Fraunhofer Institut
Experimentelles
Software Engineering

Stand der Forschung von Software-Tests und deren Automatisierung

Autoren:
Ove Armbrust
Michael Ochs
Björn Snoek

Infrastruktur für eine Forschungs- und Transferplattform am Fraunhofer IESE in Kaiserslautern für regionale, software-entwickelnde KMUs (KMU Forschungslaborplattform)

Gefördert mit Mitteln des Europäischen Fonds für Regionalentwicklung (EFRE) und des Landes Rheinland-Pfalz (Ministerium für Wirtschaft, Verkehr, Landwirtschaft und Weinbau Rheinland-Pfalz: MWVLW RLP).

Förderkennzeichen: MWVLW;
Az.: 8315 38 51 04 IESE;
Kapitel 0877 Titel 892 02.

IESE-Report Nr. 068.04/D
VERSION 1.0
18. JUNI 2004

Eine Publikation des Fraunhofer IESE

Das Fraunhofer IESE ist ein Institut der Fraunhofer-Gesellschaft.
Das Institut überträgt innovative Software-Entwicklungstechniken, -Methoden und -Werkzeuge in die industrielle Praxis. Es hilft Unternehmen, bedarfsgerechte Software-Kompetenzen aufzubauen und eine wettbewerbsfähige Marktposition zu erlangen.

Das Fraunhofer IESE steht unter der Leitung von
Prof. Dr. Dieter Rombach
Sauerwiesen 6
67661 Kaiserslautern

Abstract

Dieser Report untersucht aktuelle Forschungsansätze im Themengebiet Testen und Testautomatisierung. Vorgestellt werden jeweils Ansätze, die exemplarisch für ihre Klasse stehen. Betrachtet wird die vollautomatische Testfallgenerierung aus formalen Anforderungen, die Testfallerstellung aus halbformalen Anforderungen wie z.B. UML, sowie die Testdatengewinnung. Testdurchführung und Testauswertung werden nur kurz gestreift, intensiver behandelt wird dafür die Wiederverwendung von Testfällen, die Optimierung von Testsuiten für Regressionstest, sowie Teststrategien für objektorientierte Software.

Schlagworte: KMU research lab, testing, test planning, test coverage, test case generation, test case design

Inhaltsverzeichnis

1 Einführung

1.1 Hintergrund

Softwaresysteme werden zunehmend größer und komplexer. Große Systeme werden weltweit entwickelt, in verschiedenen Zeitzonen, unter völlig unterschiedlichen Voraussetzungen. Dafür wurden und werden neue Strategien entwickelt, um die Qualität der Produkte sicherzustellen. Im praktischen Einsatz ist jedoch der Test entwickelter Software immer noch die am meisten genutzte Vorgehensweise.

Softwaretests sind Entwicklertätigkeiten, die oft nur ungern ausgeführt werden. Dies liegt zum großen Teil daran, dass die Entwicklung guter Tests Zeit verschlingt, die in den meisten Projekten sowieso knapp ist, und daher leicht das Gefühl aufkommt, dass diese Zeit für die Entwicklung der eigentlichen Software besser genutzt werden könnte. Möglicherweise spielt es auch eine Rolle, dass sich niemand gerne Fehler nachweisen lässt; genau dies geschieht aber beim Testen.

Da der Test gemeinhin recht spät im Entwicklungszyklus stattfindet, entstehen hohe Kosten, um gefundene Fehler zu analysieren und zu korrigieren. Diese Kosten können auch mittels Testautomatisierung kaum gesenkt werden. Nicht unerheblich sind jedoch auch die Aufwände für die Definition der Testfälle, Erstellung der benötigten Testinfrastruktur (Treiber, Stubs, Laufzeitumgebung), Ausführung und Auswertung der Tests. Diese können sehr wohl mittels Automatisierung gesenkt werden.

Um Entwicklungszeit und Kosten zu sparen, versuchen die Unternehmen im Rahmen dieses Projektes daher, ihre Softwaretests zu verbessern und zu automatisieren. Dieser Report untersucht aktuelle Forschungsansätze in dieser Richtung und stellt sie kurz vor. Ziel ist es, einen Überblick zu geben über den aktuellen Stand der Forschung, um begründet Entscheidungen treffen zu können, wie in der industriellen Praxis Testen besser automatisiert werden kann.

1.2 Testen und Testautomatisierung

Beim Testen von Software handelt es sich um eine Tätigkeit, die vorhandene Fehler in einem Stück Software aufdecken soll mit dem Ziel, diese Fehler zu korrigieren. Durch Tests können immer nur Fehler gefunden werden, nicht jedoch kann die Abwesenheit von Fehlern nachgewiesen werden. Die wichtigsten Eigenschaften eines Software-Tests umfassen nach unter anderem:

- Das übersetzte, ausführbare Programm wird mit konkreten Eingabewerten versehen und ausgeführt.

- Das Programm kann in der realen Umgebung getestet werden.

Dabei kann das „Programm“ auch eine einzelne Komponente oder Unit sein, die mittels Treiber und Stubs angesteuert wird.

Automatisierung bedeutet in diesem Zusammenhang, dass einige oder alle der Schritte Testfallerstellung, Testdatengewinnung, Testtreiber- und Stuberstellung, Testausführung, Testauswertung nicht mehr von Menschen manuell ausgeführt werden, sondern ohne menschliches Zutun von Maschinen. Sind an bestimmten Stellen Eingriffe von Menschen notwendig, so spricht man von Teilautomatisierung.

1.3 Aufbau des Reports

Der vorliegende Report ist folgendermaßen aufgebaut: Kapitel 2 beschreibt den Zweck des Reports, und Kapitel 3 gibt einen Überblick über Software-Testen und Testautomatisierung allgemein. Kapitel 4 ist der eigentliche Kern des Reports: Hier werden aktuelle Forschungsansätze im Bereich Testautomatisierung beschrieben. Abschnitt 4.1 widmet sich der vollautomatischen Testfallerstellung aus formalen Anforderungen. Diese Voraussetzungen werden in Abschnitt 4.2 abgeschwächt, indem nur noch UML-basierte Anforderungen gefordert werden. Abschnitt 4.3 widmet sich der Testdatengewinnung, die Abschnitte 4.4 und 4.5 der Testdurchführung und –auswertung. Abschnitt 4.6 betrachtet die Wiederverwendung von Testfällen, Abschnitt 4.7 die Optimierung von (großen) Testsuiten speziell für Regressionstests. In Abschnitt 4.8 werden spezielle Teststrategien für objektorientierte Software diskutiert. Abschnitt 4.9 fasst die geