicklung von Software eingebetteter Systeme eine Effizienzsteigerung des Entwicklungsprozesses erreicht werden [101]. Ein weiterer positiver Effekt der MBE ist, dass die dabei entstehenden ausführbaren Modelle für den Testprozess verwendet werden können. In diesem Zusammenhang wurde insbesondere auch das Prinzip des modellbasierten Testens (MBT) erforscht [13, 96].

Das Prinzip des MBT ist ein aktuelles Forschungsfeld und deshalb ist der Begriff MBT noch nicht eindeutig definiert. Im Automobilbereich bezeichnet das MBT alle Testaktivitäten im Zusammen-

hang mit der MBE [13, 27]. In dieser Arbeit wird der Begriff MBT weiter eingeschränkt und als diejenige Praxis verstanden, bei der Testfälle von einem Testmodell abgeleitet werden, um ein ausführbares System zu testen. Das MBT bietet die Möglichkeit, durch ein explizites mentales Modell, Tests systematisch abzuleiten und die Effizienz im Hinblick auf den Umfang einer Testreihe zu steigern [95, 96]. Oft ist aber ein erhöhter Aufwand erforderlich, um das Testmodell zu erstellen und mit dem System konsistent zu halten. Ein Testverfahren für den MBT, das insbesondere im Automobilbereich Einzug gefunden hat, ist das Time Partition Testing (TPT) [71, 91]. Das modellbasierte Testverfahren TPT wurde speziell für den systematischen Test von Software eingebetteter Systeme entwickelt. Der Modellierungsansatz des zweiten in dieser Arbeit entwickelten Verfahrens (Kapitel 5) orientiert sich an der Methodologie des TPT.

Unabhängig von dem verwendeten Testverfahren ist der Eingangsdatenraum von Software eingebetteter Systeme im Regelfall zu groß, als dass jede mögliche Datenkombination getestet werden kann. Eine gängige Vorgehensweise zur Lösung dieses Problems ist die Unterteilung der Eingangsdatenräume in Äquivalenzklassen [40]. Beim Black-Box-Testen wird eine Äquivalenzklasse als ein Bereich von Werten angesehen, für den sich ein Testobjekt jeweils gleich in Bezug auf die untersuchten Eigenschaften verhält. Die Bestimmung korrekter Äquivalenzklassen ist essentiell für eine erfolgreiche Testreihe. Konkrete Testeingaben werden aus den Äquivalenzklassen durch Grenzwertanalysen oder durch die Auswahl von Stichproben bestimmt. Bei inkorrekter Äquivalenzklassendefinition führen unterschiedliche Werte aus einer erwarteten Äquivalenzklasse zu unterschiedlichem Verhalten des Testobjekts im Hinblick auf die untersuchten funktionalen Aspekte. Die Wahrscheinlichkeit Fehler zu übersehen, ist dann erheblich höher [73]. Die Qualitätssicherung von Software eingebetteter Systeme wird zusätzlich durch eine Vielzahl interner Systemzustände und Regelschleifen erschwert [133]. Es reicht nicht aus, ein Testobjekt mit jeweils genau einem Wert für jeden Eingang auszuführen. Stattdessen müssen die Eingangskanäle des Testobjekts mit Sequenzen von Daten belegt werden, um interne Zustandswechsel zu provozieren und das Testobjekt unter realen Bedingungen zu testen. Erfahrungen aus der Industrie haben gezeigt, dass die manuelle Beschreibung von Testdatensequenzen sehr zeitaufwändig sein kann.

1.2 Zielstellung und Beitrag zur Forschung

Um dem Problem des hohen Aufwands beim manuellen Testen zu begegnen, wurden in den letzten Jahren verstärkt Testautomatisierungsansätze erforscht. Ziel war es, den hohen Anteil der Kosten für die Qualitätssicherung an den Gesamtkosten zu reduzieren und die Wahrscheinlichkeit zu minimieren, dass Fehler übersehen werden. Die Kernaktivitäten des Softwaretestens sind Testmanagement, Spezifikation, Implementierung, Ausführung, Auswertung und Dokumentation. Der Großteil der bis dato existierenden Testautomatisierungsansätze für Software eingebetteter Systeme zielt lediglich auf die Automatisierung eines Teils dieser Aktivitäten ab. Allein die Automatisierung der Testdurchführung hat in den vergangenen Jahren schon zu beachtlichen Zeiteinsparungen geführt. Eine Automatisierung weiterer Aktivitäten bietet große Optimierungspotentiale.

Ein vielversprechender Ansatz, der darauf abzielt, Testfälle automatisch zu erstellen, auszuführen und zu bewerten, ist das evolutionäre Testen [106, 124]. Beim evolutionären Testen wird das Problem der Ableitung von Testfällen für ein spezifisches Testziel in ein Optimierungsproblem transformiert. Mit Hilfe evolutionärer Algorithmen wird dann nach Lösungen für das Problem gesucht. Evolutionäres Testen kann für Strukturtests, nicht-funktionale Tests und Funktionstests verwendet werden. Die im Rahmen dieser Arbeit entwickelten Verfahren realisieren einen evolutionären Funktionstest. Der evolutionäre Funktionstest ist noch eine relativ neue Technologie, die das Ziel verfolgt, automatisiert Testfälle zu bestimmen, die ein Fehlverhalten in einem System aufzeigen [79]. Ein Fehlverhalten kann beispielsweise das Überschreiten einer Oberschranke für den Ausgangskanal eines Testobjekts sein. Bei evolutionären Funktionstests wird eine hohe Anzahl zielgerichteter Testfälle ausgeführt und überwacht, ob sich ein System korrekt in Bezug auf die untersuchten Eigenschaften verhält. Voraussetzung für die Automatisierung sind ein hinreichend eingeschränkter Suchraum und die Spezifikation einer geeigneten Fitnessfunktion für die Bewertung der Testfälle.

Seit Beginn des einundzwanzigsten Jahrhunderts wurde in einigen Arbeiten die Anwendbarkeit evolutionärer Algorithmen

für Funktionstests demonstriert. Bühler und Wegener verwendeten den evolutionären Funktionstest, um einen automatisierten Parkassistenten zu testen [18]. In einer Fallstudie wurden die Abmessungen eines Parkplatzes und die Position des parkenden Fahrzeugs relativ zu dem Parkplatz variiert bis in der Simulation eine Kollision mit der Parkraumbegrenzung eintrat. Bei einer weiteren von Bühler und Wegener durchgeführten Fallstudie mit einem Bremskraftverstärker wurden mit Hilfe evolutionärer Algorithmen die Eigenschaften des Signals optimiert, welches das Bremsmoment eines Fahrzeugs steuerte [18]. Das Signal bestand aus einer festen Anzahl von Segmenten von denen jeweils nur die Breite variiert wurde. Die Generierung kontinuierlicher Testdatensequenzen wurde bei diesen Arbeiten nur bedingt durch die Anpassung der Breite der Segmente des Bremsmomentensignals unterstützt. Der Ansatz von Baresel *et al.* und Pohlheim *et al.* ermöglicht hingegen komplexere Stimuli [6, 93]: Jede Testdatensequenz besteht aus mehreren Segmenten – Breite, Amplitude und Signaltyp der einzelnen Segmente können variiert werden. Einschränkungen aller genannter Ansätze sind, dass die Länge der generierten Signale sowie die Anzahl der Segmente nicht variabel sind.

Obwohl es in der Vergangenheit einige aussichtsreiche Forschungsansätze gegeben hat, wird der evolutionäre Funktionstest heutzutage nicht in der Industrie eingesetzt. Ein Grund hierfür ist zum einen, dass die Anwendung des evolutionären Funktionstests als zu kompliziert angesehen wird. Zum anderen realisieren bisherige Ansätze kein reaktives Testverfahren (Testdatensequenzen stehen vor dem Ausführungszeitpunkt fest) und die Generierung realistischer Testsequenzen wird oft nur unzureichend unterstützt. Auch ein geeignetes Testwerkzeug hat sich bis jetzt noch nicht etablieren können. Interessante Ansätze hierfür bietet aber das EvoTest-Framework, das im EvoTest-Projekt zusammen mit dem Autor entwickelt wurde [36, 72, 75].

In dieser Forschungsarbeit werden zwei Verfahren für den evolutionären Funktionstest von Software eingebetteter Systeme vorgestellt. Ziel der Verfahren ist es, die Mängel existierender Verfahren zu beseitigen. Das Verfahren *Open-loop evolutionärer Test* (OLET) wurde von dem Autor im Rahmen des EvoTest-Projekts und das Verfahren *Modellbasierter evolutionärer Test* (MbET) im Anschluss daran entwickelt [36]. Die beiden alternativ zu verwendenden

Verfahren eignen sich insbesondere für den Test von Software eingebetteter Systeme, bei denen Verfahren zur statischen Verifikation aufgrund der Komplexität scheitern würden.

Es wurden Mechanismen entwickelt, mit denen automatisiert kontinuierliche Testdatensequenzen generiert und Testläufe ausgewertet werden können. Bei OLET werden Testdatensequenzen mit Hilfe eines Segmentansatzes beschrieben. Signalcharakteristika, wie Breite oder Amplitude der Segmente, werden während der evolutionären Suche optimiert und spannen den Suchraum auf. Zudem wird die Möglichkeit geboten, Sequenzen variabler Länge und mit einer variablen Anzahl von Segmenten zu erzeugen. Die Bewertung von Testläufen erfolgt mit Hilfe einer Fitnessfunktion, die jedem Testlauf einen atomaren Zahlenwert (Fitnesswert) zuordnet. Mit Hilfe der Fitnesswerte können Testläufe miteinander verglichen und die Suche nach Testfällen, die eine Anforderung verletzen, gesteuert werden. MbET kombiniert die Prinzipien des evolutionären Testens und des modellbasierten Testens. Abbildung 1.1 zeigt den Ablauf der Optimierung bei MbET und veranschaulicht die *Evaluation von Individuen* und die *Testausführung*. Bei MbET handelt es sich um ein automatisiertes und reaktives Testverfahren für Software eingebetteter Systeme. Es werden Testeingaben von einem Testmodell abgeleitet und die Variabilität der Testeingaben wird über veränderliche Parameter im Testmodell erreicht, die mit Hilfe evolutionärer Algorithmen optimiert werden. Mit einem Auswerteskript werden die Grenzen des gültigen Systemverhaltens festgelegt. Analog zum evolutionären Strukturtest wird das Auswerteskript instrumentiert und dient dann der automatischen Bewertung von Testfällen und der Steuerung der Suche. Das suchbasierte Testverfahren zielt darauf

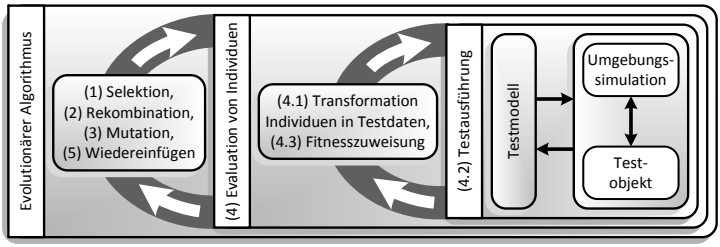


Abbildung 1.1: Modellbasiertes evolutionäres Testverfahren für den Test von Software eingebetteter Systeme.

ab, Testfälle zu finden, die im Auswerteskript festgelegte Grenzen verletzen und ein Fehlverhalten der Software aufzeigen.

Diese Arbeit leistet folgende Beiträge zum aktuellen Forschungsstand:

1. Analyse der Forschung auf dem Gebiet der Automatisierung von Black-Box-Tests für Software eingebetteter Systeme mit suchbasierten Ansätzen.
2. Präsentation des vom Autor entwickelten Verfahrens OLET für den automatisierten *open-loop*-Funktionstest mit Testdatensequenzen veränderlicher Länge und Segmentanzahl.
3. Präsentation des vom Autor entwickelten Verfahrens MbET für den automatisierten Funktionstest mit folgenden Neuerungen:
   - Entwicklung eines reaktiven suchbasierten Testverfahrens,
   - Variation von Testdatensequenzen durch Suchintervalle in Testmodellen und
   - Steuerung der Suche durch instrumentierte Testauswerteskripte.
4. Aufzeigen einer geeigneten Vorgehensweise für die Durchführung von Fallstudien mit suchbasierten Testansätzen.
5. Aufzeigen und Vergleich der Effektivität der entwickelten Verfahren anhand von Fallstudien.
6. Aufzeigen der Testplattform-übergreifenden Anwendbarkeit des evolutionären Funktionstests.

1.3 AUFBAU DER ARBEIT

Nachdem in diesem Kapitel ein Überblick über das Problemfeld und die Schwerpunkte dieser Arbeit gegeben wurde, werden in den beiden nächsten Kapiteln die wichtigsten Grundlagen für die entwickelten Verfahren vorgestellt. In Kapitel 2 wird der Test von Software eingebetteter Systeme beschrieben. Kapitel 3 stellt das

Prinzip und den aktuellen Forschungsstand des evolutionären Testens vor. Vertieft wird der evolutionäre Strukturtest (Abschnitt 3.3) und der evolutionäre Funktionstest (Abschnitt 3.4).

In Kapitel 4 wird das OLET-Verfahren beschrieben. OLET ist ein *open-loop*-Testverfahren, bei dem kontinuierliche Testdatensequenzen vor der Ausführung eines Testobjekts mit Hilfe evolutionärer Algorithmen erzeugt werden. OLET baut auf den in Abschnitt 3.4 vorgestellten Grundlagen des evolutionären Funktionstests auf.

In Kapitel 5 wird das MbET-Verfahren vorgestellt. Im Unterschied zu OLET wird hier closed-loop getestet. Durch die Verwendung von Testmodellen wird die Interaktion zwischen Testtreiber und Testobjekt ermöglicht. MbET bezieht sich auf den evolutionären Funktionstest und den evolutionären Strukturtest. Die Beschreibung von MbET beschränkt sich auf Unterschiede zu OLET.

Um OLET und MbET zu bewerten, werden in Kapitel 6 die Ergebnisse von Fallstudien mit Testobjekten aus der Serienentwicklung der Automobilindustrie vorgestellt.

In Kapitel 7 werden die Ergebnisse dieser Forschungsarbeit zusammengefasst und kritisch diskutiert. Abschließend wird in Kapitel 8 ein Ausblick auf mögliche weiterführende Arbeiten geboten.

Verweise auf Begriffe in Abbildungen, Formeln und Tabellen sowie Verweise auf Überschriften werden in dieser Arbeit *kursiv* gehalten. In Überschriften werden aus Platzgründen die Begriffe *evolutionärer Funktionstest* mit EFT, *evolutionärer Strukturtest* mit EST, *genetische Algorithmen* mit GA und *evolutionäre Algorithmen* mit EA abgekürzt.

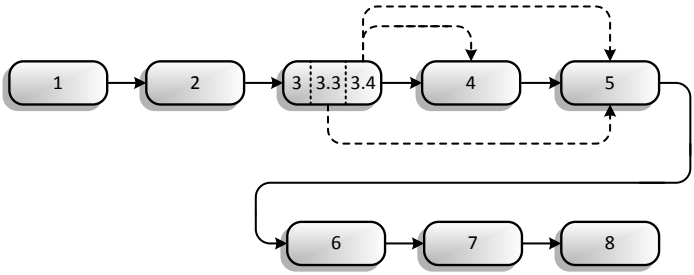


Abbildung 1.2: Zusammenhang zwischen den Kapiteln.

Teil I

GRUNDLAGEN

2 SOFTWARETEST EINGEBETTETER SYSTEME

„Wer testet, ist feige."

— Zitat eines unverbesserlichen Entwicklers

In diesem und dem nachfolgenden Kapitel werden die für den Kontext dieser Forschungsarbeit wichtigsten Grundlagen und Begrifflichkeiten erläutert sowie verwandte Arbeiten vorgestellt. Ausgehend von den Grundlagen des Softwaretestens (Abschnitt 2.1) und der Einbettung von Softwaretests in den Entwicklungsprozess (Abschnitt 2.2), wird untersucht, welche Besonderheiten der Test von Software eingebetteter Systeme aufweist (Abschnitt 2.3). Darüber hinaus werden Testansätze für eingebettete Systeme (Abschnitt 2.4) und in diesem Zusammenhang auch der modellbasierte Test als eine vielversprechende Herangehensweise an die Qualitätssicherung vorgestellt (Abschnitt 2.5).

2.1 GRUNDLAGEN DES SOFTWARETESTENS

Tests werden durchgeführt, um Fehler aufzudecken und um das Vertrauen in die korrekte Funktionalität eines Systems zu erhöhen. Der Testprozess lässt sich grob in folgende Aktivitäten einteilen: Testmanagement, Spezifikation, Implementierung, Ausführung, Auswertung und Dokumentation. Die Schritte Testmanagement, also die Planung und Koordination des Testprozesses und Dokumentation von Tests werden in dieser Arbeit nicht weiter betrachtet.

Grundlegende Testarten sind Strukturtests, nicht-funktionale Tests und Funktionstests [8, 103]. Beim Strukturtest werden Testfälle von der internen Struktur der Testobjekte abgeleitet, um einen

Programmcode nach ausgewählten Kriterien zu überdecken. *Überdecken* meint hier am Beispiel der Anweisungsüberdeckung, entsprechende Anweisungen zur Ausführung zu bringen. Durch eine vollständige Anweisungsüberdeckung wird sichergestellt, dass jede Anweisung mindestens einmal ausgeführt wird, ohne ein Fehlverhalten zu provozieren. Strukturtests werden hauptsächlich auf der Modulebene eingesetzt, weil aktuelle Testwerkzeuge nicht für komplexe Softwaresysteme anwendbar sind und die Lokalisierung von Fehlern auf der Modulebene einfacher ist. Funktionale und nicht-funktionale Tests werden dagegen auch auf der Komponenten- und Systemebene eingesetzt. Mit nichtfunktionalen Tests wird die Zuverlässigkeit, Benutzbarkeit und Effizienz eines Systems bewertet. Beispielsweise müssen bei Echtzeitsystemen die Einhaltung zeitlicher Ober– und Unterschranken sichergestellt und Speicherüberläufe ausgeschlossen werden. Strukturtests und nicht-funktionale Tests genügen nicht als alleinige Qualitätssicherungsmaßnahme, weil fehlende Funktionalität nicht erkannt wird. Funktionales Testen zielt auf die Validierung einer funktionalen Spezifikation ab. Für die Durchführung von Funktionstests werden Testfälle von der funktionalen Spezifikation abgeleitet. Als Ergebnis einer Testreihe kann eine Beurteilung über den Reifegrad eines Systems getroffen werden.

Bei komplexen Softwaresystemen, wie zum Beispiel einem Abstandsregeltempomaten, ist es unmöglich, die Abwesenheit von Fehlern zu beweisen. Es werden deshalb in der Regel so lange Testfälle manuell erstellt und ausgeführt, bis alle Bedingungen für den Abschluss des Testprozesses erfüllt sind. Für Funktionstests in der Industrie gilt allgemein die Erstellung eines Testfalls für jede Anforderung als ausreichend, beim Strukturtest genügt das Erreichen eines geforderten Überdeckungsgrads.

Testverfahren, mit denen die zuvor genannten Testarten realisiert werden, unterscheiden sich dadurch, dass ein Testobjekt mit Testfällen ausgeführt wird (dynamischer Test) oder nicht (statischer Test) [38, 103].

STATISCHER TEST

Zu den statischen Testverfahren zählen informale Techniken (Code- und Designreviews) sowie formale Techniken (statische Analysen und Model-Checking). Code- und Designreviews

werden manuell durch intensive Betrachtung durchgeführt – idealerweise von Personen, die den Programmcode nicht selbst erstellt haben. Formale Techniken können hingegen mit Werkzeugen automatisiert durchgeführt werden. Zu den bekanntesten Verfahren für die statische Analyse zählen Compiler (zum Beispiel für die Ermittlung von Syntaxverletzungen oder von nicht initialisierten Variablen), Datenflussanalysen (zum Aufdecken von Datenflussanomalien) und Kontrollflussanalysen (zum Identifizieren von Endlosschleifen und totem Code) [103]. Zudem kann durch statische Analyseverfahren die Einhaltung von Konventionen und Programmierstandards überprüft werden (zum Beispiel durch den Vergleich ermittelter Codemetriken mit zulässigen Grenzen). Model-Checking erlaubt die Verifikation eines Systems gegenüber einer formalen Spezifikation. Es versagt allerdings bei komplexen Softwaresystemen aufgrund der kombinatorischen Explosion der Zustände.

DYNAMISCHER TEST

Bei dynamischen Testverfahren wird das Testobjekt mit Testfällen zur Ausführung gebracht. Zu den Verfahren für den dynamischen Funktionstest zählen Äquivalenzklassentests (Einteilung der möglichen Eingangsdaten in Äquivalenzklassen, Testausführung jeweils mit einem konkreten Wert), Grenzwertanalysen (Testen an den Grenzen der Äquivalenzklassen) und zustandsbezogene Tests (Ableitung der Testfälle von einer Spezifikation, die auf einem Zustandsautomaten basiert). Handelt es sich um einen Strukturtest, wird beispielsweise überprüft, ob sich ein Programm korrekt verhält, wenn beispielsweise jede Anweisung (Anwendungsüberdeckung) oder jeder Pfad (Pfadüberdeckung) zur Ausführung gebracht wird [103].

Dynamische Testverfahren werden in Black-Box- und White-Box-Verfahren unterteilt [98]. Bei den White-Box-Verfahren werden Eigenschaften des Programmcodes bei der Ermittlung der Testfälle berücksichtigt. Das bekannteste White-Box-Testverfahren ist der Strukturtest. Beim Black-Box-Test erfolgt die Bewertung von Testfällen durch die Überwachung der Ausgänge des Testobjekts zum Zeitpunkt der Testausführung [9]. Funktionstests werden in der Regel mit einem Black-Box-Testverfahren realisiert (abgesehen von Verfahren zur formalen Verifikation wie beispielsweise

das Model-Checking). Die Abwesenheit von Fehlern und unerwünschtem Verhalten in einem System kann beim Black-Box-Test allerdings nicht bewiesen werden.

Die Motivation für die Entwicklung neuer Testverfahren sind die stetig steigenden Kosten der Qualitätssicherung im Verhältnis zu den Entwicklungskosten. Vor allem durch die (Teil-)Automatisierung des Testprozesses bieten sich enorme Einsparpotentiale sowohl bei den Kosten als auch beim Zeitaufwand. Der Fokus dieser Arbeit liegt auf dem Gebiet des Funktionstests. Es wurden zwei dynamische Black-Box-Testverfahren entwickelt, wobei Testfälle von einer funktionalen Systemspezifikation abgeleitet werden. Mit einem suchbasierten Ansatz wird automatisch nach Testfällen gesucht (siehe Kapitel 3), die ein Fehlverhalten eines Testobjekts gegenüber einer Spezifikation aufzeigen.

2.2 EINBETTUNG VON SOFTWARETESTS IN DEN ENTWICKLUNGSPROZESS

Für die systematische Durchführung umfangreicher Softwareentwicklungsprojekte hat sich seit Ende der 1990er Jahre in Deutschland das V-Modell etabliert. Auch in der Automobilindustrie hat das V-Modell weitgehend herkömmliche Entwicklungsmodelle, wie zum Beispiel das Spiralmodell oder das Wasserfallmodell ersetzt. Das ursprüngliche im Jahr 1992 veröffentlichte V-Modell wurde später von dem V-Modell 97 und dann im Jahr 2005 von dem V-Modell XT mit einigen Erweiterungen abgelöst [19, 20].

Abbildung 2.1 zeigt eine für den Automobilbereich typische Herangehensweise an die Entwicklung von Software für Steuergeräte gemäß dem V-Modell XT. Zuerst werden die Erwartungen an die Software mit Hilfe von *Anforderungen* festgehalten. Ausgehend von den *Anforderungen* werden detaillierte *Spezifikationen* und *Designkonzepte* für das zu realisierende System erstellt. Im Falle der modellbasierten Entwicklung kann ausführbarer Programmcode direkt aus den Spezifikationen erzeugt werden. Allerdings wird auch heutzutage noch ein Teil, insbesondere der Programmcode für die direkte Hardwareansteuerung, manuell implementiert. Sobald die ersten Module fertiggestellt sind, beginnt auf dem rechten

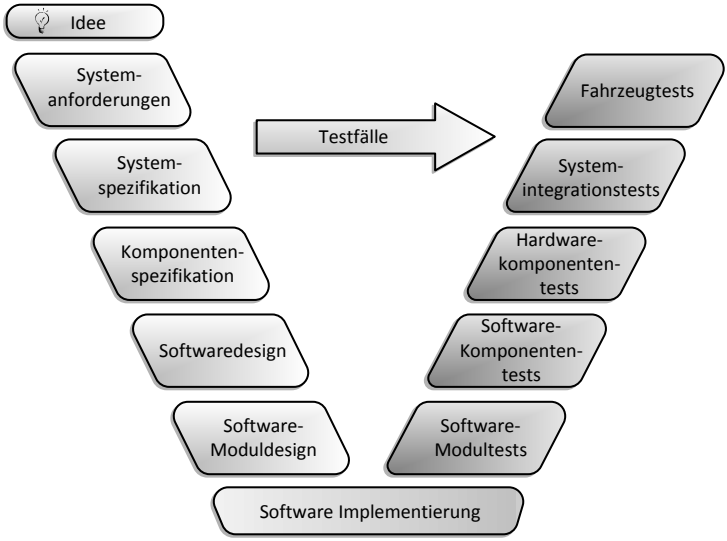


Abbildung 2.1: Software Entwicklungsprozess für Steuergeräte nach dem V-Modell XT.

Ast im V-Modell die Testphase. Dabei werden zuerst *Modultests* durchgeführt und die Module basierend auf den Ergebnissen der Testläufe überarbeitet, bis sich der gewünschte Qualitätsgrad eingestellt hat. Die Module werden dann zu Komponenten zusammengeschlossen. Die Softwarekomponenten werden sowohl auf einem PC (*Software-Komponententests*) als auch auf der im Fahrzeug zum Einsatz kommenden Hardware getestet (*Hardware-Komponententests*). Die einzelnen Komponenten werden dann zu größeren Einheiten zusammengeschlossen, um die Kommunikation und Interaktion zwischen den Komponenten zu überprüfen (*Integrationstests*). Schlussendlich wird das Verhalten der Software im Fahrzeug auf der Teststrecke und auf der Straße untersucht (*Fahrzeugtests*).

Die in dieser Arbeit vorgestellten Verfahren sind für die Validierung von funktionalen Anforderungen auf allen Teststufen einsetzbar. Der Begriff *Teststufe* bezieht sich hier auf eine Phase der Qualitätssicherung nach dem V-Modell (siehe rechter Ast in Abbildung 2.1). Sowohl das korrekte Verhalten einzelner Module als auch das Verhalten komplexer Systeme, zum Beispiel ein autonom fahrendes Fahrzeug, kann mit den Verfahren überprüft

werden. Da es sich um dynamische Testverfahren handelt, die eine hohe Anzahl von Testausführungen erfordern, sind die Verfahren vor allem für Software- und Modelltests geeignet. Bei Software- und Modelltests können Tests oft schneller als in Echtzeit ausgeführt werden und die Testausführung parallelisiert werden. Tests mit realen Steuergeräten erfordern dagegen eine Ausführung in Echtzeit und belegen somit, abhängig von der Länge der Testsequenzen und der Anzahl der Tests, die zumeist sehr kostenintensiven Prüfstände über einen langen Zeitraum.

Die Anwendbarkeit der vorgestellten Verfahren für den Test realer Steuergeräte wurde dennoch vom Autor untersucht (siehe Abschnitte 4.4.3 und 6.2.5). Bei der Verwendung von Steuergeräten für die Testausführungen werden hardwarespezifische Besonderheiten – zum Beispiel die Ausführungszeit und die Speicherauslastung – mit einbezogen. Zudem werden nicht immer alle Phasen des V-Modells von Anbietern des zu verkaufenden Endprodukts durchgeführt. Insbesondere im Automobilbereich ist es üblich, dass einzelne Fahrzeugkomponenten von Zulieferern hergestellt werden. Der Fahrzeughersteller kann in diesem Fall oft nur das fertige Steuergerät testen, weil einzig der Zulieferer Zugriff auf Modelle und Programmcode hat. Die Hauptverantwortung gegenüber den Kunden trägt allerdings allein der Fahrzeughersteller. Der Zulieferer hat darüber hinaus nicht die Möglichkeit, die Komponente im Verbund mit den anderen Funktionen des Fahrzeugs zu testen. Es ist deshalb üblich, dass der Fahrzeughersteller die korrekte Funktionsweise durch Tests mit dem realen Steuergerät überprüft. Durch die hohe Anzahl zielgerichteter Testläufe eignen sich die hierin vorgestellten Verfahren insbesondere zur Absicherung sicherheitsrelevanter Anforderungen und können das Vertrauen in die Systeme steigern.

2.3 BESONDERHEITEN VON SOFTWARETESTS EINGEBETTETER SYSTEME

Ein eingebettetes System ist ein Verbund aus Hardware und Software, der verwendet wird, um bestimmte Funktionen auszuführen. Im Unterschied zu einem PC, verfügt ein eingebettetes System

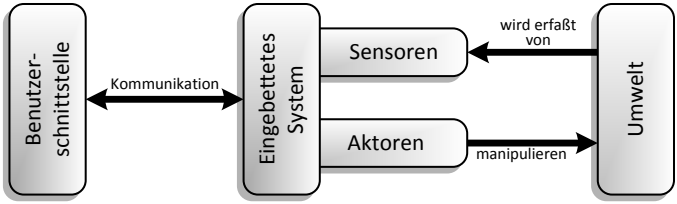


Abbildung 2.2: Interaktion von eingebetteten Systemen mit ihrer Umgebung.

zumeist nur über eine einfache Benutzerschnittstelle (beispielsweise die Bedienknöpfe einer Waschmaschine). Zu den Funktionen eingebetteter Systeme im Automobilbereich (Steuergeräte) zählen zum Beispiel Abstandsregeltempomaten oder die Ansteuerung mechanischer Komponenten, um die Bremsanforderung des Fahrers umzusetzen. Eingebettete Systeme erfassen mit *Sensoren* die *Umwelt* und stellen basierend auf den gewonnenen Erkenntnissen Informationen für den Nutzer bereit (siehe Abbildung 2.2). Durch die Ansteuerung von *Aktoren* können zudem Eingriffe in die *Umwelt* ausgelöst werden. Für die Verarbeitung der mit den *Sensoren* erfassten Ereignisse werden interne Zustandsautomaten verwendet. Die Software wird zyklisch aufgerufen und veranlasst abhängig von der erfassten Situation und dem Systemzustand eine entsprechende Reaktion. In diesem Zusammenhang spricht man von zyklischen Softwarekomponenten (vergleiche Grieskamp *et al.* [41]).

Testverfahren für eingebettete Systeme unterscheiden sich dadurch, ob sie ein *open-loop*- oder ein *closed-loop*-Testverfahren realisieren. Bei einem *closed-loop*-Testverfahren (reaktiv) besteht während der Testausführung die Möglichkeit der Interaktion zwischen der Testumgebung und dem Testobjekt oder der Umgebungssimulation. Ein Testfall kann auf das Verhalten des Testobjekts reagieren und die Eingangsdaten dem Verhalten entsprechend angepasst werden. Bei einem *open-loop*-Testverfahren stehen die Eingangsdaten immer schon vor der Ausführung des Testobjekts fest. In Kapitel 4 wird ein *open-loop*-Testverfahren vorgestellt (OLET) und in Kapitel 5 ein *closed-loop*-Testverfahren (MbET).

Unabhängig davon, ob ein *open-loop*- oder ein *closed-loop*-Testverfahren verwendet wird, reicht es für die Durchführung von Black-

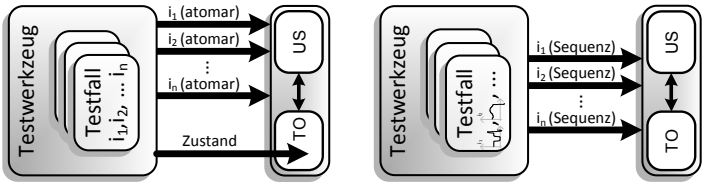


Abbildung 2.3: Test von Software eingebetteter Systeme durch direkte Zustandswahl (links) und Testdatensequenzen (rechts) mit TO: Testobjekt und US: Umgebungssimulation.

Box-Tests nicht aus, jeden Eingang des Testobjekts mit genau einem Wert zu belegen [80]. Beim Black-Box-Test von Software eingebetteter Systeme wird das Testobjekt in der Regel für jeden Testfall automatisch zurück in den initialen Zustand versetzt. Die hohe Anzahl interner Systemzustände und die komplexen Interaktionsmöglichkeiten mit der Umwelt erfordern eine angepasste Herangehensweise an die Erzeugung von Teststimuli. Als Alternativen für ein effektives Testverfahren bietet sich an, Systemzustände von außen einstellbar zu machen oder Testdatensequenzen zu erzeugen [93].

EINFACHE EINGANGSDATEN MIT DIREKTER ZUSTANDSWAHL

Bei diesem Ansatz müssen Umgebungssimulation, Testobjekt und Testtreiber erweitert werden, so dass interne Systemzustände von außen eingestellt werden können (linke Seite in Abbildung 2.3). Dadurch wird ermöglicht, dass mit der Belegung jedes Eingangsdatenkanals mit einem Wert das Verhalten eines Systems in verschiedenen Zuständen getestet werden kann. Hierbei werden allerdings keine Einschwing- und Reaktionszeiten der Systeme berücksichtigt. Der Ansatz kann auch nur dann verwendet werden, wenn Programmcode oder Modell zur Verfügung stehen. Zudem handelt es sich dann nicht mehr um einen klassischen Black-Box-Test, weil in die interne Struktur des Testobjekts eingegriffen wird.

TESTDATENSEQUENZEN

Als Alternative können Tests mit Testdatensequenzen durchgeführt werden (rechte Seite in Abbildung 2.3). Mit diesem Verfahren

können, geeignete Sequenzen vorausgesetzt, interne Systemzustände erreicht werden, ohne in die interne Struktur des Testobjekts einzugreifen. Die Erstellung effektiver Testdatensequenzen für den Test von Software eingebetteter Systeme ist sehr aufwendig. Im nächsten Abschnitt werden Verfahren für die Generierung von Testdatensequenzen vorgestellt.

Abschließend wird auf eine weitere Besonderheit der Qualitätssicherung von Software eingebetteter Systeme hingewiesen: In der Praxis müssen häufig gesetzliche Vorgaben bezüglich des Umfangs einer Testsuite eingehalten werden, anderenfalls erhält das System keine Zulassung (zum Beispiel für den Straßenverkehr). Der Begriff *Testsuite* wird in dieser Arbeit als eine Menge von Testfällen aufgefasst, die für den Test einer Komponente oder eines Systems verwendet werden. Die gesetzlichen Vorgaben sind in Standards festgehalten – beispielsweise schreibt die Norm ISO26262 bei der Entwicklung von sicherheitsrelevanter Software eingebetteter Systeme vor, dass für jede funktionale Anforderung mindestens ein Test spezifiziert und erfolgreich durchgeführt werden muss [58]. Für Strukturtests wird eine bestimmte Überdeckungsart gefordert (zum Beispiel Anweisungs- oder Zweigüberdeckung). Durch derartige Vorgaben nimmt die Anzahl erforderlicher Tests und somit auch der Testaufwand zu. Testautomatisierungsansätze können dazu beitragen, Kosten und Aufwand dennoch im Rahmen zu halten.

2.4 Black-Box-Testverfahren für eingebettete Systeme

Nachfolgend werden Ansätze vorgestellt, die alle über die Möglichkeit verfügen, Testdatensequenzen für den Test von Software eingebetteter Systemen zu generieren. Sie unterscheiden sich allerdings durch den Spezifikationsansatz und die Komplexität der generierten Testdatensequenzen sowie durch die Auswertemechanismen.

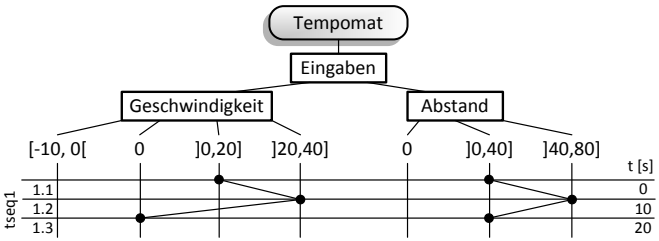


Abbildung 2.4: Klassifikationsbaum-Methode für eingebettete Systeme an einem Beispiel zur Ableitung von Testdatensequenzen für den Test eines Tempomaten.

DIREKTE SPEZIFIKATION VON TESTDATENSEQUENZEN

Bei der direkten Spezifikation werden Testdatensequenzen manuell erzeugt. Im einfachsten Fall wird eine Menge von Zeit-Wertepaaren bestimmt, die dann durch lineare Übergänge verbunden werden. Alternativ kann Software eingebetteter Systeme auch mit Daten getestet werden, die unter realen Bedingungen aufgezeichnet wurden (zum Beispiel bei Testfahrten).

SIGNALSPEZIFIKATION MIT KLASSIFIKATIONSBÄUMEN

Die Klassifikationsbaummethode (englisch Classification Tree Method, CTM) wurde Anfang der 1990er Jahre von Grochtmann und Grimm entwickelt, um die systematische Auswahl von Testeingaben zu ermöglichen [42]. Teststimuli werden durch eine Baumstruktur beschrieben. Ausgehend von der Wurzel des Baums, die das Testobjekt repräsentiert, werden die Teststimuli immer weiter verfeinert. Unter Anwendung der Äquivalenzklassenmethode werden die Blätter des Baums, die Klassen von Testeingaben repräsentieren, für die Ausführung des Testobjekts bestimmt. Schließlich werden mit Hilfe einer Kombinationstabelle Blätter für die konkreten Testfälle ausgewählt. Jede Zeile in der Tabelle entspricht einem Testfall. Erst durch spätere Erweiterungen der CTM wurde die Spezifikation und Generierung von Testdatensequenzen ermöglicht. Beispielsweise werden Testdatensequenzen in der Dissertation von Conrad durch eine Abfolge von Signalübergängen zwischen einzelnen Blättern eines Baumes beschrieben (vergleiche Abbildung 2.4) [24, 26].

SIGNALSPEZIFIKATION MIT MATHEMATISCHEN FUNKTIONEN UND PROZEDURALEN SKRIPTSPRACHEN

Einige Testwerkzeuge unterstützen die Möglichkeit, eine beliebige Formel zu spezifizieren (zum Beispiel einen Sinus), um damit Testdatensequenzen für die Ausführung eines Testobjekts zu erzeugen. Konkret heißt das, dass ein Signalverlauf – dazu zählt auch die reellwertige Zeit – durch eine Funktion in einen anderen Signalverlauf transformiert wird. Ein Beispiel für einen derartigen Ansatz bietet Broy mit der Beschreibung von stromverabeitenden Funktionen [14]. Ein weiteres Beispiel für ein Verfahren, bei dem Signale mit einer prozeduralen Skriptsprache beschrieben werden, ist TTCN-3 Embedded. Mit diesem Verfahren können kontinuierliche Testdatensequenzen generiert und der Signalverlauf reaktiv durch das Verhalten der Testobjekte angepasst werden [44].

SIGNALSPEZIFIKATION MIT TEMPORALER LOGIK

Temporale Logiken werden hauptsächlich zur formalen Beschreibung von Eigenschaften dynamischer Systeme eingesetzt. Ursprünglich wurde die temporale Logik einzig für die Modellierung von Abläufen verwendet. Ende der 1980er Jahre wurde das Konzept um die Behandlung von reellwertiger Zeit erweitert (vergleiche Ostroff [85]). Durch diese Erweiterung kann die temporale Logik dazu verwendet werden, um Signaleigenschaften und Abhängigkeiten zwischen Signalen zu spezifizieren. Zudem wird die Beschreibung deterministischer kontinuierlicher Signale ermöglicht.

SIGNALSPEZIFIKATION MIT MODELLEN

Durch die Verwendung von Modellen für die Signalspezifikation können komplexe Signale oft mit geringerem Aufwand im Vergleich zur textbasierten Spezifikation beschrieben werden. Als Modellierungsansätze werden hauptsächlich Diagramme aus der Unified Modeling Language (UML) oder Erweiterungen davon verwendet. Für die Qualitätssicherung von Software eingebetteter Systeme finden überwiegend Zustandsautomaten (zum Beispiel endliche Automaten, Timed Automata oder hybride Automaten) und Sequenzdiagramme beziehungsweise Message Sequence

Charts Anwendung (vergleiche Burton [21]). Auch für das zuvor erwähnte Verfahren TTCN-3 Embedded wird aktuell ein auf hybriden Automaten aufbauender grafischer Modellierungsansatz erforscht. In Abschnitt 2.5 wird das modellbasierte Testen weiter vertieft (siehe auch Abbildung 2.5).

Nach diesem Überblick über Ansätze für die Generierung von Testdatensequenzen werden im Folgenden Ansätze für die Testauswertung vorgestellt.

MANUELLE TESTAUSWERTUNG

Auch heutzutage werden Tests bei der Entwicklung von Steuergerätesoftware hauptsächlich durch die manuelle Betrachtung von Signalverläufen in einem Editor analysiert. Basierend auf dem Expertenwissen des Testers wird dann entschieden, ob die Signalverläufe dem in den Anforderungen spezifizierten Verhalten entsprechen.

TESTAUSWERTUNG MIT REFERENZSIGNALEN

Bei diesem Ansatz werden beim Testlauf aufgezeichnete Signale mit Referenzsignalen verglichen. Beispielsweise wird überprüft, ob eine zulässige maximale Abweichung zu einem Messsignal eines vergleichbaren Systems überschritten wurde. Referenzsignale können auch manuell basierend auf Erwartungen festgelegt werden.

TESTAUSWERTUNG MIT EINFACHEN ZUSICHERUNGEN

Mit einfachen Zusicherungen werden Ereignisse beschrieben, die nicht eintreten dürfen (zum Beispiel sollte die Drehzahl eines Motors nicht negativ sein). Ein Testlauf ist also nur dann erfolgreich durchgelaufen, wenn keine der Zusicherungen verletzt wurde.

TESTAUSWERTUNG MIT HÖHEREN PROGRAMMIERSPRACHEN

Alternativ kann das gültige Systemverhalten auch durch höhere Programmiersprachen beschrieben werden. Höhere Programmiersprachen ermöglichen die Anwendung komplexer Auswertefunk-

tionen auf die Testausgaben. Für das später vorgestellte Verfahren MbET wird eine prozedurale Skriptsprache verwendet (basierend auf dem Python-Sprachsatz [97]), um die Automatisierung der Testauswertung zu ermöglichen.

Testwerkzeuge werden in Anlehnung an CASE Werkzeuge (Computer Aided Software Engineering) als CAST Werkzeuge (Computer Aided Software Testing) bezeichnet. Anhang C zeigt eine Übersicht über die konkreten Realisierungen und CAST-Werkzeuge der zuvor vorgestellten Ansätze.

Im Folgenden wird der Begriff *Black-Box-Testautomatisierung von Software eingebetteter Systeme* näher erläutert.

Black-Box-Testautomatisierung von Software eingebetteter Systeme

Von den sechs Testaktivitäten (vergleiche Abschnitt 2.1) lassen sich am einfachsten die Dokumentation und Ausführung von Tests automatisieren. Die Automatisierung der Testspezifikation- und implementierung ist beim Black-Box-Test nur bedingt möglich. Aufgrund der Tatsache, dass Tests nicht von der Struktur der Testobjekte abgeleitet werden können, muss eine Testspezifikation vorgegeben werden oder zumindest die Möglichkeit bestehen, Tests von der funktionalen Systemspezifikation abzuleiten. Für die Testauswertung gibt es einige Verfahren, die eine automatisierte Auswertung mit einer der zuvor beschriebenen Herangehensweisen ermöglichen. Von einer Automatisierung kann allerdings nur insofern gesprochen werden, als dass beispielsweise mit Hilfe eines manuell implementierten Skripts ein Urteil im Anschluss an die Testausführung gefällt werden kann. Es muss in jedem Fall manuell ein Testorakel (vergleicht Ist- und Sollverhalten eines Testfalls) implementiert werden oder manuell eine Auswertung durch den Tester erfolgen. Die in Anhang C vorgestellten CAST-Werkzeuge wurden im Hinblick auf den Testautomatisierungsgrad eingeordnet. Auch die in dieser Arbeit vorgestellten Verfahren erfordern die manuelle Implementierung eines Testorakels. Es kann dennoch von einem hohen Automatisierungsgrad gesprochen werden, da die Aufgabe des Orakels mehr als die Bewertung einzelner Testläufe umfasst. Durch die Verwendung suchbasierter Algorithmen dient das Orakel der Steuerung der

Suche nach Testfällen, die eine Anforderungsverletzung aufzeigen. Im Verlauf einer Suche wird so eine Vielzahl von Testfällen automatisiert ausgeführt und bewertet (siehe Abschnitt 3.4).

2.5 MODELLBASIERTES TESTEN

In diesem Abschnitt wird die Methode des modellbasierten Testens (Abschnitt 2.5.1) sowie einige Modellierungsansätze dafür beschrieben (Abschnitt 2.5.2). Weiterhin wird aufgrund von Parallelen bei der Testmodellierung zu MbET das Time Partition Testing (TPT) als eine konkrete Anwendung des modellbasierten Testens vertieft (Abschnitt 2.5.3).

2.5.1 MODELLBASIERUNG UND MODELLBASIERTES TESTEN

Ziel des modellbasierten Testens ist es, die Effizienz und Effektivität des Testprozesses zu erhöhen. Mit Hilfe von Modellen können Testfälle systematisch und automatisch erstellt und zur Bewertung von Testläufen verwendet werden. Die Hauptmerkmale eines Modells sind nach Stachowiak [104]:

- *Abbildungsmerkmal* – Modelle sind Abbildungen einer konkreten in eine abstraktere Welt,
- *Verkürzungsmerkmal* – jedes Modell abstrahiert und
- *Pragmatisches Merkmal* – jedes Modell wird für einen konkreten Verwendungszweck geschaffen.

Das Paradigma der modellbasierten Entwicklung hat sich mittlerweile in den verschiedensten Forschungs- und Industriezweigen verbreitet, insbesondere auch bei der Entwicklung von Steuerungssoftware eingebetteter Systeme [94]. Als Modellierungssprachen findet bei der Softwareentwicklung für eingebettete Systeme zumeist die Matlab-Produktfamilie oder die Modellierungssprache Modelica Anwendung [83, 110]. Werden die bei der Entwicklung entstehenden, ausführbaren Modelle für den Test verwendet, spricht man vom Modelltest [24]. Wird für die Testfallerzeugung

ein eigens hierfür erstelltes Testmodell verwendet, kommt der Begriff des modellbasierten Testens zum Tragen.

Bei der Modellierung von Testmodellen kann eine Vielzahl von Beschreibungssprachen verwendet werden. Für die Qualitätssicherung von Software eingebetteter Systeme haben sich Ansätze etabliert, die auf Zustandsautomaten und Sequenzdiagrammen aufbauen. In den letzten Jahren hat es umfassende Diskussionen darüber gegeben, ob der Aufwand für die Erstellung von Testmodellen gerechtfertigt ist und zu einem effizienteren Testprozess beiträgt. Eine abschließende, verallgemeinernde Aussage wurde noch nicht getroffen (vergleiche die Arbeiten von Pretschner *et al.* [95, 96]).

2.5.2 Modellierung von Testmodellen mit Automaten

Die Beschreibung kontinuierlicher Testdatensequenzen mittels Automaten hat in den vergangenen Jahren zunehmend an Bedeutung für die Entwicklung von Steuergeräte-Software gewonnen. Wie eingangs erwähnt, steht MbET in Zusammenhang mit der Methodologie des Time Partition Testings [71]. Für die Generierung der Teststimuli werden bei TPT hybride Automaten in Kombination mit stromverarbeitetenden Funktionen verwendet (nach Broy [14]). Nachfolgend werden zuerst klassische endliche Zustandsautomaten eingeführt und dann in Bezug zu hybriden Automaten gesetzt.

Endliche Zustandsautomaten

Ein endlicher Zustandsautomat besteht aus einer endlichen Menge von Zuständen und einer Menge von Übergängen zwischen den Zuständen (Transitionen). Den Transitionen können Bedingungen und Aktionen zugeordnet werden. Sobald die Bedingungen einer Transition erfüllt sind, werden die entsprechenden Aktionen ausgeführt und ein Zustandswechsel eingeleitet. Weiterhin muss ein initialer Zustand festgelegt sein. Dargestellt werden können endliche Zustandsautomaten sowohl grafisch als Zustandsdiagramme sowie tabellarisch als Zustandsmatrix. Erweiterte endliche Zustandsautomaten bieten zusätzlich die Möglichkeit, während der Ausführung Operationsdaten abzufragen und zu manipulieren.

Die Erweiterung von endlichen Zustandsautomaten zu sogenannten Statecharts bietet darüber hinaus die Möglichkeit, Zustände hierarchisch aufzubauen und parallel zu zerlegen (vergleiche Harel *et al.* [46]).

Beim modellbasierten Testen mit Automaten wird wie folgt vorgegangen: Als erster Schritt wird, ausgehend von den Anforderungen an das zu testende System, ein Testmodell erzeugt. Danach werden nach bestimmten Kriterien Pfade durch das Testmodell bestimmt (siehe Tabelle 3.2). Die Pfade, ausgehend von dem initialen Zustand zu einem Endzustand, entsprechen den Testfällen. Das Testobjekt ist hierbei als Black-Box zu betrachten, Test Stimuli werden von dem Testmodell abgeleitet und das Verhalten des Testobjekts mit dem erwarteten Verhalten (spezifiziert im Testmodell) verglichen. Für den modellbasierten Test von Software eingebetteter Systeme ist die Erzeugung von Testmodellen mittels klassischer endlicher Zustandsautomaten unzureichend. Derartige Automaten bieten nicht die Möglichkeit, kontinuierliche Testdatensequenzen zu beschreiben. Abhilfe schaffen hybride Automaten.

HYBRIDE AUTOMATEN

Hybride Automaten – ein formales Modell für hybride Systeme – erweitern endliche Automaten um kontinuierliches Verhalten innerhalb der Zustände (vergleiche Alur *et al.* [3]). Das Besondere bei hybriden Automaten ist, dass sowohl diskretes als auch kontinuierliches Verhalten modelliert werden kann. Bei einem hybriden Automaten repräsentieren Kanten diskrete Zustandsübergänge und Knoten kontinuierliche Aktivitäten. Der große Vorteil der Testmodellierung mit hybriden Automaten ist die Beschreibung kontinuierlicher Signalverläufe. Testdatensequenzen können basierend auf dem Ausgangsverhalten des Testobjekts und der Zeit berechnet werden, die bereits bei einem Testlauf verstrichen ist. Beim modellbasierten Testen, insbesondere bei der formalen Verifikation, werden für die Modellierung mit hybriden Automaten vor allem Timed Automata verwendet (zum Beispiel Henzinger [53]). Timed Automata sind ein Spezialfall der hybriden Automaten, wobei alle Uhren mit derselben Geschwindigkeit laufen ($\dot{t} = 1$).

Das Modellierungskonzept von MbET berücksichtigt auch Konzepte für die Parallelisierung und hierarchische Komposition von

Zuständen (vergleiche Definition für Statecharts [46]). Kontinuierliches Verhalten innerhalb der Zustände wird durch mathematische Funktionen beschrieben [14]. Während der Testausführung werden Testdatensequenzen in Abhängigkeit der aktiven Zustände durch mathematische Funktionen berechnet (vergleiche Abbildung 2.5).

Die verwendeten Automaten lassen sich durch ein Tupel $H = \{S, EK, AK, R\}$ klassifizieren, wobei gilt: S ist ein Statechart, EK sind in das Testmodell eingehende Kanäle von Daten, die von den stromverarbeitenden Funktionen transformiert werden können, und AK sind ausgehende Kanäle für die aus der Verarbeitung resultierenden Testdatensequenzen. R ist eine Komponente, die jedem Zustand in den Automaten Regeln (eine Menge von Gleichungen) für die Transformation der Daten aus den Eingangskanälen EK in Daten für die Ausgangskanäle AK zuordnet. Im nachfolgenden Abschnitt wird ein Beispiel für ein MBT-Verfahren mit hybriden Automaten vorgestellt.

2.5.3 TIME PARTITION TESTING

Das modellbasierte Testverfahren Time Partition Testing wurde im Rahmen der Promotion von Lehmann entwickelt [71]. Der Name der Methode wurde durch die Tatsache inspiriert, dass jeder Zustand eines Testmodells einer zeitlich abgegrenzten Phase (Time Partition) der generierten Testdatensequenz entspricht. Nachfolgend werden die Besonderheiten von TPT bei der Testmodellierung, der Testausführung und der Testauswertung vorgestellt.

TESTMODELLIERUNG

Testfälle werden bei TPT mit einer grafischen und formalen Automatennotation modelliert, die auf hybriden Automaten basiert. Testfälle sind abstrakt auf der obersten Hierarchieebene der Automaten, konkrete Testeingaben werden auf der jeweils untersten Ebene der Zustände spezifiziert. Zustände und Transitionen werden entlang der Hierarchie der Testmodelle so lange verfeinert, bis eine ausführbare Semantik erreicht ist.

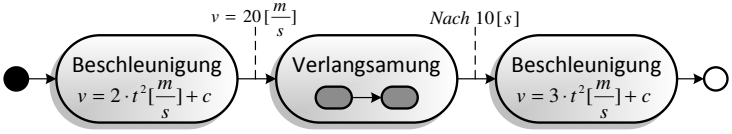


Abbildung 2.5: Testmodell für die Modellierung eines Geschwindigkeitsverlaufs.

Abbildung 2.5 zeigt ein Beispiel eines TPT-Automaten. Dabei wird mit drei Zuständen eine Testdatensequenz für den Verlauf der Geschwindigkeit, zum Beispiel eines Fahrzeugs, modelliert. Mit Hilfe mathematischer Funktionen, die den einzelnen Zuständen zugeordnet sind, werden die Teststimuli berechnet. Ein Zustandswechsel erfolgt, sobald die an den Übergängen spezifizierten Bedingungen eingetroffen sind. Der Verlauf des Signals im zweiten Zustand wird durch Unterzustände modelliert (hierarchische Komposition). Ein Testlauf terminiert, wenn ein Automat auf der obersten Hierarchieebene einen Endzustand erreicht.

INTERAKTIVITÄT

Der modellbasierte Testansatz erlaubt eine Interaktion zwischen Testtreiber und Testobjekt während der Testdurchführung (*closed-loop*-Test). Transitionsübergänge im Testmodell können an das Verhalten des Testobjekts gekoppelt und Teststimuli unter Miteinbeziehung der Ausgaben des Testobjekts generiert werden. Ein Abstandsregeltempomat kann beispielsweise eingeschaltet werden, sobald ein Fahrzeug eine bestimmte Geschwindigkeit durch die simulierte Betätigung des Gaspedals erreicht hat.

TESTAUSFÜHRUNG

Bei TPT werden Tests vollständig automatisiert ausgeführt. Um Testfälle auf unterschiedlichen Testplattformen durchzuführen, sieht das Konzept von TPT vor, dass keine oder nur wenige Anpassungen an den Testmodellen erforderlich sind. Stattdessen wird durch sogenannte *Plattform Adapter* die Kommunikation zwischen *Testmodell* und *Testobjekt* ermöglicht (siehe Abbildung 2.6). Für die Testausführung auf verschiedenen Testplattformen (MiL, SiL oder

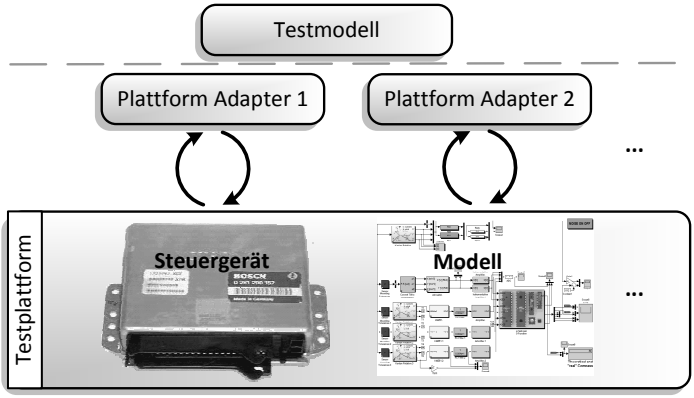


Abbildung 2.6: Testplattform-übergreifendes Testen durch Entkopplung von Testmodell und Testobjekt.

HiL) wird jeweils ein spezifischer *Plattform Adapter* benötigt – das Testmodell muss nicht zwangsläufig angepasst werden.

TESTAUSWERTUNG

Mit einer an die Besonderheiten von kontinuierlichen Signalen angepassten Skriptsprache (basierend auf Python [97]), können Testfälle während der Testausführung oder auch im Anschluss daran bewertet werden. Auswerteskripte können sowohl Gruppen von Testfällen, einzelnen Testfällen als auch einem Zustand im Testmodell zugewiesen werden. Ein Auswerteskript dient beispielsweise der Überwachung, ob ein Fahrzeug in der Simulation beim Einparken den Bordstein oder ein parkendes Fahrzeug berührt. Ergebnisse der Testausführung und Testauswertung werden in einem automatisch generierten Bericht dokumentiert.

DAS TPT-TESTWERKZEUG

Aufbauend auf der Erforschung der Methodologie des Time Partition Testings ist auch das gleichnamige Testwerkzeug TPT entstanden [13, 91]. TPT wird hauptsächlich in der Automobilindustrie, aber vereinzelt auch in anderen Bereichen für den Test von Software eingebetteter Systeme eingesetzt. Insbesondere bei der

Daimler AG hat sich das Testwerkzeug in vielen Bereichen als Standardwerkzeug etabliert.

Im Unterschied zu TPT werden Testfälle bei MbET automatisch erstellt. Bei MbET wird nach einem Testfall gesucht, der ein Fehlverhalten eines Testobjekts gegenüber einer Anforderung aufzeigt. Mit TPT lässt sich nur zeigen, dass ein System für bestimmte modellierte Eingangsdaten ein korrektes oder inkorrektes Verhalten aufweist. Jeder einzelne Testfall muss manuell mit einer konkreten Parametrierung des Testmodells implementiert werden. Dadurch kommt es zum einen häufig vor, dass sich Testfälle sehr stark ähneln. Zum Anderen können Fehler übersehen werden, falls die konkreten Testeingaben aus ungültigen Äquivalenzklassen ausgewählt wurden. OLET und MbET erheben nicht den Anspruch, ein Ersatz für existierende systematische Testansätze darzustellen, sondern eine Ergänzung. Es wird die Möglichkeit geboten, einzelne (sicherheitsrelevante) Anforderungen abzusichern.

ZUSAMMENFASSUNG

In diesem Kapitel wurden der Anwendungsbereich und das Umfeld von OLET (Kapitel 4) und MbET (Kapitel 5) beschrieben. Bei den Verfahren handelt es sich um funktionale Black-Box-Testverfahren mit dem Ziel der Testautomatisierung für Software eingebetteter Systeme. Die Verfahren sind auf mehreren Teststufen anwendbar (siehe Abschnitt 2.2) und erzeugen automatisiert kontinuierliche Testdatensequenzen (siehe Abschnitt 2.4). OLET realisiert ein *open-loop*-Testverfahren: Das Testobjekt wird mit Testdatensequenzen ausgeführt, die vor der Ausführung basierend auf einer textuellen Signalspezifikation generiert wurden. MbET realisiert ein *closed-loop*-Testverfahren und kombiniert das Prinzip des modellbasierten Testens (siehe Abschnitt 2.5) mit einem suchbasierten Ansatz (vorgestellt im nächsten Kapitel).

3 EVOLUTIONÄRES TESTEN

„Keine noch so große Zahl von Experimenten kann beweisen, dass ich recht habe; ein einziges Experiment kann beweisen, dass ich unrecht habe."

— Albert Einstein (1879 - 1955)

In diesem Kapitel werden die wichtigsten Charakteristika und der Ablauf von evolutionären Algorithmen dargestellt (Abschnitt 3.1). Evolutionäre Algorithmen sind populationsbasierte meta-heuristische Optimierungsalgorithmen und können für die Lösung einer Vielzahl von Optimierungsproblemen eingesetzt werden. Der Schwerpunkt dieser Arbeit liegt auf der Verwendung für den Softwaretest – bekannt als das evolutionäre Testen (Abschnitt 3.2). Beim evolutionären Testen werden Testfälle automatisiert erstellt, ausgeführt und ausgewertet. In der Vergangenheit wurde das evolutionäre Testen für das Erreichen verschiedenartiger Testziele eingesetzt. Zwei Anwendungsgebiete werden in diesem Kapitel vertieft: Der evolutionäre Strukturtest (Abschnitt 3.3) und der evolutionäre Funktionstest (Abschnitt 3.4). Das in Kapitel 4 vorgestellte Verfahren (OLET) baut auf dem aktuellen Forschungsstand des evolutionären Funktionstests auf. Das Verfahren in Kapitel 5 (MbET) greift zusätzlich auf das Prinzip des evolutionären Strukturtests zurück.

3.1 EVOLUTIONÄRE ALGORITHMEN

In den 1980er Jahren stellte Holland basierend auf den Evolutionstheorien von Darwin den Schemasatz vor [30, 56]. Dieser verdeutlichte das Potential genetischer Algorithmen zur Lösung

komplexer Optimierungsprobleme. Genetische Algorithmen zählen zur Familie der evolutionären Algorithmen und zeichneten sich ursprünglich durch eine binäre Kodierung und konstante Länge der Lösungen aus. Weitere Ausprägungen evolutionärer Algorithmen sind genetische Programmierung, Evolutionsstrategien, evolutionäre Programmierung und memetische Algorithmen. Einen Überblick über die verschiedenen Verfahren bieten Nissen und Pohlheim [84, 92]. Evolutionäre Algorithmen zählen zur Klasse der globalen Suchverfahren. Verfahren wie das Hill Climbing realisieren im Gegensatz dazu eine lokale Suche, die sich im Suchraum ausgehend von initialen Lösungsvorschlägen ausschließlich in Richtung besserer Lösungen bewegt [47]. Bei der lokalen Suche ist folglich die Wahrscheinlichkeit größer, dass die Suche in einem lokalen Optimum endet. Für einfache Optimierungsprobleme kann die lokale Suche allerdings effizienter sein und ist in der Regel leichter zu implementieren.

Evolutionäre Algorithmen versuchen das Problem des Auffindens von lediglich lokalen Optima im Suchraum durch den Informationsaustausch zwischen den Lösungsvorschlägen und die Mutation einzelner Lösungsvorschläge zu umgehen. In Abbildung 3.1 ist der Ablauf eines evolutionären Algorithmus dargestellt. Als erster Schritt wird eine Menge von Lösungsvorschlägen (Individuen) mit einem Zufallsgenerator erstellt (*Initialisierung*). Anschließend werden die Individuen im Hinblick auf das Ziel der Suche bewertet (*Evaluierung*) und die Suche wird abgebrochen (*Abbruch*), falls das Suchziel bereits erreicht wurde. Anderenfalls beginnt der Optimierungskreislauf, bei dem als Erstes ein Teil der Individuen der aktuellen Generation ausgewählt wird. Diese Individuen stellen das Erbgut für die nachfolgende Generation zur Verfügung. Mit den *Mutations-* und *Rekombinationsoperatoren* werden diese Individuen verändert und ein Fitnesswert durch die *Evaluation* zugewiesen. Der Fitnesswert ist ein atomarer Zahlenwert, der den Abstand eines Individuums zu dem jeweiligen Testziel beschreibt. Im Prinzip kann der Kreislauf an dieser Stelle mit den neu gewonnen Individuen von vorn beginnen. Dabei besteht aber die Gefahr, dass wichtige Informationen aus vorangegangenen Generationen verloren gehen. Stattdessen wird von dem *Wiedereinfügeoperator* ein Teil der Individuen aus der Elterngeneration ausgewählt und in die nächste Generation unverändert übernommen. Danach beginnt der Kreislauf wieder von vorn und die Suche wird fortgesetzt, bis entweder das Suchziel oder ein

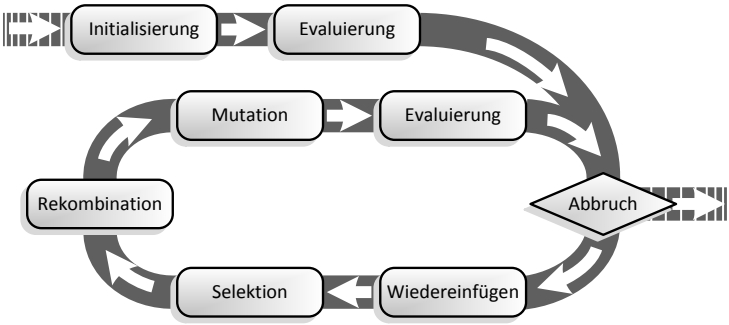


Abbildung 3.1: Optimierungskreislauf der evolutionären Algorithmen.

anderes Abbruchkriterium erreicht wird. Neben dem Suchziel ist das Erreichen einer spezifizierten maximalen Anzahl von Generationen das häufigste Abbruchkriterium. In einer Arbeit von Jain *et al.* werden alternative Abbruchkriterien vorgestellt und bewertet [59].

Im Folgenden werden die Operatoren der evolutionäre Algorithmen kurz vorgestellt. Tabelle 3.1 zeigt eine Übersicht über die wichtigsten Parameter, mit denen die Operatoren konfiguriert werden. Der Fokus liegt auf den Operatoren und Konfigurationsparametern, die bei den Fallstudien in dieser Arbeit Anwendung finden (siehe Abschnitt 6.1). Für eine bessere Veranschaulichung wird in diesem Abschnitt davon ausgegangen, dass jedes Individuum aus genau einem binär-kodierten Chromosom besteht. Die in dieser Arbeit entwickelten Verfahren unterstützten auch Individuen mit mehreren, dezimal-kodierten Chromosomen. Ein Chromosom beschreibt beim evolutionären Testen eine Menge von Testeingaben, die zu einer logischen Einheit zusammengefasst sind (zum Beispiel Stimuli für einen Eingang eines Testobjekts).

SELEKTION

Der *Selektionsoperator* steuert die Suchrichtung der Evolution durch die Auswahl von Individuen. Er wählt diejenigen Individuen aus der Elterngeneration aus, welche zur Produktion neuer Individuen für die Nachfolgegeneration zu verwenden sind. Individuen werden in Abhängigkeit ihres Fitnesswerts ausgewählt – Individuen mit einem besseren Fitnesswert werden

Parameter	Beschreibung
Populationsgröße	Die Anzahl der Individuen einer Generation.
Generationslücke	Das Verhältnis der zu produzierenden Nachkommen zur *Populationsgröße*.
Optimierungsrichtung	Legt fest, ob die Suche in Richtung von größeren oder kleineren Fitnesswerten gesteuert werden soll (Maximierung/Minimierung).
Fruchtbarkeitsrate	Der Anteil der Individuen einer Generation, deren Erbgut für die Erstellung der Nachfahren verwendet wird.
Rekombinationsrate	Der Anteil der fruchtbaren Individuen, auf die der *Rekombinationsoperator* angewendet wird.
Mutationsrate	Der Anteil der fruchtbaren Individuen, auf die der *Mutationsoperator* angewendet wird.
Mutationsschritt	Das Ausmaß der durch den *Mutationoperator* hervorgerufenen Änderung. Wird abhängig vom Wertebereich der mutierten Variablen berechnet (siehe Pohlheim [92]).
Selektionsoperator	Vorgehensweise bei der Auswahl von Individuen für die nächste Zwischenpopulation oder Generation.
Elite	Der Anteil der Individuen, die direkt in die Zwischen- beziehungsweise die finale Population übernommen werden. Die Auswahl erfolgt in der Regel proportional zur Fitness.
Art von Elitismus	Legt fest, ob elitäre Individuen direkt in die nächste Generation übernommen (stark) oder ob Sie zuerst in die Zwischenpopulation eingefügt werden (schwach).

Tabelle 3.1: Konfigurationsparameter für den evolutionären Algorithmus.

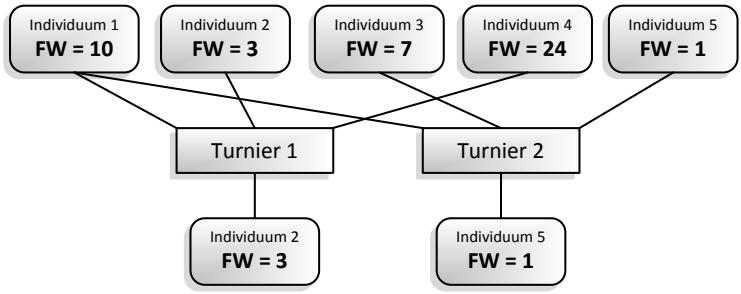


Abbildung 3.2: Auswahl von Individuen mittels *Turnierselektion*. Individuen werden zufällig zugeteilt, das Individuum mit dem besten Fitnesswert gewinnt das jeweilige Turnier. Die Turniere werden solange ausgeführt, bis die benötigte Anzahl von Individuen erreicht ist.

bei der nachfolgenden *Rekombination* bevorzugt herangezogen. Der Fitnesswert steht somit in direktem Zusammenhang mit der Selektionswahrscheinlichkeit der einzelnen Individuen. Abbildung 3.2 zeigt ein Beispiel für ein typisches Selektionsverfahren – die *Turnierselektion* – das auch für die Fallstudien in Kapitel 6 eingesetzt wurde (vergleiche Anhang A). Die *Turnierselektion* bevorzugt die besten Individuen; es werden aber auch Individuen mit durchschnittlichen Fitnesswerten ausgewählt, falls die Fitnesswerte aller Teilnehmer an einem Turnier durchschnittlich sind. Weitere prominente Selektionsverfahren sind die *Rouletteselektion* und das *stochastic universal sampling* (Auswahl von Individuum i mit der Wahrscheinlichkeit $p_i = f_i / \sum F$) sowie die *Truncation-Selektion* (Individuen entsprechend der Fitnesswerte aufsteigend sortieren und die schlechteren abschneiden) [92].

REKOMBINATION

Bei der *Rekombination* werden zwei oder mehr Elternindividuen herangezogen, um neue Individuen für die nachfolgende Generation zu produzieren. Das Verhältnis der zu produzierenden Nachkommen zur *Populationsgröße* wird durch die *Generationslücke* festgelegt. Die als Eltern in Frage kommenden Individuen werden im Vorfeld durch den *Selektionsoperator* ausgewählt. Ähnlich der biologischen Fortpflanzung werden die Informationen der Elternindividuen ausgetauscht und miteinander kombiniert.

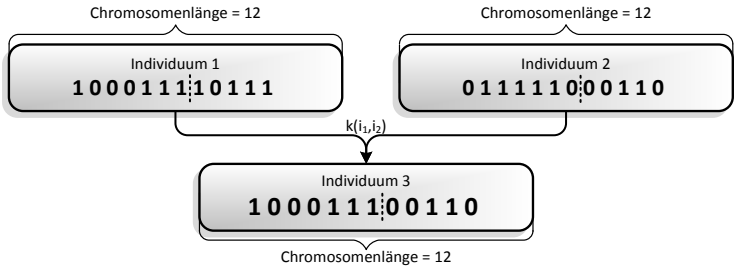


Abbildung 3.3: Rekombination von Individuen mit genetischen Algorithmen. Die Chromosomen sind binär kodiert, der Rekombinationsoperator schneidet das erste Chromosom an einer zufälligen Stelle ab und füllt die restlichen Stellen mit Teilen eines Chromosoms von einem anderen Individuum auf.

Die *Rekombinationsrate* gibt an, auf wie viel Prozent aller Individuen der *Rekombinationsoperator* angewendet wird. Abbildung 3.3 zeigt ein Beispiel für die *Rekombination* zweier Individuen. Für die *Rekombination* existieren unterschiedliche Verfahren, die in Abhängigkeit der Kodierung der Individuen und der Problemstellung ausgewählt werden können (vergleiche Pohlheim [92]).

REKOMBINATION MIT CHROMOSOMEN VARIABLER LÄNGE

In den meisten Fällen entstehen bei der *Rekombination* Chromosomen gleicher Länge, wie jedes der Elternindividuen. Für komplexe Problemstellungen ist es allerdings sinnvoll, die Flexibilität der Chromosomen durch eine variable Länge zu erhöhen. Auch in dieser Arbeit wird auf einen derartigen Ansatz zurückgegriffen (vergleiche Abschnitte 4.1 und 5.1). Das Vorgehen ist wie folgt: Bei der Rekombination werden zwei Chromosomen jeweils an einer zufällig gewählten Stelle gespalten. Der vordere Teil des ersten Chromosoms wird dann mit dem hinteren Teil des zweiten Chromosoms verbunden. Es ist oft sinnvoll, eine Maximallänge festzulegen – beim Überschreiten der Maximallänge wird der Rest abgeschnitten. Eine Übersicht über genetische Algorithmen mit Chromosomen variabler Länge bieten zum Beispiel Stringer und Wu [108]. Bei komplexen Datenstrukturen muss sichergestellt werden, dass ein gültiger Nachfahre entsteht, der für die Evaluierung verwendbar ist. Abbildung 3.4 zeigt ein Beispiel für

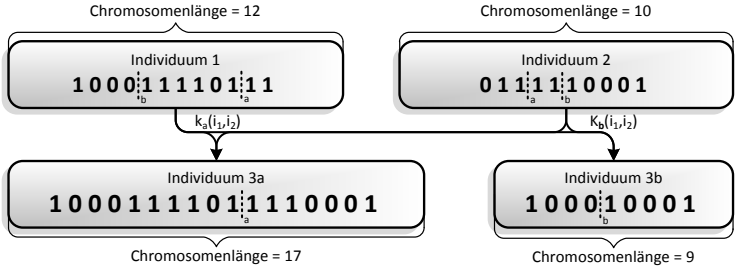


Abbildung 3.4: Rekombination von Individuen am Beispiel der genetischen Algorithmen mit variabler Chromosomenlänge. Die Chromosomen sind binär kodiert, der Rekombinationsoperator trennt die Chromosomen zweier Individuen jeweils an einer zufällig gewählten Stelle und fügt die Teile zusammen.

einen *Rekombinationsoperator*, der Chromosomen variabler Länge erzeugt.

MUTATION

Die *Mutation* erfolgt im Anschluss an die *Rekombination*. Das Ziel der *Mutation* ist, eine Konvergenz in lokale Optima zu verhindern sowie die Diversität der Population sicherzustellen. Neben der *Rekombination* ist die *Mutation* der zweite Operator, der genetische Änderungen bei den Individuen hervorruft. Diese Veränderungen (*Mutationsschritte*) sind zumeist gering und werden mit einer bestimmten Wahrscheinlichkeit (*Mutationsrate*) auf die Individuen angewendet. Zur Festlegung des *Mutationsschrittes* und der *Mutationsrate* wird eine von zwei verschiedenen Herangehensweisen verwendet: Entweder bleibt der *Mutationsschritt* und die *Mutationsrate* während des gesamten Verlaufs der Suche konstant oder sie werden stetig verringert (vergleiche auch Pohlheim [92]). Die zweite Methode hat zur Folge, dass der Algorithmus zu Beginn der Suche den Suchraum weitläufig erkundet und erst im weiteren Verlauf der Suche eine feinere Auflösung verwendet. In dieser Arbeit wird die Methode für die *Mutation* verwendet, bei der sich *Mutationsschritt* und *Mutationsrate* nicht im Verlauf der Optimierung ändern. In Abbildung 3.5 wird das Prinzip der Mutation anhand zweier Individuen veranschaulicht.

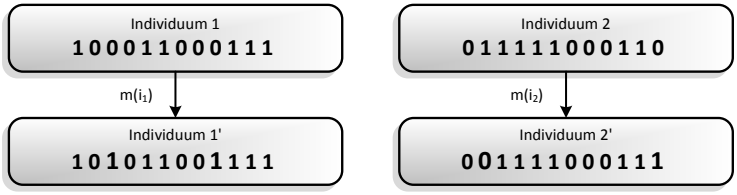


Abbildung 3.5: Mutation von Individuen am Beispiel der genetischen Algorithmen. Die Individuen sind binär kodiert, der Mutationsoperator ändert je nach Konfiguration ein oder mehrere Bits (Gene).

EVALUIERUNG

Bei der *Evaluierung* wird jedem Individuum ein Fitnesswert zugewiesen. Der Fitnesswert ist ein Maß dafür, wie weit ein Individuum vom Erreichen des jeweiligen Suchziels entfernt ist. Ein besseres Individuum hat demzufolge einen niedrigeren Fitnesswert als ein schlechteres Individuum (oder umgekehrt, wenn die Suche auf einen maximalen Fitnesswert abzielt). Die Fitnesszuweisung erfolgt entweder proportional zu dem mit der Fitnessfunktion errechneten Wert, nach der Reihenfolge, oder durch die mehrkriterielle Fitnesszuweisung. Die mehrkriterielle Fitnesszuweisung wird dann verwendet, wenn einem Individuum nicht genau ein Fitnesswert zugewiesen werden kann (zum Beispiel wenn gleichzeitig der Verbrauch und die Leistung eines Fahrzeugs optimiert werden sollen). In dieser Arbeit wird ein Verfahren verwendet, bei dem die Fitnesszuweisung proportional zum Fitnesswert erfolgt.

WIEDEREINFÜGEN

Der *Wiedereinfügeoperator* bestimmt diejenigen Individuen einer Generation, die durch neu produzierte Nachkommen zu ersetzen sind. Der Fitnesswert bestimmt die Überlebenswahrscheinlichkeit eines Individuums. Individuen mit einem besonders schlechten Fitnesswert besitzen zwar die Chance Bestandteil der nächsten Generation zu sein, jedoch ist diese zumeist sehr gering (je nach eingesetztem Verfahren). Beim einfachen *Wiedereinfügen* ist die Anzahl der Nachkommen identisch mit der Anzahl der Elternindividuen. Dies entspricht einem kompletten Ersetzen der Elterngeneration bei einer *Wiedereinfügerate* von $1,0$. Um zu verhindern,

dass wichtige Informationen aus vorangegangen Generationen verloren gehen und unter Umständen das beste Individuum der nachfolgenden Generation einen schlechteren Fitnesswert hat als das beste Individuum der Elterngeneration, wird eine *Wiedereinfügerate* kleiner als 1,0 verwendet. Beispielsweise kann bei der Kombination eines guten Individuums mit einem anderen guten Individuum ein schlechteres Individuum entstehen. Bei einer *Wiedereinfügerate* von kleiner als 1,0 muss eine Auswahl der Individuen erfolgen, die nicht durch Nachkommen ersetzt werden. Diese werden dann unverändert in die nachfolgende Generation übernommen. Auch für das *Wiedereinfügen* gibt es eine Vielzahl von Methoden (vergleiche Pohlheim [92]).

In Abbildung 3.6 ist die Funktionsweise des *Wiedereinfügeoperators* veranschaulicht, der für die Durchführung der Fallstudien verwendet wurde (siehe Kapitel 6) [28]. Dabei bilden die durch *Mutation* und *Rekombination* neu generierten Individuen zusammen mit den *fruchtbaren, nicht-elitären* Individuen die sogenannte *Nachfolgepopulation*. Durch den Einsatz des *Selektionoperators* werden aus der *Nachfolgepopulation* Kandidaten für die *Zwischenpopulation* ausgewählt. Bei schwachem *Elitismus* sind die elitären Individuen aus der *Elterngeneration* auch Bestandteil der *Zwischenpopulation*. Für die Fallstudien wurde starker *Elitismus* verwendet – die elitären Individuen werden dann direkt in die *finale Population* übernommen. In jedem Fall ist der letzte Schritt beim Wiedereinfügen, mit einem erneuten Einsatz des *Selektionoperators*, Individuen aus der *Zwischenpopulation* auszuwählen. Das Ergebnis ist dann die *finale Population*, die über die gleiche Anzahl von Individuen verfügt wie die *Elterngeneration*.

Die Anwendung der hier vorgestellten Techniken zur Lösung komplexer Probleme zählt zu dem Forschungsfeld der suchbasierten Softwaretechnik (SBSE). Untersuchungen haben ergeben, dass die SBSE für zahlreiche Probleme schneller und bessere Lösungen findet, als klassische Ansätze. Unter anderem wurde in einer Arbeit von Souza *et al.* die suchbasierte Softwaretechnik mit mehreren Versuchspersonen verglichen [102]. Erkenntnis der experimentellen Untersuchungen mit einer Auswahl von Problemen aus der Komplexitätsklasse NP-schwer ist, dass die suchbasierte Softwaretechnik effizienter und effektiver beim Auffinden einer Lösung ist. In einer Arbeit von Harman *et al.* wird ein umfassender Überblick über die verschiedenen Forschungsfelder im

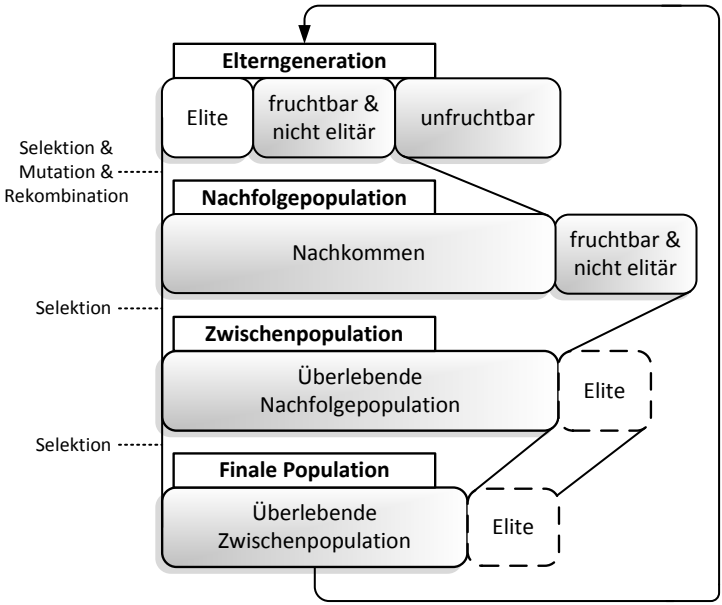


Abbildung 3.6: Prozess der Auswahl von Individuen für die nächste Generation (*finale Population*) mit einer Nachkommensanzahl gleich der Ausgangspopulationsgröße. In der Abbildung wird auch der Unterschied zwischen schwachem *Elitismus* (einfügen in *Zwischenpopulation*) und starkem *Elitismus* (einfügen in *finale Population*) veranschaulicht.

Bereich der suchbasierten Softwaretechnik geboten [49]. Eine Auswertung darin verdeutlicht, dass der Schwerpunkt der Forschung bei über 50% der Veröffentlichungen beim Testen und Debuggen liegt. Eine weitere Erkenntnis in der Arbeit von Harman *et al.* ist, dass die Anzahl der Veröffentlichungen in diesem Bereich in den letzten zwanzig Jahren kontinuierlich zugenommen hat. Die Verwendung von Suchverfahren für den Softwaretest wird im Forschungsfeld suchbasierter Softwaretest (SBST) zusammengefasst. In den folgenden Abschnitten wird das evolutionäre Testen, eine konkrete Anwendung des SBST, vertieft.

3.2 Automatisierte Testfallerstellung mit genetischen Algorithmen

Der am einfachsten zu realisierende Ansatz für die automatisierte Testfallerstellung ist der Zufallstest. Bereits in den 1970er Jahren wurde die dynamische Generierung von Testeingaben mit einem Zufallsgenerator untersucht (vergleiche Ince [57]). Auch heutzutage kommt der Zufallstest aufgrund der einfachen Anwendbarkeit noch für eine Vielzahl von Problemen zum Einsatz. Da die Suche nach Testfällen nicht gesteuert wird, sinkt allerdings bei zunehmender Größe des Suchraums die Wahrscheinlichkeit, das Suchziel zu erreichen. Eine Alternative bietet das evolutionäre Testen.

Der Begriff evolutionäres Testen ist allgemein bekannt als die Anwendung von genetischen Algorithmen für den Softwaretest [47]. Eine für das Suchziel geeignete Fitnessfunktion vorausgesetzt, wird beim evolutionären Testen die Suche nach Testfällen automatisch in Richtung der relevanten Bereiche im Suchraum gesteuert. Evolutionäres Testen kann für die vollständige Automatisierung der Erstellung, Ausführung und Bewertung von Testfällen für verschiedene Testarten eingesetzt werden [78, 105]. Eine große Anzahl von Testfällen kann durchgeführt werden, ohne dass ein manueller Eingriff eines Testers notwendig ist. Der Suchraum für die evolutionäre Suche ist der Eingangsdatenraum der jeweiligen Testobjekte. In der Forschung wird der zuvor beschriebene Zufallstest oft zum Vergleich für die Effektivität des evolutionären Testens herangezogen. Der evolutionäre Test liefert in fast allen Fällen bessere Ergebnisse – sowohl bei der Effizienz als auch bei der Effektivität [74, 122].

Grundvoraussetzung für die Durchführung evolutionärer Tests ist ein Testziel, das numerisch mit Hilfe einer Fitnessfunktion beschrieben werden kann. Unabhängig von dem jeweiligen Testziel ist der automatisierte Ablauf eines evolutionären Tests wie folgt: Zuerst wird eine Menge von Individuen nach dem Zufallsprinzip erzeugt und das Testobjekt mit den in den Individuen kodierten Testeingaben ausgeführt. Basierend auf den Ergebnissen der Testausführung berechnet die Fitnessfunktion Fitnesswerte für die Individuen. Die nächste Generation wird mit den im vorangegangenen Abschnitt beschriebenen *Selektions-*, *Mutations-*,

Rekombinations-, *Evaluierungs-* und *Wiedereinfügungsoperatoren* erzeugt (vergleiche Abbildung 3.1). Die Ausprägungen des evolutionären Testens für die Realisierung unterschiedlicher Testarten unterscheiden sich in erster Linie durch den Evaluierungsoperator. Ein evolutionärer Algorithmus kann sowohl dazu verwendet werden, einen Satz von Testfällen für die Erfüllung von Überdeckungskriterien zu bestimmen (Strukturtest), als auch für die Suche nach Anforderungsverletzungen eines Systems (Funktionstest). Bei der Erfüllung von Überdeckungskriterien entspricht ein Fitnesswert dem Abstand zum Überdecken beispielsweise einer durch eine *if*-Anweisung hervorgerufenen Verzweigung im Programmcode. Ist es das Ziel, eine Anforderungsverletzung aufzuzeigen, repräsentiert der Fitnesswert dagegen die Kritikalität des Testfalls im Hinblick auf das Verletzen der Anforderung.

Die meisten veröffentlichten Arbeiten im Bereich evolutionäres Testen beschäftigten sich mit dem Auffinden einer Menge von Testfällen mit dem Ziel, Programmcode zu überdecken [49]. Evolutionäres Testen wurde aber auch für den Funktionstest, den Grey-Box Test und den Test von nicht-funktionalen Eigenschaften angewendet. Einen Überblick über die verschiedenen suchbasierten Testansätze wird in Arbeiten von Harman *et al.* geboten [49]. Weiterhin haben Afzal *et al.* eine Übersicht über Arbeiten im Bereich suchbasiertes Testen von nicht-funktionalen Eigenschaften erstellt [1]. In den nachfolgenden Abschnitten werden der evolutionäre Strukturtest und der evolutionäre Funktionstest vertieft.

3.3 EVOLUTIONÄRER STRUKTURTEST

Erstmals verwendeten Xanthakis *et al.* im Jahr 1992 evolutionäre Algorithmen für den Strukturtest [131]. Der evolutionäre Strukturtest hat das Ziel, eine Menge von Testfällen zu bestimmen, die ein Programm oder eine Funktion nach bestimmten Kriterien überdeckt [61, 82, 107]. Der Schwerpunkt in der Forschung liegt seit Ende der 1990er Jahre auf der Überdeckung von prozeduralem Programmcode (in erster Linie ANSI-C). Die einzelnen Arbeiten versuchen die Effektivität, gemessen an dem erreichten Überdeckungsgrad, zu steigern. Ein höherer Überdeckungsgrad kann durch eine verbesserte Fitnessfunktion oder

eine optimierte Suchlandschaft erreicht werden (vergleiche Jones und Harman *et al.* [48, 62]). Zu Beginn des einundzwanzigsten Jahrhunderts wurde zudem auch die erfolgreiche Anwendung des evolutionären Strukturtests für objektorientierte Sprachen (siehe Wappler und Wegener [121]) und Simulink-Modelle mit Stateflow-Diagrammen demonstriert (siehe Windisch und Ghani *et al.* [39, 128]).

Der evolutionäre Strukturtest von prozeduralem Code wird seit fast zwanzig Jahren erforscht. Dennoch wird der Ansatz nur vereinzelt in der Praxis bei der Entwicklung von Software für eingebettete Systeme verwendet. Ursachen liegen zum einen darin, dass abgesehen von einigen akademischen Ansätzen (zum Beispiel Tracey *et al.* [115]) nur unzureichend Werkzeugunterstützung vorhanden ist. Abhilfe zu schaffen versucht beispielsweise das EU-Förderprojekt EvoTest durch die Entwicklung des EvoTest-Frameworks [36]. Weiterhin hatte die Überdeckung von Programmcode in vielen Industriezweigen bisher einen geringen Stellenwert. Insbesondere aber im Automobilbereich wird in Zukunft verstärkt durch Standards und Regulierungen gefordert, Programmcode sicherheitsrelevanter Systemfunktionen mit Testfällen zu überdecken (siehe Abschnitt 2.3) [58]. Die Bedeutung von Verfahren zur Überdeckung von Programmcode wird folglich in naher Zukunft zunehmen.

Der evolutionäre Strukturtest ist ein dynamisches Verfahren. Dabei wird das Testobjekt mit verschiedenen Eingaben ausgeführt, um zu überprüfen, ob das jeweilige Überdeckungsziel durch einen Testfall erreicht wird. Die Suche nach Testfällen erfolgt vollständig automatisch. Die Automatisierung wird erreicht, indem die Bestimmung von Testfällen zur Überdeckung von Programmcode in ein Optimierungsproblem transformiert wird. Mit Hilfe der evolutionären Algorithmen wird versucht, das Optimierungsproblem zu lösen, das heißt, die Überdeckungsziele zu erreichen. Voraussetzung ist allerdings, dass eine Testumgebung vorhanden ist, die eine Programmiersprache und die geforderte Überdeckungsart unterstützt. Tabelle 3.2 zeigt eine Übersicht über die gängigsten kontrollflussorientierten Überdeckungsarten, die beim Strukturtest angestrebt werden. Für den Schwerpunkt dieser Arbeit ist die Zweigüberdeckung relevant.

Der Ablauf der Suche erfolgt analog zu der in Abbildung 3.1 aufgezeigten Vorgehensweise. Der Schlüssel zur Anwendbarkeit

Überdeckungsart	Beschreibung
Anweisungs-überdeckung (C_0)	Bei der Anweisungsüberdeckung muss jede Anweisung im Programmcode einmal zur Ausführung gebracht werden.
Zweigüberdeckung (C_1)	Für die Zweigüberdeckung muss jede Verzweigung im Programmcode mindestens einmal durchlaufen werden.
Bedingungs-überdeckung (C_2, C_3)	Für die einfache Bedingungsüberdeckung muss jede Bedingung vor Verzweigungen im Programmcode sowohl zu wahr als auch zu falsch ausgewertet werden. Mehrfache Bedingungsüberdeckung erfordert zusätzlich Kombinationen der atomaren Ausrücke.
Pfadüberdeckung (C_4)	Die Pfadüberdeckung erfordert die Ausführung aller möglichen Pfade durch einen Programmcode.

Tabelle 3.2: Übersicht über kontrollflussorientierte Überdeckungsarten beim Strukturtest (C_X sind gängige Kurzbezeichnungen).

für den Strukturtest liegt in der Überwachung des Kontrollflusses während der Testausführung und der problemspezifischen Definition einer Fitnessfunktionen für die *Evaluierung* der Individuen. In den Individuen sind die für die Ausführung der Testobjekte erforderlichen Testeingaben kodiert. Das Ergebnis der *Evaluierung* sind Fitnesswerte, die ein Messwert dafür sind, wie nah ein Testfall am Erreichen eines Überdeckungsziels ist. Um den Kontrollfluss bei der Ausführung mit einem Testfall messbar zu machen, muss der Programmcode zuerst instrumentiert werden. Instrumentieren ist der Prozess, der alle Verzweigungen im Programmcode analysiert und mit Hilfe der Abstandsfunktionen transformiert.

Abbildung 3.7 veranschaulicht eine Vorgehensweise bei der Instrumentierung von Programmcode und der Evaluierung von Testfällen. Als Erstes wird über den Zwischenschritt der Erzeugung eines abstrakten Syntaxbaums der *Kontrollflussgraph* von dem Testobjekt abgeleitet. Der *Kontrollflussgraph* fasst Informationen über alle Verzweigungen im Programmcode zusammen. Der *Instrumentierer* fügt dann die für die Messbarmachung erforderli-

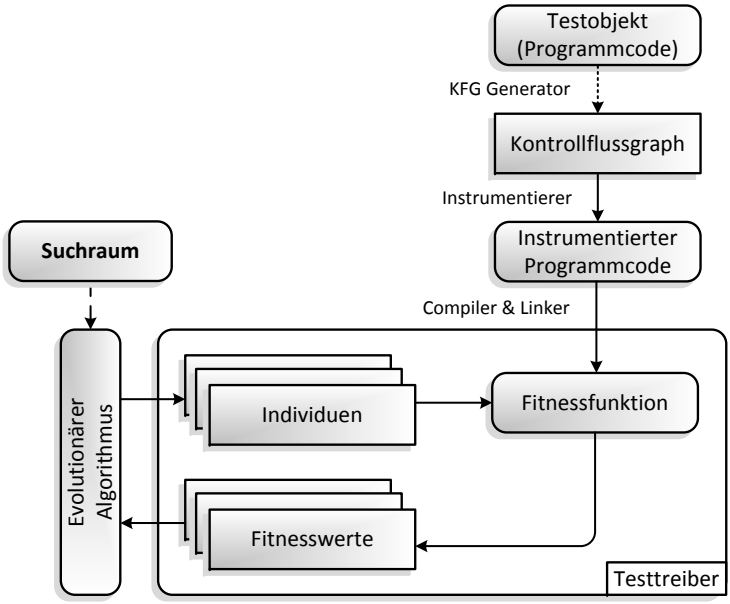
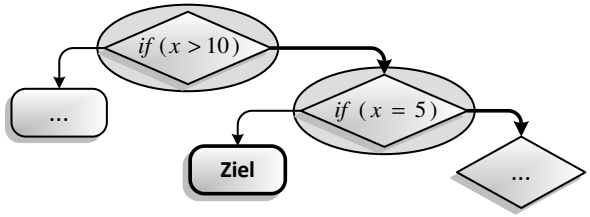


Abbildung 3.7: Ablauf der Instrumentierung und Evaluierung von Testfällen beim evolutionären Strukturtest mit KFG: Kontrollflussgraph.

chen Erweiterungen in den *Kontrollflussgraphen* ein und generiert den *instrumentierten Programmcode*. Anschließend muss der Programmcode noch durch den Einsatz eines *Compilers* und eines *Linkers* in ein ausführbares Programm (hier: *Fitnessfunktion*) übersetzt werden. Dieses Programm liefert dann bei der Ausführung mit einem Testfall den dazugehörigen *Fitnesswert* (Abstand zum Erreichen des Testziels). Im unteren Teil der Abbildung 3.7 wird der Schritt der Evaluation von Individuen skizziert. Die Individuen einer Generation werden durch die *Fitnessfunktion* bewertet. Die *Fitnesswerte* werden dann in den Kreislauf als Gewichte für die evolutionären Operatoren zurückgeführt.

Für die Berechnung von Fitnesswerten für den evolutionären Strukturtest hat sich folgender Ansatz in der Forschung etabliert (siehe Wegener *et al.* und Kalaji *et al.* [64, 125]):

$$\mathrm{Fitnesswert} = \mathrm{Abstandsstufe} + \mathrm{Zweigabstand} \qquad (3.1)$$




```
1:  void f( int x )
2:  {
3:      if( x > 10 )
4:          …
5:      else
6:          if( x = 5 )
7:              Ziel;
8:          else
9:              …
10: }
```

```
1:  int f_ins( int x )
2:  {
3:      if( x > 10 )
4:          return 1+(1+ε)^-|10-x|;
5:      else
6:          if( x = 5 )
7:              return -1;
8:          else
9:              return (1+ε)^-|x-5|;
10: }
```

Abbildung 3.8: Programmcode (links), Kontrollflussgraph (oben) und instrumentierter Programmcode (rechts).

Die *Abstandsstufe* beschreibt die Anzahl von Verzweigungen zwischen dem Zielknoten und der Verzweigung, bei dem der Kontrollfluss in die falsche Richtung abgezweigt ist. Die falsche Richtung bezeichnet einen Pfad, der nicht mehr zum gewünschten Ziel führen kann. Um Individuen unterscheiden zu können, die das Ziel bei derselben Verzweigung verfehlt haben, wird bei der Berechnung des *Fitnesswertes* auch der *Zweigabstand* mit einbezogen. Der *Zweigabstand* ist ein Maß dafür, wie weit ein Testfall davon entfernt ist, bei dieser Verzweigung dem anderen Zweig zu folgen. Wenn zum Beispiel eine Verzweigung mit der Bedingung $x = y$ ausgewertet werden soll, kann der *Zweigabstand* als $|x - y|$ berechnet werden. Damit der Wert für den Zweigabstand nicht größer als die *Abstandsstufe* werden kann, wird der *Zweigabstand* auf einen Wert: $0 \leqslant \mathrm{Zweigabstand} < 1$ normiert. Insgesamt ergibt sich also für den normierten *Zweigabstand* an der Verzweigung $x = y$ (vergleiche Baresel und Wappler [4, 120]):

$$\mathrm{Zweigabstand} = 1 - (1 + \epsilon)^{-|x-y|} \tag{3.2}$$

wobei gilt: $0 < \epsilon \ll 1$.

Abbildung 3.8 zeigt den Ausschnitt eines Programmcodes, den davon abgeleiteten Kontrollflussgraphen sowie den instrumentierten Programmcode. Um das *Ziel* zu erreichen, muss der Programmcode mit dem Wert $x = 5$ aufgerufen werden. Bei einem Eingabewert

Bedingung	Abstand zu *wahr*	Abstand zu *falsch*
$x = y$	$1 - (1 + \epsilon)^{-\lvert x-y \rvert}$	trivial
$x \neq y$	trivial	Abst$(x = y)$
$x < y$	$1 - (1 + \epsilon)^{y-x} \cdot (1 - \kappa)$	$1 - (1 + \epsilon)^{x-y}$
$x \leqslant y$	$1 - (1 + \epsilon)^{y-x}$	$1 - (1 + \epsilon)^{x-y} \cdot (1 - \kappa)$
$e_1 \wedge e_2$	$\frac{1}{2}$Abst(e_1) + $\frac{1}{2}$Abst(e_2)	min(Abst(e_1), Abst(e_2))
$e_1 \vee e_2$	min(Abst(e_1), Abst(e_2))	$\frac{1}{2}$Abst(e_1) + $\frac{1}{2}$Abst(e_2)
$\neg e$	Negation auflösen	Negation auflösen

Tabelle 3.3: Funktionen für die Berechnung des *Zweigabstands* mit $0 < \epsilon \ll 1$ und $0 < \kappa \ll 1$ (κ ist kleinstmöglicher positive Wert eines Datentyps).

ungleich 5 wird der Abstand aus Zeile 4 oder 9 zurückgegeben. Tabelle 3.3 zeigt eine Übersicht über die benötigten Abstandsfunktionen für die Überdeckungen von gängigen prozeduralen Programmiersprachen (zum Beispiel ANSI-C). Diese bilden die Grundlage für die Bewertungsmethode von MbET (vorgestellt in Abschnitt 5.2).

3.4 Evolutionärer Funktionstest

Der evolutionäre Funktionstest hat das Ziel, automatisiert Testfälle zu bestimmen, die eine funktionale Anforderung an ein Testobjekt verletzen. Erreicht wird die Automatisierung durch die Transformation des Testeingaben-Generierungsprozesses in ein Optimierungsproblem. Mit Hilfe evolutionärer Algorithmen wird dann versucht, das Optimierungsproblem zu lösen. Vor der Durchführung eines evolutionären Funktionstests müssen zuerst drei testspezifische Voraussetzungen erfüllt werden. Im Folgenden

werden die drei Voraussetzungen, Fitnessfunktion, Individuen-/Suchraumspezifikation und Testtreiber, beschrieben:

FITNESSFUNKTION

Die Aufgabe der Fitnessfunktion ist es zu bestimmen, wie weit ein Testfall vom Erreichen des jeweiligen Suchziels, beziehungsweise dem Brechen einer Anforderung, entfernt ist. Die Fitnessfunktion wird mit den bei der Testausführung aufgezeichneten Ein- und Ausgaben aufgerufen und liefert dann einen Fitnesswert zurück, der die Güte des Testfalls repräsentiert. Die Fitnesswerte werden anschließend von dem Selektionsoperator verwendet, um die einzelnen Individuen miteinander zu vergleichen und die Suche zu steuern.

INDIVIDUEN- UND SUCHRAUMSPEZIFIKATION

Als nächster Schritt müssen die Individuenspezifikation und der Suchraum für den evolutionären Algorithmus spezifiziert werden. Die Individuenspezifikation beschreibt, wie die Testeingaben in den Individuen kodiert sind. Der Suchraum legt die Wertebereiche für die in den Individuen kodierten Testeingaben fest und sollte auf die für das jeweilige Suchziel relevanten Bereiche reduziert werden. Dabei muss sichergestellt werden, dass das Ziel der Suche nicht außerhalb des Suchraums liegt. Die Größe des Suchraums beeinflusst die Wahrscheinlichkeit für das Erreichen des Suchziels innerhalb eines zeitlich vorgegebenen Rahmens.

TESTTREIBER

Der Testtreiber führt das Testobjekt mit den generierten Testeingaben aus. Die Testeingaben werden dafür auf die Eingangskanäle des Testobjekts abgebildet. Im Anschluss an die Testausführung werden die Testein- und Testausgaben an die Fitnessfunktion zur Bewertung weitergeleitet. Für die Kommunikation zwischen Optimierungseinheit und Testobjekt wird ein Adapter realisiert. Der Adapter ist aber nicht testobjektspezifisch sondern testplattformspezifisch und kann für andere Testobjekte wiederverwendet werden.

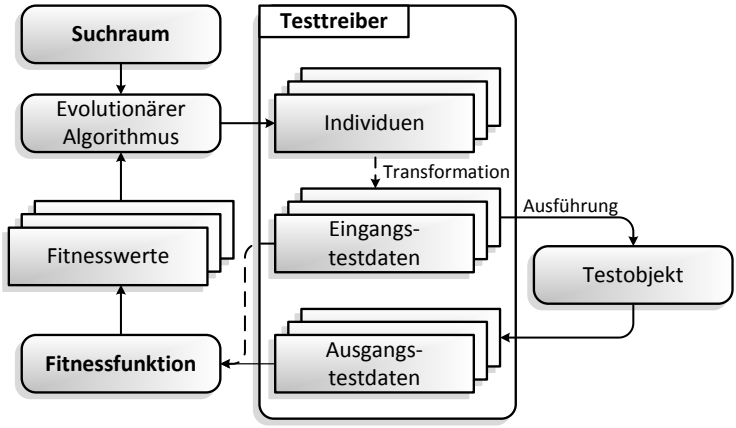


Abbildung 3.9: Ablauf der Evaluierung von Testfällen beim evolutionären Funktionstest. Fett gedruckt sind die drei Testobjekt-spezifischen Anteile. Gestrichelte Linien kennzeichnen einen optionalen Datenfluss.

Abbildung 3.9 verdeutlicht das Zusammenspiel der für die Durchführung eines evolutionären Funktionstests erforderlichen Komponenten. Sind die drei testspezifischen Voraussetzung erfüllt, wird eine durch das Zufallsprinzip erzeugte Menge von *Individuen* an den *Testtreiber* übergeben. Der *Testtreiber* transformiert die *Individuen* in *Eingangstestdaten* (zum Beispiel in Testdatensequenzen) und nutzt diese für die Ausführung des *Testobjekts*. Nach der Ausführung werden die *Eingangs-* und *Ausgangstestdaten* an die *Fitnessfunktion* weitergeleitet. Der *evolutionäre Algorithmus* generiert dann, basierend auf den Erkenntnissen aus der Fitnessberechnung, die nächste Generation von *Individuen*.

Tabelle 3.4 zeigt einen Aufwandsvergleich zwischen evolutionärem Strukturtest und evolutionärem Funktionstest. Hierdurch werden die Ursachen für die unterschiedliche Akzeptanz des evolutionären Strukturtests und des evolutionären Funktionstests verdeutlicht. Insbesondere für den Einsatz in der Industrie wurde der evolutionäre Funktionstest zumeist als zu aufwändig und zu kompliziert angesehen.

Die beiden suchbasierten Testverfahren OLET (Kapitel 4) und MbET (Kapitel 5) knüpfen an den nachfolgend vorgestellten Stand der Forschung an und erweitern diesen um Konzepte für den effi-

	Evolutionärer Strukturtest	Evolutionärer Funktionstest
Testziel	Überdeckung von Programmcode	Auffinden von Anforderungsverletzungen
Fitness-funktion	automatisch, abgeleitet vom Programmcode	problemspezifisch
Suchraum	automatisch, manuelle Einschränkung möglich	problemspezifisch
Testtreiber	automatisch	problem- und plattformspezifisch, teilweise automatisierbar

Tabelle 3.4: Vergleich der Voraussetzungen des evolutionären Strukturtests mit denen des evolutionären Funktionstest hinsichtlich des Aufwands für den Tester.

zienten und effektiven Funktionstest von Software eingebetteter Systeme.

STAND DER FORSCHUNG EVOLUTIONÄRER FUNKTIONSTEST

Die Anwendbarkeit genetischer Algorithmen für den Funktionstest wurde erstmals im Jahr 1995 von Jones *et al.* demonstriert [60]. Das Testobjekt wurde formal mit der Z-Notation spezifiziert und die Testfälle automatisiert von der Spezifikation abgeleitet. Ende der 1990er Jahre wurde dieser Ansatz von Tracey *et al.* erweitert [114, 116]. Weitere Arbeiten wurden Anfang des einundzwanzigsten Jahrhunderts von Baresel *et al.* und Pohlheim *et al.* mit einem Abstandsregeltempomaten durchgeführt [6, 93]. Zeitgleich wurde der evolutionäre Funktionstest von Bühler und Wegener auf einen Parkassistenten und einen Bremskraftverstärker angewendet [15, 17, 18]. Im Rahmen des NATUS-Projekts stellten Vieira *et al.* im Jahr 2006 einen weiteren Ansatz vor, bei dem Testsequenzen unter Verwendung von genetischen Algorithmen

erstellt wurden. Der Schwerpunkt lag auf der Qualitätssicherung von webbasierten Anwendungen [117]. Lefticaru und Ipate verfolgten ein Jahr später einen Ansatz, bei dem mit genetischen Algorithmen die Überdeckung endlicher Zustandsautomaten angestrebt wurde, die funktionale Spezifikationen darstellen [70]. Als Testobjekte verwendeten Lefticaru und Ipate JAVA-Klassen. Im Jahr 2009 wurde von Kruse *et al.* eine Fallstudie mit einem ABS durchgeführt [68] – zeitgleich zur Forschung des Autors, welche die Grundlage dieser Arbeit bildet. Schließlich haben Hänsel *et al.* und Hemmati *et al.* im Jahr 2010 Verfahren entwickelt, die mit Hilfe evolutionärer Algorithmen auf die Maximierung der Überdeckung von Testmodellen (Timed Automata und UML Zustandsautomaten) abzielen [52, 55]. Nachfolgend werden die Arbeiten aus dem Kontext der Qualitätssicherung von Software eingebetteter Systeme genauer betrachtet.

Im Rahmen der von Bühler und Wegener durchgeführten Fallstudie wurde untersucht, ob ein Parkassistent beim Einparkvorgang in bestimmten Situationen ein anderes Fahrzeug oder den Bordstein berührt. Mit Hilfe eines evolutionären Algorithmus wurden die Abmessungen eines potentiellen Parkplatzes und die Position des einparkenden Fahrzeugs relativ zu dem Parkplatz variiert [16, 18]. Der eigentliche Parkvorgang erfolgte dann automatisch in der Simulation mit der entsprechenden Steuerungssoftware. Jedes Individuum enthält insgesamt fünf Werte (Gene): Länge und Breite des Parkplatzes sowie x-Koordinate, y-Koordinate und Winkel relativ zum Parkplatz. Durch die evolutionäre Suche konnten Testfälle gefunden werden, die in der Simulation eine Kollision provozierten. In einer anderen Arbeit von Bühler und Wegener wurde eine Fallstudie mit einem Bremskraftverstärker durchgeführt [17, 18]. Aufgabe des Bremskraftverstärkers ist es, den Bremsdruck in Situationen zu verstärken, bei denen die angeforderte Bremskraft des Fahrers nicht ausreichend ist, um einen Unfall zu verhindern. Dafür wurde in einem festgelegten Szenario mit zwei Fahrzeugen der Zeitpunkt, an dem der Fahrer die Bremse auslöst sowie die Stärke und Dauer der Bremsung mit evolutionären Algorithmen, simuliert. Durch die evolutionäre Suche konnte ein Szenario identifiziert werden, bei dem der Bremsassistent den Bremsdruck in einer unkritischen Situation fälschlicherweise verstärkt hatte.

Bei den von Baresel *et al.* und Pohlheim *et al.* durchgeführten Experimenten mit einem Abstandsregeltempomaten wurden bereits kontinuierliche Testdatensequenzen durch die Aneinanderreihung von Signalabschnitten (Segmenten) generiert [6, 93]. Ziel war es, eine Verletzung des Mindestabstands zum vorausfahrenden Fahrzeug durch den Tempomaten aufzuzeigen. Der verwendete Ansatz wies allerdings noch einige Limitierungen auf. Dazu zählen die feste Länge und die feste Anzahl von Segmenten für jede Testdatensequenz sowie die nicht unterstützte Interaktivität zwischen Testobjekt und Testtreiber. Die Testdatensequenzen wurden immer statisch vor der Ausführung festgelegt. Weiterhin war der Praxisbezug begrenzt, weil kein Testobjekt aus dem industriellen Kontext verwendet wurde, sondern ein Versuchsobjekt aus dem akademischen Umfeld.

Kruse *et al.* beschreiben einen Testansatz, der das EvoTest-Framework verwendet, um automatisiert Testeingaben für ein ABS zu erzeugen [36, 68, 123]. Das Besondere bei dieser Arbeit ist, dass die Testfälle auf einem realen Steuergerät evaluiert wurden. Das Ziel dieser Fallstudie war, den Bremsweg zu maximieren, den ein Fahrzeug bei einer bestimmten Ausgangsgeschwindigkeit zurücklegt. Es konnte allerdings kein Fehlverhalten bei dem untersuchten ABS festgestellt werden. Der verwendete Signalgenerierungsansatz war ebenfalls auf die Erstellung von Testdatensequenzen mit einer festen Länge und einer festen Anzahl von Segmenten beschränkt. Weiterhin wurde auch hierbei keine Interaktivität zwischen Testobjekt und Testtreiber unterstützt.

Im Rahmen der Forschung von Hänsel *et al.* wurde *EvoTA* (Evolutionary test generation from Timed Automata) entwickelt – ein evolutionäres Testverfahren, das zeitbehaftete Test-Traces von UPPAAL Timed Automata Modellen ableitet [7, 55]. Für die Evaluierung von *EvoTA* wurde ebenfalls ein ABS-Modell verwendet. Als Ergebnis der Fallstudie konnte aufgezeigt werden, dass mit *EvoTA* eine höhere Überdeckung des Testmodells erreicht wurde als mit dem UPPAAL Model Checker und dem Zufallstest. In einer Arbeit von Hemmati *et al.* werden Testfälle von UML-Zustandsautomaten abgeleitet [52]. Der suchbasierte Ansatz zielt darauf ab, mit einer möglichst kleinen Anzahl von Testfällen (Pfaden durch das Testmodell) eine maximale Diversifizierung der Testsuite zu erreichen. Der Diversifizierungsgrad wird basierend auf der Anzahl von identischen Transitionen der Testfälle berechnet. Es wird die An-

nahme getroffen, dass eine hohe Diversifizierung der Testsuite in einer hohen Fehleraufdeckungsrate resultiert [51]. Die Annahme wird durch eine Fallstudie mit einer in C++ implementierten Komponente zur Sicherheitsüberwachung eines Steuergeräts validiert.

Zur besseren Abgrenzung zu den zuvor beschriebenen Ansätzen werden nachfolgend die Neuerungen der entwickelten Verfahren OLET und MbET zusammengefasst.

ABGRENZUNG ZU EXISTIERENDEN VERFAHREN

Der Schwerpunkt dieser Arbeit liegt auf dem evolutionären Funktionstest (Black-Box) von Software eingebetteter Systeme [72, 75]. Kontinuierliche Testdatensequenzen werden generiert, um die internen Systemzustände der Testobjekte zu erreichen [81, 133]. Das OLET-Verfahren unterstützt eine variable Segmentanzahl und Länge der Testdatensequenzen. Die Variabilität der Segmentanzahl wird durch eine veränderliche Chromosomenlänge erreicht (vergleiche Abbildung 3.4).

Zudem wird eine Kombination des evolutionären Funktionstests und des modellbasierten Tests realisiert. Mit dem entstehenden Verfahren, MbET, können komplexe Eingangsdaten durch hybride Automaten beschrieben werden. Der Suchraum wird über veränderliche Parameter in den Automaten aufgespannt. Eine Hürde für die Akzeptanz des evolutionären Funktionstests im industriellen Umfeld ist die in bisherigen Ansätzen fehlende Interaktivität zwischen dem Testobjekt und dem Testtreiber während der Testausführung. Das MbET-Verfahren ergänzt den Testprozess um diese Funktionalität.

Darüber hinaus wird in dieser Arbeit eine neue Herangehensweise an die Beschreibung der Fitnessfunktion und die Berechnung der Fitness präsentiert (MbET) [75]. Ziel ist es, den Einsatz des evolutionären Funktionstest in einem industriellen Umfeld zu vereinfachen und somit die Akzeptanz bei Testern zu steigern, die nicht mit evolutionären Algorithmen vertraut sind. Konkret wird bei dieser Herangehensweise ein Auswerteskript analog zum evolutionären Strukturtest instrumentiert. Das instrumentierte Auswerteskript wird dann mit den Testein- und Testausgaben aufgerufen, mit dem Ziel, Zweige zu überdecken, die ein Fehlverhalten des Testobjekts repräsentieren.

Die in dieser Arbeit vorgestellte Testumgebung zielt auf eine Plattform-unabhängige Herangehensweise für den evolutionären Funktionstest ab [74]. Durch die Trennung der Testplattform (MiL, SiL und HiL) von der evolutionären Engine kann die Fitnessfunktion und die Individuen-/Suchraumspezifikation für unterschiedliche Testplattformen wiederverwendet werden.

Die Herangehensweise an die Durchführung von Fallstudien im Bereich suchbasierter Softwaretests war in der Vergangenheit sehr unterschiedlich, darum ist ein Vergleich der einzelnen Arbeiten nur schwer möglich [2, 34]. In der vorliegenden Arbeit wird besonderer Wert darauf gelegt, die Fallstudien nachvollziehbar und mit anderen Ansätzen vergleichbar zu gestalten. Darüber hinaus werden Richtlinien für die Durchführung von Experimenten im Bereich suchbasiertes Testen vorgeschlagen.

ZUSAMMENFASSUNG

Ausgehend von einer Einführung in die Theorie der evolutionären Algorithmen wurde in diesem Kapitel das Prinzip des evolutionären Testens vorgestellt. Beim evolutionären Testen wird das Problem der Testfallgenerierung in ein Optimierungsproblem transformiert, welches dann mit Hilfe evolutionärer Algorithmen gelöst wird. Der Fokus dieser Arbeit liegt auf dem evolutionären Funktionstest von Software eingebetteter Systeme. Das MbET-Verfahren (Kapitel 5) greift für die Fitnesszuweisung auch auf das Prinzip des evolutionären Strukturtests zurück. Der evolutionäre Funktionstest hat den Zweck, durch automatisch generierte Testfälle Fehlverhalten bei den untersuchten Systemen aufzuzeigen. Der evolutionäre Strukturtest dient der Überdeckung von Programmcode. Aufbauend auf dem in diesem Kapitel beschriebenen Forschungsstand des evolutionären Funktionstests, werden nachfolgend die im Rahmen dieser Arbeit entwickelten Verfahren präsentiert. Diese zielen darauf ab, die beschriebenen Probleme und Nachteile existierender Ansätze zu überwinden.

Teil II

VERFAHREN EVOLUTIONÄRER FUNKTIONSTEST VON SOFTWARE EINGEBETTETER SYSTEME

4 EVOLUTIONÄRES OPEN-LOOP-TESTVERFAHREN

„Das Problem zu erkennen ist wichtiger, als die Lösung zu erkennen, denn die genaue Darstellung des Problems führt zur Lösung."

— Albert Einstein (1879 - 1955)

OLET (Open-loop evolutionärer Test) ist ein Testverfahren mit dem Ziel, Testfälle zu identifizieren, die ein Fehlverhalten einer Software gegenüber einer Spezifikation aufzeigen. Der Schwerpunkt in diesem Kapitel liegt auf der Generierung realistischer Testdatensequenzen (Abschnitt 4.1) und auf der Fitnessfunktion für die Evaluierung der Testfälle (Abschnitt 4.2). Weiterhin werden die Erweiterungen eines klassischen evolutionären Algorithmus für die Realisierung von OLET (Abschnitt 4.3) und der Aufbau und die Hauptbestandteile einer Testumgebung für die Durchführung der Fallstudien beschrieben (Abschnitt 4.4). Unterschieden wird hier zwischen Tests von Modellen und Software in einer Simulationsumgebung und Tests von realen Steuergeräten (vergleiche auch Lindlar und Windisch [74] und Vos *et al.* [118]). Abschließend werden Stärken und Limitierungen von OLET zusammengefasst, die die Entwicklung von MbET motiviert haben (vorgestellt in Kapitel 5).

4.1 SUCHRAUM UND TESTDATENSEQUENZEN

Eine Möglichkeit für die automatisierte Erzeugung einer Testdatensequenz wäre, mehrere voneinander unabhängige Werte

mit evolutionären Algorithmen zu optimieren und aneinanderzureihen. Problematisch ist allerdings, dass dann mit hoher Wahrscheinlichkeit Stimuli für das Testobjekt erzeugt werden, die in der Realität nicht auftreten können. Die Nachvollziehbarkeit bei der Auswertung von Testfällen und das Auffinden einer Fehlerursache wird durch unrealistische Teststimuli erschwert. Wenn beispielsweise ein Eingangskanal eines Testobjekts die Geschwindigkeit eines Fahrzeugs ist, sind zu starke Sprünge physikalisch nicht möglich. Wird jeder Zeitschritt einer Testdatensequenz einzeln bestimmt, muss zudem eine hohe Anzahl von Werten optimiert werden. Eine hohe Anzahl veränderlicher Werte wirkt sich negativ auf die Wahrscheinlichkeit für das Erreichen des Suchziels aus, da der Suchraum dadurch sehr groß wird.

Um die Größe des Suchraums zu begrenzen und gleichzeitig realistische Testdatensequenzen zu generieren, wurde aufbauend auf der Arbeit von Pohlheim *et al.* ein Signalgenerierungsansatz für OLET entwickelt [93]. OLET unterstützt eine variable Anzahl von Segmenten und eine variable Länge der Testdatensequenzen durch die Verwendung genetischer Algorithmen mit Chromosomen variabler Länge. Zusätzlich wurde die Menge der unterstützten Signaltypen erweitert (siehe Anhang B). Ein ähnlicher Ansatz wurde von Windisch und Al Moubayed entwickelt [129]. Windisch und Al Moubayed verwenden Techniken der genetischen Programmierung, um automatisiert Testdatensequenzen für den Strukturtest von Simulink-Modellen und Stateflow-Diagrammen zu erzeugen.

Das Prinzip des Signalgenerierungsverfahrens von OLET besteht darin, dass eine Anzahl von Standardsignalen, zum Beispiel Impulse oder lineare Übergänge, aneinandergereiht werden. Die Aufgabe des Testers ist, die maximale und minimale Anzahl der Segmente und Bereiche für zulässige Parametrierungen festzulegen (oberer Teil in Abbildung 4.1). Zu den Parametrierungen zählen die Signaltypen für die Übergänge zwischen den Segmenten sowie die Ober- und Untergrenzen der Breite und der Amplitude. Für die Länge eines Chromosoms C_k ergibt sich (entsprechend der Anzahl der zu optimierenden Parameter einer Testdatensequenz):

$$l(C_k) = 1 + 3 * n, \tag{4.1}$$

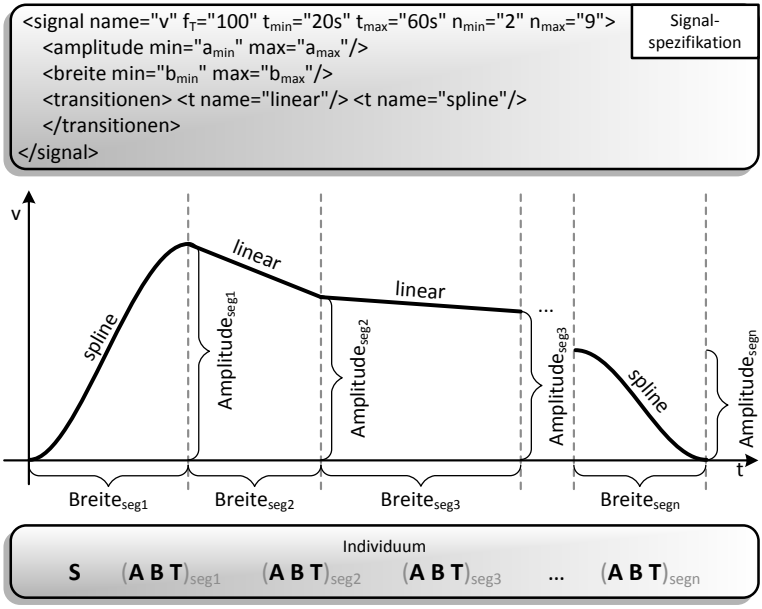


Abbildung 4.1: Mit OLET erzeugtes Signal (Mitte), die dazugehörige Spezifikation (oben) und die Optimierungssequenz (unten) mit S: Startwert, A: Amplitude, B: Breite und T: Signaltyp.

wobei n der Anzahl der Segmente entspricht. Der Wert 1 steht für die initiale Amplitude und der Faktor 3 resultiert daraus, dass in jedem Segment drei Werte, nämlich der Signaltyp, die Breite und die Endamplitude, kodiert sind. Die Abtastrate wird einmalig vom Tester für einen Optimierungslauf festgelegt und die Simulationsdauer für jeden Testfall einmalig ermittelt.

Der Signalgenerierungsansatz unterscheidet zwischen der Optimierungssequenz und der Testdatensequenz. Die Optimierungssequenz umfasst die Menge aller Daten, die in den Chromosomen eines Individuums kodiert sind und von der evolutionären Engine optimiert werden (unterer Teil in Abbildung 4.1). Für eine effiziente Suche muss die Optimierungssequenz so klein wie möglich sein, ohne die Effektivität der Testdatensequenzen zu beeinträchtigen. Eine Testumgebung, die einen derartigen Ansatz realisiert, muss in der Lage sein, die Optimierungssequenz in eine Testdatensequenz zu transformieren (siehe Abschnitt 4.4). Die generierte

Testdatensequenz wird dann für die Ausführung des Testobjekts verwendet. Abbildung 4.1 zeigt ein Beispiel einer mit diesem Ansatz generierten Testdatensequenz. Eine derartige Testdatensequenz könnte beispielsweise die Geschwindigkeit eines Fahrzeugs simulieren. Verwendete Signaltypen für die Übergänge zwischen zwei Segmenten sind *linear* und *spline*. Bei einer Abtastrate von $f_T = 100$ und einer Testlaufzeit von 60 Sekunden müssten ohne den Segmentansatz $60 * 100 = 6.000$ Werte einzeln optimiert werden. Dabei müsste natürlich auch sichergestellt werden, dass die erzeugten Signale nicht die Grenzen des physikalisch Möglichen verletzen (zum Beispiel durch Unstetigkeiten der Signale). Bei dem Segmentansatz ergeben sich nach Gleichung 4.1: $1 + 3 * n$ zu optimierende Parameter oder konkret an dem Beispiel in Abbildung 4.1: $1 + 3 * 9 = 28$ Parameter. Für 9 der 28 Parameter gilt zudem ein Wertebereich mit lediglich zwei möglichen Werten (Signaltyp `linear` oder `spline`), die anderen Parameter sind reelle Zahlen (Amplitude und Breite).

Sobald die Optimierung gestartet wird, erzeugt die evolutionäre Engine Individuen, die konkrete Werte für die spezifizierten Bereiche enthalten. Der Begriff evolutionäre Engine bezieht sich auf die Softwarekomponente, die den evolutionären Kreislauf realisiert (vergleiche Abbildung 3.1). Jedes Individuum kann aus mehreren Chromosomen bestehen und jedes Chromosom repräsentiert eine Testdatensequenz für einen Eingang des Testobjekts. Das Individuum in Abbildung 4.1 besteht aus einem einzigen Chromosom.

Genetische Änderungen an den Chromosomen werden durch Mutation und Rekombination herbeigeführt. Bei der Mutation wird ein zufälliger (kleiner) Wert auf einen oder mehrere der Parameter der Chromosomen addiert. Bei der Rekombination wird ein Chromosom an einer zufällig gewählten Stelle abgeschnitten und mit dem hinteren Teil eines ebenfalls geteilten Chromosoms von einem anderen Individuum zusammengefügt (vergleiche Abbildung 4.2). Wurde eine Grenze für die maximale Anzahl von Segmenten spezifiziert, werden Segmente über die maximale Anzahl hinaus automatisch abgeschnitten. Im Anschluss an die genetischen Anpassungen der Individuen wird der Signalverlauf aus der Signalspezifikation und den Chromosomen abgeleitet. Als letzter Schritt werden für jedes Individuum alle Sequenzen auf eine einheitliche Testdauer skaliert. Dadurch wird gewährleistet, dass alle Eingangskanäle über die gesamte Simulationsdauer belegt sind.

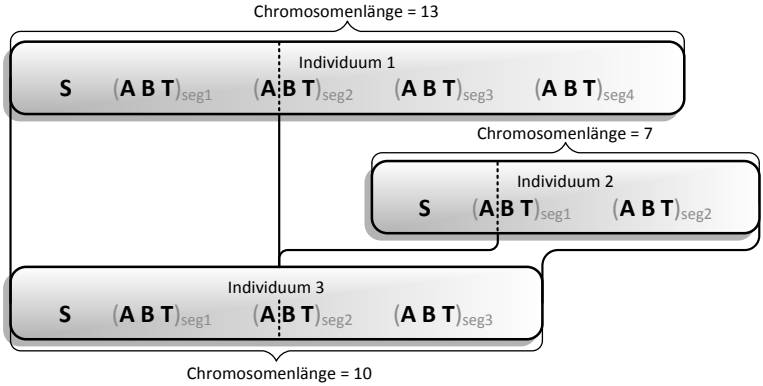


Abbildung 4.2: Rekombination zweier Individuen mit OLET.

Für die Transformation der Optimierungssequenz in die Testdatensequenzen muss sichergestellt werden, dass keine ungültigen Teststimuli erzeugt werden. Bei der Rekombination beeinflusst der Schnittpunkt des ersten Chromosoms den des Zweiten. Wird das erste Chromosom nach dem Amplitudenwert abgeschnitten (vergleiche Abbildung 4.2), muss das zweite Chromosom auch nach dem Amplitudenwert abgeschnitten werden (bei einem zufällig ausgewählten Segment). Anderenfalls wäre das Individuum ungültig, weil das letzte Segment nicht vollständig definiert ist. Zudem darf durch die Mutation der Individuen keine spezifizierte Grenze der Parameter verletzt werden. Führt ein Mutationsschritt zum Über- oder Unterschreiten einer Ober- oder Untergrenze, wird dem Gen der spezifizierte Grenzwert zugewiesen.

Anhang B stellt eine Auswahl möglicher Signaltypen für die Übergänge zwischen den Segmenten vor. Außerdem werden die Formeln aufgeführt, mit denen die Testumgebung die Übergänge interpoliert. Weitere Signaltypen sind denkbar – für die Durchführung von Fallstudien im Rahmen dieser Arbeit hat sich die vorgestellte Menge als ausreichend erwiesen.

4.2 FITNESSFUNKTION

Ziel des evolutionären Funktionstests ist, einen Testfall zu finden, der eine Anforderungsverletzung aufzeigt. Der Testfall demonstriert dann anhand eines konkreten Szenarios (Gegenbeispiel), dass die in der Anforderung spezifizierten Bedingungen nicht immer erfüllt sind. Als erster Schritt für die Durchführung eines evolutionären Funktionstests muss eine geeignete Anforderung ausgewählt werden. Geeignet bedeutet in diesem Zusammenhang, dass die Anforderung durch eine numerische Funktion, die sogenannte Fitnessfunktion, beschrieben werden kann. Die Aufgabe der Fitnessfunktion ist es, zu bewerten, wie weit ein Testfall vom Erreichen des angestrebten Testziels entfernt ist. Beim evolutionären Strukturtest (Fitness kann vom Kontrollfluss abgeleitet werden) und dem evolutionären Zeitverhaltenstest (Fitness leitet sich von der Ausführungszeit ab) kann die Fitnessfunktion automatisch erstellt werden. Liegen beim evolutionären Funktionstest keine formalisierten Anforderungen vor, muss dagegen für jedes Suchziel manuell eine Fitnessfunktion implementiert werden. Für die Erstellung einer solchen Fitnessfunktion müssen die Ausgänge des Testobjekts analysiert und in Zusammenhang mit der untersuchten Anforderung gebracht werden. In Anforderungen werden häufig zulässige Oberschranken für das Ausgangsverhalten eines Systems festgelegt. Die Suche lässt sich dann in Richtung der Verletzung der Schranken und somit der Verletzung der Anforderung steuern. Bezieht sich eine Anforderung auf einen Systemausgang, der lediglich wenige Werte annehmen kann (beispielsweise ein binäres Ausgangssignal), ist die Steuerung der Suche nicht möglich.

Die Fitnessfunktion für OLET wird als mathematische Funktion realisiert (vergleiche Abbildung 4.3). Als Eingabe stehen sowohl die während der Ausführung des Testobjekts aufgezeichneten Testdaten als auch die Testeingaben (Testdatensequenzen) zur Verfügung. Um die Qualität zweier Individuen im Hinblick auf das Suchziel zu vergleichen, müssen die Testdatensequenzen mit Hilfe der Fitnessfunktion in einen atomaren Wert transformiert werden. Je nach Problemstellung besteht eine mögliche Herangehensweise darin, die *Fitnessfunktion* auf jeden Zeitschritt eines Testfalls anzuwenden und dann den kleinsten oder größten errechneten Wert als *Fitnesswert* zuzuweisen. Alternativ kann, abhängig von der Problemstellung, auch das Integral oder die Ableitung, sowohl

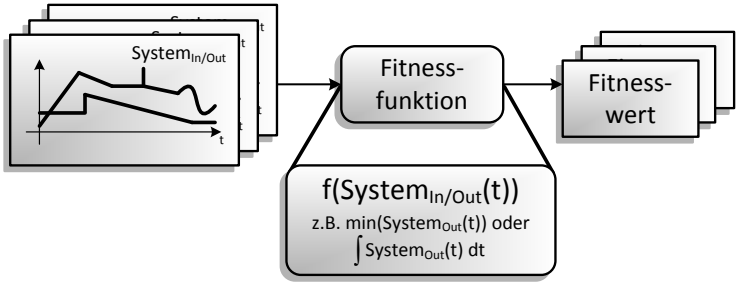


Abbildung 4.3: Berechnung atomarer *Fitnesswerte* aus kontinuierlichen Datensequenzen mit System$_{In}$: Eingaben für das Testobjekt (Testdatensequenzen) und System$_{Out}$: Ausgaben des Testobjekts.

der Testdaten als auch von den mit der *Fitnessfunktion* errechneten Werten, gebildet werden. Für die Fallstudien in Kapitel 6 wurde jeweils der kleinste mit einer Fitnessfunktion errechnete Fitnesswert in den evolutionären Kreislauf zurückgeführt.

Insbesondere für die Klasse der eingebetteten Systeme kann die Entwicklung einer geeigneten Fitnessfunktion eine aufwändige Voraussetzung für den evolutionären Funktionstest sein. Der Aufwand wird aber durch die automatisierte Suche und die Möglichkeit zur Ausführung einer hohen Anzahl von zielgerichteten Testfällen kompensiert. Im einfachsten Fall errechnet sich die Fitness aus dem kleinsten Abstand eines Ausgangskanals zum Überschreiten eines Schwellenwerts. Häufig ist aber eher das Zusammenspiel von Eingangs- und Ausgangsverhalten sowie das Verhältnis zwischen den Signalen ausschlaggebend. Da die zu spezifizierende Fitnessfunktion sehr stark vom Testobjekt und der untersuchten Anforderung abhängig ist, wird für eine weitere Veranschaulichung auf die vom Autor durchgeführten Fallstudien verwiesen (Abschnitte 6.2 und 6.3).

4.3 Optimierungskreislauf

Im Folgenden wird betrachtet, welche Anpassungen mit den vorgestellten Neuerungen für die Suche mit evolutionären Algorith-

men nötig sind. Die Anpassungen beschränken sich auf den Rekombinationsoperator (siehe Abschnitt 4.1) und die Evaluierung von Testfällen – der generelle Ablauf des evolutionären Kreislaufs kann unverändert bleiben (vergleiche Abbildung 3.1).

Die *Evaluierung* von Testfällen dient der Zuweisung von Fitnesswerten zu Individuen. Bei OLET lässt sich die *Evaluierung* in vier Schritte unterteilen:

1. Jedes Individuum besteht aus einem oder mehreren Chromosomen. Jedes Chromosom besteht wiederum aus einer Anzahl von Werten, die die Eigenschaften einer Testdatensequenz beschreiben. Als erster Schritt bei der Evaluierung erzeugt die Testumgebung mit den Werten aus den Chromosomen und der Signalspezifikation die Testdatensequenzen (vergleiche Abbildung 4.1).

2. Das Testobjekt wird dann mit den Testdatensequenzen ausgeführt und die Ausgabewerte werden aufgezeichnet. Es ergibt sich folglich eine Menge von Testdatensätzen, bestehend aus den Werten der Teststimuli und der Testausgaben.

3. Mit der *Fitnessfunktion* wird für jeden Testdatensatz ein Fitnesswert berechnet, der dem Abstand zum *Brechen* einer Anforderung entspricht (vergleiche Abbildung 4.3).

4. Die *Fitnesswerte* fließen zurück in den Optimierungskreislauf und werden als Gewichte für die Operatoren des evolutionären Algorithmus verwendet (vergleiche Abbildung 3.9).

Die Suche wird solange vollautomatisch fortgeführt, bis ein Abbruchkriterium erfüllt wird – bei einem erfolgreichen Testlauf ein Fitnesswert, der dem Testziel entspricht [59]. Wird ein Testfall identifiziert, der eine Anforderungsverletztung aufzeigt, muss der Tester den Testfall untersuchen und manuell die für die Beseitigung des Fehlers notwendigen Anpassungen an der Software vornehmen. Sobald der zuvor identifizierte Testfall keine Anforderungsverletzung mehr hervorruft, kann die Suche erneut gestartet werden. Werden keine weiteren Anforderungsverletzungen identifiziert, stellt das Ergebnis einen Vertrauensgewinn in die korrekte Funktionalität der Software dar.

4.4 TESTUMGEBUNG

Mit der OLET- und der MbET-Testumgebung (Abschnitt 5.4) wurden Fallstudien durchgeführt (Kapitel 6). Eine Hauptkomponente für die Realisierung des Optimierungskreislaufs war das im EvoTest-Förderprojekt entwickelte EvoTest-Framework (Abschnitt 4.4.1). In den nächsten beiden Abschnitten werden Erweiterungen des Frameworks für die Durchführung von Tests mit OLET beschrieben. Unterschieden wird hierbei zwischen Modelltests beziehungsweise Model-in-the-Loop (MiL)-Tests und Softwaretests beziehungsweise Software-in-the-Loop (SiL)-Tests (Abschnitt 4.4.2) sowie den Tests mit einem realen Steuergerät, den Hardware-in-the-Loop (HiL)-Tests (Abschnitt 4.4.3). Die Bezeichnung *in-the-loop* bezieht sich auf die Kommunikation zwischen Testobjekt und Umgebungssimulation während der Testausführung, nicht aber auf die Kommunikation zwischen Testumgebung und Testobjekt beziehungsweise Umgebungssimulation. Eine Einschränkung von OLET ist, dass kein reaktives Testen ermöglicht wird. Testdatensequenzen stehen bereits vor der Testausführung fest und können während der Ausführung nicht mehr angepasst werden.

4.4.1 EVOTEST-FRAMEWORK

Das EvoTest-Framework (ETF) stellt Komponenten und Schnittstellen zur Verfügung, um die Generierung, Ausführung, Überwachung und Auswertung von Testfällen mit evolutionären Algorithmen zu automatisieren. Es wurde speziell für den Test von großen und komplexen Softwaresystemen entwickelt. Das ETF kann für das Erreichen beliebiger Testziele verwendet werden. Im Rahmen des EvoTest-Projekts [36] wurde die Überdeckung von Programmcode und der Funktionstest mit dem ETF untersucht [43, 130].

Implementiert wurde das ETF als Erweiterung der Entwicklungsumgebung Eclipse [109]. Kern des Frameworks stellt die evolutionäre Engine dar, welche von GUIDE (Graphical User Interface for DREAM Experiments) unter Verwendung der Evolving Objects Library generiert wurde [45, 65]. GUIDE stellt darüber hinaus eine grafische Benutzeroberfläche zur Verfügung, mit der die evolutionäre Engine konfiguriert werden kann (vergleiche Tabelle 3.1).

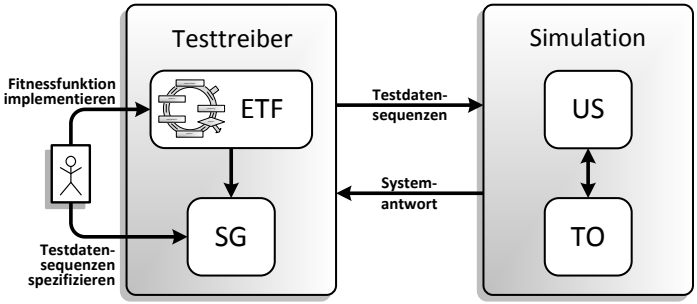


Abbildung 4.4: Model-in-the-Loop Architektur mit ETF: EvoTest-Framework, SG: Signalgenerator, US: Umgebungssimulation und TO: Testobjekt.

Die Evolving Objects Library enthält alle Klassen, die erforderlich sind, um mit Hilfe der konfigurierten Parameter die evolutionäre Engine zu generieren. Das ETF ist nicht auf die Verwendung von GUIDE und der Evolving Objects Bibliothek beschränkt. Durch entsprechende Bibliotheken kann das ETF beispielsweise auch für lokale Suchverfahren wie Hill-Climbing oder Simulated Annealing erweitert werden. Eine ausführliche Beschreibung des EvoTest-Frameworks wurde in einer Arbeit von Dimitrov *et al.* vorgestellt [33].

4.4.2 TESTUMGEBUNG FÜR MIL- UND SIL-TESTS

Voraussetzung für die Durchführung von MiL-Tests ist ein ausführbares Modell. Die Durchführung eines evolutionären Funktionstests in einer MiL-Testumgebung kann auf einem einzigen Computer realisiert werden. Insbesondere im industriellen Kontext empfiehlt es sich bei komplexen Testobjekten, die Simulation aufgrund der hohen Ausführungszeit auf mehreren Computern zu parallelisieren oder zumindest auf mehrere CPU Kerne zu verteilen.

Abbildung 4.4 zeigt eine Übersicht über die an der evolutionären Suche beteiligten Komponenten. Das ETF realisiert einen evolutionären Algorithmus und steuert die Suche. Bevor die Individuen an die *Simulationskomponente* weitergereicht werden können,

müssen sie von einem *Signalgenerator* in kontinuierliche Testdatensequenzen transformiert werden (vergleiche Abschnitt 4.1). Die Testdatensequenzen werden dann für die Ausführung des *Testobjekts* verwendet. Je nach Anwendungsfall gibt es zusätzlich einen Modellteil, der für die *Umgebungssimulation* zuständig ist und direkt mit dem *Testobjekt* kommuniziert.

Für die in dieser Arbeit durchgeführten MiL-Experimente wurde die Simulationsumgebung Matlab/Simulink verwendet (Abschnitte 6.2.4 und 6.3.4). Dabei ist sowohl das Testobjekt als auch die Umgebungssimulation in Form von Simulink-Modellen und Stateflow-Diagrammen realisiert [110–112]. Zusätzlich zu der MiL-Simulation der Testfälle kann auch aus den Modellen generierter Programmcode in einer SiL-Simulation ausgeführt werden. Die Durchführung von SiL-Tests als Ergänzung zu MiL-Tests ist insbesondere dann sinnvoll, wenn nicht sichergestellt werden kann, dass das Verhalten des aus dem Modell generierten Programmcodes identisch zu dem des Modells ist.

4.4.3 Testumgebung für Steuergeräte-Tests

Im Anschluss an die Durchführung von MiL- und SiL-Tests ist der nächste Schritt im Entwicklungsprozess ein Test mit einem realen Steuergerät – der HiL-Test (siehe Abbildung 2.1). Dafür wird aus dem Modell ausführbarer Programmcode erzeugt und zusammen mit Treibern für die Ansteuerung der Hardware auf das Steuergerät übertragen. Die Treiber für die Hardwareansteuerung werden zumeist nicht modellbasiert entwickelt.

Zu den Vorteilen der Durchführung von HiL- im Vergleich zu MiL-Tests zählt die Berücksichtigung hardwarespezifischer Eigenschaften der Steuergeräte. Um das Zusammenspiel zwischen Hardware und Software beziehungsweise das Zeitverhalten unter Echtzeitbedingungen zu überprüfen, ist ein HiL-Test einem MiL-Test vorzuziehen. Nachteilig sind allerdings die hohen Anschaffungskosten eines HiL-Prüfstands verbunden mit der hohen Anzahl erforderlicher Ausführungen in Echtzeit bei einem evolutionären Funktionstest. Der evolutionäre Funktionstest an einem HiL-Prüfstand wird aber noch durch einen weiteren entscheidenden Aspekt motiviert. Insbesondere im Automobilbereich ist es üblich, dass Steuergeräte zum Teil von Zulieferern hergestellt

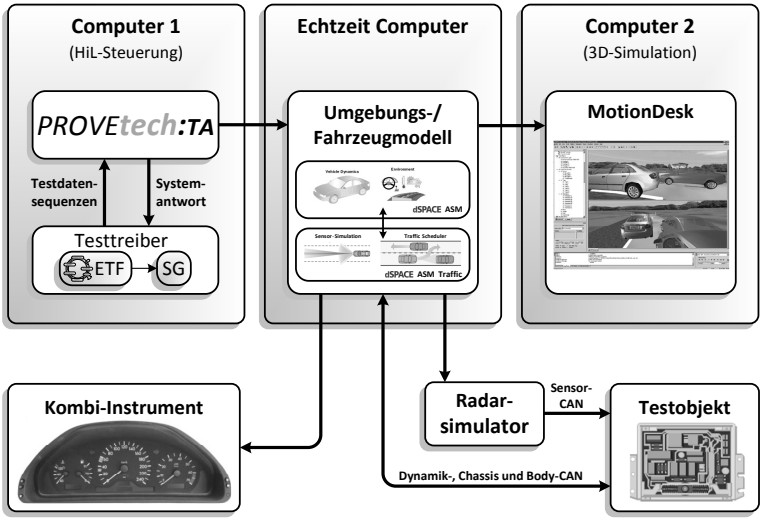


Abbildung 4.5: Hardware-in-the-Loop Architektur für den Test von Steuergeräten mit ETF: EvoTest-Framework und SG: Signalgenerator.

werden und nur der Zulieferer Zugriff auf das Modell hat. Deshalb kann ein MiL-Test folglich nicht beim Fahrzeughersteller durchgeführt werden und es kann einzig auf einen HiL-Test zurückgegriffen werden. In dem Fall, dass sich das Steuergerät nicht gemäß der Spezifikation verhält, muss der Fahrzeughersteller für einen Teil der möglichen Schäden haften. Deshalb ist es im Interesse des Fahrzeugherstellers, die Funktionalität mit einem effektiven Testansatz selbst zu validieren.

Abbildung 4.5 zeigt den Aufbau der HiL-Testumgebung. Mit Hilfe dieses Testsystems wurde die Fallstudie in Abschnitt 6.2.5 mit einem Steuergerät aus der Serienentwicklung durchgeführt. Insgesamt besteht das Testsystem aus drei Computern: *Computer 1* verfügt über eine Kopplung zwischen den Komponenten für die Realisierung eines evolutionären Funktionstests und dem Tool *ProveTech:TA* für die Ansteuerung des *Echtzeit Computers* [77]. *ProveTech:TA* fungiert als Plattform-Adapter und hat einzig die Funktion, die generierten Teststimuli an den *Echtzeit Computer* weiterzuleiten. *Computer 2* visualisiert den aktuellen Testverlauf in einer 3D-Simulation.

Der *Echtzeit Computer* verfügt über drei *CAN*-Anschlüsse für die Kommunikation mit dem *Kombi-Instrument*, dem *Radarsimulator* und dem *Testobjekt*. *CAN* steht für *Controller Area Network* und ist ein asynchrones, serielles Bussystem, das vor allem im Automobilbereich verwendet wird. Die Komponente *Radarsimulator* imitiert aus der Sicht des *Testobjekts* ein reales Radar. Das *Kombi-Instrument* zeigt Geschwindigkeit, Drehzahl und Warnhinweise für den Fahrer an. Mit den durch das Zusammenspiel aus *ETF* und *Signalgenerator* erzeugten Testdatensequenzen wird auf dem *Echtzeit Computer* ein *Umgebungs*- und ein *Fahrzeugmodell* simuliert. Basierend auf den Ausgaben von *Umgebungs*- und *Fahrzeugmodell* werden die für das *Testobjekt* relevanten Daten auf die *CAN*-Busse geschrieben. Das *Testobjekt* gibt dann die Reaktion auf die Eingaben durch den *CAN*-Bus zurück. Alle für die Berechnung der Fitnesswerte relevanten Testdaten werden schließlich über den *Echtzeit Computer* und die *ProveTech:TA*-Komponente an die evolutionäre Engine übermittelt. Eine ausführliche Beschreibung des HiL-Testsystems sowie eine damit durchgeführte Fallstudie wurde bereits in einer Veröffentlichung des Autors vorgestellt [74].

ZUSAMMENFASSUNG

Funktionstests von Software eingebetteter Systeme können mit OLET automatisiert durchgeführt werden. Die Aufgabe des Testers beschränkt sich auf die Spezifikation der Teststimuli, die Implementierung einer Fitnessfunktion sowie geringe Anpassungen des Testtreibers. Realistische Testdatensequenzen werden durch die Aneinanderreihung von Standardsignalen erzeugt. Mit Hilfe eines genetischen Algorithmus mit Chromosomen variabler Länge werden die Segmentanzahl und die zeitliche Länge der Teststimuli variiert. Die Fitnessfunktion dient der Quantifizierung der Qualität der Testfälle im Hinblick auf das Erreichen der Testziele und somit der Steuerung der Suche. Weiterhin wurde die OLET-Testumgebung beschrieben, die eine Testplattform-übergreifende Evaluation von Testfällen in einer MiL-, SiL- und auch in einer HiL-Umgebung ermöglicht. Signalspezifikation und Fitnessfunktion müssen dafür nicht angepasst werden. Im nächsten Kapitel wird ein alternatives Verfahren vorgestellt, das mit Hilfe hybrider Automaten ein reaktives Testverfahren realisiert.

5 MODELLBASIERTES EVOLUTIONÄRES TESTVERFAHREN

„Solange der Mensch Software schreibt, wird die Software mit Fehlern behaftet sein, unabhängig davon, in welcher Sprache bzw. auf welcher semantischen Ebene er seine Gedanken zum Ausdruck bringt."

— Abraham Maslow (*1940)

Bei MbET (Modellbasierter evolutionärer Test) werden Testdatensequenzen mit Hilfe hybrider Automaten generiert (vergleiche Lindlar *et al.* [75]). Durch die Verwendung hybrider Automaten werden reaktive Tests von Software eingebetteter Systeme ermöglicht (Abschnitt 5.1). MbET beinhaltet auch eine neue Herangehensweise an die Erstellung einer Fitnessfunktion für den evolutionären Funktionstest (Abschnitt 5.2). Die Bewertung von Testläufen zur Überprüfung funktionaler Anforderungen erfolgt mit Hilfe des evolutionären Strukturtests [119]. Für das modellbasierte Verfahren und die neue Fitnessfunktion sind Erweiterungen des evolutionären Algorithmus erforderlich (Abschnitt 5.3). In Abschnitt 5.4 wird die Testumgebung für die *closed-loop*-Fallstudien vorgestellt. Schließlich wird MbET mit dem *open-loop*-Verfahren OLET verglichen.

5.1 SUCHRAUM UND TESTDATENSEQUENZEN

Der in Abschnitt 4.1 vorgestellte Signalgenerierungsansatz für den *open-loop*-Test (OLET) basiert auf der Beschreibung von Testdatensequenzen mit Hilfe der in Abschnitt 4.1 beschriebenen

hierarchischen Auszeichnungssprache. Die Variabilität für die evolutionäre Suche wird durch die Vorgabe von Wertebereichen für Breite und Amplitude einzelner Signalabschnitte erreicht. Weiterhin können Signaltypen für die Übergänge zwischen den Abschnitten ausgewählt werden. Teststimuli werden bei MbET mit Hilfe erweiterter hybrider Automaten spezifiziert (vorgestellt in Abschnitt 2.5.2). Jeder Automat dient der Beschreibung eines oder mehrerer Signalverläufe, die für die Ausführung des Testobjekts verwendet werden. Entsprechend der Zustände der Automaten sind die Teststimuli in einzelne Phasen eingeteilt. Der Verlauf der Testdatensequenzen kann in Abhängigkeit des Ausgangsverhaltens des Testobjekts angepasst werden. Beispielsweise könnte der erste Zustand eines Automaten für die Initialisierung eines Geräts zuständig sein und der darauf folgende simuliert den Betrieb. Der Zustandswechsel erfolgt, sobald das Testobjekt den Abschluss der Initialisierung signalisiert hat.

Die Variabilität im Suchraum für die evolutionäre Suche wird durch die Optimierung von Parametern im Testmodell realisiert. Die Spezifikation veränderlicher Testdatensequenzen für ein konkretes Problem erfolgt in zwei Schritten:

1. Der Tester spezifiziert das Testmodell mit Hilfe hybrider Automaten.
2. Der Tester kennzeichnet variable Parameter im Testmodell und legt Wertebereiche für die Parameter fest.

Die Wertebereiche entsprechen dem Suchraum des evolutionären Algorithmus. Bei der Festlegung von Wertebereichen muss einerseits beachtet werden, dass der Suchraum nicht zu groß wird, da sonst die Suche sehr lange dauern kann. Andererseits dürfen relevante Bereiche auch nicht von der Suche ausgeschlossen werden. Bei der Erstellung des Testmodells kann der Tester einzelne Zustände durch weitere Zustände hierarchisch verfeinern. Außerdem kann bei komplexen Testobjekten, mit einer hohen Anzahl von Eingängen, eine beliebige Anzahl von parallelen Automaten modelliert werden. Kontinuierliches Verhalten wird innerhalb der Zustände durch Gleichungen beschrieben. Jede Gleichung ist einem Eingangskanal des Testobjekts zugeordnet. Variablen in den Gleichungen sind entweder:

1. Von der evolutionären Engine errechnete Parameter beziehungsweise Gene aus der Optimierungssequenz,

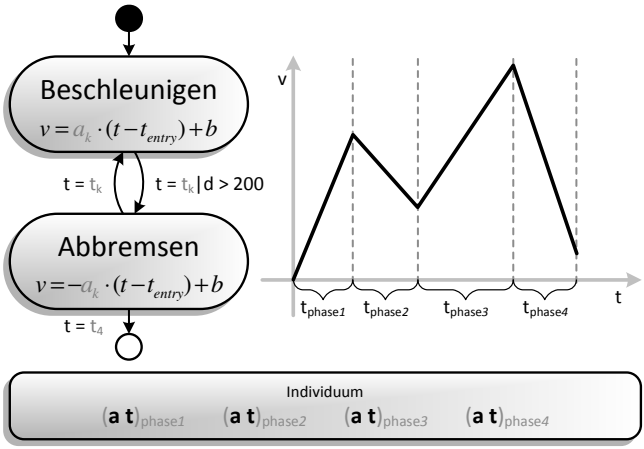


Abbildung 5.1: Mit MbET erzeugte Testdatensequenz (rechts) und die dazugehörige Spezifikation (links). Testmodell-Parameter (*grau*) werden variiert, zum Beispiel $a_k \in [0;5]$ und $t_k \in [0;60]$. Der untere Teil zeigt die Optimierungssequenz.

2. die kontinuierliche Zeit (wird für jeden Testlauf auf 0 zurückgesetzt) oder
3. ein rückgekoppelter Ausgangskanal des Testobjekts.

Über den gesamten Testverlauf muss sichergestellt werden, dass Daten durchgängig für jeden Eingangskanal des Testobjekts festgelegt sind. Anderenfalls muss von der Testumgebung eine Warnung ausgegeben werden.

Abbildung 5.1 zeigt ein einfaches Beispiel eines MbET-Testmodells. Das Testobjekt hat einen Eingangskanal v (zum Beispiel für die Simulation der Geschwindigkeit eines Fahrzeugs). Der Parameter b im Testmodell wird mit 0 initialisiert und beim Zustandswechsel automatisch auf den letzten Wert von v gesetzt, um Unstetigkeiten zu vermeiden. t_{entry} bezeichnet den Zeitpunkt, zu dem der jeweilige Zustand aktiviert wurde. Auf der rechten Seite ist eine generierte Testdatensequenz mit einer möglichen Belegung der Modellparameter dargestellt. Durch Änderungen der Parameter ändert sich der Verlauf der Testdatensequenz. Die Steigungen ändern sich, wenn also die Parameter a_k verändert und die Übergangszeitpunkte zwischen *Beschleunigen* und *Abbremsen*, wenn die

Zeitpunkte t_k verändert werden. Der Übergang von *Beschleunigen* nach *Abbremsen* wird zusätzlich durch den Parameter d beeinflusst. d ist ein Ausgangskanal des Testobjekts und beschreibt den Abstand zu einem weiteren hinterherfahrenden Fahrzeug. Um den Abstand zwischen den Fahrzeugen nicht zu groß werden zu lassen, wird dieser automatisch ab einer Entfernung von $200m$ verringert (Reaktivität). Die Anzahl der zu optimierenden Parameter hängt in diesem Beispiel von der Anzahl der Übergänge zwischen den Zuständen ab. Für jeden Zustandswechsel k muss der Zeitpunkt t_k sowie die Steigung a_k vom evolutionären Algorithmus bestimmt werden. Der Vorteil von MbET gegenüber OLET ist, dass für beide Zustände unterschiedliche Vorgaben für die maximal zulässige Steigung spezifiziert werden können. Soll die Geschwindigkeit eines Fahrzeugs simuliert werden, ist der Betrag des Maximums der negativen Beschleunigung ($-8m/s^2$) größer als der Maximalwert der positiven Beschleunigung ($5m/s^2$). Der untere Teil in Abbildung 5.1 zeigt die Optimierungssequenz. Das Individuum besteht aus einem einzigen Chromosom mit 8 Genen (Parametern für das Testmodell). MbET sieht vor, dass veränderliche Parameter eines Zustandsautomatens jeweils in einem Chromosom zusammengefasst werden. Bei einem Automaten, der sich auf der obersten Hierarchieebene aus mehreren parallelen Zustandsautomaten zusammensetzt, würde das Individuum automatisch um entsprechende Chromosomen erweitert werden.

MbET sieht außerdem die Verwendung von Template-Zuständen vor. Template-Zustände sind vorgefertigte, parametrierbare Zustände, die die Spezifikation von Testmodellen bei häufig wiederkehrenden Problemen erleichtern. Auf ein spezielles Anwendungsfeld zugeschnitten, kann die Effizienz des Testprozesses durch die Verwendung von Template-Zuständen erhöht werden. Abbildung 5.2 veranschaulicht dieses Konzept anhand von zwei Template-Zuständen. Auf der linken Seite ist der *Alternieren*-Zustand und auf der rechten Seite der *Impuls*-Zustand dargestellt. Der untere Teil zeigt die Optimierungssequenzen und mit den Templates erzeugte Signalverläufe. Der *Alternieren*-Zustand wechselt zwischen ansteigendem und abfallendem Signalverlauf (zum Beispiel für die Simulation eines abwechselnd beschleunigenden und abbremsenden Fahrzeugs). Der *Impuls*-Zustand erzeugt Impulse zu wechselnden Zeitpunkten, bei Bedarf können sich auch die Breite und die Amplitude ändern (zum Beispiel für die Simulation einer Schalterbetätigung). Der Tester muss für die Verwendung dieser

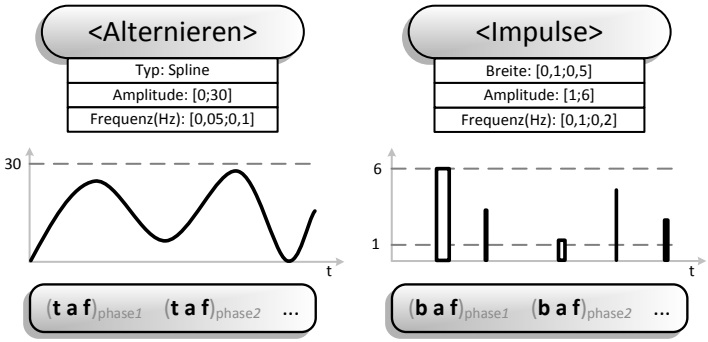


Abbildung 5.2: MbET Template-Zustände: Der *Alternieren*- und der *Impuls*-Zustand. Konfigurierbar sind *Frequenzen* und *Amplitudenbereiche* sowie der *Typ* (*Alternieren*-Zustand) und die *Breite* (*Impuls*-Zustand).

Zustände lediglich die in der Tabelle angegebenen Parameter festlegen, die Testumgebung erzeugt automatisch die Optimierungs- und Testdatensequenzen.

In Abschnitt 4.1 wurde beschrieben, wie die Länge der Chromosomen durch die Rekombination variiert werden kann. Bei MbET findet diese Vorgehensweise bei den Template-Zuständen Anwendung. Wenn mit dem *Impuls*-Zustand aus Abbildung 5.2 beispielsweise eine Sequenz der Länge 60s erzeugt werden soll, würden bei einer Frequenz von $0,1$ jede zehnte und bei einer Frequenz von $0,2$ jede fünfte Sekunde ein Impuls erzeugt werden. Daraus ergeben sich dann für Breite, Amplitude und Auslösungszeitpunkte der Impulse zwischen $3 \cdot 6 = 18$ und $3 \cdot 12 = 36$ zu optimierende Parameter. Die Aufgabe des Anwenders von MbET beschränkt sich dabei auf die Spezifikation des Frequenzbereichs. Die Testumgebung erzeugt automatisch eine variable Anzahl von Parametern in Abhängigkeit der Frequenz. Die Variation der Frequenz wird durch die Variation der Anzahl der Übergänge durch den Rekombinationsoperator erreicht (siehe Abbildung 4.2).

Die Optimierungssequenz ist auch bei MbET sehr viel kleiner als die Testdatensequenz. Es müssen nur wenige Parameter im Testmodell variiert werden, um die Eigenschaften der Testdatensequenzen signifikant zu verändern. In Abhängigkeit der aktiven Zustände werden Testdaten erzeugt, ohne dass jeder Zeitschritt einzeln optimiert werden muss.

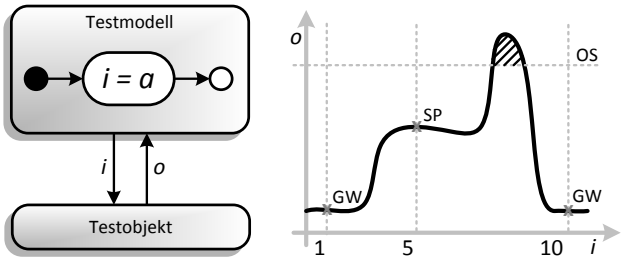


Abbildung 5.3: Äquivalenzklassentests mit der Klasse $\ddot{A}K_i = [1, 10]$, SP: Stichprobe, GW: Grenzwert und OS: Oberschranke.

Schließlich findet noch ein weiterer Aspekt bei der Spezifikation der Teststimuli mit MbET Berücksichtigung: Mit MbET kann überprüft werden, ob Äquivalenzklassen für Teststimuli korrekt gewählt sind [73]. Bei der manuellen Testauswahl werden in der Regel nur die Grenzen der Äquivalenzklassen sowie vereinzelte Stichproben getestet. Wird eine Äquivalenzklasse nicht korrekt gewählt, besteht die Gefahr, dass Fehler übersehen werden. Abbildung 5.3 zeigt ein einfaches Beispiel: Für die erwartete Äquivalenzklasse $\ddot{A}K_i = [1, 10]$ werden Testläufe mit den Grenzwerten 1 und 10 (*GW*) sowie der Stichprobe 5 (*SP*) durchgeführt. Auf der rechten Seite sind die Ausgaben des Testobjekts in Abhängigkeit der Eingaben dargestellt. Obwohl keiner der drei Testfälle fehlschlägt, verhält sich das Testobjekt nicht für alle Werte aus der erwarteten Äquivalenzklasse korrekt (Überschreitung von *OS*). Mit MbET kann ein Fehlverhalten weitestgehend ausgeschlossen werden. Es wird nur ein einziger Suchlauf für die Äquivalenzklasse $\ddot{A}K_i = [1, 10]$ spezifiziert und die Suche automatisch in Richtung der Verletzung der Schranke gesteuert. Positiver Seiteneffekt ist eine geringere Anzahl von Testfällen, die spezifiziert werden muss.

5.2 STRUKTURTEST AUF AUSWERTESKRIPTEN

Die Aufgabe der Fitnessfunktion ist es, Individuen im Hinblick auf das Erreichen des Testziels zu bewerten, um deren Vergleichbarkeit zu ermöglichen. Im Unterschied zu der in Abschnitt 4.2

beschriebenen Herangehensweise an die Erstellung der Fitnessfunktion basiert das Grundprinzip der Fitnessberechnung des MbET-Verfahrens auf dem evolutionären Strukturtest (vorgestellt in Abschnitt 3.3). Die Aufgabe des Testers ist es, mit Hilfe einer Skriptsprache (idealerweise domänenspezifisch, zum Beispiel für Fahrerassistenzsysteme), die Grenzen zulässigen Verhaltens für die Ausgänge des zu testenden Systems zu beschreiben. Als Ausgangspunkt dienen die funktionalen Anforderungen aus der Systemspezifikation. Die Herangehensweise, die Grenzen des zulässigen Systemverhaltens mit einer Skriptsprache zu beschreiben, ist an die Methodologie des TPT angelehnt (vergleiche Abschnitt 2.5.3). Das resultierende *Auswerteskript* dient bei TPT allerdings nur dazu, ein Urteil über einen durchgeführten Testlauf zu fällen (Test bestanden oder nicht). Bei MbET wird das Auswerteskript von der Testumgebung analog zum evolutionären Strukturtest instrumentiert. Das Resultat, ein *instrumentiertes Auswerteskript*, berechnet beim Aufruf mit den Testein- und Testausgaben den Abstand zum Erreichen eines Testziels. Folglich wird eine Aussage darüber möglich, welches von zwei Individuen näher am jeweiligen Testziel ist.

Abbildung 5.4 zeigt ein vereinfachtes Beispiel eines Auswerteskripts für den Test eines Abstandsregeltempomats (linke Seite). Testdatensequenzen, die während der Ausführung eines Testfalls aufgezeichnet wurden, werden mit dem Auswerteskript hinsichtlich des Unterschreitens spezifizierter Schranken bewertet. Konkret gilt ein Testfall als fehlgeschlagen, falls zu einem Zeitpunkt im Testverlauf $v_{rel}(t)$ größer und $a(t)$ kleiner als Null sind, beziehungsweise $v_{rel}(t)$ kleiner als Null ist während $a(t)$ größer als Null und $d(t)$ kleiner als Eins sind. Der Abstandsregeltempomat muss folglich bei positiver Relativgeschwindigkeit beschleunigen und bei negativer Relativgeschwindigkeit abbremsen. Außerdem muss der Abstand zum vorausfahrenden Fahrzeug größer als $1\,m$ sein. Der erste Schritt für die Durchführung eines evolutionären Strukturtests ist, einen Kontrollflussgraphen aus dem Programmcode abzuleiten (rechte Seite in Abbildung 5.4). Die Blätter im Kontrollflussgraphen repräsentieren Testergebnisse. Als nächster Schritt muss ein Zielknoten (Testziel) ausgewählt werden, der ein negatives Testergebnis repräsentiert. Aus dem Kontrollflussgraph wird dann automatisch ein instrumentiertes Auswerteskript generiert, das den normierten Abstand zum ausgewählten Testziel berechnet (rechte Seite in Abbildung 5.5). Das instrumentierte

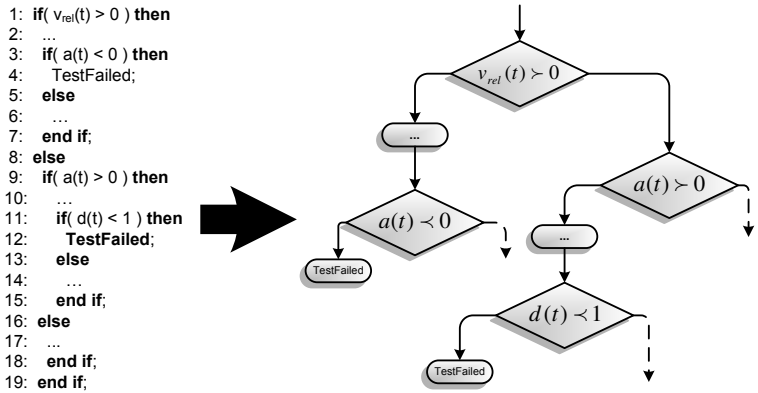


Abbildung 5.4: Zusammenhang zwischen Testauswerteskript und Kontrollflussgraphen. Jeder *TestFailed*-Knoten stellt ein unabhängiges Ziel für einen Optimierungslauf dar.

Auswerteskript wird mit den während der Testausführung aufgezeichneten Testein- und Testausgaben aufgerufen. Das Ergebnis entspricht dem Fitnesswert für ein Individuum. Mit Hilfe der Fitnesswerte wird die Suche gezielt auf den Knoten gelenkt, der dem aktuellen Testziel entspricht. In diesem Beispiel ist das Testziel der Knoten, der Zeile 12 im Auswerteskript entspricht. Die fettgedruckte Linie auf der linken Seite von Abbildung 5.5 zeigt einen möglichen Verlauf beim Aufruf mit $v_{rel}(t)$ und $a(t)$ kleiner als Null. Da der Kontrollfluss für dieses Beispiel an der Verzweigung $a(t) > 0$ in die falsche Richtung verzweigt, wird als Ergebnis der Fitnesswert aus Zeile 17 in den evolutionären Kreislauf zurückgegeben.

Der Prozess der Instrumentierung transformiert automatisch jede Verzweigung im Auswerteskript nach folgendem Schema: Sobald keine Verbindung mehr zwischen dem aktuellen Knoten und dem Testziel existiert, wird die Summe aus *Abstandsstufe* und *Zweigabstand* zurückgegeben (siehe Formel 3.2 auf Seite 48). Beim Erreichen des Testziels wird der Rückgabewert auf -1 gesetzt. Die Rückgabewerte werden dahingehend normiert, dass der *Zweigabstand* immer kleiner als die *Abstandsstufe* ist (siehe Tabelle 3.3). Untersuchungen haben gezeigt, dass die *Abstandsstufe* einen größeren Einfluss auf den Erfolg der Suche hat als der *Zweigabstand* (vergleiche zum Beispiel Wegener *et al.*) [125]. Durch diese Art der

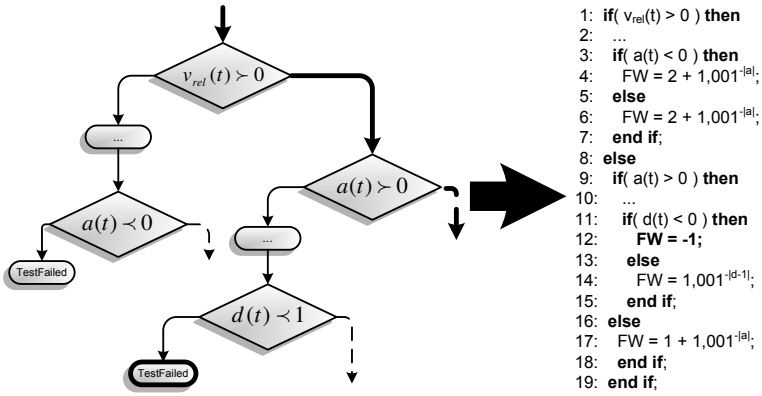


Abbildung 5.5: Generierung des instrumentierten Testauswerteskripts aus dem Kontrollflussgraphen. Durch die Instrumentierung liefert das Testauswerteskript beim Aufruf mit Testdaten als Ergebnis den Abstand zum Testziel.

Fitnessberechnung erfolgt die Normierung der Fitnesswerte bei MbET automatisch.

Der Befehlssatz für das Auswerteskript basiert auf dem Sprachsatz von Python [97]. Vorteile der Verwendung von Python sind, dass Python allgemein als einfach zu erlernen und gut lesbar gilt, die Syntax sehr kurz und ausdrucksstark ist, und dass Python-Programme leicht erweitert werden können. MbET wurde speziell für die Qualitätssicherung von Software eingebetteter Systeme entwickelt. Um den Anforderungen bei der Auswertung von Testdatensequenzen gerecht zu werden, wurde der Befehlssatz von MbET entsprechend erweitert. Zu den Erweiterungen zählen in erster Linie mathematische Operatoren, die nicht nur eine Momentaufnahme sondern auch den bisherigen Verlauf einer Sequenz von Daten berücksichtigen (zum Beispiel Integrale oder Ableitungen). Darüber hinaus können auch Filter verwendet werden, um unerwünschte Frequenzen zu unterdrücken.

Tabelle 5.1 zeigt einen Vergleich zwischen den beiden in dieser Arbeit vorgestellten Verfahren für den evolutionären Funktionstest. Auf der Grundlage von Tabelle 5.1 kann abhängig von der zu untersuchenden Problematik das geeignetere Testverfahren auswählt werden. Die oberen fünf Zeilen beziehen sich auf die Generierung

	OLET	MbET
Beschreibungssprache für Testdatensequenzen	Auszeichnungssprache für hierarchisch strukturierte Daten	hybride Automaten mit Bereichen für Parameter
Reaktivität	Nein	Ja
Variable Länge der Testdatensequenzen	Ja	Ja
Variable Chromosomenlänge	Ja	Ja
Einhaltung physikalischer Grenzen	bedingt möglich	kann durch Testmodellierung sichergestellt werden
Fitness-Beschreibungssprache	Sprache der Testumgebung oder des Testobjekts	basiert auf Python-Sprachsatz
Normierung der Fitnesswerte	manuell (nicht immer erforderlich)	automatisch mit Gleichung 3.2

Tabelle 5.1: Gegenüberstellung OLET (*open-loop*) und MbET (*closed-loop*).

von Testdatensequenzen (vorgestellt in den Abschnitten 4.1 und 5.1) und die unteren beiden Zeilen auf die Fitnessberechnung (vorgestellt in den Abschnitten 4.2 und 5.2). Die Hauptvorteile von MbET gegenüber OLET sind die unterstützte Reaktivität (*closed-loop*), ein höherer Abstraktionsgrad bei der Testspezifikation und die Unterstützung bei der Steuerung der Suche durch Techniken des Strukturtests. Zudem hat sich die modellbasierte Entwicklung und auch das modellbasierte Testen in der Vergangenheit bei der Entwicklung von Software für Steuergeräte etablieren können. Die Akzeptanz von MbET in einem industriellen Umfeld wird folglich begünstigt. In Abhängigkeit der Problemstellung kann aber durch das Testmodell ein erhöhter Programmieraufwand entstehen.

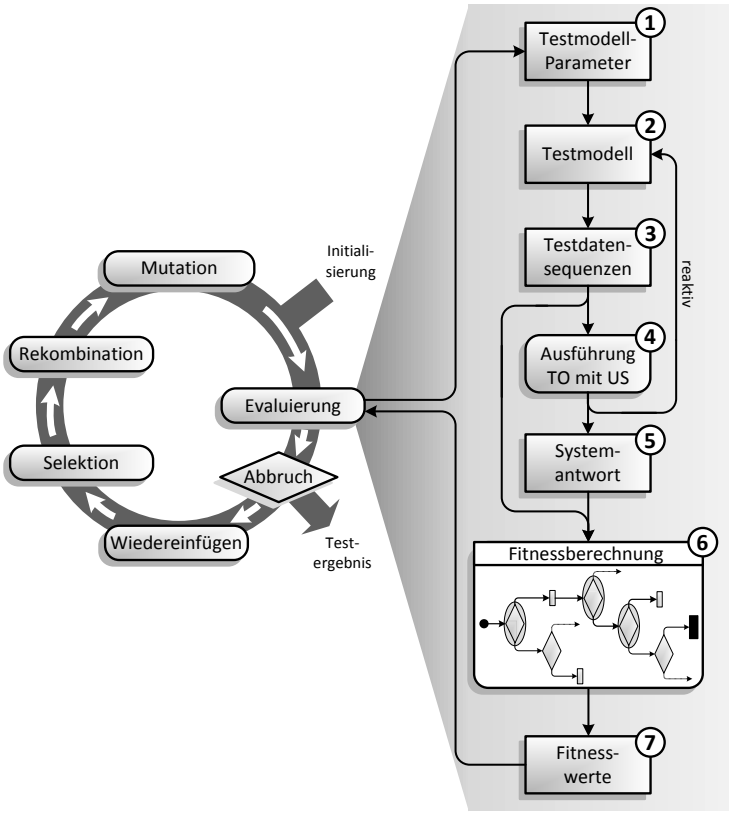


Abbildung 5.6: Workflow des MbET-Verfahrens mit dem Fokus auf der *Evaluierung* von Individuen mit TO: Testobjekt und US: Umgebungssimulation.

5.3 Optimierungskreislauf

Im Folgenden werden die durch die neue Herangehensweise bei der Beschreibung von Testdatensequenzen und der Fitnessberechnung herbeigeführten Änderungen an dem Optimierungskreislauf beschrieben. Analog zu der in Abschnitt 4.3 vorgestellten Vorgehensweise des *open-loop*-Testverfahrens wird auch hier nur der Schritt der Evaluierung von Individuen vertieft. Eine Veranschaulichung des Ablaufs der Suche bietet Abbildung 5.6. Auf der linken Seite wird der Kreislauf der evolutionären Algorith-

men dargestellt (vergleiche Abschnitt 3.1). Auf der rechten Seite wird aufgezeigt, wie den Individuen *Fitnesswerte* zugewiesen werden: In jedem Individuum sind die *Testmodell-Parameter* (1) kodiert. Mit den *Testmodell-Parametern* und dem *Testmodell* (2) kann die Testumgebung *Testdatensequenzen* (3) für die *Ausführung der Umgebungssimulation und des Testobjekts* (4) generieren. Durch die Rückkopplung (reaktiv) kann das Ausgangsverhalten des Testobjekts Einfluss auf den weiteren Verlauf der *Testdatensequenzen* nehmen. Während der *Testausführung* (4) wird die *Systemantwort* (5) aufgezeichnet und dann zusammen mit den *Testdatensequenzen* (3) für die Berechnung der *Fitness* (6) verwendet. Die *Fitnessberechnung* (6) liefert für jedes Individuum einen *Fitnesswert* (7), der dann für den weiteren Ablauf der Optimierung verwendet wird.

Jedes Blatt im Kontrollflussgraphen (vom Auswerteskript abgeleitet) entspricht entweder einem positiven oder einem negativen Testergebnis. Ein konkreter Optimierungslauf zielt immer darauf ab, ein Blatt zu erreichen, das ein negatives Testergebnis repräsentiert. Folglich muss ein separater Optimierungslauf für jedes derartige Testziel gestartet werden. Im Verlauf der Suche können allerdings auch weitere Testziele erreicht werden, die nicht dem aktuellen Suchziel entsprechen. Testfälle, die Testziele zufällig überdecken, können gegebenenfalls weitere Optimierungsläufe einsparen und werden deshalb auch berücksichtigt. Die Suche wird vollautomatisch durchgeführt, bis eines der Abbruchkriterien eintrifft.

5.4 TESTUMGEBUNG

Für die Realisierung einer prototypischen Testumgebung wird auch bei MbET auf das EvoTest-Framework zurückgegriffen (vergleiche Abschnitt 4.4.1). Nachfolgend werden die Unterschiede zu der in Abschnitt 4.4 beschriebenen OLET-Testumgebung dargestellt. Abbildung 5.7 veranschaulicht das Zusammenspiel der einzelnen Komponenten der Testumgebung und die Aufgaben des Testers. Zu den Aufgaben des Testers zählen, das *Testmodell* zu erstellen und die zu optimierenden Parameter festzulegen. Die Testumgebung liest das Testmodell ein und identifiziert automatisch alle variablen Parameter. Als nächstes muss der Tester das

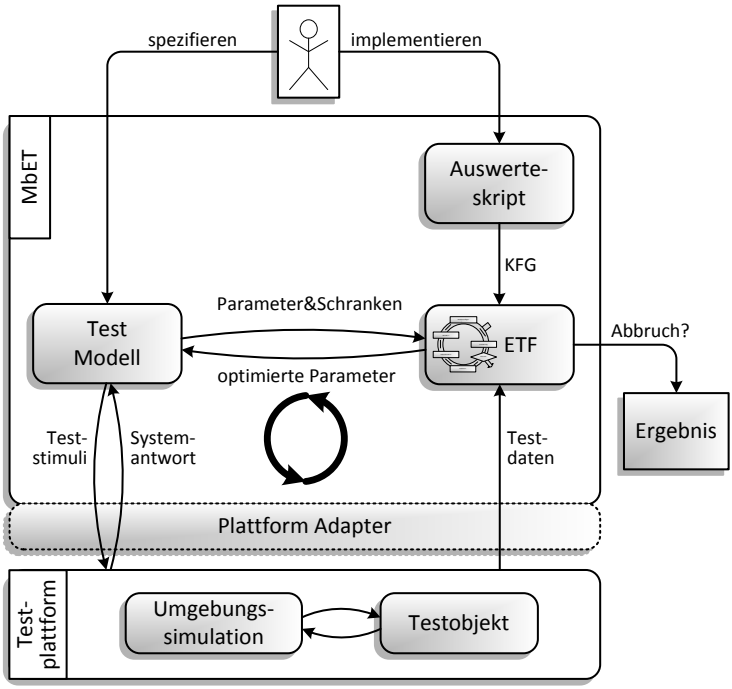


Abbildung 5.7: Die Rolle des Testers und das Konzept der Testplattform-übergreifenden Testausführung mit KFG: Kontrollflussgraph und ETF: EvoTest-Framework.

Auswerteskript implementieren. Aus der Menge möglicher Testziele, die von dem *Kontrollflussgraphen* abgeleitet werden, muss eine Auswahl getroffen werden. Für jedes Testziel wird dann automatisch mit dem *EvoTest*-Framework ein instrumentiertes Auswerteskript erzeugt und daraus eine ausführbare Fitnessfunktion generiert. Stehen die Testziele fest, muss der Tester entweder global für alle Testziele oder separat für jedes Testziel die *Schranken* der variablen *Parameter* im *Testmodell* spezifizieren. Bevor die Suche gestartet werden kann, müssen die Eingangs- und Ausgangskanäle des *Testmodells* mit der *Umgebungssimulation* und dem *Testobjekt* verbunden werden.

Im unteren Teil von Abbildung 5.7 wird die Funktionsweise des *Plattform Adapters* veranschaulicht. Der *Plattform Adapter* ermöglicht die Kommunikation zwischen *Testmodell* und *Umgebungssi-*

mulation sowie *Testobjekt*. Das *Testmodell* kann interaktiv basierend auf dem Ausgangsverhalten der simulierten Komponenten den weiteren Verlauf der Testdatengenerierung anpassen. Beispielsweise kann ein Zustandswechsel davon abhängig sein, dass ein Ausgangskanal des *Testobjekts* einen bestimmten Schwellenwert überschreitet. Durch das plattformübergreifende Konzept kann die Testausführung durch geringe Anpassung sowohl in der Simulation mit einem Modell als auch durch Ausführung auf einem Steuergerät erfolgen. Testmodelle und Auswerteskripte besitzen somit hohes Potential im Hinblick auf die plattformübergreifende Wiederverwendbarkeit [88].

ZUSAMMENFASSUNG

MbET ist ein reaktives Verfahren für den evolutionären Funktionstest von Software eingebetteter Systeme. Testdatensequenzen werden mit Hilfe hybrider Automaten beschrieben und der Suchraum durch Stellschrauben in den Automaten realisiert. Für die Bewertung von Testläufen werden die Grenzen gültigen Systemverhaltens mit einer auf dem Python-Sprachsatz basierenden Skriptsprache beschrieben. Mit den Techniken des evolutionären Strukturtests versucht die evolutionäre Engine, die darin festgelegten Grenzen zu überschreiten. Die Parametrierung des Testmodells wird automatisch in Richtung der Verletzung der Schranken gesteuert. MbET wurde speziell für den Einsatz in einem industriellen Umfeld konzipiert. Ein Ziel bei der Entwicklung des Verfahrens war deshalb, die Anwendung des evolutionären Funktionstests durch Tester ohne umfassende Kenntnisse auf dem Gebiet des SBST zu ermöglichen. Sowohl OLET als auch MbET sind vor allem als Ergänzung für bestehende funktionale Testverfahren anzusehen. Für die meisten Probleme ist es auf Grund des damit verbundenen Aufwands nicht sinnvoll, jede Anforderung mit einem evolutionären Funktionstest zu überprüfen. Mit den Verfahren OLET und MbET sollten gezielt komplexe oder sicherheitsrelevante Problemstellungen untersucht werden, die ohne einen Automatisierungsansatz schwer beherrschbar sind.

6 FALLSTUDIEN

„Es spielt keine Rolle, wie großartig deine Theorie ist, es spielt auch keine Rolle wie klug du bist – wenn es sich nicht durch Experimente belegen lässt, dann ist es einfach falsch."

— Richard Feynman (1918 - 88)

Anhand von Fallstudien wurde die Eignung von OLET und MbET für den Test von Software eingebetteter Systeme untersucht. In der Vergangenheit war die Herangehensweise an die Durchführung von Fallstudien im Bereich des suchbasierten Testens sehr heterogen [2, 34]. Diese Arbeit ebnet einer einheitlichen Herangehensweise und vor allem Vergleichbarkeit der Vielzahl von Arbeiten auf dem Forschungsgebiet den Weg (Abschnitt 6.1). Als Testobjekte der Fallstudien dienten zwei komplexe Softwaresysteme der neusten Generation aus dem Automobilumfeld: ein Abstandsregeltempomat (Abschnitt 6.2) und ein aktiver Notbremsassistent (Abschnitt 6.3). Die Systeme zeichnen sich durch einen hohen Grad der Verantwortungsübernahme – auch in komplexen Verkehrssituationen – durch das Fahrzeug aus [67]. Der Abstandsregeltempomat (ART) dient als Tempomat, reduziert aber auch automatisch den Abstand zu vorausfahrenden Fahrzeugen, falls ein zu dichtes Auffahren droht. Der aktive Notbremsassistent (ABA) warnt den Fahrer im Fall einer bevorstehenden Kollision und löst im letztmöglichen Moment automatisch eine Notbremsung aus, um einen Unfall zu verhindern. Fallstudien wurden sowohl mit OLET (Kapitel 4) als auch mit MbET (Kapitel 5) durchgeführt und mit dem Zufallstest verglichen [72, 75, 118, 127]. Um die Eignung evolutionärer Algorithmen für den Funktionstest an einem Hardware-Prüfstand zu untersuchen, wurden Testläufe mit einem realen ART-Steuergerät durchgeführt (Abschnitt 6.2.5) [74].

Schließlich werden die Ergebnisse der Fallstudien in Bezug auf die in Abschnitt 6.4 vorgestellten Propositionen bewertet.

6.1 AUFBAU DER FALLSTUDIEN

Analog zu der von Perry *et al.* und Kitchenham *et al.* empfohlenen Vorgehensweise werden im Folgenden der Kontext der Fallstudien (Abschnitt 6.1.1), die Auswahl der Fälle (Abschnitt 6.1.2), das Design der Fallstudien (Abschnitt 6.1.3) und die Gefahren für ihre Validität (Abschnitt 6.1.4) beschrieben [66, 89, 90].

6.1.1 KONTEXT DER FALLSTUDIEN

Zum Kontext der Fallstudien zählen die vorgestellten Verfahren für den evolutionären Funktionstest (Kapitel 4 und 5) sowie der aktuelle Stand der empirischen Forschung im Bereich Softwaretest. Angeregt durch die Forschung von Do *et al.* wurden im Rahmen dieser Arbeit eine Vielzahl von Veröffentlichungen mit empirischen Untersuchungen aus dem Testbereich analysiert [34]. Basierend auf den daraus resultierenden Erkenntnissen wurde eine Infrastruktur entwickelt, um Experimente mit Softwaretest-Techniken und insbesondere auch evolutionären Testansätzen durchzuführen.

In einer Arbeit von Juristo *et al.* werden Experimente mit verschiedenen Testansätzen aus über 25 Jahren analysiert, um den Reifegrad des Wissens im Bereich Softwaretest und der verschiedenen Techniken zu bewerten [63]. Dort wird unter anderem bemängelt, dass die ausgewählten Testobjekte oft nicht repräsentativ in Bezug auf die Komplexität sind. Einige Jahre später veröffentlichte Briand eine Bewertung des Status der empirischen Forschung im Bereich Softwaretest [12]. Der Fokus lag auf der Identifikation typischer Probleme derartiger empirischer Untersuchungen. Dazu zählen Probleme mit der Validität der Ergebnisse (beispielsweise wurde die Anzahl der Tests als Synonym für den Testaufwand angesehen), eine unklare Beschreibung der Fehler, die mit dem Ansatz identifiziert werden sollten, zu kleine Testobjekte ohne

Relevanz zur realen Welt und injizierte Fehler ohne Bezug zu realen Fehlern. Insbesondere weist Briand auch auf die Wichtigkeit von empirischen Untersuchungen für den evolutionären Test im Hinblick auf das Erreichen von Testzielen und die Skalierbarkeit auf komplexe Testobjekte und Suchräume hin.

Wie bereits in Abschnitt 3.4 verdeutlicht, haben bisherige Ansätze für den evolutionären Funktionstest noch nicht den erforderlichen Reifegrad für einen Einsatz in einem industriellen Kontext erreicht. Es gibt allerdings ein gesteigertes Interesse an der empirischen Untersuchung von suchbasierten Testansätzen und deren Skalierbarkeit für den industriellen Einsatz. In dieser Arbeit wird deshalb bewertet, ob die beiden vorgestellten Verfahren für den Einsatz in der Industrie geeignet sind und für welche Art von Problemstellungen sich die Verwendung empfiehlt.

6.1.2 Auswahl der Fälle

Die Auswahl der Testobjekte erfolgte unter Berücksichtigung des repräsentativen Charakters für die Ergebnisse. Sowohl beim ART als auch beim ABA handelt es sich um komplexe Steuergerätesoftware aus der Serienentwicklung der Automobilindustrie. Zum Zeitpunkt der Untersuchung standen die Systeme unmittelbar vor der Serienfreigabe. Die Tatsache, dass es sich bei beiden Systemen um sicherheitsrelevante Funktionen handelt, motiviert die Erforschung alternativer Testverfahren. Ein Fehlverhalten muss um jeden Preis ausgeschlossen werden. Die Verwendung des ART wurde darüber hinaus durch die Möglichkeit motiviert, sowohl in der Modell-Simulation als auch an einem Hardwareprüfstand mit einem realen Steuergerät zu testen. Abbildung 6.1 zeigt einen Überblick darüber, auf welcher Testplattform, mit welcher der vorgestellten Testverfahren, Fallstudien durchgeführt wurden. Von der Ausführung von SiL-Testläufen wurde abgesehen, weil der Fokus hier nicht auf der Korrektheit von Codegeneratoren liegt. Es wird davon ausgegangen, dass die Ausgaben des Testmodells beim MiL-Test und des davon abgeleiteten Programmcodes beim SiL-Test übereinstimmen.

Für jedes der beiden Testobjekte wurden Fallstudien mit zwei verschiedenen Parametrierungen der Testumgebung durchgeführt:

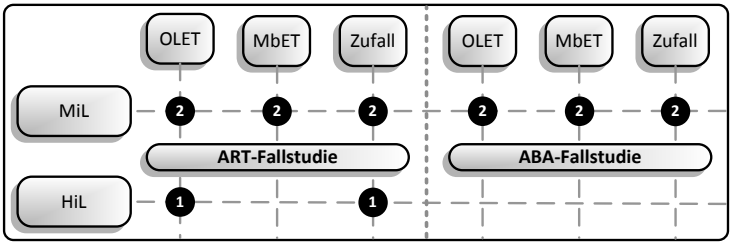


Abbildung 6.1: Überblick über Plattformen für die Testausführung und im Rahmen der Fallstudien angewendete Testverfahren.

evolutionäre Suche und Zufallssuche. Bei der evolutionären Suche wurden Fallstudien sowohl mit OLET als auch mit MbET durchführt. OLET und MbET realisieren Testautomatisierungsverfahren für komplexe Softwaresysteme, die automatisch nach einer Anforderungsverletzung suchen. Bis dato existiert außer dem Zufallstest kein Testverfahren für den industriellen Einsatz, mit dem die in dieser Arbeit entwickelten Verfahren verglichen werden können. Fallstudien wurden außerdem mit in das Modell injizierten Fehlern beziehungsweise mit einem älteren Entwicklungsstand wiederholt. Schließlich wurde der ART an einem Hardwareprüfstand mit OLET getestet. Für weitere Experimente stand der Hardwareprüfstand nicht zur Verfügung. Insgesamt ergibt sich eine Summe von vierzehn Fallstudien: Vier Fallstudien *closed-loop* evolutionär und fünf *open-loop* evolutionär sowie fünf Fallstudien mit dem Zufallstest.

6.1.3 DESIGN DER FALLSTUDIEN

Eine Bewertung der Effektivität und Anwendbarkeit der entwickelten Verfahren für den evolutionären Funktionstest erfolgt auf Basis der Propositionen P1 bis P5 (Absatz *Propositionen*). Nachfolgend wird beschrieben, wie welche Daten im Rahmen der Fallstudien erhoben wurden (Absätze *Unabhängige Variablen* und *Abhängige Variablen*) und welche Kriterien verwendet werden, um die Ergebnisse der Fallstudien in Bezug auf die aufgestellten Propositionen zu interpretieren (Absatz *Kriterien für die Interpretation der Ergebnisse*).

PROPOSITIONEN

- **P1** Durch den Einsatz evolutionärer Algorithmen und modellbasierter Testtechniken kann ein automatisiertes und reaktives Testverfahren für Software eingebetteter Systeme realisiert werden.
- **P2** Die vorgestellten Verfahren sind auch für den Test von hochkomplexen Softwaresystemen aus dem industriellen Kontext anwendbar.
- **P3** Die vorgestellten Verfahren für den evolutionären Funktionstest sind effizienter und effektiver beim Auffinden von Testfällen, die einen fehlerhaften Zustand demonstrieren, als der Zufallstest.
- **P4** Die vorgestellten Verfahren lassen es zu, dass der gleiche Testlauf sowohl in der Simulation (MiL) als auch unter Verwendung realer Steuergeräte (HiL) mit minimalen Anpassungen durchgeführt werden kann.
- **P5** Komplexe Testdatensequenzen können effektiv für das Auffinden von Fehlern und für einen industriellen Kontext angemessen im Hinblick den Zeit- und Programmieraufwand des Testers erzeugt werden.

Die Fallstudien wurden durchgeführt, um die Propositionen P1 bis P5 zu belegen. Für die Bewertung von Proposition P3 werden die Ergebnisse der evolutionären Suche mit der Zufallssuche verglichen. Für Proposition P2 wurde auf komplexe Steuergeräte-Software aus der Serienentwicklung von Kraftfahrzeugen zurückgegriffen. Ein reales Steuergerät, das für den Einsatz in einem Fahrzeug hergestellt wurde, wurde an einem speziellen Hardwareprüfstand getestet, um Proposition P4 zu untersuchen. Ziel dieser Vorgehensweise ist, Effizienz, Effektivität und Anwendbarkeit der Verfahren in einem industriellen Kontext zu bewerten.

UNABHÄNGIGE VARIABLEN

Bei der Erhebung von Daten für die Bewertung der Fallstudien im Hinblick auf die Propositionen wurde zwischen unabhängigen und abhängigen Variablen unterschieden. Unabhängige Variablen

blieben während der Durchführung einer Experimentreihe konstant, während die abhängigen Variablen veränderlich waren. Die unabhängigen Variablen der Fallstudien waren:

- U1 Als zu verwendende Testobjekte wurden ART und ABA ausgewählt, es handelt sich um zwei komplexe Softwaresysteme aus der Automobilindustrie.
- U2 Die jeweils verwendeten Verfahren für den evolutionären Funktionstest (OLET oder MbET).
- U3 Die Konfiguration der Testumgebung bei der Durchführung der Fallstudien. Es wird zwischen zwei Parametersätzen unterschieden: Der erste Parametersatz wurde für die Zufallssuche verwendet, der zweite beschrieb die Konfiguration der evolutionären Engine für die evolutionäre Suche. Für die Bestimmung der Parameter der evolutionären Engine wurden die Benchmark-Funktionen Griewank und Rastrigin verwendet [126]. Griewank und Rastrigin gelten allgemein als gute Indikatoren für die Bewertung der Effektivität und Effizienz von suchbasierten Testverfahren. Mit diesen Funktionen wurden mehrere Optimierungsläufe mit verschiedenen Parameterbelegungen durchgeführt; die Parameter, die am schnellsten konvergierten, sind in Tabelle 6.1 aufgeführt. Eine detaillierte Beschreibung der Benchmark-Experimente ist in Anhang A aufgeführt. Die Annahme bei der Verwendung dieser Konfiguration für die Fallstudien war, dass diese Parameter auch für die in dieser Arbeit verwendeten komplexen Testobjekte geeignet sind.

  Im Folgenden werden die ermittelten Parameter für die evolutionäre Suche kurz beschrieben: Aus der Elterngeneration wurden 80% (*Fruchtbarkeitsrate*) der Individuen als Kandidaten für die Fortpflanzung mittels *Turnierselektion* ausgewählt. Diese Individuen wurden dann mit einer *Rekombinationsrate* von 80% kombiniert und anschließend mit einer *Mutationsrate* von 60% mutiert. Aus den Nachkommen und den fruchtbaren und nicht-elitären Individuen wurden dann jeweils 70% durch eine erneute Anwendung der *Turnierselektion* ausgewählt. Die verbleibenden Individuen bildeten die *Zwischenpopulation*. In die finale Population wurden schließlich aus der Elterngeneration die besten 10% (*Elite*) eingefügt und die weiteren Individuen mittels

Parameter	*Zufallstest*	*Evolutionärer Test*
Algorithmus		
Populationsgröße	100	100
Generationslücke	-	1
Optimierungsrichtung	-	Minimieren
Nachfolgepopulation		
Anzahl Nachkommen	100	100
Fruchtbarkeitsrate	-	80%
Rekombinationsrate	-	80%
Mutationsrate	-	60%
Selektionsoperator	-	Turnier
Zwischenpopulation		
Überlebende Nachkommen	-	70%
Überlebende nicht-elitäre	-	70%
Selektionsansatz	-	Turnier
Finale Population		
Art des Elitismus	-	Stark
Elite	-	10%
Finale Selektion	-	Truncation
Abbruchbedingung		
Generationen	100	100
Fitness	$\leqslant 0$	$\leqslant 0$

Tabelle 6.1: Konfiguration des Zufallstests und der evolutionären Suche. Für OLET und MbET werden die gleichen Parametersätze verwendet. Die Bewertung der Zufallstests erfolgt durch die Fitnessfunktion der suchbasierten Verfahren.

Truncation-Selektion aus der *Zwischenpopulation* aufgefüllt. Abbruchbedingungen für die Optimierung waren ein Fitnesswert kleiner als Null sowie das Erreichen der 100. Generation.

ABHÄNGIGE VARIABLEN

Die abhängigen Variablen für die Fallstudien waren:

- **A1** Die Anzahl evaluierter Testfälle. Diese Zahl entspricht der Anzahl der Ausführungen der Testobjekte und der Anzahl der Auswertungen durch die Fitnessfunktion.
- **A2** Die Anzahl ungültiger Testfälle, die generiert wurden. Testdatensequenzen wurden einer Gültigkeitsprüfung im Hinblick auf die Verletzung physikalischer Grenzen unterzogen (zum Beispiel die maximal mögliche positive und negative Beschleunigung).
- **A3** Die Anzahl aufgedeckter Fehler. Unterschieden wird zwischen Fehlern, die im Verlauf des Entwicklungsprozesses nicht identifiziert werden konnten, und injizierten Fehlern.
- **A4** Der im Durchschnitt bei den MiL-Tests erreichte Überdeckungsgrad[1].
- **A5** Die Rechenzeit für einen Optimierungslauf.
- **A6** Die Anzahl von Programmcode-Zeilen und Zuständen für die Suchraumspezifikation.
- **A7** Die Anzahl von Programmcode-Zeilen der Fitnessfunktion und des Auswerteskripts.
- **A8** Die Anzahl von Programmcode-Zeilen für den Testtreiber.
- **A9** Die Verbesserungen des Fitnesswerts. Es wird betrachtet, ob und wie stark sich der Fitnesswert im Verlauf der Optimierung verbesserte. Gleichzeitig wird untersucht, ob mit der evolutionären Suche bessere Fitnesswerte erzielt werden konnten, als mit der Zufallssuche.

1 Durch MiL-Tests ermittelte Testfälle erreichen im Allgemeinen einen vergleichbaren Überdeckungsgrad bei der Simulation des Modells sowie bei der Ausführung des daraus generierten Programmcodes [5].

A10 Der Zeitaufwand für die Konfiguration der evolutionären Engine.

KRITERIEN FÜR DIE INTERPRETATION DER ERGEBNISSE

Die abhängigen und unabhängigen Variablen wurden vor dem Hintergrund ausgewählt, dass eine Aussage bezüglich der Propositionen getroffen werden kann. Nachfolgend werden kurz die Propositionen zusammengefasst und die Vorgehensweise für ihre Bewertung erläutert:

P1 Proposition P1 bezieht sich auf die Automatisierung des Testprozesses durch die Kombination modellbasierter und suchbasierter Testansätze. Eine Bewertung erfolgt basierend auf den Erkenntnissen der Fallstudien mit MbET und den abhängigen Variablen A1, A2, A3 und A9.

P2 Proposition P2 bezieht sich auf den Einfluss der Komplexität der Testobjekte auf die Anwendbarkeit von OLET und MbET. Komplexitätsmetriken der Testobjekte werden in den Abschnitten 6.2.1 und 6.3.1 vorgestellt. Auf der Grundlage der abhängigen Variablen A3 und A9 wird eine Aussage darüber getroffen, ob etwaige Fehler trotz der hohen Komplexität der Testobjekte gefunden werden können. Durch Variable A5 wird die Ausführungsdauer für einen Optimierungslauf gemessen. Variable A4 misst den erreichten Überdeckungsgrad und gibt Aufschluss darüber, wie umfassend die Struktur des Modells während der Suche zur Ausführung gebracht wurde.

P3 Mit Proposition P3 werden OLET und MbET mit dem Zufallstest verglichen. Eine kontinuierliche Verbesserung des Fitnesswerts verdeutlicht, dass das Testobjekt immer näher an den Grenzen des zulässigen Bereichs ausgeführt wurde. Ein optimaler Fitnesswert repräsentiert eine Verletzung der Grenzen. Der Fitnessverlauf wurde mit Variable A9 gemessen, gefundene Fehler mit Variable A3. Weiterhin wird auch der jeweils beste Testfall untersucht, um einen besseren Einblick in die Bedeutung der Ergebnisse zu erlangen.

P4 Proposition P4 bezieht sich auf die plattformübergreifende Anwendbarkeit der Verfahren. Grundlage dafür bilden die

in Abschnitt 6.2.5 vorgestellten Ergebnisse mit der HiL-Testumgebung aus Abschnitt 4.5 (abhängige Variable A9) und die Anzahl ungültiger Testfälle (Variable A2).

P5 Schließlich dient Proposition P5 der Bewertung der Effektivität und Effizienz der Verfahren im Hinblick auf das Auffinden von Fehlern und des Aufwands seitens der Anwender. Zur Bewertung werden die Variablen A3, A6, A7, A8 und A10 herangezogen.

6.1.4 GEFAHREN FÜR DIE VALIDITÄT

In diesem Abschnitt werden mögliche Gefahren für die *Konstruktvalidität*, sowie Gefahren für die *externe und interne Validität* der Ergebnisse der Fallstudien diskutiert.

KONSTRUKTVALIDITÄT

Eine Gefahr für die Konstruktvalidität ist die Zulässigkeit von Aussagen über die Propositionen auf Basis der gemessenen Daten. Es stellt sich die Frage, ob die gemessenen Daten den Daten entsprechen, die zu messen beabsichtigt wurden. Mögliche Gefahren für die Konstruktvalidität sind:

- Fehler, die bei einem vorzeitigen Abbruch der Suche identifiziert wurden, müssen nicht zwangsläufig eine Verletzung der untersuchten Anforderung darstellen. Um dieses Risiko zu vermindern, werden die aufgedeckten Fehler intensiv untersucht, um zu überprüfen, ob sie mit der entsprechenden Anforderung in Bezug stehen.
- Die in Kapitel 4.1 und 5.1 vorgestellten Verfahren für die Signalgenerierung erlauben es eventuell nicht, vollständig realistische Signale zu erzeugen. Um dieses Risiko zu vermindern, wurden alle ungültigen Testfälle mit Hilfe der Variable A2 erfasst.

INTERNE VALIDITÄT

Die interne Validität bezeichnet die Interpretierbarkeit der Ergebnisse – die Gewissheit, mit der Änderungen an den abhängigen Variablen auf Änderungen der unabhängigen Variablen zurückzuführen sind. Es wird das Ausmaß erfasst, mit dem das Design der Fallstudien und die Analyse der Ergebnisse durch die Existenz von Störvariablen und weiterer unerwarteter Fehlerquellen kompromittiert sein könnte. Mögliche Gefahren für die interne Validität sind:

- Die intrinsische Zufälligkeit evolutionärer Algorithmen. Diese Gefahr wird dadurch gemindert, dass jede Experimentreihe zehnmal wiederholt wurde.
- Die Ergebnisse der Fallstudien könnten durch eine fehlerhafte Implementierung der Testumgebung beeinträchtigt sein.

EXTERNE VALIDITÄT

Gefahren für die externe Validität können das Ausmaß beeinflussen, inwieweit Schlussfolgerungen aus den Fallstudien generalisiert werden können. Mögliche Gefahren für die externe Validität sind:

- Der repräsentative Charakter der für die Fallstudien ausgewählten Systeme. Es handelt sich bei beiden Testobjekten um Steuerungssysteme aus dem Automobilbereich. Das kann dazu führen, dass Ergebnisse nur in diesem Kontext gültig sind. Die Durchführung von Fallstudien mit Testobjekten aus anderen Domänen könnte eine Verallgemeinerung der Ergebnisse unterstützen.
- Die Anzahl erforderlicher Wiederholungen der Experimente. Diese kann zu viel Zeit in Anspruch nehmen, um eine Aussage über die statistische Signifikanz der Ergebnisse zu treffen. Insbesondere bei der Verwendung komplexer Softwaresysteme aus der Automobilindustrie kann die Ausführung der Testläufe lange dauern. In dieser Arbeit wurde jede Experimentreihe zehnmal wiederholt.

6.2 DATENERHEBUNG ART-FALLSTUDIE

Nach einer kurzen Beschreibung des ART-Systems (Abschnitt 6.2.1) werden die spezifischen Voraussetzungen für die Durchführung eines evolutionären Funktionstests beschrieben. Zu den Voraussetzungen gehören die Beschreibung des Suchraums (Abschnitt 6.2.2) und der Fitnessfunktion (Abschnitt 6.2.3). In den Abschnitten 6.2.4 und 6.2.5 werden die Ergebnisse der MiL- und HiL-Tests ausgewertet. Entsprechend der in Abschnitt 6.1.2 beschriebenen Vorgehensweise, wurden Experimente mit Zufallstest, mit OLET und mit MbET durchgeführt. Zusätzlich wurde das System auch an einem Hardwareprüfstand mit OLET getestet. In Abschnitt 6.2.6 werden die Ergebnisse von Experimenten mit Fehlerinjektionstechniken vorgestellt. Fehlerinjektionstests wurden durchgeführt, um die Effektivität der Verfahren besser bewerten zu können. Abschließend werden die Ergebnisse der Messung der abhängigen Variablen für die ART-Fallstudie diskutiert (Abschnitt 6.2.7).

6.2.1 SYSTEMBESCHREIBUNG

Die Motivation für die Entwicklung eines Abstandsregeltempomaten ist, die hohe Anzahl von Auffahrunfällen insbesondere auf Autobahnen zu verringern. Eine interne statistische Auswertung bei der Daimler AG hat ergeben, dass Fahrzeuge mit einem derartigen System seltener für Reparaturen an der Frontpartie in die Werkstatt müssen als Fahrzeuge, die über keinen ART verfügen. Eine weitere Studie erbrachte die Erkenntnis, dass bis zu 50% aller schweren LKW-Unfälle auf Autobahnen verhindert werden könnten, wenn alle Lastwagen über einen Abstandsregeltempomaten verfügen würden [29]. Darüber hinaus ermöglicht der ART eine Steigerung des Fahrkomforts und eine Verminderung des Kraftstoffverbrauchs durch eine effiziente Fahrweise. Die erste Generation von ART kam bei der Daimler AG bereits 1998 zum Einsatz. Bei dem für die Fallstudie verwendeten System handelt es sich um die dritte Generation.

Die Aufgabe des ART ist es, eine vorgegebene Geschwindigkeit (Setzgeschwindigkeit) zu halten. Die benötigte Beschleunigung wird automatisch angepasst, unabhängig von der Beschaffenheit

der Straße und der Trägheit des Fahrzeugs. Schaltvorgänge werden auch vollautomatisch durchgeführt. Im Unterschied zu einem klassischen Tempomaten reduziert der ART die Geschwindigkeit, falls ein zu dichtes Auffahren auf das Vorderfahrzeug droht oder wenn die Querbeschleunigung in einer Kurve zu groß wird (unterer Teil in Abbildung 6.2). Die Geschwindigkeit wird sowohl über den mechanischen Widerstand des Motors als auch über eine direkte Ansteuerung des Bremssystems reduziert. Wenn der Weg durch einen Spurwechsel oder eine starke Beschleunigung des Vorderfahrzeugs frei beziehungsweise der Kurvenradius größer wird, stellt das System wieder die Setzgeschwindigkeit ein.

Der Abstand zu vorausfahrenden Fahrzeugen wird mit Radar-, Laser- oder Kamerasystemen gemessen. Der verwendete Sensor muss in der Lage sein, Verkehrssituationen zuverlässig zu erkennen; beispielsweise muss in einer Kurve das vorausfahrende Fahrzeug genau identifiziert werden. Weiterhin dürfen Verkehrsschilder und Brücken über der Fahrbahn nicht mit Fahrzeugen verwechselt werden. Das in dieser Arbeit untersuchte System verwendet einen Radar für die Abstandsbestimmung und ist so ausgelegt, dass es mit einer maximalen negativen Beschleunigung von $-2\mathrm{m}/\mathrm{s}^2$ bremst. Reicht die maximal zulässige Bremskraft nicht aus, um einen sicheren Mindestabstand einzuhalten, wird der Fahrer durch optische und akustische Signale gewarnt. Der Fahrer muss dann wieder die Kontrolle über die Geschwindigkeit des Fahrzeugs übernehmen und bremsen oder dem Hindernis ausweichen.

KOMPLEXITÄT DES ART-MODELLS

Der verwendete ART wurde mit der Modellierungssprache Matlab Simulink implementiert und besteht aus 3.284 Blöcken [110, 111]. Interne Systemzustände wurden mit der Erweiterung Stateflow modelliert und gliederten sich in 8 Zustandsdiagramme mit insgesamt 25 Zuständen [112]. Das Testobjekt kommunizierte mit dem Umgebungsmodell über 26 Eingänge und 11 Ausgänge. Der ART weist eine vergleichsweise hohe Komplexität zu anderen Steuerungssystemen im Bereich eingebetteter Systeme auf. Zusätzlich zu dem ART-Systemmodell wurde ein weiteres Modell für die Umgebungssimulation verwendet. Zu den Aufgaben des Umgebungsmodells zählte, Komponenten zu simulieren, die im

realen Fahrzeug mit dem ART kommunizieren. Weiterhin wurden beispielsweise das Fahrverhalten des vorausfahrenden Fahrzeugs und der Straßenverlauf simuliert. Das Umgebungsmodell stellte sicher, dass die Ansteuerung des ART auf eine Art erfolgt, die mit einem realen Fahrszenario vergleichbar ist.

FAHRSZENARIO

Jeder Testlauf entsprach einem Fahrszenario mit einer Simulationsdauer von ungefähr einer Minute. In dem Szenario wurden zwei Fahrzeuge mit Hilfe eines Umgebungsmodells simuliert: Ein Fahrzeug mit dem ART und ein vorausfahrendes Fahrzeug (oberer Teil in Abbildung 6.2). Beide Fahrzeuge fuhren über den gesamten Testverlauf in derselben Spur. Ein konkretes Szenario begann immer mit einer kurzen Initialisierungssequenz, wobei das Fahrzeug mit dem ART vom Stillstand bis zu einer vorgegebenen Geschwindigkeit beschleunigte[2]. Im Anschluss an die Initialisierung wurde der ART aktiviert und erhielt die Kontrolle über die Geschwindigkeit des Fahrzeugs. Die Interaktion des Fahrers mit dem ART war dann auf die Betätigung des *Tempomatenhebels* (Anpassung der Setzgeschwindigkeit) und des *Abstandshebels* (Anpassung des Abstandsmodus) beschränkt. Das vorausfahrende Fahrzeug fuhr die Geschwindigkeit, die von der Testumgebung vorgegeben wurde.

TESTZIEL

Ziel der Suche ist, einen Testfall zu finden, bei dem der Mindestabstand zwischen den beiden Fahrzeugen verletzt wird, ohne dass rechtzeitig die Fahrerübernahmewarnung aktiviert wird. Aufgrund der hohen Komplexität und des unendlich großen Eingangsdatenraums des ART, ist der evolutionäre Funktionstest besonders für die Überprüfung der Anforderung geeignet. Die negative Beschleunigung des ART ist auf $-2\mathrm{m/s^2}$ begrenzt. Reicht diese nicht aus, muss der Fahrer rechtzeitig informiert werden, um einen manuellen Bremseingriff auszulösen. Für die Fallstudien wurde von einer Reaktionszeit des Fahrers von 1s und einer maximal möglichen negativen Beschleunigung von $-6\mathrm{m/s^2}$ ausgegangen. Eine Situation, in der keine Warnung aktiviert ist und

2 Für den Black-Box-Test des verwendeten Testobjekts erforderlich.

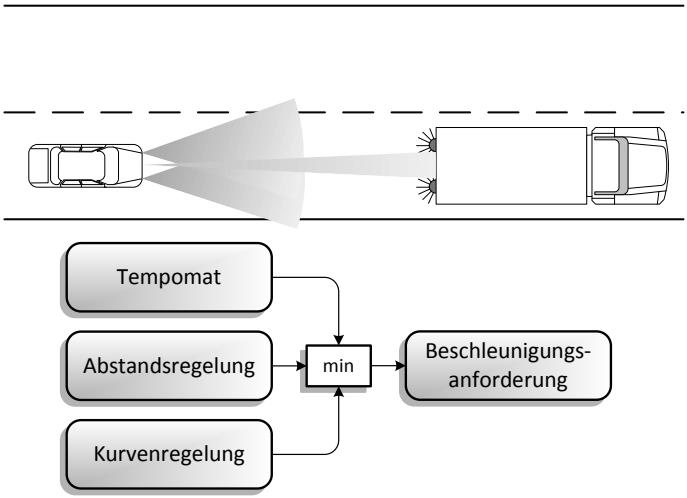


Abbildung 6.2: Fahrszenario für die ART-Experimente. Im unteren Teil der Abbildung wird die Funktionsweise des ART skizziert: *Tempomat*, *Abstands-* und *Kurvenregelung* geben jeweils Beschleunigungswerte vor, das System wählt den kleinsten Wert aus.

der Abstand nicht ausreicht, um bei diesen Vorgaben eine Kollision zu verhindern, gilt als Anforderungsverletzung.

6.2.2 Beschreibung des Suchraums

Im Folgenden wird veranschaulicht, wie der Suchraum für den evolutionären Funktionstest des ART spezifiziert wurde. Dabei wird zwischen der Herangehensweise von OLET und MbET differenziert. Insgesamt wurden Testdatensequenzen für vier Eingänge des Testobjekts und der Umgebungssimulation generiert: Die Geschwindigkeit des vorausfahrenden Fahrzeugs, die Interaktion des Fahrers mit dem Tempomatenhebel und dem Abstandshebel sowie der Kurvenradius (bei kleinem Kurvenradius begrenzt der ART die Geschwindigkeit).

OPEN-LOOP

Abbildung 6.3 zeigt die *Signalspezifikation* für den *open-loop*-Test des ART. Für jedes der vier variablen Signale wurde ein Signaltyp zugelassen. Die Geschwindigkeit des vorausfahrenden Fahrzeugs ist eine Aneinanderreihung von *Spline*-Segmenten, die jeweils mit einer Steigung von Null beginnen und enden. Dadurch wurde ein stetiger und differenzierbarer Signalverlauf realisiert. Betätigungen des Tempomatenhebels wurden durch *Impulse* simuliert, die in unregelmäßigen Zeitabständen ausgelöst wurden. Eine Amplitude mit dem Wert *eins* steht für den Fahrerwunsch, die Sollgeschwindigkeit zu erhöhen, und ein Wert von *zwei* dafür, die Geschwindigkeit zu reduzieren. Beim Abstandshebel verhält es sich ähnlich, *eins* steht für den Fahrerwunsch den Abstand zu verringern und *zwei* den Abstand zu vergrößern. Der Kurvenradius wurde durch ein *Cosinus*-Signal simuliert. Die Initialisierungssequenz wurde direkt vom Testtreiber erzeugt. Während der Initialisierung wurde das Fahrzeug mit dem ART durch die Betätigung des Gaspedals aus dem Stand beschleunigt.

CLOSED-LOOP

Auch bei MbET wurden die vier Signale – Geschwindigkeit des vorausfahrenden Fahrzeugs, Interaktion mit Tempomaten- und Abstandshebel sowie Kurvenradius – variiert. Allerdings wurden hierbei die Testdatensequenzen mit hybriden Automaten spezifiziert. Abbildung 6.4 zeigt das Testmodell, von dem die Signale für die Testausführung abgeleitet wurden. Bei dem Automaten, der das Verhalten des Fahrers modelliert, wurde auf das Konzept der Reaktivität zwischen Testmodell und Testobjekt zurückgegriffen. Der Übergang vom Zustand *Init* in den Zustand *Keine Fahrerinteraktion* wurde erst dann vollzogen, wenn eine bestimmte Geschwindigkeit des ART-Fahrzeugs erreicht wurde (ausgelöst durch eine konstante Betätigung des Gaspedals). Die Geschwindigkeit des ART-Fahrzeugs ist eine Ausgangsgröße des Testobjekts.

Signal-spezifikation

```
<signale f_T="100" t_min="30" t_max="70">
<signal name="Vorausfahrend" n_min="1" n_max="9">
   <amplitude min="15" max="40"/>
   <breite min="2" max="7"/>
   <transitionen> <t name="spline"/> </transitionen>
</signal>
<signal name="Tempomatenhebel" type="integer" n_min="6" n_max="10">
   <amplitude min="1" max="2"/>
   <transitionen> <t name="impulse"/> </transitionen>
</signal>
<signal name="Abstandshebel" type="integer" n_min="1" n_max="5">
   <amplitude min="1" max="2"/>
    <transitionen> <t name="impulse"/> </transitionen>
</signal>
<signal name="Kurvenradius" n_min="1" n_max="5">
   <amplitude min="100" max="2000"/>
   <breite min="1" max="4"/>
   <transitionen> <t name="cosine"/> </transitionen>
</signal>
</signale>
```

Abbildung 6.3: Auszug aus der Suchraumbeschreibung für den *open-loop*-Test des ART.

6.2.3 Erstellung der Fitnessfunktion

Als nächster Schritt erfolgte die Herleitung und Spezifikation der Fitnessfunktion. Zu erst werden hierzu Vorüberlegungen präsentiert, die sowohl für OLET als auch für MbET relevant sind. Anschließend wird die konkrete Umsetzung für die beiden Verfahren beschrieben.

Die Fitnessfunktion muss die Kritikalität einer Situation in Bezug auf das rechtzeitige Anzeigen der Übernahmewarnung bewerten. Ein Fitnesswert kleiner als Null bedeutet, dass die zu untersuchende Anforderung verletzt wurde. Folgende Definitionen wurden bei der Beschreibung der Fitnessfunktion verwendet:

- Geschwindigkeit des Eigenfahrzeugs v_{ego} [m/s]: Geschwindigkeit des Fahrzeugs mit dem ART-System.
- Geschwindigkeit des Zielfahrzeugs v_{ziel} [m/s]: Geschwindigkeit des Fahrzeugs, das vor dem Eigenfahrzeug fährt.

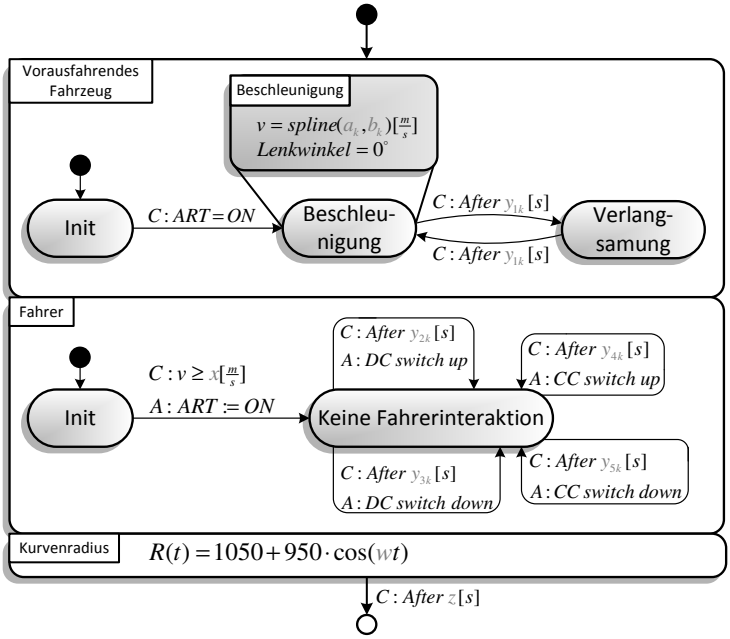


Abbildung 6.4: Suchraumbeschreibung für den *closed-loop*-Test des ART. Parameter in *grau* werden von dem evolutionären Algorithmus variiert. Für die Wertebereiche der variablen Parameter gilt: $a_k \in [0;0,03]$, $b_k \in [0;0,3]$, $w \in [0;0,2]$, $x \in [5;20]$, $y_{(1..5)k} \in [0;60]$ und $z \in [30;70]$.

- Relativgeschwindigkeit v_{rel} [m/s]: Zielfahrzeuggeschwindigkeit minus Eigengeschwindigkeit.
- Minimale Relativgeschwindigkeit $v_{rel,min}$ [m/s]: Ein kleiner positiver Wert.
- Reaktionszeit T_R [s]: Zeit zwischen dem Erscheinen der Warnung und dem Beginn der Bremsung durch den Fahrer (1s).
- Bremszeit t_B [s]: Zeit vom Beginn der Bremsung bis die Relativgeschwindigkeit gleich Null ist.
- Bremsabstand d_B [s]: Abstand zu dem Zeitpunkt, bei dem die Geschwindigkeit des Eigenfahrzeugs Null ist.

- Bremsweg $s_{B,ego}$ und $s_{B,ziel}$ [m]: Wege, den die Fahrzeuge während der Bremsung zurücklegen.
- Maximale Bremskraft des Fahrers $a_{ego,max}$ [m/s^2]: Begrenzt auf $-6 m/s^2$.
- Ungebremster Zeitabstand ZA_U [s]: Zeit bis zum Erreichen der Position des Zielfahrzeugs bei konstanter Geschwindigkeit. Analog lässt sich der Zeitabstand auch als Distanz [m] verstehen.
- Gebremster Zeitabstand ZA_B [s]: Zeit bis zum Erreichen der Position des Zielfahrzeugs, wobei nach der Reaktionszeit mit $a_{ego,max}$ gebremst wird (Reaktionszeit + Bremszeit).

Der *Zeitabstand* ZA_U unter der Annahme, dass die Fahrzeuge nicht bremsen, wurde wie folgt berechnet:

$$\begin{aligned} ZA_U(t) &= \frac{d(t)}{\max(v_{ego}(t) - v_{ziel}(t), v_{rel,min})} \\ &= \frac{d(t)}{\max(v_{rel}(t), v_{rel,min})} \end{aligned} \tag{6.1}$$

Als nächster Schritt wurde der *Zeitabstand* für den Fall berechnet, dass der Fahrer beim Erscheinen der Warnung bremst und das vorausfahrende Fahrzeug mit konstanter Geschwindigkeit weiterfährt. Dafür musste zuerst der Abstand nach der Reaktionszeit $d_R(t)$ bestimmt werden:

$$\begin{aligned} d_R(t) &= d(t) - v_{ego}(t) \cdot T_R + v_{ziel}(t) \cdot T_R \\ &= d(t) - v_{rel}(t) \cdot T_R \end{aligned} \tag{6.2}$$

Für den Fall, dass der *Zeitabstand* nach der Reaktionszeit schon kleiner als Null ist, wurde Formel 6.1 für den ungebremsten *Zeitabstand* verwendet. Ein Unfall kann in diesem Fall nicht verhindert werden. Wenn der Abstand nach der Reaktionszeit größer als Null ist, kann noch gebremst werden. Der Abstand nach dem abgeschlossenen Bremsmanöver d_B wurde dann wie folgt berechnet:

$$d_B(t) = d_R(t) - s_{B,ego}(t) + s_{B,ziel}(t) \tag{6.3}$$

Da im Falle einer Kollision von einem Abstand von *Null* ausgegangen wurde, galt: $d_B(t) = 0$ und die *Bremszeit* $t_B(t)$ konnte wie folgt berechnet werden:

$$t_B(t) = \frac{v_{rel}(t) + \sqrt{v_{rel}(t)^2 + 2 \cdot a_{ego,max} \cdot d_R(t)}}{a_{ego,max}} \tag{6.4}$$

Die *Bremszeit* t_B ist die Zeit vom Beginn der Bremsung (nach der Reaktionszeit T_R) bis der Abstand zwischen den beiden Fahrzeugen gleich Null ist und somit eine Kollision stattfindet. Für den gebremsten *Zeitabstand* ZA_B ergibt sich:

$$ZA_B(t) = t_B(t) + T_R \tag{6.5}$$

ZA_B ist undefiniert wenn der Ausdruck unter der Wurzel kleiner als Null ist, also wenn $v_{rel}(t_0)^2 + 2 \cdot a_{ego,max} \cdot d(t_R) < 0$. In diesem Fall konnte die Berechnung für den ungebremsten *Zeitabstand* verwendet werden, um die Steuerung der Suche zu ermöglichen. Insgesamt ergab sich für den *Zeitabstand*:

$$ZA(t) = \begin{cases} ZA_U(t), & d_R(t) \leqslant 0 \\ ZA_U(t), & v_{rel}(t)^2 + 2 \cdot a_{ego,max} \cdot d_R(t) < 0 \\ ZA_B(t), & sonst \end{cases} \tag{6.6}$$

Als letzter Schritt musste noch der Zeitabschnitt, in dem die Warnung aktiv ist, mit einbezogen werden. Ziel bei der Suche war, einen Testfall zu finden, bei dem der ART die Warnung zu spät ausgelöst hatte. Deshalb ist der *Zeitabstand*, solange die Warnung aktiv ist, nicht relevant für die Fitnessberechnung. Der Zustand der Warnung ist ein Ausgangskanal des Testobjekts. Um zu verhindern, dass Werte von der Fitnessfunktion ausgewählt werden, zu denen die Warnung aktiv ist, kann dieses Signal mit einem großen Wert δ multipliziert und zum *Zeitabstand* hinzu addiert werden. Für die finale Fitnessfunktion ergab sich folglich:

$$Fitnesswert = min(ZA(t) + \delta \cdot Warnung(t)) \tag{6.7}$$

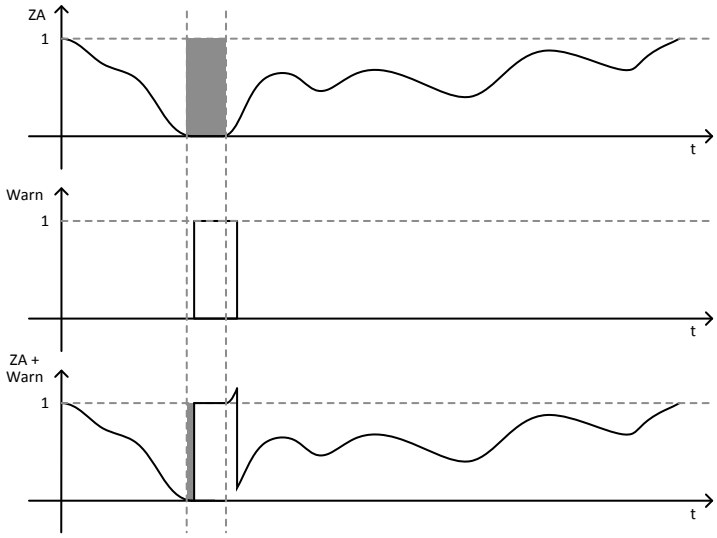


Abbildung 6.5: Fitnessberechnung für einen ART-Testfall. Im oberen Diagramm wird der normierte Zeitabstand (ZA), in der Mitte die Warnung (Warn) und im unteren Diagramm die Summe des normierten Zeitabstands und der Warnung dargestellt.

In Abbildung 6.5 wird das Prinzip des Miteinbeziehens der Warnung in die Fitnessberechnung grafisch veranschaulicht. Nachfolgend werden die Besonderheiten bei der Realisierung der Fitnessfunktion für OLET und MbET vorgestellt.

OPEN-LOOP

Entsprechend der in Abschnitt 4.2 beschriebenen Vorgehensweise, wurde Gleichung 6.7 als mathematische Funktion implementiert und auf jeden Zeitschritt der bei einem Testlauf aufgezeichneten Signale angewendet. Als Programmiersprache empfiehlt es sich, entweder die Sprache der Testumgebung (Java) oder die Implementierungssprache des Testobjekts zu verwenden (Matlab). Der kleinste mit der Fitnessfunktion errechnete Wert repräsentiert den Fitnesswert für den Testfall und wird in den Optimierungskreislauf zurückgeführt.

CLOSED-LOOP

Bei dem *closed-loop*-Verfahren wurde ebenfalls Gleichung 6.7 als Ausgangspunkt verwendet. Entsprechend der in Abschnitt 5.2 beschriebenen Vorgehensweise wurde mit einem Python-basierten Sprachsatz ein Auswerteskript von Formel 6.7 abgeleitet. Das Auswerteskript wurde dann automatisch wie beim Strukturtest instrumentiert (vergleiche Abschnitt 3.3 und Tabelle 3.3). Das instrumentierte Auswerteskript 1 wurde mit den Testeingaben und Testausgaben aufgerufen. Durch die Instrumentierung war es möglich, den Kontrollfluss zu observieren und den Abstand zum Zielknoten zu berechnen. Der Abstand zum Zielknoten entspricht dem Fitnesswert. Da es sich um eine Sequenz von Testdaten handelt, wird jeder Zeitschritt einzeln betrachtet und der kleinste Abstand im Testverlauf als Fitnesswert übernommen.

6.2.4 MODEL-IN-THE-LOOP-EXPERIMENTE

In diesem Abschnitt werden die Ergebnisse der MiL-Experimente mit OLET, MbET und dem Zufallstest[3] vorgestellt. Die Konfiguration der evolutionären Engine und des Zufallstests erfolgte entsprechend der in Tabelle 6.1 aufgeführten Parameter.

Bevor die Experimente gestartet werden konnten, musste der Testtreiber für die MiL-Experimente angepasst werden. Notwendige Anpassungen beinhalteten die Verbindung der Eingangs- und Ausgangskanäle des Testobjekts mit dem Testtreiber. Weiterhin mussten die während eines Testlaufs aufgezeichneten Daten an die Auswertefunktion weitergeleitet werden. Danach konnte die Optimierung gestartet werden.

Abbildung 6.6 zeigt ein Beispiel eines Testfalls, der während eines Optimierungslaufs mit MbET ermittelt wurde. Die Diagramme zeigen den Verlauf der Testdatensequenzen, die von der Testumgebung erzeugt wurden (Interaktionen mit Tempomatenhebel und Abstandshebel, Geschwindigkeit des vorausfahrenden Fahrzeugs,

3 Teststimuli für die Zufallstest wurden in dieser Arbeit mit MbET-Testmodellen erzeugt. Die Bewertung erfolgte durch MbET-Auswerteskripte.

Auswerteskript 1 returns FitnessWert

Input: $v_{rel}, d, warnung, T_R, a_{ego,max}$

1: $d_R(t) = d(t) - v_{rel}(t) \cdot T_R$
2: $c(t) = v_{rel}(t)^2 + 2 \cdot a_{ego,max} \cdot d_R(t)$
3: **if** $warnung! = 1$ **then**
4: **if** $d_R(t) \leqslant 0$ **then**
5: **if** $c(t) \geqslant 0$ **then**
6: $t_B = (v_{rel}(t) + sqrt(v_{rel}(t)^2 + 2 \cdot a_{ego,max} \cdot d_R(t)))/a_{ego,max}$
7: **if** $t_B + T_R < 0$ **then**
8: $FW = -1$
9: **else**
10: $FW = 1 - 1,001^{-|t_B - T_R|}$;
11: **end if**
12: **else**
13: $FW = 2 - 1,001^{-|c(t)|}$
14: **end if**
15: **else**
16: $FW = 3 - 1,001^{-|d_R(t)|}$
17: **end if**
18: **else**
19: $FW = 4$
20: **end if**

Kurvenradius) sowie die Antwort an den Ausgängen des Testobjekts (Warnungen, Abstände, Geschwindigkeit und Setzgeschwindigkeit des Fahrzeugs mit dem ART). Die Übernahmewarnung des ART wird durch ein starkes Bremsmanöver des vorausfahrenden Fahrzeugs ausgelöst. Zum Zeitpunkt des Erscheinens der Warnung ist hier noch ausreichend Zeit für den Fahrer gegeben, um einen Unfall zu verhindern.

In Abbildung 6.7 werden der durchschnittliche Fitnessverlauf und der in jeder Generation gefundene beste Wert der Experimente dargestellt. Der Durchschnittswert wird in dieser und in weiteren Abbildungen, die den Fitnessverlauf darstellen, immer mit

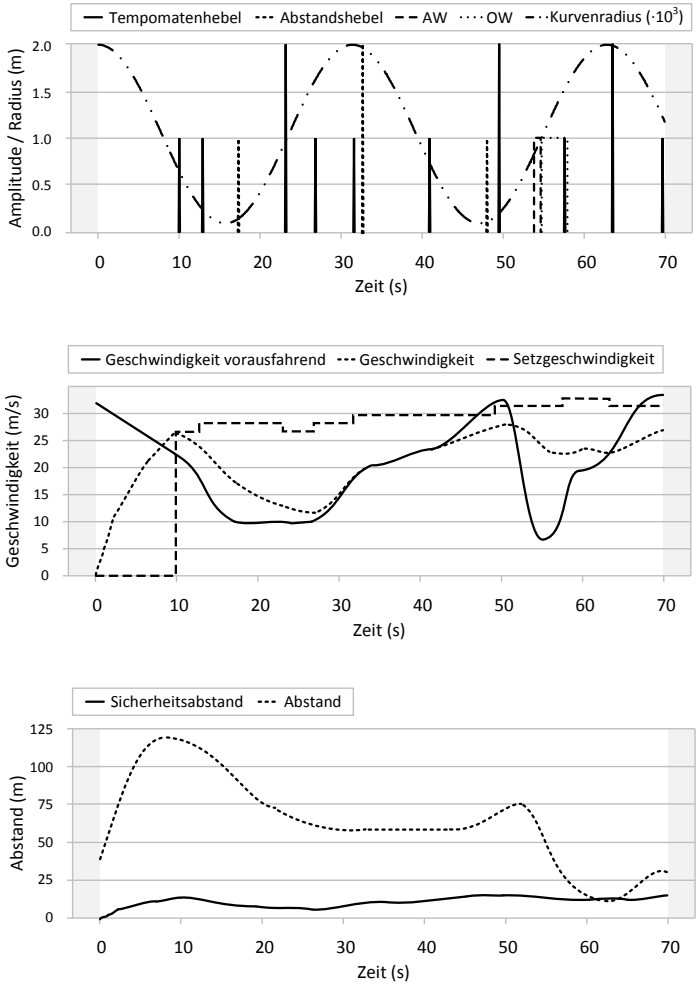


Abbildung 6.6: Generierte Signale und Ausgangsverhalten eines Fahrmaneuvers mit dem ART. OW steht für optische und AW für akustische Warnung.

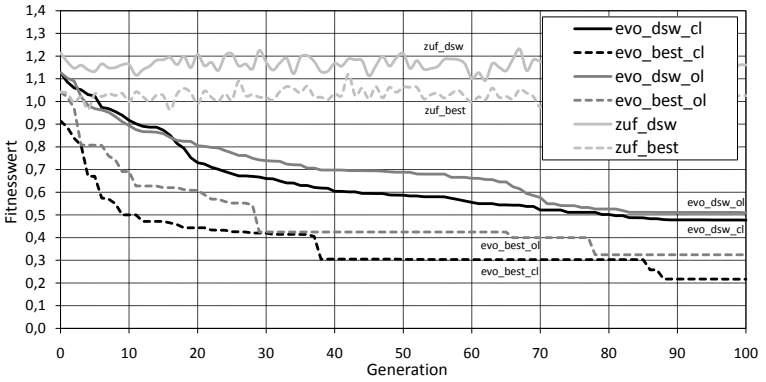


Abbildung 6.7: Fitnessverlauf der ART-Experimente. Dargestellt sind die Ergebnisse mit dem *open-loop* (ol)- und *closed-loop* (cl)-Verfahren im Vergleich zum Zufallstest (zuf). Die Experimente wurden mit jeder Konfiguration zehnmal wiederholt.

dsw abgekürzt, die besten gefundenen Werte mit *best*. Die Experimentreihen wurden je zehnmal mit dem *open-loop*[4] – und dem *closed-loop*-Verfahren sowie zehnmal mit dem Zufallstest durchgeführt. Die beiden suchbasierten Verfahren erzielten vergleichbare Ergebnisse; beim Zufallstest konnten dagegen nur Testfälle mit einem deutlich schlechteren Fitnesswert identifiziert werden. Durch Mann-Whitney-Wilcoxon Tests konnte für den jeweils besten ermittelten Fitnesswert gezeigt werden, dass die Ergebnisse von OLET und MbET mit einer hohen statistischen Signifikanz besser sind, als die des Zufallstests ($P < 0{,}001$, zweiseitiger Test). Der Unterschied zwischen MbET und OLET ist nicht statistisch signifikant ($P < 0{,}05$).

Bei der Durchführung der Experimente konnte kein Testfall gefunden werden, der eine Verletzung der untersuchten Anforderung aufzeigt. Da kein vorzeitiger Abbruch durch das Erreichen des Testziels eingetreten ist, wurden insgesamt 10 [Optimierungsläufe] $\cdot$100 [Generationen] $\cdot$100 [Individuen] = 100.000 [Testfälle] für jede Konfiguration ausgeführt und ausgewertet. Bei einer Dauer von $\approx$ 10s pro Testfall ergab das eine gesamte Ausführungszeit

4 Die Fitnesswerte für den *open-loop*-Test wurden für eine bessere Vergleichbarkeit mit einem konstanten Faktor skaliert.

(ohne Parallelisierung) von 100.000 [Testfällen] ·10s [Dauer je Testausführung] = 1.000.000s, was ungefähr 278 Stunden, beziehungsweise 27,8 Stunden[5] für einen einzelnen Testlauf entspricht. Der Zeitaufwand war identisch für die Durchführung der *open-loop-*, *closed-loop-* und Zufallstests. Da die Testausführungen innerhalb einer Generation unabhängig sind, kann die Ausführungsdauer durch Parallelisierung signifikant verringert werden.

Bei der Durchführung der Optimierungsreihen wurde die kumulierte Überdeckung des ART-Modells gemessen. Sicherheitsstandards verlangen oft eine vollständige Überdeckung. Die ermittelten Testfälle können als Ausgangspunkt für einen Strukturtest verwendet werden, der auf eine vollständige Überdeckung abzielt. Für die evolutionären Testverfahren ergab sich bei den Optimierungsreihen im Durchschnitt ein Überdeckungsgrad von 79% für die C_1-Zweigüberdeckung (siehe Tabelle 3.2) und 85% für die D_1-Bedingungsüberdeckung (vergleiche Conrad und Sadeghipour [25]). Ein höherer Überdeckungsgrad kann erreicht werden, indem die Stimulation auf mehrere Eingangskanäle des Testobjekts ausgeweitet wird. Fahreraktivitäten, wie zum Beispiel Blinkerbetätigungen, wurden im Rahmen dieser Fallstudie nicht simuliert. Durch die Stimulation weiterer Eingangskanäle besteht allerdings die Gefahr, dass die Suche erst später relevante Bereiche des Suchraums aufdeckt. Beim Zufallstest wurden etwas geringere Werte für den Überdeckungsgrad gemessen (C_1: 77% und D_1: 84%) – dazu waren die im Verlauf der Suche ermittelten Fitnesswerte und somit die Qualität der Testfälle signifikant schlechter (siehe Abbildung 6.7).

6.2.5 HARDWARE-IN-THE-LOOP-EXPERIMENTE

Um das plattformübergreifende Konzept zu demonstrieren, wurde mit OLET ein evolutionärer Funktionstest an einem HiL-Prüfstand durchgeführt. Der Aufbau des Prüfstands wurde in Abschnitt 4.4.3 beschrieben. Das Testobjekt ist ein Steuergerät mit aus dem ART-Modell generiertem Code. Hervorzuheben ist, dass sowohl die Suchraumbeschreibung als auch die Fitnessfunktion ohne Anpassung aus den MiL-Tests wiederverwendet werden konnten.

5 WinXP, Dual-Core mit je 1,8GHz, 2GB Ram

Es mussten allerdings einige Anpassungen am Testtreiber vorgenommen werden. Diese umfassten hauptsächlich die Weiterleitung der generierten Testdatensequenzen an die Eingänge der HiL-Testumgebung. Weiterhin konnten die Testdatensequenzen im Unterschied zu der MiL-Simulation nicht direkt übermittelt werden, sondern mussten in Textdateien zwischengespeichert werden. Die Ansteuerung des Echtzeitrechners wurde von ProveTech:TA vorgenommen [77]. ProveTech:TA liest die generierten Signale aus den Textdateien aus und übermittelt diese dann an den Echtzeitrechner (siehe Abbildung 4.5). Nach der Testausführung wurden die aufgezeichneten Ergebnisse wieder in Textdateien geschrieben, von dem Testtreiber ausgelesen und mittels der Fitnessfunktion bewertet.

Der Prozess der Erzeugung von Testdatensequenzen ist identisch mit dem MiL-Test, deshalb wird hier auf die Plots in Abbildung 6.6 verwiesen. Das verwendete Steuergerät stammte aus der Serienfertigung und ist mittlerweile auf der Straße im Einsatz. Dementsprechend ist es auch nicht verwunderlich, dass keine Auffälligkeiten gefunden werden konnten. Zu den wesentlichen Vorteilen des HiL-Tests gegenüber dem MiL- und SiL-Test zählt die Möglichkeit, Tests auch bei Nichtvorhandensein des Modells beziehungweise des Programmcodes automatisiert durchzuführen. Weiterhin werden die Hardwareeigenschaften der Steuergeräte (zum Beispiel das Laufzeitverhalten) berücksichtigt. Der Fitnessverlauf des Zufallstests im Vergleich zum evolutionären Funktionstest wird in Abbildung 6.8 dargestellt[6]. Auch hier lässt sich erkennen, dass der evolutionäre Test im Vergleich zum Zufallstest schneller einen besseren Fitnesswert erreichen konnte. Als Ergebnis dieser Untersuchung lässt sich festhalten, dass der evolutionäre Funktionstest in der Lage ist, automatisiert zielgerichtete Testfälle für den HiL-Test zu erzeugen, und dass Suchraumbeschreibung und Fitnessfunktion aus dem MiL-Test wiederverwendet werden können.

6 Für die HiL-Tests wurde eine Vorabversion der Fitnessfunktion verwendet.

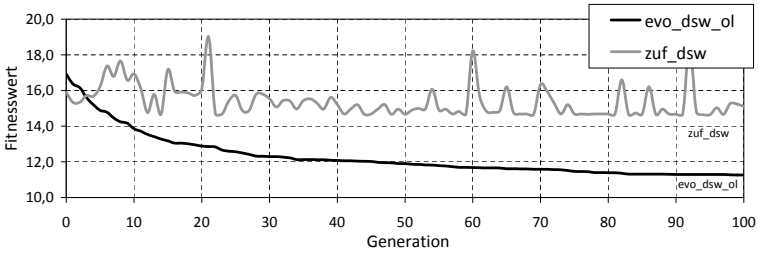


Abbildung 6.8: Fitnessverlauf für die HiL-Experimente (*open-loop*). In dieser Arbeit wurden aufgrund der hohen Ausführungszeit der Testfälle und der nur für begrenzte Zeit zur Verfügung stehenden Testumgebung jeweils drei Optimierungsdurchläufe durchgeführt.

6.2.6 FEHLERINJEKTIONSTESTS

Mithilfe von Techniken der Fehlerinjektion wurde die Effektivität von OLET und MbET im Vergleich zum Zufallstest untersucht (vergleiche auch Clark und Pradhan [22]). Im Allgemeinen wird die Fehlerinjektion verwendet, um festzustellen, wie fehlertolerant ein System ist. Es wird also überprüft, ob ein System trotz injiziertem Fehler ein korrektes Ergebnis liefert beziehungsweise durch eine Fehlermeldung zu erkennen gibt, dass sich das System nicht gemäß der Spezifikation verhält. In dieser Arbeit wurde die Fehlerinjektion dazu verwendet, ein Testverfahren zu bewerten (vergleiche auch Eldh *et al.* [35]). In einer Arbeit von Schlingloff und Vulinovic wird das Prinzip der Fehlerinjektion im Zusammenhang mit der modellbasierten Entwicklung vorgestellt [100]. Dabei werden auch Fehlerklassen festgelegt. Im Rahmen dieser Fallstudie wurde auf die Fehlerklasse zurückgegriffen, bei der mit Hilfe von Verstärkerblöcken das Signal zwischen zwei Blöcken verändert wird. In Frage kamen nur Fehler, die mit der untersuchten Anforderung korrelieren und somit den Erscheinungszeitpunkt der Warnung beeinflussen.

BESCHREIBUNG DES INJIZIERTEN FEHLERS

Um einen hohen Komfort für den Fahrer zu gewährleisten und ein zu abruptes Abbremsen oder Beschleunigen zu verhindern,

wurde der ART mit einer Vielzahl von Regelkreisen und Filtern modelliert. Ein Modul des ART berechnet zu jedem Zeitpunkt in Abhängigkeit der Eingangsgrößen die Schwelle, bei der die maximal zulässige Bremsleistung des Systems nicht mehr ausreicht und der Fahrer eingreifen muss. Ein fehlerhaftes Verhalten wurde herbeigeführt, indem Parameter eines Reglers in diesem Modul durch einen Verstärkerblock verändert wurden. Durch den injizierten Fehler erschien die Warnung in bestimmten Situationen im Hinblick auf die untersuchte Anforderung zu spät. Diese Situationen sind dadurch charakterisiert, dass das vorausfahrende Fahrzeug stark abbremst, der Abstand zwischen den Fahrzeugen klein ist und sich das nachfolgende Fahrzeug schneller als eine bestimmte Geschwindigkeit bewegt.

ERGEBNISSE

Insgesamt wurden mit OLET, MbET und dem Zufallstest jeweils fünf Experimentreihen[7] durchgeführt. Abbildung 6.9 zeigt den durchschnittlichen Fitnessverlauf der Experimentreihen. MbET erreichte das Optimierungsziel im Durchschnitt in der 75. Generation und im besten Fall in der 45. Generation (Fitnesswert kleiner als Null). Das Verfahren lieferte somit als Ergebnis einen Testfall, der ein Fehlverhalten der Software belegt. Zum Zeitpunkt des Erscheinens der Warnung kann ein Unfall nicht mehr durch ein Bremsmanöver verhindert werden. Bei OLET[8] verhält es sich ähnlich: Im Durchschnitt dauerte es bis zur 87. Generation und im besten Fall bis zur 50. Generation, bis ein entsprechender Testfall gefunden wurde. Die Rechenzeit verkürzte sich proportional um die Anzahl der Generationen, die nicht mehr ausgeführt werden mussten. Nach der Behebung des Fehlers sollte der Tester zuerst überprüfen, ob der zuvor identifizierte Testfall kein Fehlverhalten mehr auslöst. Um das Vertrauen in die Software weiter zu steigern, sollte danach ein erneuter Optimierungslauf durchgeführt werden. Bei den Experimenten mit dem Zufallstest wurde kein Fehlverhalten identifiziert.

7 Die maximale Anzahl von Generationen wurde für die Fehlerinjektionstests auf 150 erhöht, um die Errechnung von Durchschnittswerten zu ermöglichen.

8 Die Fitnesswerte für den *open-loop*-Test wurden für eine bessere Vergleichbarkeit mit einem konstanten Faktor skaliert.

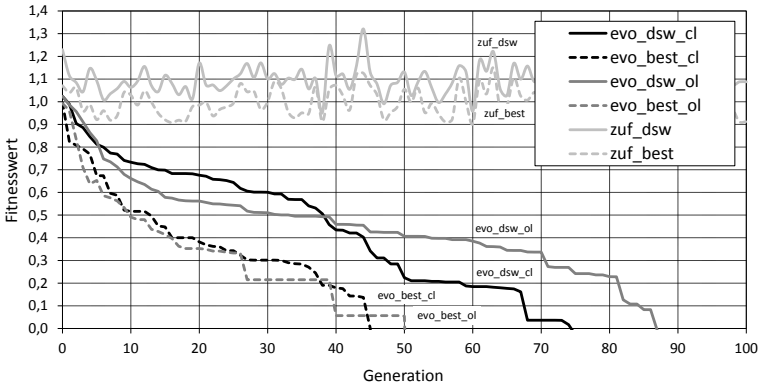


Abbildung 6.9: Fitnessverlauf bei injiziertem Fehler in das ART-Modell, im Vergleich der *open-loop* (ol)-, *closed-loop* (cl)- und der Zufallstest (zuf). Es wurden jeweils fünf Optimierungsläufe durchgeführt.

6.2.7 MESSWERTE DER ABHÄNGIGEN VARIABLEN

Abschließend werden die Ergebnisse der Messung der abhängigen Variablen bei der Durchführung der ART-Fallstudie präsentiert. In Tabelle 6.2 sind die Ergebnisse für die Variablen A1 bis A8 für die suchbasierten Testverfahren zusammengefasst (Variablen A1 bis A5 sind Durchschnittswerte für einen Optimierungslauf). Zusätzlich werden nachfolgend die Ergebnisse im Zusammenhang mit den Variablen A9 und A10 vorgestellt.

Variable A9 bezieht sich auf die Verbesserung der Fitnesswerte. Die Abbildungen 6.7, 6.8 und 6.9 veranschaulichen, dass mit den evolutionären Testverfahren eine große Anzahl problemorientierter Testfälle gefunden werden konnte. Die Kritikalität der Testfälle, repräsentiert durch den Fitnesswert, nahm im Verlauf der Optimierung signifikant zu. Im Vergleich zu den Zufallstests konnten Testfälle mit deutlich besseren Fitnesswerten identifiziert werden. Das Ergebnis der Optimierungsläufe waren Testfälle, bei denen die Warnung spät ausgelöst wurde. Trotz der hohen Anzahl von zielgerichteten Testfällen wurde – abgesehen von den Fehlerinjektionstests – keine Verletzung der untersuchten Anforderung aufgezeigt. Ein weiteres Ergebnis der Fallstudien ist ein erhöhtes Vertrauen in die korrekte Funktionalität der Software.

Variable A10 ist die Messgröße für die Ermittlung des Aufwands für die Bestimmung eines geeigneten Parametersatzes für die evolutionäre Engine. Für die Bestimmung der Parametersätze wurden Experimente mit Benchmark-Funktionen durchgeführt (siehe Anhang A). Insgesamt wurden 8 Arbeitsstunden aufgewendet, um die Parameter aus Tabelle 6.1 zu bestimmen. Für weitere Experimente können diese Parameter wiederverwendet werden und müssen nicht erneut bestimmt werden. Analog zu der Vorgehensweise bei der ART-Fallstudie wird im nächsten Abschnitt ein aktiver Notbremsassistent (ABA) untersucht.

	OLET	MbET
Evaluierte Testfälle (A1)	10.000	10.000
Evaluierte Testfälle FIT (A1′)	8.700	7.500
Ungültige Testfälle (A2)	0	0
Aufgedeckte Fehler (A3)	0	0
Aufgedeckte Fehler bei FIT (A3′)	100%	100%
Überdeckungsgrad (A4)	79% (C_1) & 85% (D_1)	79% (C_1) & 85% (D_1)
Rechenzeit (A5)	28h	28h
Rechenzeit FIT (A5′)	24h	21h
Codezeilen/Zustände für Suchraumspezifikation (A6)	78 (LoC)	6 (Zustände)
Codezeilen der Fitnessfunktion (A7)	63 (LoC)	58 (LoC)
Codezeilen des Testtreibers (A8)	94 (LoC,MiL) 192 (LoC,HiL)	62 (LoC)

Tabelle 6.2: Messergebnisse der abhängigen Variablen der ART-Fallstudie mit FIT: Fehlerinjektionstests.

6.3 DATENERHEBUNG ABA-FALLSTUDIE

Dieser Abschnitt beginnt mit einer kurzen Beschreibung des Testobjekts – ein aktiver Notbremsassistent (Abschnitt 6.3.1) – und einer Auflistung der spezifischen Voraussetzung für die Durchführung eines evolutionären Funktionstests: Suchraum (Abschnitt 6.3.2) und Fitnessfunktion (Abschnitt 6.3.3). In Abschnitt 6.3.4 werden die Ergebnisse der MiL Tests vorgestellt. Auch beim ABA handelt es sich um ein System, das bereits den Stand der Serienreife erreicht hat. Es konnte aber zusätzlich auf einen früheren Entwicklungsstand aus dem Jahr 2009 zurückgegriffen werden. Ergebnisse von Experimenten mit dem früheren Entwicklungsstand werden in Abschnitt 6.3.5 beschrieben. Abschließend werden die Messwerte der abhängigen Variablen diskutiert (Abschnitt 6.3.6).

6.3.1 SYSTEMBESCHREIBUNG

Die Hauptaufgabe des ABA ist es, in kritischen Situationen im letztmöglichen Moment einen Unfall durch eine Vollbremsung zu verhindern oder zumindest die Unfallfolgen zu verringern. Zusätzlich wird der Fahrer vor der Auslösung durch optische und akustische Warnmeldungen auf mögliche Gefahrensituation hingewiesen. Ursache für die Auslösung können Unachtsamkeiten seitens des Fahrers aufgrund von Müdigkeit, falsch eingeschätzte Verkehrssituationen oder riskante Fahrweisen anderer Verkehrsteilnehmer sein. Verbaut wird ABA hauptsächlich in Lastkraftwagen, kommt aber vereinzelt auch in Personenkraftwagen zum Einsatz. Der Einbau in Lastkraftwagen ist vor allem durch die weiten zurückgelegten Strecken und die hohe kinetische Energie im Falle eines Aufpralls motiviert. Die erste Version des ABA ist bei der Daimler AG im Jahr 2006 in Serie gegangen. Mittlerweile sind bereits mehr als 10.000 Fahrzeuge mit dem System im Einsatz. In dieser Fallstudie wird die Nachfolgegeneration betrachtet, die im Jahr 2011 den Vorgänger ablösen wird. Auch bei ABA wird ein Radarsystem für die Abstandsbestimmung zu vorausfahrenden Fahrzeugen verwendet.

Das Warn- und Bremsverhalten des Systems entspricht einer Kaskade: Je nach Kritikalität einer Situation wird zuerst eine optische und eine akustische Warnung, dann ein leichter Bremseingriff

und als letzter Schritt eine Notbremsung ausgelöst (siehe Abbildung 6.10). Die Kaskade wird automatisch abgebrochen, falls eine Situation nicht mehr kritisch ist oder falls der Fahrer, zum Beispiel durch einen deutlichen Lenkeingriff, einen Abbruchwunsch zum Ausdruck bringt.

KOMPLEXITÄT DES ABA-MODELLS

Der verwendete ABA wurde mit der Modellierungssprache Matlab/Simulink implementiert und besteht aus 4.832 Blöcken [110, 111]. Interne Systemzustände wurden mit der Erweiterung Stateflow modelliert und gliedern sich in 6 Zustandsdiagramme mit insgesamt 37 Zuständen [112]. Das Testobjekt kommuniziert mit dem Umgebungsmodell über 71 Eingänge und 31 Ausgänge. ABA weist, wie auch der ART, eine vergleichsweise hohe Komplexität zu anderen Systemen im Bereich der Software eingebetteter Systeme auf. Für die Simulation aller weiteren Fahrzeugkomponenten, der Straße und anderen Verkehrsteilnehmer wurde ein Umgebungsmodell verwendet. Dieses Umgebungsmodell beschränkte sich auf die für die Durchführung der Fallstudie erforderliche Funktionalität.

FAHRSZENARIO

Das Fahrszenario für die Experimente war wie folgt aufgebaut: Ein Lastkraftwagen mit dem aktiven Notbremsassistenten fährt mit konstanter Geschwindigkeit. Hinter einer Kurve erscheint ein abbremsendes Fahrzeug – zum Beispiel aufgrund eines Staus. Nach der Erkennung des vorausfahrenden Fahrzeugs fahren beide Fahrzeuge abwechselnd schneller und langsamer, bis der Testfall terminiert. Der Testfall terminiert sobald eine Notbremsung ausgelöst wird oder wenn die maximale Dauer des Testfalls erreicht ist. In Abbildung 6.10 wird das Fahrszenario skizziert.

TESTZIEL

Das Testziel war, einen Testfall zu finden, bei dem der Notbremsassistent zu spät in Bezug auf die Systemspezifikation bremste. Ein vermeidbarer Unfall wäre dann nicht durch das System verhindert worden. Die konkrete Anforderung aus der Systemspezifikation

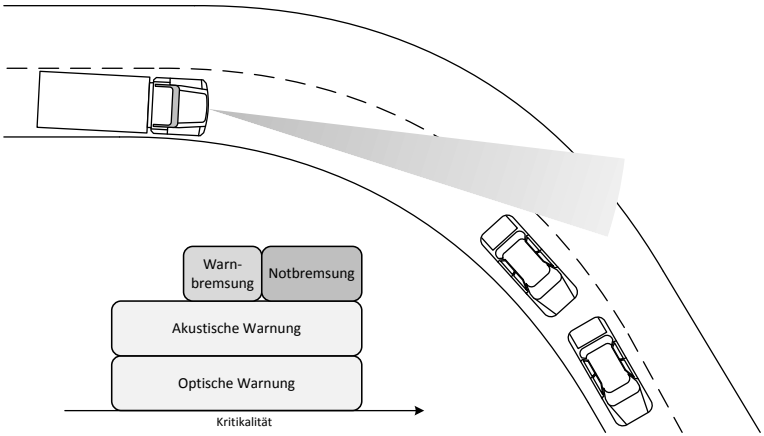


Abbildung 6.10: Fahrszenario für die ABA-Experimente. Dargestellt wird auch die Warnkaskade: Mit zunehmender Kritikalität löst ABA eine optische - und akustische Warnung, eine Warnbremsung und dann eine Notbremsung aus.

wird durch Formel 6.9 beschrieben. Mit der Formel wird die erforderliche negative Beschleunigung ermittelt, um in kritischen Situationen einen Zusammenstoß zu vermeiden.

6.3.2 BESCHREIBUNG DES SUCHRAUMS

Die für den Test des ABA generierten Teststimuli umfassten die Geschwindigkeit des Lastkraftwagens, das Fahrverhalten des vorausfahrenden Fahrzeugs und die Kurvenkrümmung der Straße beziehungsweise den damit einhergehenden Erkennungszeitpunkt des vorausfahrenden Fahrzeugs. Von dem Radarwinkel des ABA und dem Kurvenradius lässt sich der Zeitpunkt der Erkennung ableiten. Bei der Testausführung musste durch die Suchraumbeschreibung sichergestellt werden, dass sich die Fahrzeuge nur innerhalb der Grenzen des physikalisch Möglichen bewegen. Nachfolgend wird vorgestellt, wie die Suchraumbeschreibungen für OLET und MbET realisiert wurden.

Signal-spezifikation

```
<signale fT="100" tmin="20" tmax="40">
<signal name="Vorausfahrend" nmin="3" nmax="6">
   <amplitude min="0" max="100"/>
   <breite min="1" max="3"/>
   <transitionen>
      <t name="spline"/>
      <t name="linear"/>
   </transitionen>
</signal>
<signal name="Fahrer" type="integer" nmin="4" nmax="8">
   <amplitude min="0" max="100"/>
   <breite min="1" max="3"/>
   <transitionen>
      <t name="spline"/>
      <t name="linear"/>
   </transitionen>
</signal>
</signale>
```

Abbildung 6.11: Auszug aus der Suchraumbeschreibung für den *open-loop*-Test des ABA.

OPEN-LOOP

Die Geschwindigkeit des vorausfahrenden Fahrzeugs wurde zu Beginn des Szenarios kontinuierlich reduziert und dann ab einem variablen Zeitpunkt zwischen positiver - und negativer Beschleunigung alterniert. Das nachfolgende Fahrzeug fuhr zuerst mit konstanter Geschwindigkeit und alternierte dann auch zwischen moderater Beschleunigung und Abbremsung. Der ABA war aktiviert und konnte zu jeder Zeit durch eine Warn- beziehungsweise eine Notbremsung eingreifen. Abbildung 6.11 zeigt die *Signalspezifikation* für veränderliche Testdatensequenzen. Auch für den Test des ABA ist eine Initialisierungssequenz erforderlich, die direkt von dem Testtreiber erzeugt wird.

CLOSED-LOOP

Abbildung 6.12 zeigt das Testmodell, von dem die Signale für die *closed-loop*-Tests abgeleitet wurden. Nach einer kurzen Beschleunigungsphase wurde der Notbremsassistent durch den *Fahrer*

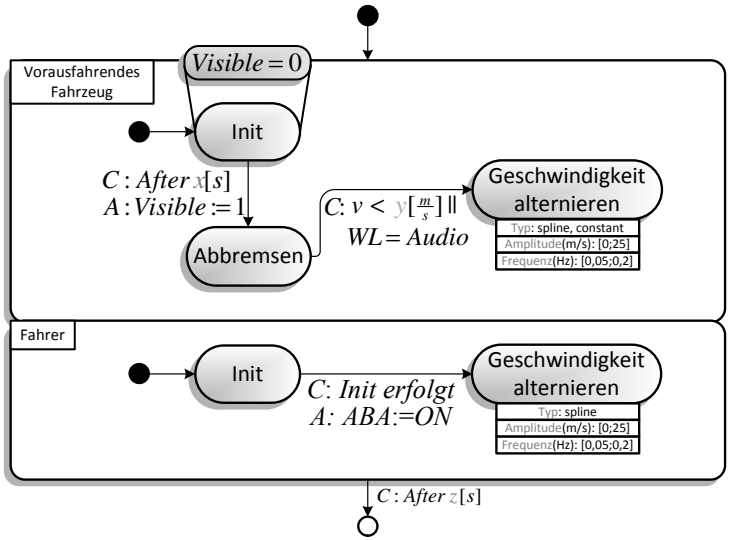


Abbildung 6.12: Suchraumbeschreibung für den *closed-loop*-Test des ABA mit WL: Warnlevel. Parameter in *grau* werden von dem evolutionären Algorithmus innerhalb folgender Schranken variiert: $x \in [0;10]$, $y \in [5;15]$ und $z \in [30;50]$

aktiviert und danach die Geschwindigkeit variiert. Das *vorausfahrende Fahrzeug* wurde aufgrund einer Kurve erst zum Zeitpunkt x im Testverlauf durch den Radar des nachfolgenden Fahrzeugs erfasst. Reaktives Verhalten zeigte sich darin, dass das *vorausfahrende Fahrzeug* mit dem Erscheinen einer optischen Warnung des ABA beginnt, die Geschwindigkeit zu alternieren. Die optische Warnung wurde über einen Ausgang des Testobjekts signalisiert. Dasselbe passierte, wenn die Geschwindigkeit des *vorausfahrenden Fahrzeugs* beim Abbremsen den Schwellenwert y unterschritt. Dieses Vorgehen zielt darauf ab, die Suche in Richtung relevanter Testfälle zu begünstigen.

6.3.3 ERSTELLUNG DER FITNESSFUNKTION

Die Fitnessfunktion steuert die Suche in die Richtung von Testfällen, bei denen eine Kollision durch rechtzeitiges Bremsen hätte

vermieden werden können. Ein Fitnesswert kleiner als Null repräsentiert einen Testfall, bei dem die Notbremsung zu spät im Hinblick auf die Spezifikation erfolgt ist. Folgende Definitionen wurden bei der Beschreibung der Fitnessfunktion verwendet:

- Geschwindigkeit des Eigenfahrzeugs v_{ego} [m/s]: Geschwindigkeit des Fahrzeugs mit dem ABA-System.
- Geschwindigkeit des Zielfahrzeugs v_{ziel} [m/s]: Geschwindigkeit des Fahrzeugs, das vor dem Eigenfahrzeug fährt.
- Relativgeschwindigkeit v_{rel} [m/s]: Zielfahrzeuggeschwindigkeit minus Eigengeschwindigkeit.
- Beschleunigung des Eigenfahrzeugs a_{ego} [m/s^2]: Beschleunigung des Fahrzeugs mit dem ABA-System.
- Beschleunigung des Zielfahrzeugs a_{Ziel} [m/s^2]: Beschleunigung des vorausfahrenden Fahrzeugs.
- Mindestabstand d_{min} [m]: Mindestabstand nach einer Notbremsung zwischen zwei Fahrzeugen.
- Notbremsabstand d_{NB} [m]: Erforderlicher Abstand bei gegebener Beschleunigung und Geschwindigkeit, um einen Unfall zu verhindern.
- Erforderliche Bremsverzögerung a_{NB} [m/s^2]: Erforderliche Bremsverzögerung, um eine Relativgeschwindigkeit von 0 [m/s] unter Berücksichtigung des Mindestabstands zu erreichen.
- Maximale Bremskraft des aktiven Notbremsassistenten $a_{ABA,max}$ [m/s^2]: Begrenzt auf $-6 m/s^2$.

Ausgangspunkt war eine Formel zur Berechnung der erforderlichen Bremsverzögerung $a_S(t)$, um ein Fahrzeug aus der Fahrt zum Stillstand zu bringen:

$$a_S(t) = \frac{v^2(t)}{2 \cdot s(t)} \tag{6.8}$$

Dabei wurde allerdings noch nicht berücksichtigt, dass sich das vorausfahrende Fahrzeug auch bewegt. Um den Zeitpunkt zu bestimmen, bei dem die Notbremsung spätestens ausgelöst werden

muss, wurde die *erforderliche Bremsverzögerung* $a_{NB}(t)$ errechnet. Die *erforderliche Bremsverzögerung* $a_{NB}(t)$ beschreibt die Beschleunigung, die erforderlich ist, um einen Unfall mit einem vorausfahrenden Fahrzeug zu vermeiden. Dafür wurde die Geschwindigkeit aus Formel 6.8 mit der Relativgeschwindigkeit ersetzt und die aktuelle Beschleunigung des vorausfahrenden Fahrzeugs mit einbezogen. Die *erforderliche Bremsverzögerung* $a_{NB}(t)$ errechnete sich folglich als:

$$a_{NB}(t) = a_{ziel}(t) - \frac{v_{rel}^2(t)}{2 \cdot (d(t) - d_{min})} \tag{6.9}$$

Als nächstes wurde Gleichung 6.9 nach d_{NB} (*Notbremsabstand*) aufgelöst. Der *Notbremsabstand* ist der Abstand, in Abhängigkeiten der Relativgeschwindigkeit und Beschleunigungen, bei dem die Notbremsung ausgelöst werden muss:

$$d_{NB}(t) = -\frac{v_{rel}^2(t)}{2 \cdot (a_{ABA,max} - a_{ziel}(t))} + d_{min} \tag{6.10}$$

Eine Notbremsung muss spätestens ausgelöst werden, wenn die Differenz zwischen dem in Gleichung 6.10 errechneten *Notbremsabstand* und dem realen Abstand Null ist. Ist die Differenz negativ, würde die Notbremsung einen Unfall nicht mehr verhindern können und es läge somit eine Anforderungsverletzung vor. Bei der Fitnessberechnung wurden Zeitpunkte, zu denen die Notbremsung aktiv ist, nicht berücksichtigt. Folglich ergab sich für die finale Fitnessfunktion (mit einem großen Wert δ):

$$Fitnesswert = min(d_{NB}(t) - d(t) + \delta \cdot NB_{akt}(t)) \tag{6.11}$$

Bei Formel 6.11 wird nicht der Sonderfall berücksichtigt, wenn das vorausfahrende Fahrzeug beim Abbremsen zum Stehen kommt. Aus Gründen der Geheimhaltung wird hier auf eine weitere Ausführung verzichtet.

In Abbildung 6.13 wird das Prinzip der Fitnessberechnung grafisch veranschaulicht. Spätestens beim Schnittpunkt der beiden Verläufe muss die Notbremsung ausgelöst werden – das heißt,

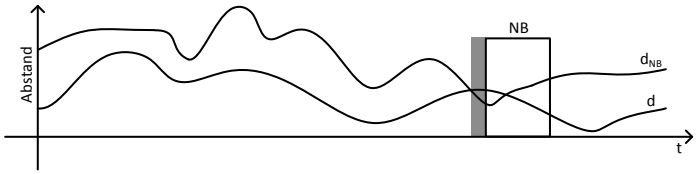


Abbildung 6.13: Fitnessberechnung für einen ABA-Testfall mit d: Abstand, NB: Notbremsung und d_{NB}: Notbremsabstand.

wenn der Abstand d gleich dem *Notbremsabstand* d_{NB} ist. Nachfolgend werden die Besonderheiten bei der Implementierung der Fitnessfunktion für OLET und MbET vorgestellt.

Für OLET musste Formel 6.11 mit der Sprache des Testobjekts (Matlab) oder der Sprache des Testtreibers (JAVA) implementiert werden. Dabei mussten allerdings einige Sonderfälle des ABA berücksichtigt werden – beispielsweise bremst ABA nur dann, wenn eine Mindestrelativgeschwindigkeit überschritten wird. Weitere Sonderfälle werden hier aus Gründen der Geheimhaltung nicht weiter diskutiert.

Für MbET wurde ausgehend von Gleichung 6.11 ein Auswerteskript erstellt, das der Bewertung von Testläufen diente. Das Auswerteskript wurde automatisch instrumentiert und dann mit den aufgezeichneten Testdaten aufgerufen. Damit das Auswerteskript instrumentiert werden konnte, musste im Auswerteskript ein Ziel definiert werden (entspricht einer Zeile im Auswerteskript). Durch die Instrumentierung wurde automatisch der Abstand zu dem ausgewählten Ziel berechnet. Auswerteskript 2 zeigt einen Ausschnitt aus dem instrumentierten Auswerteskript[9] für die ABA-Experimente.

6.3.4 Model-in-the-Loop-Experimente

In diesem Abschnitt werden die Ergebnisse von Experimenten mit dem evolutionären Funktionstest im Vergleich zum Zufallstest aufgezeigt. Die Experimente wurden mit der Konfiguration aus Tabelle 6.1 durchgeführt. Auch bei der ABA-Fallstudie musste der

9 Das vollständige Auswerteskript ist vertraulich und kann nur von Mitarbeitern der Daimler AG eingesehen werden.

Auswerteskript 2 returns FitnessWert

Input: $v_{rel}, v_{rel,min}, a_{ego}, a_{tar}, a_{ABA,max}, d, NB_{akt}$

1: $d_{NB}(t) = -v_{rel}^2(t)/(2(a_{NB}(t) - a_{Ziel})) + d_{min}$
2: **if** $NB_{akt}! = 1$ **then**
3: **if** $v_{rel}(t) > v_{rel,min}$ **then**
4: **if** $d(t) < d_{NB}(t)$ **then**
5: $FW = -1$
6: **else**
7: $FW = 1 - 1,001^{-|d(t)-d_{NB}(t)|}$
8: **end if**
9: **else**
10: $FW = 2 - 1,001^{-|v_{rel}(t)-v_{rel,min}|}$
11: **end if**
12: **else**
13: $FW = 3$
14: **end if**

Testtreiber um die Verlinkung der Eingangs- und Ausgangskanäle des Testobjekts mit der Komponente für die Erzeugung der Testdatensequenzen und der Weiterleitung der aufgezeichneten Daten an die Bewertungsfunktion erweitert werden.

Abbildung 6.14 zeigt einen exemplarischen Testfall, der bei einem Optimierungslauf mit MbET identifiziert wurde. Die beiden oberen Diagramme zeigen Werte für Geschwindigkeit und Beschleunigung der beiden Fahrzeuge – Größen, die für die Berechnung des Notbremsabstands d_{NB} benötigt werden. In dem unteren Diagramm ist zu erkennen, wie mit zunehmender Kritikalität einer Situation die Warnkaskade durchlaufen wird (siehe Abbildung 6.10). Der Fahrer wird zuerst durch eine optische und akustische Warnung (*OW/AW*) auf eine potentiell gefährliche Situation hingewiesen. Nimmt die Kritikalität weiter zu, wird der Fahrer zusätzlich durch eine Warnbremsung (*WB*) gewarnt und schließlich wird als letzter Schritt von dem System eine Notbremsung ausgelöst, um einen Unfall zu verhindern.

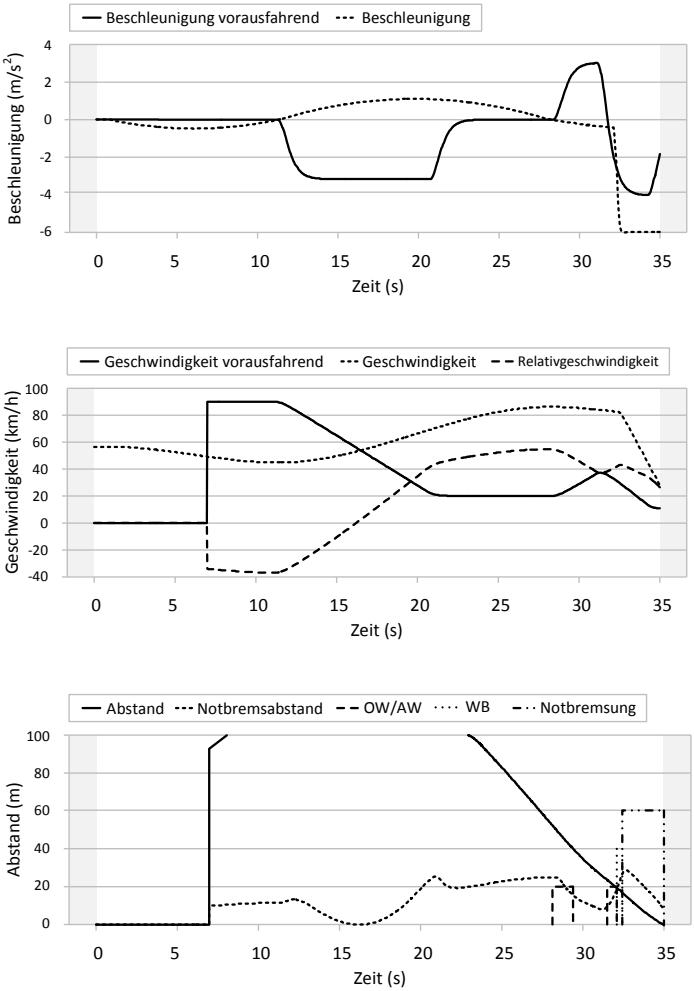


Abbildung 6.14: Generierte Signale und Ausgangsverhalten eines Fahrmaneuvers mit dem ABA. OW/AW steht für optische und akustische Warnung und WB steht für Warnbremsung.

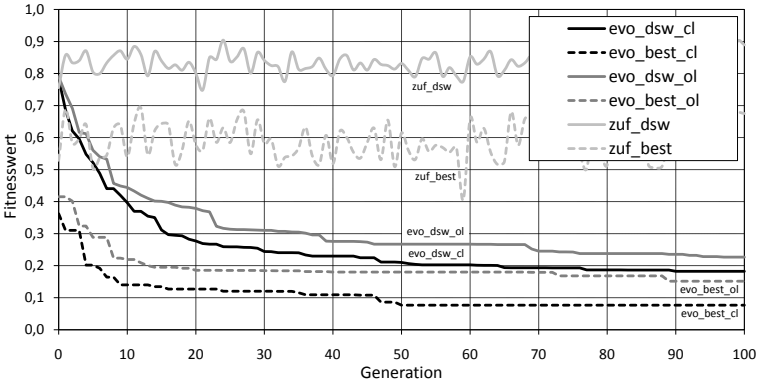


Abbildung 6.15: Fitnessverlauf der ABA-Experimente. Dargestellt sind die Ergebnisse mit dem *open-loop* (ol)- und *closed-loop* (cl)-Verfahren im Vergleich zu dem Zufallstest (zuf). Die Experimente wurden mit jeder Konfiguration zehnmal wiederholt.

Abbildung 6.15 zeigt den durchschnittlichen Fitnessverlauf. Auch hierbei wurden jeweils zehn Optimierungsdurchläufe mit OLET[10] und MbET sowie mit dem Zufallstest durchgeführt. Erwartungsgemäß schnitten die evolutionären Suchverfahren deutlich besser ab als der Zufallstest. Der finale Fitnesswert der evolutionären Testverfahren war mit hoher statistischer Signifikanz besser, als der des Zufallstests ($P < 0{,}001$). Trotz der hohen Anzahl ausgeführter Testfälle unterschritt der Zufallstest keinen Wert, der von den suchbasierten Verfahren im Durchschnitt bereits nach wenigen Generationen unterschritten wurde. Das etwas schlechtere Abschneiden von OLET im Vergleich zu MbET ist darauf zurückzuführen, dass nicht auf das Erfolgen der Warnung reagiert werden konnte (siehe Abbildungen 6.11 und 6.12). Der Unterschied zwischen MbET und OLET ist aber nicht statistisch signifikant ($P < 0{,}05$).

Für die ABA-Fallstudie wurden insgesamt 10 [Optimierungsläufe] $\cdot 100$ [Generationen] $\cdot 100$ [Individuen] $= 100.000$ [Testfälle] für jede Konfiguration ausgeführt. Bei einer Ausführungsdauer von ≈ 11s pro Testfall ergab das eine gesamte Ausführungszeit (ohne Parallelisierung) von 100.000 [Testfällen] $\cdot 11$s [Dauer je Testausführung]

10 Die vom OLET-Verfahren errechneten Fitnesswerte wurden für eine bessere Vergleichbarkeit mit einem konstanten Faktor skaliert.

$= 1.100.000s$ was ungefähr 306 Stunden entspricht, beziehungsweise 30, 6 Stunden für einen einzelnen Optimierungslauf. Für die Durchführung der Zufallstests wurde ein vergleichbarer Zeitaufwand benötigt.

Während der Durchführung der Optimierungsreihen mit dem ABA-Modell wurde der Überdeckungsgrad gemessen. Mit dem *closed-loop*-Verfahren wurde ein Überdeckungsgrad von 89% für die C_1-Zweigüberdeckung und 90% für die D_1-Bedingungsüberdeckung ermittelt. Beim *open-loop*-Verfahren und bei den Zufallstests ergaben die Messungen einen etwas geringeren Wert: 88% für die C_1- und auch für die D_1-Überdeckung.

6.3.5 Experimente mit älterem Entwicklungsstand

Um die Effektivität der Testverfahren besser bewerten zu können, werden nun die Ergebnisse von Experimenten mit einem früheren Entwicklungsstand vorgestellt. Konkret wurde ein Entwicklungsstand der ABA-Software verwendet, bei dem im weiteren Verlauf der Qualitätssicherung noch Fehler identifiziert werden konnten. Die MiL-Experimentreihen wurden ein weiteres Mal durchgeführt, um zu untersuchen, ob ein Fehlverhalten mit dem evolutionären Funktionstest und dem Zufallstest durch einen entsprechenden Testfall aufgezeigt werden kann. Die untersuchte Anforderung besagt, dass die Notbremsung rechtzeitig erfolgen muss, um einen Unfall zu verhindern. Ausgenommen sind Situationen, bei denen ein Unfall im Rahmen der physikalischen Grenzen nicht mehr vermieden werden kann (zum Beispiel, wenn ein langsam fahrendes Fahrzeug bei geringem Abstand einschert).

Ergebnisse

Ein Fehlverhalten zeichnete sich wie folgt aus: Durch Formel 6.10 wurde der Zeitpunkt festgelegt, zu dem die Notbremsung spätestens ausgelöst werden musste. Etwaige Fehler mussten den Auslösezeitpunkt der Warnung beeinflussen. Damit die Suche nicht bereits bei einem Grenzwert abbrach, wurde zusätzlich noch eine Toleranz von 20% berücksichtigt. Dieser Toleranzfaktor wurde auf den Notbremsabstand d_{NB} aufaddiert. Spätestens wenn der Notbremsabstand d_{NB} gleich dem realen Abstand ist, muss

die Notbremsung ausgelöst werden (vergleiche Abbildung 6.13). Berechnet man die erforderliche Beschleunigung a_{NB} (Formel 6.9) mit den Werten zum Zeitpunkt der Auslösung der Notbremsung, ergibt sich folglich bei einem fehlgeschlagenen Test ein Wert kleiner als $a_{ABA,max}$. Als alternatives Abbruchkriterium könnte auch einzig das Erreichen der maximalen Anzahl von Generationen berücksichtigt werden. Das Ergebnis der Suche wäre dann der maximale zeitliche Abstand zwischen dem in der Anforderung spezifizierten Auslösezeitpunkt und dem tatsächlichen Zeitpunkt der Notbremsauslösung. Hier wurde auf die zuerst beschriebene Variante zurückgegriffen.

Im Durchschnitt brach die Suche nach 62 Generationen beim *closed-loop*-Verfahren und nach 69 Generationen beim *open-loop*-Verfahren[11] ab. Im besten Fall waren es 34 Generationen beim *closed-loop*-Test und 54 Generationen beim *open-loop*-Test. Bei den Zufallstests konnte kein Fehlverhalten aufgezeigt werden (kein Abbruch vor dem Erreichen der 100. Generation). Allerdings konnte bei einem Durchlauf in der 73. Generation ein Testfall mit einem vergleichsweise niedrigen Fitnesswert identifiziert werden. Abbildung 6.16 zeigt den durchschnittlichen Fitnessverlauf, gemittelt über je fünf Optimierungsläufe.

Im Anschluss an die Durchführung der Optimierungsläufe wurde eine manuelle Auswertung der Szenarien durchgeführt, die ein Fehlverhalten aufzeigten. Als Ergebnis konnte festgestellt werden, dass die Szenarien dadurch charakterisiert waren, dass nach einer erfolgten optischen oder haptischen Warnung das vorausfahrende Fahrzeug erst stark beschleunigte und dann wieder abrupt abbremste. Die Warnung wurde beim Fahrzeug mit dem ABA deaktiviert und hätte kurz danach wieder aktiviert werden müssen. Aufgrund hoher Mindestaufenthaltsdauern in den Warnstufen, wurde die Notbremsung verzögert ausgeführt. Mindestaufenthaltsdauern wurden festgelegt, um das Problem des Prellens zu umgehen. Deren Dauer musste dann aber reduziert werden, um das Einhalten der Anforderung sicherzustellen.

11 Die Fitnesswerte für den *open-loop*-Test wurden für eine bessere Vergleichbarkeit mit einem konstanten Faktor skaliert.

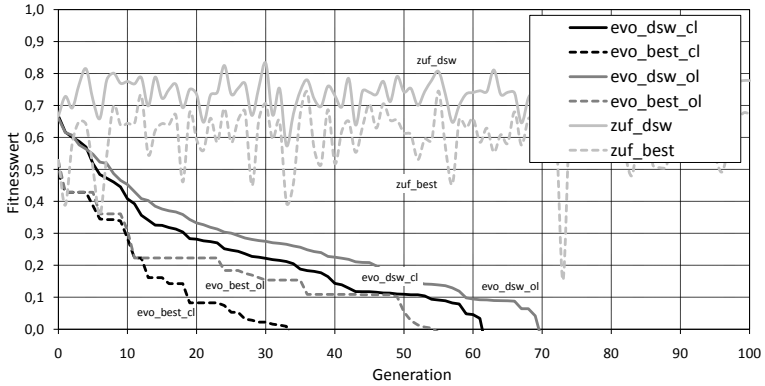


Abbildung 6.16: Fitnessverlauf bei Experimenten mit älterem ABA Entwicklungsstand im Vergleich der *open-loop* (ol)-, der *closed-loop* (cl)- und der Zufallstest (zuf). Es wurden jeweils fünf Optimierungsläufe durchgeführt.

6.3.6 Messwerte der abhängigen Variablen

Ergebnisse für die Variablen A1 bis A8 werden in Tabelle 6.3 zusammengefasst (Variablen A1 bis A5 sind Durchschnittswerte für einen Optimierungslauf). Ergebnisse im Zusammenhang mit den Variablen A9 und A10 werden nachfolgend vorgestellt.

Variable A9 ist ein Messwert für die Verbesserung des Fitnesswerts im Verlauf der Optimierung. Anhand der Diagramme 6.15 und 6.16 lässt sich erkennen, dass die Kritikalität der gefundenen ABA-Testfälle – repräsentiert durch den Fitnesswert – im Verlauf der Optimierung zugenommen hat. Bei der evolutionären Suche konnten erwartungsgemäß bessere Testfälle gefunden werden, als beim Zufallstest. Da kein negativer Fitnesswert aufgetreten ist, ist das Ergebnis auch hier ein Vertrauensgewinn in die korrekte Funktionalität der Software. Bei den Testläufen mit einem älteren Entwicklungsstand konnten dagegen Testfälle mit Fitnesswerten kleiner als Null gefunden werden. Mit Hilfe der Testfälle kann das Testobjekt überarbeitet werden, um ein derartiges Fehlverhalten in der finalen Version auszuschließen.

Für die evolutionäre Engine wurde derselbe Parametersatz verwendet, wie für die ART-Fallstudie. Es ergab sich demzufolge für

Variable A10 ein Aufwand von 8 Arbeitsstunden. Sowohl für die ART- als auch die ABA-Experimente konnten mit den ermittelten Parametern eine kontinuierliche Verbesserung der Fitnesswerte gemessen werden und somit positive Rückschlüsse auf die verwendeten Parameter gezogen werden.

	OLET	MbET
Evaluierte Testfälle (A1)	10.000	10.000
Evaluierte Testfälle fehlerbehaftetes TO (A1′)	6.900	6.200
Ungültige Testfälle (A2)	0	0
Aufgedeckte Fehler (A3)	0	0
Aufgedeckte Fehler fehlerbehaftetes TO (A3′)	100%	100%
Überdeckungsgrad (A4)	88% (C_1) & 88% (D_1)	89% (C_1) & 90% (D_1)
Rechenzeit (A5)	31h	31h
Rechenzeit bei fehlerbehaftetem TO (A5′)	21h	19h
Codezeilen/Zustände für Suchraumspezifikation (A6)	55 (LoC)	5 (Zustände)
Codezeilen der Fitnessfunktion (A7)	102	93
Codezeilen des MiL-Testtreibers (A8)	85	53

Tabelle 6.3: Messergebnisse der abhängigen Variablen der ABA-Fallstudie mit TO:Testobjekt.

6.4 BEWERTUNG DER PROPOSITIONEN

Basierend auf den Ergebnissen der Experimente mit dem Abstandsregeltempomaten (Abschnitt 6.2) und dem aktiven Notbremsassistenten (Abschnitt 6.3) werden OLET und MbET in Bezug auf die Propositionen (Abschnitt 6.1.3) bewertet. Grundlage der Bewertung sind die im Rahmen der Fallstudien gewonnenen Erkenntnisse und insbesondere auch die abhängigen Variablen (Abschnitt 6.1.3). Ziel ist die Bewertung der Effektivität und Effizienz von OLET und MbET und auch ihrer Anwendbarkeit in einem industriellen Kontext.

Mit Proposition P1 wurde die Behauptung aufgestellt, dass durch die Kombination eines suchbasierten mit einem modellbasierten Testverfahren ein automatisiertes und reaktives Testverfahren für Software eingebetteter Systeme realisiert werden kann. Anhand der Fallstudien wurde gezeigt, dass mit MbET automatisiert eine Vielzahl realistischer Testdatensequenzen erzeugt werden konnten (Variablen A1 und A2). Bei Fehlerinjektionstests und Experimentreihen mit einem älteren Entwicklungsstand konnten zudem Fehler bei den Testobjekten aufgezeigt werden (Variable A3'). Da MbET ein reaktives Testverfahren realisiert, wird der exakte Verlauf der Testdatensequenzen erst während der Testausführung ermittelt. Das bisherige Ausgangsverhalten des Testobjekts und die Testzeit werden bei der Erstellung der Teststimuli mit einbezogen. Im Vergleich zu dem *open-loop*-Verfahren OLET konnte beobachtet werden, dass mit MbET durch den reaktiven Ansatz das Suchziel schneller erreicht wird (Variable A9).

Proposition P2 bezieht sich auf etwaige Probleme bei der Anwendbarkeit des evolutionären Funktionstests auf komplexe Softwaresysteme aus der Industrie. Insbesondere die Vielzahl interner Systemzustände von Software eingebetteter Systeme kann die Suche nach relevanten Testdaten erschweren. Für die Fallstudien wurden zwei komplexe Softwaresysteme verwendet (Abschnitte 6.2.1 und 6.3.1). Der gemessene Fitnessverlauf (Variable A9), die hohe Anzahl ausgeführter Testfälle (Variable A1), die Rechenzeit von ungefähr einem Tag (Variable A5) und insbesondere auch das Auffinden von Fehlern (Variable A3') verdeutlichen, dass die vorgestellten Verfahren für dieses Anwendungsfeld geeignet sind. Fehler konnten nur mit OLET und MbET gefunden werden, nicht aber mit dem Zufallstest. Die gemessene Überdeckung (Variable

A4) gibt Aufschluss darüber, dass bei der Suche ein großer Anteil der Struktur der Testobjekte durchlaufen wurde. Die identifizierten Testfälle können auch für den Strukturtest weiterverwendet werden und somit das Erreichen einer vollständigen Überdeckung signifikant beschleunigen. Durch die Bewertung dieser Proposition wird die besondere Stärke des evolutionären Funktionstests gegenüber der formalen Verifikation deutlich. Derartige Verfahren versagen bei vergleichbaren Problemstellungen aufgrund der kombinatorischen Explosion der Zustandsräume.

Proposition P3 besagt, dass der evolutionäre Funktionstest effektiver beim Auffinden von Fehlern in Software eingebetteter Systeme ist, als der Zufallstest. Beide in den Fallstudien verwendeten Systeme stammen aus der Serienentwicklung. Es ist deshalb nicht verwunderlich, dass weder mit den Verfahren für den evolutionären Funktionstest noch mit dem Zufallstest ein Fehlverhalten aufgezeigt werden konnte. Dennoch ist anhand der Fitnessverläufe (Variable A9) klar zu erkennen, dass mit dem Zufallstest keine vergleichbaren Ergebnisse erzielt werden konnten (hohe statistische Signifikanz). Bereits nach wenigen Generationen waren die im Durchschnitt mit OLET und MbET ermittelten Fitnesswerte besser als die Besten des Zufallstests. Weiterhin konnte anhand von Fehlerinjektionstests (ART) und Tests mit einem fehlerbehafteten Entwicklungsstand (ABA) demonstriert werden, dass die evolutionären Testverfahren effektiver beim Auffinden von Fehlern sind, als der Zufallstest (Variable A3'). Im Gegensatz zu den evolutionären Testverfahren konnten beim Zufallstest auch bei diesen Testobjekten keine Testfälle gefunden werden, die ein Fehlverhalten aufzeigen.

Mit Proposition P4 wird behauptet, dass der evolutionäre Funktionstest auch für den HiL-Test (Testobjekt ist ein reales Steuergerät) eingesetzt werden kann. Dabei wird auch der Aufwand berücksichtigt, eine MiL-Testspezifikation (Fitnessfunktion, Signalspezifikation und Testtreiber) für einen HiL-Test wiederzuverwenden. Für die Bewertung der Proposition wurde im Rahmen dieser Forschung eine Testumgebung entwickelt (Abschnitt 4.4.3), mit der ein reales Steuergerät mit den Techniken des evolutionären Funktionstests getestet werden kann. Für die Fallstudie wurde ein in der Serienproduktion hergestelltes Steuergerät – mit der Software des Abstandsregeltempomats – verwendet (Abschnitt 6.2.5). Erkenntnisse und Ergebnisse der Fallstudie zeigen, dass Such-

raumspezifikation und Fitnessfunktion wiederverwendet werden können, um das Steuergerät an einem Hardwareprüfstand zu testen (Variablen A2 und A8). Es müssen lediglich einige Anpassungen am Testtreiber vorgenommen werden. Vorteil des HiL-Tests ist die Einbeziehung von Hardware-Eigenschaften (z.B. Laufzeitverhalten), allerdings erfolgt die Testausführung in Echtzeit. Auch bei den HiL-Experimenten konnten mit OLET schneller bessere Fitnesswerte identifiziert werden, als beim Zufallstest (Variable A9).

Schließlich wurde mit Proposition P5 die Behauptung aufgestellt, dass komplexe Testdatensequenzen effektiv im Hinblick auf das Auffinden von Fehlern und für einen industriellen Kontext angemessen, im Hinblick auf den Aufwand für den Tester, erzeugt werden können. Mit Hilfe von Fehlerinjektionstests (ART) und Tests mit einem fehlerbehafteten Entwicklungsstand (ABA) wurde gezeigt, dass mit den von MbET und OLET automatisch generierten Testdatensequenzen ein Fehlverhalten aufgezeigt werden kann (Variable A3'). Abbildungen 6.6 und 6.14 zeigen Beispiele für Testfälle, die im Rahmen der evolutionären Suche ausgeführt wurden. Bei beiden Verfahren muss jeweils nur eine Spezifikation erstellt werden, die Variation der Testdatensequenzen erfolgt automatisch durch die evolutionären Algorithmen. Es wurde eine Vielzahl komplexer und zielgerichteter Testfälle vollautomatisch ausgeführt.

Im Rahmen dieser Arbeit war es nicht möglich, die Fallstudien unter Einbeziehung einer repräsentativen Anzahl von Testern aus dem Bereich der Qualitätssicherung von Software eingebetteter Systeme durchzuführen. Eine klare Aussage, ob der Aufwand für die Durchführung eines evolutionären Funktionstests in der Industrie als zu hoch angesehen wird, kann folglich nicht getroffen werden. Eine erste Einschätzung bieten aber die erforderlichen Codezeilen für die testobjektspezifischen Anteile des Experimentaufbaus (Variablen A6, A7 und A8). Der Umfang ist vergleichbar mit alternativen Testverfahren aus dem Bereich der Qualitätssicherung von Software eingebetteter Systeme (siehe Anhang C). Es handelt sich dabei allerdings nur um Implementierungen einzelner Testfälle und nicht, wie bei den hier vorgestellten Verfahren, um eine automatisierte Suche. Weiterhin wurde in dieser Arbeit ein Verfahren für die Ermittlung geeigneter Parameter für die Konfiguration der evolutionären Engine diskutiert und der Auf-

wand mit Variable A10 gemessen (siehe Anhang A). Die dabei bestimmten Parameter können für Experimente mit dem EvoTest-Framework sowohl in der Forschung als auch in der industriellen Praxis wiederverwendet werden und erfordern keinen weiteren Aufwand seitens der Tester. Kommt eine alternative evolutionäre Engine zum Einsatz, muss die Experimentreihe aus Anhang A unter Umständen wiederholt werden. Aktuelle Forschung beschäftigt sich außerdem mit dem *Automated Parameter Tuning*, um geeignete Parameter der evolutionären Engine automatisch zur Laufzeit zu bestimmen. Der Aufwand für die Parameterbestimmung würde dann vollständig entfallen [76].

ZUSAMMENFASSUNG

Die Anwendung von OLET und MbET wurde für die Testautomatisierung von Software eingebetteter Systeme untersucht. Als Testobjekte dienten zwei komplexe Systeme aus dem Automobilbereich: Ein Abstandsregeltempomat und ein aktiver Notbremsassistent. Beide Systeme wurden mit den suchbasierten Verfahren getestet und die Ergebnisse mit dem Zufallstest verglichen. Die Suche nach relevanten Testfällen verlief beim evolutionären Test deutlich schneller als beim Zufallstest. Im Rahmen der Durchführung der Fallstudien konnten keine *echten* Fehler bei den Testobjekten aufgedeckt werden. Die Systeme hatten bereits den Status der Serienreife erreicht. Um die Effektivität der Verfahren dennoch zu bewerten, wurden Fehler manuell mittels Fehlerinjektion in das ART-Modell eingebaut und ein fehlerbehafteter Entwicklungsstand des ABA-Modells verwendet. Mit den suchbasierten Verfahren gelang es daraufhin, Testfälle zu identifizieren, die ein Fehlverhalten aufzeigten. Zudem wurde mit MbET die Möglichkeit aufgezeigt, während der Testdurchführung auf das Ausgangsverhalten des getesteten Systems zu reagieren. Aufgrund der unterstützten Reaktivität kann die Suche bei entsprechenden Problemstellungen schneller in relevante Bereiche des Suchraums vordringen. Schließlich wurde der evolutionäre Funktionstest auch für Tests mit einem realen Steuergerät verwendet (OLET). Suchraumbeschreibung und Fitnessfunktion aus den MiL-Tests mussten nicht angepasst werden, sondern konnten plattformübergreifend wiederverwendet werden.

Teil III

SCHLUSS

7 CONCLUSIO

„Die Wissenschaft fängt eigentlich erst da an interessant zu werden, wo sie aufhört."

— Justus von Liebig (1803 - 73)

Das Testen – ein integraler Bestandteil von Softwareentwicklungsprojekten – leistet einen erheblichen Beitrag zur Sicherstellung der Qualität. Der Anteil der Kosten der Qualitätssicherung an den Gesamtkosten großer Entwicklungsprojekte, insbesondere im Bereich eingebetteter Systeme, und auch die Kosten für das nachträgliche Beheben von Fehlern, haben in den vergangenen Jahrzehnten stetig zugenommen. Verfahren zur Testautomatisierung – und damit auch die hierin vorgestellten Verfahren für den evolutionären Funktionstest – wirken diesem Missverhältnis entgegen.

Ausgangspunkt dieser Arbeit war eine Untersuchung des aktuellen Forschungsstands von Black-Box-Testverfahren für Software eingebetteter Systeme und des evolutionären Testens (Kapitel 2 und 3). Basierend auf den daraus gewonnenen Erkenntnissen wurden zwei Verfahren entwickelt: Das *open-loop*-Testverfahren OLET und das *closed-loop*-Testverfahren MbET (Kapitel 4 und 5). OLET und MbET wurden speziell für den Funktionstest von Software eingebetteter Systeme aus dem industriellen Umfeld konzipiert. Die Verfahren zielen insbesondere auf folgende Punkte ab:

- die Erzeugung realistischer Testdatensequenzen,
- die automatisierte Bewertung von Testfällen,
- den Test des reaktiven Verhaltens von eingebetteten Softwaresystemen (nur MbET) und
- die Konzeption einer Testumgebung, die für den Einsatz in einem industriellen Umfeld geeignet ist.

Realistische kontinuierliche Signale werden bei OLET durch die Aneinanderreihung von Standardsignalen (Abschnitt 4.1) und bei MbET mit Hilfe von Testmodellen (Abschnitt 5.1) erzeugt. Im Unterschied zu bisherigen Ansätzen wird bei OLET eine variable Segmentanzahl für die Signale unterstützt. Darüber hinaus kann auch die Länge der Testdatensequenzen variiert werden. Bei MbET werden Testdatensequenzen durch hybride Automaten beschrieben. Der Suchraum wird über variable Parameter in Testmodellen aufgespannt. Vorteile dieses modellbasierten Ansatzes sind ein höherer Abstraktionsgrad, die damit verbundene verbesserte Übersicht und ein höheres Potential für die Wiederverwendbarkeit von Modellen, insbesondere bei komplexen Anwendungen. Allerdings kann die Erstellung der Testmodelle einen höheren Aufwand erfordern. Für eine vertiefende Diskussion der Effizienz der Verwendung von Testmodellen wird hier auf einschlägige Arbeiten verwiesen (zum Beispiel Pretschner [95]). Im Rahmen der Fallstudien mit einem Abstandsregeltempomaten (ART) und einem aktiven Notbremsassistenten (ABA) aus der Serienentwicklung der Daimler AG wurden beide Signalgenerierungsansätze untersucht. Sowohl mit OLET als auch mit MbET konnten realistische Testdatensequenzen für die Simulation physikalischer Zusammenhänge erzeugt werden.

Für die Erstellung einer Fitnessfunktion werden bei OLET mathematische Funktionen implementiert, die basierend auf den aufgezeichneten Testdaten einen Fitnesswert berechnen (Abschnitt 4.2). Bei MbET werden instrumentierte Auswerteskripte verwendet, um Testläufe zu bewerten (Abschnitt 5.2). Mit Techniken des Strukturtests werden Fitnesswerte automatisch normiert. Der Tester wählt ein beliebiges Strukturelement im Auswerteskript aus und die evolutionäre Engine versucht dann automatisch, das Element zu erreichen. Da es sich bei den untersuchten Testdaten um Sequenzen handelt, wird für jeden Zeitschritt ein separater Fitnesswert berechnet. Der jeweils kleinste über den Verlauf eines Testfalls bestimmte Fitnesswert wird in den Optimierungskreislauf zurückgeführt.

Der Begriff *Reaktivität* bezeichnet bei einer Testumgebung die Eigenschaft, den Verlauf der generierten Teststimuli basierend auf dem Ausgangsverhalten des Testobjekts anzupassen. Im Unterschied zu bisherigen Arbeiten im Bereich evolutionärer Funktionstests, wird dieses Konzept von MbET unterstützt (Abschnitt

4.4). MbET wird den aktuellen Anforderungen der Industrie an ein Testwerkzeug gerecht. Bei der Modellierung der Automaten kann auf die Ausgänge des Testobjekts zugegriffen und der weitere Verlauf der Testdatensequenzen angepasst werden (zum Beispiel durch einen Zustandswechsel im Testmodell). Dieses Konzept wurde anhand von Fallstudien demonstriert. Bei der ABA-Fallstudie wurde das vorausfahrende Fahrzeug erst dann durch den Testtreiber beschleunigt, wenn bei dem Fahrzeug mit dem ABA-System eine Warnung ausgegeben wurde. Dadurch konnte die Suche in einen Bereich gesteuert werden, der als besonders kritisch angesehen wurde. Zudem wurde bei der ART-Fallstudie erst mit dem Alternieren der Geschwindigkeit begonnen, als eine durch die Betätigung des Gas-Pedals hervorgerufene Geschwindigkeitsschwelle überschritten wurde.

In den Abschnitten 4.4 und 5.4 wurden Konzepte einer Testumgebung für automatisierte SiL-, MiL- und HiL-Tests vorgestellt. Im Rahmen der Fallstudien (Abschnitte 6.2.4, 6.2.5 und 6.3.4) wurde die Funktionsweise der Testumgebung auf den Testplattformen MiL und HiL überprüft. Dabei konnte aufgezeigt werden, dass der evolutionäre Funktionstest mit lediglich geringen Anpassungen plattformübergreifend eingesetzt werden kann. Der evolutionäre Funktionstest an einem HiL-Aufbau ermöglicht die Miteinbeziehung von Hardware-Eigenschaften der Steuergeräte und Testläufe bei Nichtvorhandensein des Programmcodes und Modells (zum Beispiel wenn der Zugriff auf den Programmcode nur durch einen Zulieferer möglich ist).

Weiterhin wurden die suchbasierten Verfahren im Rahmen der Fallstudien mit dem Zufallstest verglichen (Kapitel 6). Die suchbasierten Testverfahren erzielten deutlich bessere Ergebnisse hinsichtlich der Effizienz und Effektivität. OLET und MbET untereinander lieferten vergleichbare Ergebnisse; MbET konnte allerdings, aufgrund der unterstützten Reaktivität, etwas besser abschneiden. Durch Fehlerinjektionstests (ART) und Tests mit einem älteren Entwicklungsstand (ABA) wurde die Effektivität der suchbasierten Testverfahren aufgezeigt. Fehler konnten einzig mit den suchbasierten Testverfahren gefunden werden – nicht aber beim Zufallstest. Zusammenfassend lässt sich sagen, dass OLET in erster Linie für einfachere Problemstellungen geeignet ist. MbET ist dagegen generell einsetzbar, insbesondere wenn mehrere Anforderungen

in separaten Optimierungsläufen überprüft werden sollen oder ein reaktives Testverfahren erforderlich ist.

Mit den vorgestellten Verfahren werden Testern von Software eingebetteter Systeme effektive Werkzeuge zur Verfügung gestellt, mit denen komplexe Sachverhalten automatisiert überprüft werden können. Von den Testern wird dabei lediglich ein grundlegendes Verständnis der Funktionsweise von Optimierungsverfahren gefordert.

8 WEITERFÜHRENDE ARBEITEN

„Es gibt keine einfachen Lösungen für sehr komplizierte Probleme. Man muss den Faden geduldig entwirren, damit er nicht reißt."

— Michail Gorbatschow (*1931)

Im Folgenden werden Vorschläge für Verbesserungen und Weiterentwicklungen der vorgestellten Verfahren diskutiert. Die Vorschläge sollen die Akzeptanz des evolutionären Funktionstests in der Industrie weiter erhöhen.

Durch den Einsatz von Techniken des Automated Parameter Tunings kann die Konfiguration der evolutionären Engine automatisch an das jeweilige Problem angepasst werden. In dieser Arbeit wurde ein Parametersatz bestimmt, der sich als geeignet für die durchgeführten Fallstudien erwiesen hat. Für andere Probleme kann sich unter Umständen eine andere Konfiguration als effektiver erweisen. Automated Parameter Tuning versucht für jedes Problem eine ideale Konfiguration automatisch zu bestimmen. Einen Überblick über den aktuellen Forschungsstand bietet die Arbeit von Lobo *et al.* [76].

Eine weitere Verbesserungsmöglichkeit bietet die Parallelisierung der Testausführung auf mehreren Rechenkernen oder Computern. Die Evaluierung von Individuen ist der aufwändigste Schritt beim evolutionären Testen. Da kein Datenaustausch zwischen den Individuen einer Generation erforderlich ist, würde sich der Zeitaufwand nahezu proportional zu der Anzahl der Prozessoren verringern. Untersuchungen von Pargas *et al.* mit der Testumgebung TGen deuten auf hohe Zeiteinsparungen hin [87]. Darüber hinaus kann mit der Parallel Computing Toolbox von Matlab die Ausführung von Schleifeniterationen parallelisiert werden [86, 110]. Bei ersten Experimenten im Rahmen dieser Arbeit führte die Miteinbeziehung des zweiten Kerns auf einem Zweikernprozessor bereits

zu Zeiteinsparungen in Höhe von 40% gegenüber der sequentiellen Ausführung. Weiterhin kann die Suchdauer reduziert werden, indem Testfälle anderer Testwerkzeuge als Ausgangspunkt für die evolutionäre Suche verwendet werden (Seeding).

Um die Verbreitung der Verfahren in der Industrie zu beschleunigen, sollte eine Fallstudie unter Einbeziehung einer repräsentativen Anzahl von Testern durchgeführt werden. Die dadurch gewonnenen Daten würden die Vorteile der hier untersuchten Verfahren weiter in Bezug auf die Effizienz belegen.

Für die Verwendung in der Praxis wäre es für OLET und MbET hilfreich, den Tester bei der Erstellung der Fitnessfunktion beziehungsweise der Auswerteskripte mit einem Wizard zu unterstützen. Alternativ könnten auf Basis formaler Systemspezifikationen automatisiert Fitnessfunktionen abgeleitet werden. In der Vergangenheit wurde allerdings bei der Entwicklung von Software für eingebettete Systeme im Automobilbereich von einer Formalisierung der Anforderungen weitestgehend abgesehen. Gründe hierfür sind vor allem ein erhöhter Aufwand und fehlende Werkzeugunterstützung. Bei MbET besteht auch die Möglichkeit, den Suchraum auszuweiten. Beispielsweise kann es sinnvoll sein, weitere Zustände mit Hilfe der evolutionären Engine während der Optimierung zu erzeugen, um die Flexibilität der Testdatensequenzen zu erhöhen. Dafür müssten dann allerdings Regeln festgelegt werden, nach denen die Zustände erzeugt werden und die Testumgebung entsprechend erweitert werden.

Schließlich kann es in speziellen Fällen hilfreich sein, wenn die Durchführung einer mehrkriteriellen Suche unterstützt wird (vergleiche Lakhotia *et al.* [69]). Beispielsweise könnte nach einem Fehlverhalten einer Software und gleichzeitig nach der maximalen Ausführungszeit gesucht werden. Ein derartiges Vorgehen ist vor allem dann sinnvoll, wenn ein Zusammenhang zwischen der Ausführungszeit und einem Fehlverhalten vermutet wird.

LITERATURVERZEICHNIS

[1] W. Afzal, R. Torkar und R. Feldt (2009). *A systematic review of search-based testing for non-functional system properties*. Information and Software Technology, **51**(6), 957–976. ISSN 0950-5849. (Zitiert auf Seite 44.)

[2] S. Ali, L. C. Briand, H. Hemmati und R. K. Panesar-Walawege (2009). *A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation*. IEEE Computer Society, **99**(RapidPosts), 742–762. ISSN 0098-5589. (Zitiert auf den Seiten 56 und 87.)

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis und S. Yovine (1995). *The algorithmic analysis of hybrid systems*. Theoretical Computer Science, **138**, 3–34. ISSN 0304-3975. (Zitiert auf Seite 28.)

[4] A. Baresel (2000). *Automatisierung von Strukturtests mit evolutionären Algorithmen*. Diplomarbeit, Humboldt Universität Berlin. (Zitiert auf Seite 48.)

[5] A. Baresel, M. Conrad, S. Sadeghipour und J. Wegener. (2003). *The Interplay between Model Coverage and Code Coverage*. In EUROCAST - 9th International Workshop on Computer Aided Systems. (Zitiert auf Seite 94.)

[6] A. Baresel, H. Pohlheim und S. Sadeghipour (2003). *Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms*. In Proceedings of the 3rd Genetic and Evolutionary Computation Conference (GECCO 2003), Seiten 2428–2441. ISBN 3-540-40603-4. (Zitiert auf den Seiten 6, 52 und 54.)

[7] G. Behrmann, A. David und K. G. Larsen (2004). *A Tutorial on* Uppaal. In M. Bernardo und F. Corradini, Herausgeber,

Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, Seiten 200–236. Nummer 3185 in LNCS, Springer–Verlag. (Zitiert auf Seite 54.)

[8] B. Beizer (1990). *Software Testing Techniques*. International Thomson Computer Press. ISBN 978-0442206727. (Zitiert auf den Seiten 2 und 13.)

[9] B. Beizer (1995). *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471-12094-4. (Zitiert auf Seite 15.)

[10] BITKOM – Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. (2011). *Wachstumsschub für Eingebettete Systeme*. `http://www.bitkom.org/files/documents/BITKOM_PI_Hannovermesse_04_04_2011_V6.pdf`. (Zitiert auf Seite 1.)

[11] Bosch (2006). *Magazin zur Bosch-Geschichte*. `http://www.bosch.com/content/language1/downloads/Magazin_05_de.pdf`. Letzter Zugriff: April 2011. (Zitiert auf Seite 1.)

[12] L. C. Briand (2007). *A Critical Analysis of Empirical Research in Software Testing*. In Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Seiten 1–8. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2886-4. (Zitiert auf den Seiten 88 und 89.)

[13] E. Bringmann und A. Krämer (2008). *Model-Based Testing of Automotive Systems*. In Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008), Seiten 485–493. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3127-4. (Zitiert auf den Seiten 3, 4 und 31.)

[14] M. Broy (1997). *Refinement of Time*. In Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software (ARTS 1997), Seiten 44 – 63. LNCS 1231, TCS. ISBN 3-540-63010-4. (Zitiert auf den Seiten 23, 27 und 29.)

[15] O. Bühler (2007). *Evolutionärer Funktionstest von eingebetteten Systemen für abstandsbasierte Fahrerassistenzfunktionen im Automobil*. Dissertation, Eberhard-Karls-Universität Tübingen. (Zitiert auf Seite 52.)

[16] O. Bühler und J. Wegener (2004). *Automatic Testing of an Autonomous Parking System using Evolutionary Computation*. In Proceedings of the SAE World Congress. (Zitiert auf Seite 53.)

[17] O. Bühler und J. Wegener (2005). *Evolutionary Functional Testing of a Vehicle Brake Assistant System*. In Proceedings of 6th Metaheuristics International Conference (Mic 2005). (Zitiert auf den Seiten 52 und 53.)

[18] O. Bühler und J. Wegener (2008). *Evolutionary functional testing*. Computers & Operations Research, **35**(10), 3144–3160. ISSN 0305-0548. (Zitiert auf den Seiten 6, 52 und 53.)

[19] Bundesministerium des Innern (1997). *AU 250, Entwicklungsstandard für IT Systeme des Bundes, Vorgehensmodell*. `http://www.v-modell.iabg.de/`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 16 und 169.)

[20] Bundesministerium des Innern (2006). *Das V-Modell XT*. `http://www.v-modell-xt.de`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 16 und 169.)

[21] S. Burton (2002). *Automated Generation of High Integrity Test Suites from Graphical Specifications*. Dissertation, University of York. (Zitiert auf Seite 24.)

[22] J. A. Clark und D. K. Pradhan (1995). *Fault Injection: A Method for Validating Computer-System Dependability*. Computer, **28**, 47–56. ISSN 0018-9162. (Zitiert auf Seite 114.)

[23] J. Coldewey (2009). *Schlechte Noten für Informatik-Ausbildung*. OBJEKTspektrum, **05-09**, 12 – 15. (Zitiert auf Seite 2.)

[24] M. Conrad (2004). *Auswahl und Beschreibung von Testszenarien für den Modell-basierten Test eingebetteter Software im Automobil*. Dissertation, Technische Universität Berlin. (Zitiert auf den Seiten 2, 22 und 26.)

[25] M. Conrad und S. Sadeghipour (2002). *Einsatz von Überdeckungskriterien auf Modellebene - Erfahrungsbericht und experimentelle Ergebnisse*. Softwaretechnik-Trends, **22**(2), 1 – 6. (Zitiert auf Seite 112.)

[26] M. Conrad, H. Dörr, I. Fey und A. Yap (1999). *Model-based Generation and Structured Representation of Test Scenarios*. In Proceedings of the Workshop on Software-Embedded Systems Testing (WSEST 1999). Gaithersburg, USA. (Zitiert auf Seite 22.)

[27] M. Conrad, I. Fey und S. Sadeghipour (2004). *Model–based Testing of Embedded Control Software — The MB3T Approach.* In Proceedings of the ICSE Workshop: Software Engineering for Automotive Systems, Seiten 1 – 9. Edinburgh. (Zitiert auf Seite 4.)

[28] L. Da Costa und M. Schoenauer (2009). *Bringing evolutionary computation to industrial applications with guide*. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO 2009), Seiten 1467–1474. ACM, New York, NY, USA. ISBN 978-1-60558-325-9. (Zitiert auf Seite 41.)

[29] Daimler AG (2009). *Meilensteine der Fahrzeugsicherheit. Die Vision vom unfallfreien Fahren.* `http://www.daimler.com/Projects/c2c/channel/documents/1718371_Sicherheitsbroschuere_deutsch_FINAL_080609.pdf`. Letzter Zugriff: April 2011. (Zitiert auf Seite 98.)

[30] C. Darwin (1859). *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. London: John Murray, 1. Auflage. ISBN 978-0486450063, 532 Seiten. (Zitiert auf Seite 33.)

[31] Deutsche Presse-Agentur (2008). *Motorenprobleme bei BMW*. `http://www.autogazette.de/Motorenprobleme-bei-BMW/artikel_1077297_1.htm`. Letzter Zugriff: April 2011. (Zitiert auf Seite 2.)

[32] E. W. Dijkstra (1972). *The humble programmer*. Communications of the ACM, **15**(10), 859–866. (Zitiert auf Seite 2.)

[33] D. M. Dimitrov, I. Manova und I. Spasov (2008). *Evotest - Framework for customizable implementation of Evolutionary Testing*. In Proceedings of the International Workshop on Software and Services. Sofia, Bulgaria. (Zitiert auf Seite 68.)

[34] H. Do, S. Elbaum und G. Rothermel (2004). *Infrastructure Support for Controlled Experimentation with Software Testing and Regression Testing Techniques*. In Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2004), Seiten 60–70. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2165-7. (Zitiert auf den Seiten 56, 87 und 88.)

[35] S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson und D. Sundmark (2006). *A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques*. In Proceedings of the Testing: Academic & Industrial Conference on Practice and Research Techniques (TAIC PART 2006), Seiten 159–170. ISBN 0-7695-2672-1. (Zitiert auf Seite 114.)

[36] EvoTest (2006 – 2009). `http://evotest.iti.upv.es/`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 6, 45, 54 und 67.)

[37] M. E. Fagan (2002). *A history of software inspections*. Software pioneers: contributions to software engineering, **12**, 562–573. ISSN 0098-5589. (Zitiert auf Seite 2.)

[38] German Testing Board e.V. (2010). *ISTQB®/GTB Standardglossar der Testbegriffe*. `http://www.german-testing-board.info/downloads/pdf/CT_Glossar_DE_EN_V21.pdf`. (Zitiert auf Seite 14.)

[39] K. Ghani, J. A. Clark und Y. Zhan (2009). *Comparing algorithms for search-based test data generation of* MATLAB®SIMULINK®*models*. In Proceedings of the 11th conference on Congress on Evolutionary Computation (CEC 2009), Seiten 2940–2947. IEEE Press, Piscataway, NJ, USA. ISBN 978-1-4244-2958-5. (Zitiert auf Seite 45.)

[40] J. B. Goodenough und S. L. Gerhart (1975). *Toward a theory of test data selection*. In Proceedings of the International Con-

ference on Reliable Software, Seiten 493–510. ACM Press, New York, NY, USA. (Zitiert auf Seite 4.)

[41] W. Grieskamp, M. Heisel und H. Dörr (2001). *Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components*. Science of Computer Programming, **40**(1), 31–57. ISSN 0167-6423. (Zitiert auf Seite 19.)

[42] M. Grochtmann und K. Grimm (1993). *Classification trees for partition testing*. Software Testing, Verification and Reliability, **3**, 63–82. (Zitiert auf Seite 22.)

[43] H. Gross, P. M. Kruse, J. Wegener und T. Vos (2009). *Evolutionary White-Box Software Test with the EvoTest Framework: A Progress Report*. In Proceedings of the 2nd International Conference on Software Testing Verification and Validation Workshop (ICSTW 2009), Seiten 111–120. IEEE Computer Society, Los Alamitos, CA, USA. ISBN 978-0-7695-3671-2. (Zitiert auf Seite 67.)

[44] J. Grossmann, D. Serbanescu und I. Schieferdecker (2009). *Testing Embedded Real Time Systems with TTCN-3*. In Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST 2009), Seiten 81–90. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3601-9. (Zitiert auf Seite 23.)

[45] GUIDE (2010). *Graphical User Interface for DREAM Experiments*. `http://guide.gforge.inria.fr/`. Letzter Zugriff: April 2011. (Zitiert auf Seite 67.)

[46] D. Harel, A. Pnueli, J. P. Schmidt und R. Sherman (1987). *On the Formal Semantics of Statecharts*. In Proceedings of the 2nd Symposium on Logic in Computer Science (LICS 1987), Seiten 54–64. IEEE Press, New York, USA. (Zitiert auf den Seiten 28, 29 und 169.)

[47] M. Harman und P. McMinn (2010). *A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search*. IEEE Transactions on Software Engineering, **36**(2), 226–247. ISSN 0098-5589. (Zitiert auf den Seiten 34 und 43.)

[48] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel und M. Roper (2004). *Testability Transformation*.

IEEE Transactions on Software Engineering, **30**(1), 3–16. ISSN 0098-5589. (Zitiert auf Seite 45.)

[49] M. Harman, S. A. Mansouri und Y. Zhang (2009). Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Technischer Bericht TR-09-03, Department of Computer Science, King's College London. (Zitiert auf den Seiten 41, 42 und 44.)

[50] A. Helmerich, N. Koch, L. Mandel, P. Braun, P. Dornbusch, A. Gruler, P. Keil, R. Leisibach, J. Romberg, B. Schaetz, T. Wild und G. Wimmel (2005). *Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area*. `ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf`. Letzter Zugriff: April 2011. (Zitiert auf Seite 1.)

[51] H. Hemmati (2011). *Similarity-Based Test Case Selection: Toward Scalable and Practical Model-Based Testing*. Phd thesis, Simula Research Laboratory and Informatic Department, University of Oslo. (Zitiert auf Seite 55.)

[52] H. Hemmati, L. Briand, A. Arcuri und S. Ali (2010). *An enhanced test case selection approach for model-based testing: an industrial case study*. In Proceedings of the 18th International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2010), Seiten 267–276. ACM, New York, NY, USA. ISBN 978-1-60558-791-2. (Zitiert auf den Seiten 53 und 54.)

[53] T. A. Henzinger (1996). *The theory of hybrid automata*. In Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS 1996), Seite 278. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-7463-6. (Zitiert auf Seite 28.)

[54] K.-D. Hilf, I. Matheis, J. Mauss und J. Rauh (2010). *Automated simulation of scenarios to guide the development of a crosswind stabilization function*. In Proceedings of the 6th International Federation of Automatic Control Symposium Advances in Automotive Control (IFAC AAC 2010), Seiten 1 – 5. Munich, Germany. (Zitiert auf Seite 185.)

[55] J. Hänsel, D. Rose, P. Herber und S. Glesner (2011). *An Evolutionary Algorithm for the Generation of Timed Test Traces for Embedded Real-Time Systems*. In Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST 2011). (Zitiert auf den Seiten 53 und 54.)

[56] J. H. Holland (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. University of Michigan Press, Ann Arbor, MI, USA. ISBN 978-0472084609, 206 Seiten. (Zitiert auf Seite 33.)

[57] D. C. Ince (1984). *The Automatic Generation of Test Data*. The Computer Journal, **30**(1), 63–69. (Zitiert auf Seite 43.)

[58] International Organization for Standardization (2010). *ISO/IEC 9126*. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 21 und 45.)

[59] B. J. Jain, H. Pohlheim und J. Wegener (2001). *On Termination Criteria of Evolutionary Algorithms*. In Proceedings of the 1st Genetic and Evolutionary Computation Conference (GECCO 2001), Seite 768. Morgan Kaufmann, San Francisco, California, USA. ISBN 1-55860-774-9. (Zitiert auf den Seiten 35 und 66.)

[60] B. Jones, H. Sthamer, X. Yang und D. Eyres (1995). *The automatic generation of software test data sets using adaptive search techniques*. In Proceedings of the 3rd International Conference on Software Quality Management (SQM 1995), Seiten 435–444. ISBN 978-1853124174. (Zitiert auf Seite 52.)

[61] B. F. Jones, H. Sthamer und D. E. Eyres (1996). *Automatic Test Data Generation using Genetic Algorithms*. Software Engineering Journal, **11**(5), 299–306. (Zitiert auf Seite 44.)

[62] T. Jones (1995). *Evolutionary Algorithms, Fitness Landscapes and Search*. Dissertation, University of New Mexico. (Zitiert auf Seite 45.)

[63] N. Juristo, A. M. Moreno und S. Vegas (2004). *Reviewing 25 Years of Testing Technique Experiments*. Empirical Software

Engineering, **9**(1-2), 7–44. ISSN 1382-3256. (Zitiert auf Seite 88.)

[64] A. S. Kalaji, R. M. Hierons und S. Swift (2008). Automatic Generation of Test Sequences from EFSM Models using Evolutionary Algorithms. Technischer Bericht UB8 3PH, School of Information Systems, Computing and Mathematics. Brunel University. (Zitiert auf Seite 47.)

[65] M. Keijzer, J. J. M. Guervós, G. Romero und M. Schoenauer (2002). *Evolving Objects: A General Purpose Evolutionary Computation Library*. In Selected Papers from the 5th European Conference on Artificial Evolution, Seiten 231–244. Springer-Verlag, London, UK. ISBN 3-540-43544-1. (Zitiert auf Seite 67.)

[66] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. E. Emam und J. Rosenberg (2002). *Preliminary guidelines for empirical research in software engineering*. IEEE Transactions on Software Engineering, **28**(8), 721–734. ISSN 0098-5589. (Zitiert auf Seite 88.)

[67] K. Kompass (2008). *Forschung für das Auto von morgen: Aus Tradition entsteht Zukunft*, Kapitel Fahrerassitenzsysteme der Zukunft – auf dem Weg zum autonomen PKW?, Seiten 261 – 285. Springer. ISBN 3-54-074150-X. (Zitiert auf Seite 87.)

[68] P. M. Kruse, J. Wegener und S. Wappler (2009). *A highly configurable test system for evolutionary black-box testing of embedded systems*. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO 2009), Seiten 1545–1552. ACM, New York, NY, USA. ISBN 978-1-60558-325-9. (Zitiert auf den Seiten 53 und 54.)

[69] K. Lakhotia, M. Harman und P. McMinn (2007). *A multi-objective approach to search-based test data generation*. In Proceedings of the 9th annual conference on Genetic and evolutionary computation, Seiten 1098–1105. ACM, New York, NY, USA. ISBN 978-1-59593-697-4. (Zitiert auf Seite 144.)

[70] R. Lefticaru und F. Ipate (2007). *Automatic State-Based Test Generation Using Genetic Algorithms*. Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on, **0**, 188–195. (Zitiert auf Seite 53.)

[71] E. Lehmann (2003). *Time Partition Testing*. Dissertation, Technische Universität Berlin. (Zitiert auf den Seiten 4, 27 und 29.)

[72] F. Lindlar (2009). *Search-Based Functional Testing of Embedded Software Systems*. In Doctoral Symposium in conjunction with The 2nd International Conference on Software Testing, Verification and Validation (ICST 2009). ISBN 978-1-4244-8341-9. (Zitiert auf den Seiten 6, 55 und 87.)

[73] F. Lindlar und A. M. Pérez (2009). *Using evolutionary algorithms to select parameters from equivalence classes*. In Proceedings of Dagstuhl-Workshop Evolutionary Test Generation (2008). 08351, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. ISSN 1862-4405. (Zitiert auf den Seiten 4 und 78.)

[74] F. Lindlar und A. Windisch (2010). *A Search-Based Approach to Functional Hardware-in-the-Loop Testing*. In Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010), Seiten 111–119. IEEE, Benevento, Italy. (Zitiert auf den Seiten 43, 56, 59, 71 und 87.)

[75] F. Lindlar, A. Windisch und J. Wegener (2010). *Integrating Model-Based Testing with Evolutionary Functional Testing*. In Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2010), Seiten 163–172. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-4050-4. (Zitiert auf den Seiten 6, 55, 73 und 87.)

[76] F. Lobo, C. Lima und Z. Michalewicz (2007). *Parameter Setting in Evolutionary Algorithms*. Springer, Berlin, 1. Auflage. ISBN 3-5406-9431-5. (Zitiert auf den Seiten 136 und 143.)

[77] MBtech Group GmbH & Co. KGaA (2010). *ProoveTech:TA*. `http://www.mbtech-group.com/eu-de/electronics_solutions/tools_equipment/provetechta_test_automation.html`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 70 und 113.)

[78] P. McMinn (2004). *Search-based software test data generation: a survey: Research Articles*. Software Testing, Verification

& Reliability, **14**, 105–156. ISSN 0960-0833. (Zitiert auf Seite 43.)

[79] P. McMinn (2011). *Search-Based Software Testing: Past, Present and Future*. In Proceedings of the 4th International Workshop on Search-Based Software Testing (SBST 2011), Seiten 1 – 11. (Zitiert auf Seite 5.)

[80] P. McMinn und M. Holcombe (2003). *The state problem for evolutionary testing*. In Proceedings of the 3rd international conference on Genetic and evolutionary computation (GECCO 2003), Seiten 2488–2498. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-40603-4. (Zitiert auf Seite 20.)

[81] P. McMinn und M. Holcombe (2005). *Evolutionary testing of state-based programs*. In Proceedings of the 5th conference on Genetic and evolutionary computation (GECCO 2005), Seiten 1013–1020. ACM, New York, NY, USA. ISBN 1-59593-010-8. (Zitiert auf Seite 55.)

[82] C. C. Michael, G. E. McGraw, M. A. Schatz und C. C. Walton (1997). *Genetic algorithms for dynamic test data generation*. In Proceedings of the 12th international conference on Automated software engineering (ASE 1997), Seite 307. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-7961-1. (Zitiert auf Seite 44.)

[83] Modelica Association (2010). *Modelica®*. `http://www.modelica.org/`. Letzter Zugriff: April 2011. (Zitiert auf Seite 26.)

[84] V. Nissen (2000). *Einführung in evolutionäre Algorithmen*. Vieweg. ISBN 978-3-528-05499-1, 355 Seiten. (Zitiert auf Seite 34.)

[85] J. S. Ostroff (1989). *Temporal logic for real time systems*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471-92402-4. (Zitiert auf Seite 23.)

[86] Parallel Computing Toolbox® (2011). `http://www.mathworks.com/products/parallel-computing/`. Letzter Zugriff: April 2011. (Zitiert auf Seite 143.)

[87] R. P. Pargas, M. J. Harrold und R. R. Peck (1999). *Test-data Generation Using Genetic Algorithms*. Software Testing, Verification and Reliability, **9**(4), 263–282. (Zitiert auf Seite 143.)

[88] A. M. Perez und S. Kaiser (2010). *Top-Down Reuse for Multilevel Testing*. In Proceedings of the 17th International Conference on the Engineering of Computer-Based Systems (ECBS 2010), Band 1, Seiten 150–159. IEEE Computer Society, Los Alamitos, CA, USA. ISBN 978-0-7695-4005-4. (Zitiert auf Seite 86.)

[89] D. E. Perry, A. A. Porter und L. G. Votta (2000). *Empirical studies of software engineering: a roadmap*. In Proceedings of the Conference on The Future of Software Engineering (FOSE 2000), Seiten 345–355. ACM Press. ISBN 1-58113-253-0. (Zitiert auf Seite 88.)

[90] D. E. Perry, S. E. Sim und S. Easterbrook (2006). *Case studies for software engineers*. In Proceedings of the 28th international conference on Software engineering (ICSE 2006), Seiten 1045–1046. ACM, New York, NY, USA. ISBN 1-59593-375-1. (Zitiert auf Seite 88.)

[91] PikeTec GmbH (2010). *Time Partition Testing*. `http://www.piketec.com/index.php?lang=en`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 4, 31 und 186.)

[92] H. Pohlheim (2000). *Evolutionäre Algorithmen : Verfahren, Operatoren und Hinweise für die Praxis*. Springer, Berlin, Heidelberg [u.a.]. ISBN 3-540-66413-0, 317 Seiten. (Zitiert auf den Seiten 34, 36, 37, 38, 39 und 41.)

[93] H. Pohlheim, M. Conrad und A. Griep (2005). *Evolutionary Safety Testing of Embedded Control Software by Automatically Generating Compact Test Data Sequences*. In SAE 2005 World Congress, Seiten 804–814. ISSN 0096-736X. (Zitiert auf den Seiten 6, 20, 52, 54 und 60.)

[94] A. Pretschner (2003). *Zum modellbasierten funktionalen Test reaktiver Systeme*. Dissertation, Technische Universität München, Germany. (Zitiert auf Seite 26.)

[95] A. Pretschner (2006). *Zur Kosteneffektivität des modellbasierten Testens*. In Proceedings of Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme (2006), Seiten 85–94. Schloss Dagstuhl. (Zitiert auf den Seiten 4, 27 und 140.)

[96] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch und T. Stauner (2005). *One evaluation of model-based testing and its automation.* In Proceedings of the 27th international conference on Software engineering (ICSE 2005), Seiten 392–401. ACM. ISBN 1-58113-963-2. (Zitiert auf den Seiten 3, 4 und 27.)

[97] Python (2010). `http://www.python.org/`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 25, 31 und 81.)

[98] T. Repasi (2009). *Software testing - State of the art and current research challenges*. In Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Seiten 47–50. IEEE. ISBN 978-1-4244-4477-9. (Zitiert auf Seite 15.)

[99] W. Schäfer und H. Wehrheim (2008). *Eingebettete Software*. `http://www.transmechatronic.de/fileadmin/Fachbeitrag/Eingebettete_Software_V3.pdf`. (Zitiert auf Seite 1.)

[100] B.-H. Schlingloff und S. Vulinovic (2005). *Zuverlässigkeitsprüfung eingebetteter Steuergeräte mit modellgetriebener Fehlerinjektion*. In Jahrestagung der ASIM/GI-Fachgruppe 4.5.5 (Simulation technischer Systeme), Seiten 125 – 133. Humboldt-Universität zu Berlin, Institut für Informatik. ISSN 1436-9915. (Zitiert auf Seite 114.)

[101] B. Schätz, A. Pretschner, F. Huber und J. Philipps (2002). *Model-Based Development of Embedded Systems*. In Proceedings of the Workshops on Advances in Object-Oriented Information Systems (OOIS 2002), Seiten 298–312. Springer-Verlag, London, UK. ISBN 3-540-44088-7. (Zitiert auf Seite 3.)

[102] J. Souza, C. Maia, F. Freitas und D. Coutinho (2010). *The Human Competitiveness of Search Based Software Engineering*. In Proceedings of the 2nd International Symposium on

Search Based Software Engineering (SBSE 2010). Benevento, Italy. ISBN 978-1-4244-8341-9. (Zitiert auf Seite 41.)

[103] A. Spillner und T. Linz (2005). *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, Heidelberg, 3. Auflage. ISBN 978-3-89864-358-0. (Zitiert auf den Seiten 2, 13, 14 und 15.)

[104] H. Stachowiak (1973). *Allgemeine Modelltheorie*. Springer, Wien [u.a.]. ISBN 3-211-81106-0, 494 Seiten. (Zitiert auf den Seiten 3 und 26.)

[105] H. Sthamer und J. Wegener (2002). *Using Evolutionary Testing to improve Efficiency and Quality in Software Testing*. In Proceedings of 2nd Asia-Pacific Conference on Software Testing (APSEC 2002). (Zitiert auf Seite 43.)

[106] H. Sthamer, A. Baresel und J. Wegener (2001). *Evolutionary Testing of Embedded Systems*. In Proceedings of the 14th International Internet & Software Quality Week (QW 2001), Seiten 1–34. (Zitiert auf Seite 5.)

[107] H. H. Sthamer (1995). *The Automatic Generation of Software Test Data Using Genetic Algorithms*. Dissertation, University of Glamorgan / Prifvsgol Morgannwg. (Zitiert auf Seite 44.)

[108] H. Stringer und A. S. Wu (2005). *Variable-Length Genetic Algorithms and an Analysis of Changes in Chromosome Length Absent Selection Pressure*. `http://www.cs.ucf.edu/~stringer/ECJ%t20Article.pdf`. Letzter Zugriff: April 2011. (Zitiert auf Seite 38.)

[109] The Eclipse Foundation (2010). *Eclipse integrated devopment enviroment*. `http://www.eclipse.org`. Letzter Zugriff: April 2011. (Zitiert auf Seite 67.)

[110] The MathWorks™, Inc. (2011). MATLAB®. `http://www.mathworks.de/products/matlab/`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 26, 69, 99, 119, 143 und 168.)

[111] The MathWorks™, Inc. (2011). SIMULINK®. `http://www.mathworks.de/products/simulink/`. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 3, 99, 119 und 169.)

[112] The MathWorks™, Inc. (2011). STATEFLOW®. http://www.mathworks.de/products/stateflow/. Letzter Zugriff: April 2011. (Zitiert auf den Seiten 69, 99, 119 und 169.)

[113] Toyota (2010). *Toyota Deutschland ruft Prius III in die Werkstatt*. http://www.toyota.de/about/news/details_2010_06.aspx. Letzter Zugriff: April 2011. (Zitiert auf Seite 2.)

[114] N. Tracey, J. Clark und K. Mander (1998). *Automated Program Flaw Finding using Simulated Annealing*. SIGSOFT Software Engineering Notes, **23**(2), 73–81. ISSN 0163-5948. (Zitiert auf Seite 52.)

[115] N. Tracey, J. Clark, K. Mander und J. McDermid (1998). *An Automated Framework for Structural Test-Data Generation*. In Proceedings of the 13th International Conference on Automated Software Engineering (ASE 1998), Seiten 285–288. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-8750-9. (Zitiert auf Seite 45.)

[116] N. J. Tracey (2000). *A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software*. Dissertation, University of York. (Zitiert auf Seite 52.)

[117] F. E. Vieira, F. Martins, R. Silva, R. Menezes und M. Braga (2006). *Using Genetic Algorithms to Generate Test Plans for Functionality Testing*. In Proceedings of the 44th annual Southeast regional conference (SERC 2006), Seiten 140–145. ACM, New York, NY, USA. ISBN 1-59593-315-8. (Zitiert auf den Seiten 52 und 53.)

[118] T. Vos, F. Lindlar, B. Wilmes, A. Windisch, A. Baars, P. Kruse, H. Gross und J. Wegener (2012). *Evolutionary functional black-box testing in an industrial setting*. Software Quality Journal, **02/2012**, 1–30. ISSN 0963-9314. (Zitiert auf den Seiten 59 und 87.)

[119] T. E. J. Vos, A. I. Baars, F. Lindlar, P. M. Kruse, A. Windisch und J. Wegener (2010). *Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool*. In Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST 2010), Seiten 175–184. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3990-4. (Zitiert auf Seite 73.)

[120] S. Wappler (2007). *Automatic Generation Of Object-Oriented Unit Tests Using Genetic Programming*. Dissertation, Technische Universität Berlin. (Zitiert auf Seite 48.)

[121] S. Wappler und J. Wegener (2006). *Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm*. In Proceedings of the Congress on Evolutionary Computation (CEC 2006), Seiten 3193–3200. IEEE Press, Vancouver. (Zitiert auf Seite 45.)

[122] J. Wegener und M. Grochtmann (1998). *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*. Real-Time Systems, **15**(3), 275–298. (Zitiert auf Seite 43.)

[123] J. Wegener und P. M. Kruse (2009). *Search-Based Testing with in-the-loop Systems*. In Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09), Seiten 81–84. IEEE Computer Society. ISBN 978-0-7695-3675-0. (Zitiert auf Seite 54.)

[124] J. Wegener, H. Sthamer und A. Baresel (2001). *Application Fields for Evolutionary Testing*. In Proceedings of the 9th European International Conference on Software Testing Analysis & Review (EuroSTAR 2001). (Zitiert auf Seite 5.)

[125] J. Wegener, K. Buhr und H. Pohlheim (2002). *Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing*. In Proceedings of the 2nd Genetic and Evolutionary Computation Conference (GECCO 2002), Seiten 1233–1240. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-878-8. (Zitiert auf den Seiten 47 und 80.)

[126] D. Whitley, S. Rana, J. Dzubera und K. E. Mathias (1996). *Evaluating evolutionary algorithms*. Artificial Intelligence, **85**, 245–276. ISSN 0004-3702. (Zitiert auf den Seiten 92 und 173.)

[127] B. Wilmes, F. Lindlar und A. Windisch (2011). *Suchbasierter Test für den industriellen Einsatz*. In 4. Symposium Testen im System- und Software-Life-Cycle, Technische Akademie Esslingen. ISBN 3-92-4813-92-2. (Zitiert auf Seite 87.)

[128] A. Windisch (2009). *Search-Based Testing of Complex* Simulink®*Models containing* Stateflow®*Diagrams*. In Proceedings of the 1st International Workshop on Search-Based Software Testing (SBST 2009). Denver, Colorado. ISBN 978-1-4244-3495-4. (Zitiert auf Seite 45.)

[129] A. Windisch und N. Al Moubayed (2009). *Signal Generation for Search-Based Testing of Continuous Systems*. In Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2009), Seiten 121–130. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3671-2. (Zitiert auf Seite 60.)

[130] A. Windisch, F. Lindlar, S. Topuz und S. Wappler (2009). *Evolutionary functional testing of continuous control systems*. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO 2009), Seiten 1943–1944. ACM, New York, NY, USA. ISBN 978-1-60558-325-9. (Zitiert auf Seite 67.)

[131] S. E. Xanthakis, C. C. Skourlas und A. LeGall (1992). *Application of Genetic Algorithms to Software Testing*. In Proceedings of the 5th International Conference on Software Engineering and its Applications (SEA 1992), Seiten 625–636. (Zitiert auf Seite 44.)

[132] J. Zander-Nowicka (2008). *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. Dissertation, Technische Universität Berlin. (Zitiert auf Seite 184.)

[133] Y. Zhan und J. A. Clark (2006). *The state problem for test generation in* Simulink®. In Proceedings of the 6th annual conference on Genetic and evolutionary computation (GECCO 2006), Seiten 1941–1948. ACM. ISBN 1-59593-186-4. (Zitiert auf den Seiten 4 und 55.)

ABKÜRZUNGSVERZEICHNIS

A

ABA Aktiver Notbremsassistent

ART Abstandsregeltempomat

C

CL closed-loop

D

DSW Durchschnittswert

E

EA Evolutionäre Algorithmen

EFT Evolutionärer Funktionstest

ES Eingebettetes System

EST Evolutionärer Strukturtest

ETF Evolutionary Testing Framework

F

FW Fitnesswert

G

GA Genetische Algorithmen

GW Grenzwert

H

HIL Hardware-in-the-Loop

K

KFG Kontrollflussgraph

M

MBET *Modellbasierter evolutionärer Test*

MIL Model-in-the-Loop

N

NB Notbremsung

O

OL open-loop

OLET *Open-loop evolutionärer Test*

S

SG Signalgenerator

SIL Software-in-the-Loop

SP Stichprobe

T

TO Testobjekt

TPT Time Partition Testing

U

US Umgebungssimulation

W

WL Warnlevel

GLOSSAR

A

ABSTANDSREGELTEMPOMAT Ein *Abstandsregeltempomat* ist ein Fahrerassistenzsystem, das aus einem Geschwindigkeits- und einem Abstandsregler besteht. Wenn die des Fahrers gewünschte Geschwindigkeit zu einem zu dichten Auffahren auf das Vorderfahrzeug führen würde, regelt der Abstandsregler eine niedrigere Geschwindigkeit ein, um einen sicheren Abstand zu wahren.

AKTIVER NOTBREMSASSISTENT Ein *aktiver Notbremsassistent* ist ein Fahrerassistenzsystem, das in kritischen Situationen eine Notbremsung auslöst. Zudem warnt das System den Fahrer durch akustische und optische Hinweise vor möglichen Gefahrensituationen.

C

CHROMOSOM Im Rahmen dieser Arbeit wird ein Individuum als eine Menge von *Chromosomen* aufgefasst. In den *Chromosomen* sind Daten kodiert, auf Basis derer der Testtreiber die Teststimuli erzeugt. Jedes Chromosom enthält Daten für einen Eingangskanal des Testobjekts (OLET) beziehungsweise für einen Zustandsautomaten (MbET).

CLOSED-LOOP-TEST Beim *closed-loop*-Test können Eingangsdaten entsprechend des Ausgangsverhaltens eines Testobjekts zur Laufzeit angepasst werden. Beispielweise besteht die Möglichkeit, einen Zustandswechsel in einem Testmodell durchzuführen, sobald ein Ausgangskanal eines Testobjekts einen Grenzwert überschritten hat.

E

EINGEBETTETES SYSTEM Ein *eingebettetes System* ist eine Einheit aus Hardware und Software, die verwendet wird, um eine bestimmte Menge von Funktionen auszuführen. *Eingebettete Systeme* erfassen mit Sensoren die Umwelt und stellen basierend auf den gewonnenen Erkenntnissen Informationen für den Nutzer bereit oder lösen durch Ansteuerung von Aktoren einen Eingriff in die Umwelt aus. Im Automobilbereich werden eingebettete Systeme aufgrund des Aufgabenschwerpunkts der Steuerung und Regelung als Steuergeräte bezeichnet.

ENDLICHE ZUSTANDSAUTOMATEN Ein *endlicher Zustandsautomat* setzt sich aus einer endlichen Menge von Zuständen und einer Menge von Zustandsübergängen zwischen den Zuständen (Transitionen) zusammen. Zu den Hauptanwendungsfeldern *endlicher Zustandsautomaten* zählt die Modellierung von Systemverhalten.

ERWEITERTE ENDLICHE ZUSTANDSAUTOMATEN *Erweiterte endliche Zustandsautomaten* erweitern klassische endliche Zustandsautomaten um die Möglichkeit, während der Ausführung Operationsdaten abzufragen und zu manipulieren.

EVOLUTIONARY TESTING FRAMEWORK Das *Evolutionary Testing Framework* wurde im Rahmen des EvoTest-Förderprojekts entwickelt. Das Framework realisiert ein evolutionäres Suchverfahren und bietet grundlegende Funktionalitäten für die Generierung von Teststimuli für den Struktur- und den Funktionstest.

EVOLUTIONÄRE ALGORITHMEN *Evolutionäre Algorithmen* gehören zu der Klasse der metaheuristischen Suchverfahren. Mit evolutionären Algorithmen wird ein globales Suchverfahren realisiert. Die Wahrscheinlichkeit, dass die Suche in einem lokalen Optimum endet, ist geringer als bei lokalen Suchverfahren.

EVOLUTIONÄRE ENGINE Eine *evolutionäre Engine* ist eine Softwarekomponente, die einen evolutionären Algorithmus realisiert.

EVOLUTIONÄRER FUNKTIONSTEST Beim *evolutionären Funktionstest* wird mit Hilfe evolutionärer Algorithmen nach einem

Testfall gesucht, der ein Fehlverhalten eines Systems aufzeigt (Anforderungsverletzung). Wird kein solcher Testfall gefunden, geht ein Vertrauensgewinn in die korrekte Funktionalität der Software einher.

EVOLUTIONÄRER STRUKTURTEST Beim *evolutionären Strukturtest* werden evolutionäre Algorithmen verwendet, um Programmcode nach ausgewählten Kriterien zu überdecken (mit Testfällen zur Ausführung bringen). Das Ergebnis ist ein Satz von Testfällen, der beispielsweise alle Verzweigungen in einem Programmcode durchläuft.

EVOLUTIONÄRER ZEITVERHALTENSTEST Beim *evolutionären Zeitverhaltenstest* werden evolutionäre Algorithmen verwendet, um maximale - und minimale Ausführungszeiten einer Funktion zu identifizieren. Der Zeitverhaltenstest gehört zu der Kategorie der nicht-funktionalen Testverfahren.

EVOLUTIONÄRES TESTEN Als *evolutionäres Testen* wird die Verwendung genetischer Algorithmen für den Softwaretest bezeichnet. Gängige Testarten für das globale Sucherverfahren sind der Strukturtest, der Funktionstest und der nicht-funktionale Test.

G

GA MIT CHROMOSOMEN VARIABLER LÄNGE *Genetische Algorithmen mit Chromosomen variabler Länge* zeichnen sich dadurch aus, dass genetischen Veränderungen von Individuen die Länge der Chromosomen beeinflussen können. Beim evolutionären Testen bietet dieser Ansatz eine erhöhte Flexibilität für die Generierung von Teststimuli.

GEN Ein Chromosom besteht aus mehreren *Genen*. Im Rahmen dieser Arbeit wird ein *Gen* als ein atomarer Zahlenwert aufgefasst, der einem Charakteristikum einer Testdatensequenz oder einem Parameter eines Testmodells entspricht.

GENETISCHE ALGORITHMEN *Genetische Algorithmen* sind eine Unterkategorie der evolutionären Algorithmen. Die Funktionsweise der genetischen Algorithmen orientiert sich sehr stark an der biologischen Evolution. Der Übergang von einer Generation zur nächsten erfolgt mit Hilfe von Selektions-,

Mutations-, Rekombinations- und Wiedereinfügeoperatoren.

H

HYBRIDE AUTOMATEN *Hybride Automaten*, eine Erweiterung von endlichen Automaten, sind ein formales Modell für die Beschreibung hybrider Systeme. Transitionen in einem *hybriden Automaten* repräsentieren diskrete Zustandsübergänge und Zustände kontinuierliches Verhalten.

HYBRIDE SYSTEME *Hybride Systeme* erlauben die Modellierung sowohl von diskretem als auch von kontinuierlichem Verhalten.

M

MATLAB *Matlab* ist ein kommerzielles Software-Entwicklungswerkzeug, das von der Firma MathWorks vertrieben wird [110]. In erster Linie wird *Matlab* zur Lösung und Visualisierung mathematischer Probleme eingesetzt. Simulink und Stateflow sind Erweiterungen von *Matlab* für die modellbasierte Entwicklung.

MODELLBASIERTE ENTWICKLUNG Die *modellbasierte Entwicklung* ist ein Entwicklungsansatz, bei dem ein Programm oder eine Funktion mit Hilfe von Modellen beschrieben wird. Zu den Vorteilen dieses Ansatzes zählen ein höherer Abstraktionsgrad und die Möglichkeit, ausführbare Modelle für den Testprozess zu verwenden.

MODELLBASIERTER EVOLUTIONÄRER TEST Das zweite in dieser Arbeit entwickelte Verfahren für den Funktionstest *MbET* – Modellbasierter Evolutionärer Test – vereinigt das Prinzip des modellbasierten Testens mit dem des evolutionären Testens. Teststimuli werden von hybriden Automaten abgeleitet und die Ergebnisse einer Testausführung mit instrumentierten Auswerteskripten bewertet.

MODELLBASIERTES TESTEN Das *modellbasierte Testen* ist ein Testansatz, bei dem Testfälle von einem Modell (zum Beispiel einem Automaten) abgeleitet werden. *Modellbasierte Testansätze* zielen im Allgemeinen auf eine Automatisierung des Testprozesses ab.

O

OPEN-LOOP EVOLUTIONÄRER TEST Das erste in dieser Arbeit entwickelte Verfahren für den funktionalen Test *OLET* – *Open-Loop* Evolutionärer Test – realisiert einen Testautomatisierungsansatz für Software eingebetteter Systeme. Teststimuli werden bei *OLET* mit Hilfe einer hierarchischen Auszeichnungssprache beschrieben.

OPEN-LOOP-TEST Beim *open-loop*-Test hat das Ausgangsverhalten eines Testobjekts während der Ausführung keinen Einfluss auf die generierten Eingangsdaten.

S

SIMULINK *Simulink* gehört zu der Entwicklungsumgebung Matlab und ist eine blockbasierte Modellierungssprache für die modellbasierte Entwicklung [111].

STATECHARTS *Statecharts* nach Harel *et al.* erweitern klassische endliche Zustandsautomaten um die Möglichkeit, Zustände und Zustandsautomaten hierarchisch anzuordnen und zu parallelisieren [46].

STATEFLOW *Stateflow* gehört zu der Entwicklungsumgebung Matlab und ist eine Beschreibungssprache für Automaten. Die Automaten dienen der Spezifikation interner Systemzustände im Rahmen der modellbasierten Entwicklung mit Simulink [112].

T

TESTPLATTFORM Eine *Testplattform* ist eine Ausführungsplattform, die die notwendige Infrastruktur für die Durchführung von Tests bereitstellt.

TESTSTUFE Als *Teststufen* werden die einzelnen Phasen bei der Qualitätssicherung nach dem V-Modell bezeichnet [19, 20]. Abhängig von dem zu entwickelnden System können *Teststufen* beispielsweise Modul-, Komponenten- und Systemtests sein.

TESTSUITE Eine *Testsuite* ist eine endliche Menge von Testfällen, die zu einer Einheit zusammengefasst sind.

Teil IV

ANHANG

A BESTIMMUNG DER EA-PARAMETER

Für die Ermittlung der Parameter der *evolutionären Engine* wurden Benchmark-Experimente mit den Funktionen *Rastrigin* und *Griewangk* durchgeführt. Die Funktionen *Rastrigin* und *Griewangk* gelten allgemein als gute Indikatoren für die Performance von Suchverfahren [126]. Ausgehend von einer initialen Belegung wurde jeweils ein Konfigurationsparameter aus Tabelle 3.1 variiert, während die anderen konstant blieben (zum Beispiel die *Populationsgröße*). Die Experimente wurden einhundert Mal für jede Konfiguration wiederholt. Eine Durchführung von Benchmark-Experimenten mit den in dieser Arbeit verwendeten Testobjekten konnte aufgrund der hohen Anzahl von benötigten Ausführungen für eine realistische Bewertung und der hohen Ausführungszeit der Testobjekte nicht realisiert werden. Es wird die Annahme getroffen, dass die ermittelten Parameter auch auf die Testobjekte der Fallstudien skalieren.

Nachfolgend werden Auszüge aus den Ergebnissen der Experimentreihen vorgestellt. Bei der Betrachtung der *Populationsgröße* muss berücksichtigt werden, dass eine erhöhte *Populationsgröße* auch eine höhere Anzahl an erforderlichen Evaluierungen bewirkt. Für die Durchführung der Fallstudien wird deshalb eine *Populationsgröße* von 100 verwendet (Abbildungen A.1 und A.2). Insgesamt ergeben sich somit bei 100 Generationen maximal 10.000 erforderliche Evaluierungen. Die *Fruchtbarkeitsrate* wird – eindeutig erkennbar aus den Verläufen – auf den Wert 80 gesetzt (Abbildungen A.3 und A.4). Als *Selektionsoperator* wird die *Turnierselektion* gewählt (Abbildungen A.5 und A.6). *Rekombinationsrate* und *Mutationsrate* werden mit den Werten $0,8$ und $0,6$ belegt (Abbildungen A.7 und A.8 sowie Abbildungen A.9 und A.10). Die Ermittlung weiterer Konfigurationsparameter für den evolutionären Algorithmus erfolgte analog, soll hier aber nicht weiter aufgeführt werden. Tabelle 6.1 fasst die Ergebnisse der Benchmark-Experimente zusammen.

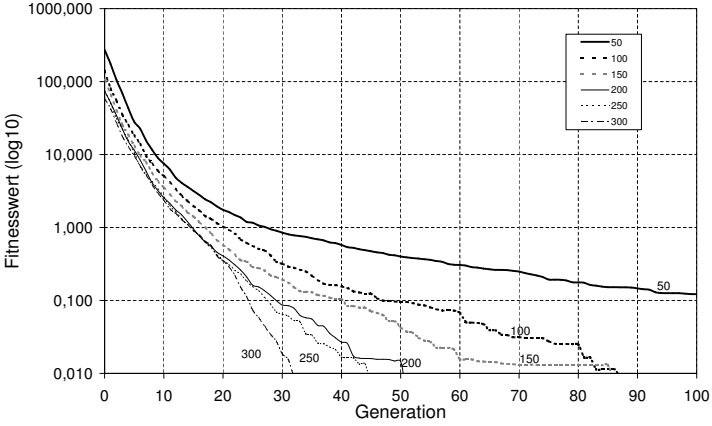


Abbildung A.1: Experimente mit der *Rastrigin*-Funktion für die Ermittlung einer geeigneten *Populationsgröße*. Die Plots sind über 100 Ausführungen gemittelt.

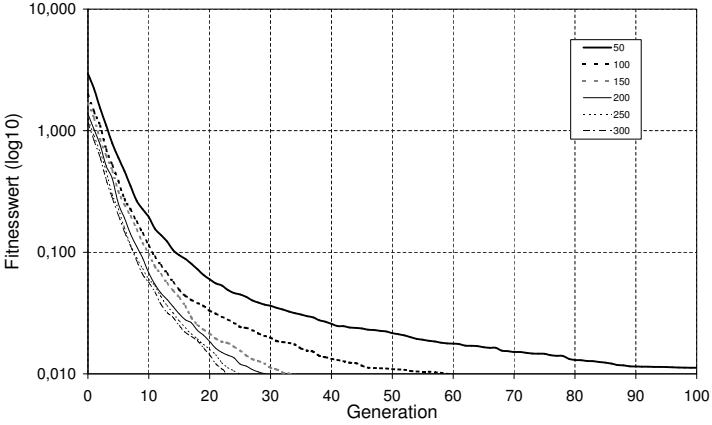


Abbildung A.2: Experimente mit der *Griewangk*-Funktion für die Ermittlung einer geeigneten *Populationsgröße*. Die Plots sind über 100 Ausführungen gemittelt.

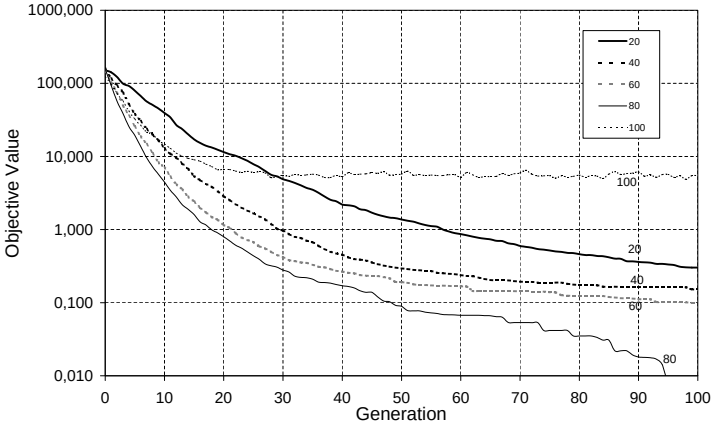


Abbildung A.3: Experimente mit der *Rastrigin*-Funktion für die Ermittlung einer geeigneten *Fruchtbarkeitsrate*. Die Plots sind über 100 Ausführungen gemittelt.

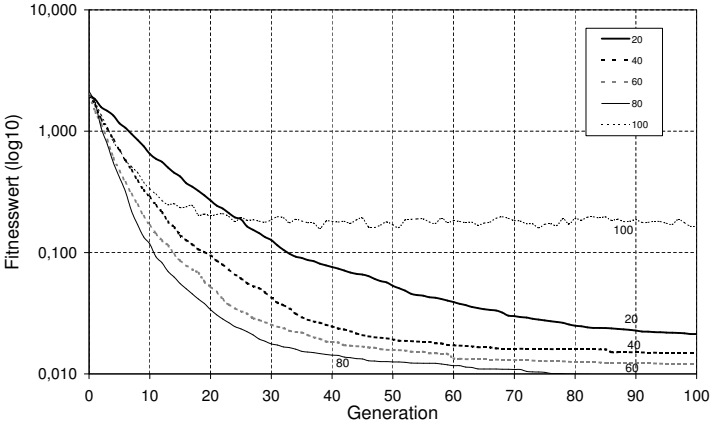


Abbildung A.4: Experimente mit der *Griewangk*-Funktion für die Ermittlung einer geeigneten *Fruchtbarkeitsrate*. Die Plots sind über 100 Ausführungen gemittelt.

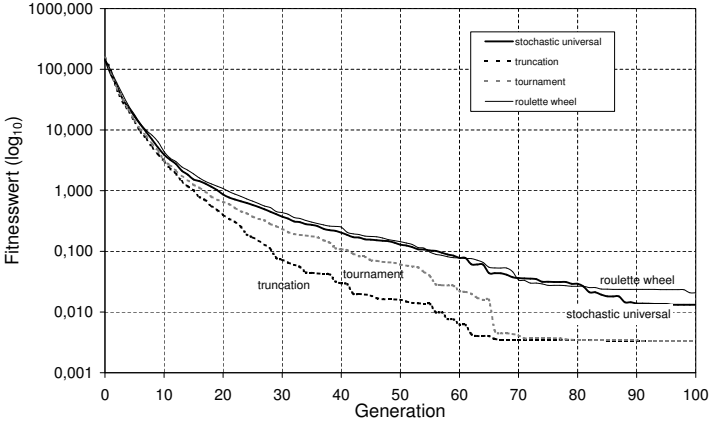


Abbildung A.5: Experimente mit der *Rastrigin*-Funktion für die Ermittlung eines geeigneten *Selektionsoperators*. Die Plots sind über 100 Ausführungen gemittelt.

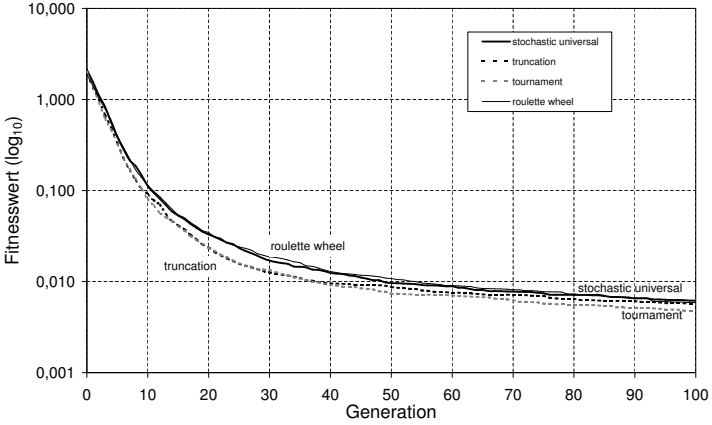


Abbildung A.6: Experimente mit der *Griewangk*-Funktion für die Ermittlung eines geeigneten *Selektionsoperators*. Die Plots sind über 100 Ausführungen gemittelt.

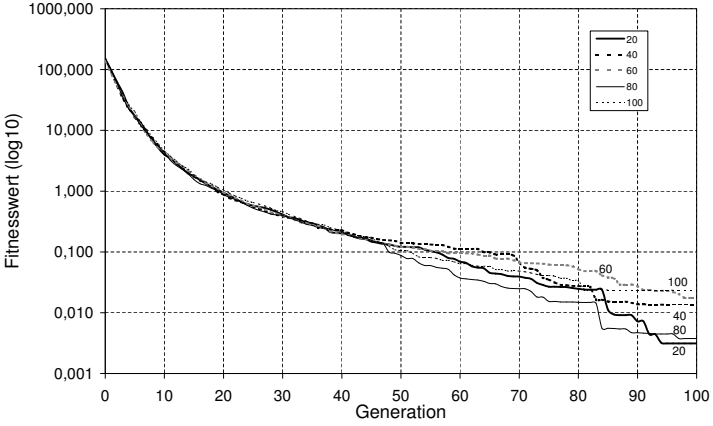


Abbildung A.7: Experimente mit der *Rastrigin*-Funktion für die Ermittlung einer geeigneten *Rekombinationsrate*. Die Plots sind über 100 Ausführungen gemittelt.

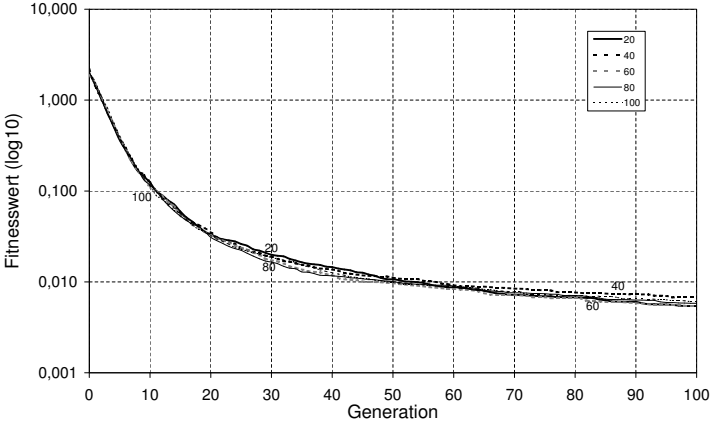


Abbildung A.8: Experimente mit der *Griewangk*-Funktion für die Ermittlung einer geeigneten *Rekombinationsrate*. Die Plots sind über 100 Ausführungen gemittelt.

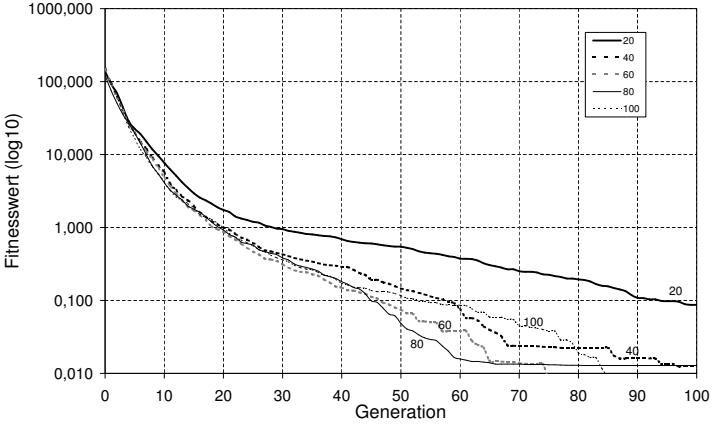


Abbildung A.9: Experimente mit der *Rastrigin*-Funktion für die Ermittlung einer geeigneten *Mutationsrate*. Die Plots sind über 100 Ausführungen gemittelt.

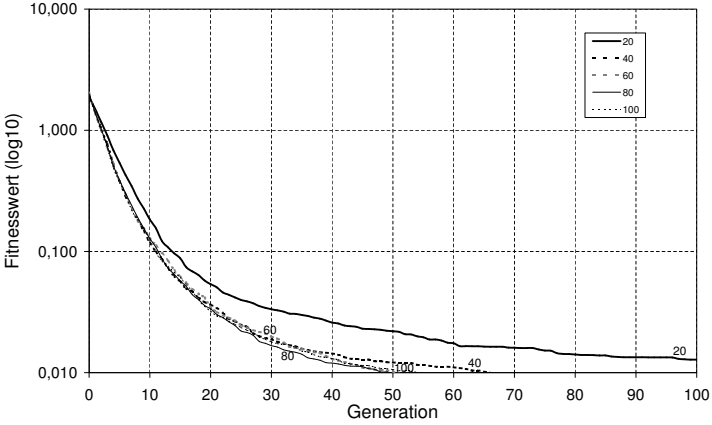
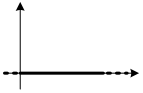
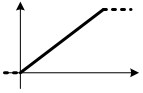
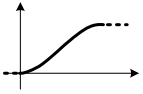
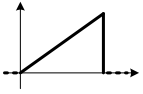
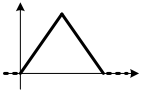
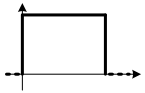
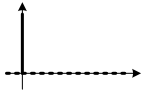
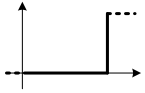


Abbildung A.10: Experimente mit der *Griewangk*-Funktion für die Ermittlung einer geeigneten *Mutationsrate*. Die Plots sind über 100 Ausführungen gemittelt.

B SIGNALTYPEN FÜR OLET

Mit OLET (Kapitel 4) generierte Testdatensequenzen bestehen aus aneinandergereihten, parametrierten Standardsignalen. Auf den nachfolgenden Seiten wird eine Übersicht über die von OLET unterstützten Signaltypen geboten. Mit Hilfe der Formeln werden konkrete Werte (für jeden Abtastschritt) für die Übergänge zwischen den Segmenten bestimmt.

Typ	Zeichnung	Formel (mit A: Amplitude, B: Breite und Seg: Segment)
Konstant		$f(t) = A_{Seg_{n-1}}$
Linear		$f(t) = (A_{Seg_n} - A_{Seg_{n-1}}) \cdot t + A_{Seg_{n-1}}$
Sinus		$f(t) = \frac{A_{Seg_n} - A_{Seg_{n-1}}}{2} \cdot \sin(\frac{\pi}{B_{Seg_n}} t - \frac{\pi}{2}) + \frac{A_{Seg_n} + A_{Seg_{n-1}}}{2}$
Sägezahn		$f(t) = \begin{cases} (A_{Seg_n} - A_{Seg_{n-1}}) \cdot t + A_{Seg_{n-1}} & \text{für } t < B_{Seg_n} \\ A_{Seg_{n-1}} & \text{für } t = B_{Seg_n} \end{cases}$
Dreieck		$f(t) = \begin{cases} (A_{Seg_n} - A_{Seg_{n-1}}) \cdot t + A_{Seg_{n-1}} & \text{für } t < \frac{B_{Seg_n}}{2} \\ (A_{Seg_{n-1}} - A_{Seg_n}) \cdot t + 2 \cdot A_{Seg_n} & \text{für } \frac{B_{Seg_n}}{2} \leqslant t \leqslant B_{Seg_n} \end{cases}$

Typ	Zeichnung	Formel (mit A: Amplitude, B: Breite und Seg: Segment)
Rechteck		$f(t) = \begin{cases} A_{Seg_n} & \text{für } t < B_{Seg_n} \\ A_{Seg_{n-1}} & \text{für } t = B_{Seg_n} \end{cases}$
Spline		$f(t) = spline^1(A_{Seg_{n-1}}, A_{Seg_n}, B_{Seg_n}, t)$
Impuls		$f(t) = \begin{cases} A_{Seg_n} & \text{für } t = 0 \\ A_{Seg_{n-1}} & \text{für } 0 < t \leqslant B_{Seg_n} \end{cases}$
Sprung		$f(t) = \begin{cases} A_{Seg_{n-1}} & \text{für } t < B_{Seg_n} \\ A_{Seg_n} & \text{für } t = B_{Seg_n} \end{cases}$

1 Formel nach Sky McKinley and Megan Levine (Steigung Null im Start- und Endpunkt): „Cubic Spline Interpolation“ (`http://online.redwoods.cc.ca.us/instruct/darnold/laproj/Fall98/SkyMeg/Proj.PDF`)

C BLACK-BOX TESTWERKZEUGE FÜR ES

In der Tabelle auf den nachfolgenden Seiten werden Black-Box Testwerkzeuge vorgestellt, die über die Möglichkeit verfügen, kontinuierliche Testdatensequenzen für die Testausführung zu generieren. Für einen effektiven Test von Software eingebetteter Systeme ist eine derartige Funktionalität unabdingbar. Im Unterschied zu OLET (Kapitel 4) und MbET (Kapitel 5) können diese Werkzeuge keine neuen Testfälle automatisiert erstellen und bewerten. Bis dato existieren auch noch keine derartigen Testwerkzeuge. Die Tabelle bietet eine Übersicht darüber, welche Verfahren aktuell im Bereich der Entwicklung von Software eingebetteter Systeme Anwendung finden. Im Fokus stehen die Herangehensweise an die Generierung der Testdatensequenzen und die Auswertung von Testfällen.

Bezeichnung & Hersteller	Testdatensequenzen	Auswertung	Automatisierung
CTE/ES (Razorcat) und CTE/XL (Berner&Mattner)	Klassifikationsbäume, Testdatensequenzen über Interpolation zwischen zwei Punkten	manuell, bei CTE/XL auch mit Messina Testplattform	Automatisierte Testfallableitung
EXAM (MicroNova)	Sequenzdiagramme	vordefinierte kombinierbare Regelsätze	Automatisierte Testfallableitung und Ausführung
MiLEST (Justyna Zander [132])	Zustandsautomaten und Aneinanderreihung von Signaleigenschaften	Validierungs-funktionen nach dem Schema IF *precondition* THEN *assertion*	Automatisierte Testfallableitung, Ausführung und Auswertung
MSC2C-Testcode Generator (Validas)	Message Sequence Charts, kontinuierliche Signale in C implementieren	Abgleich mit erwartetem Verhalten in MSCs	Automatisierte Testfallableitung, Ausführung und Auswertung

Bezeichnung & Hersteller	Testdatensequenzen	Auswertung	Automatisierung
MTest (dSpace)	grafisch mit Klassifikationsbäumen, kontinuierliche Signale mit Signalbuilder	Vergleich mit Referenzsignalen und Schranken	Automatisierte Testfallableitung und Ausführung
Silest (Forschungsprojekt Silest)	Signalbuilder, Signale im XML Format	Referenzsignale und Schranken für Ausgangssignale	Automatisierte Testausführung und Auswertung
Simulink Verification and Validation (MathWorks)	Signalbuilder	Schranken für Ausgangssignale	–
Testweaver (QTronic GmbH)	Erzeugung kontinuierlicher Testdatensequenzen mit der Erweiterung *Silver* [54]	Schranken für Ausgangssignale	Automatisierte Testausführung und Auswertung

Bezeichnung & Hersteller	Testdatensequenzen	Auswertung	Automatisierung
TPT (PikeTec [91])	hybride Automaten und stromverarbeitende Funktionen	Skriptsprache für die Beschreibung des gültigen Systemverhaltens	Automatisierte Testausführung und Auswertung
TTCN3 Embedded (Forschungs- projekt TEMEA)	Textuell oder mit hybriden Automaten	Assertions und Signalvergleich	Automatisierte Testausführung und Auswertung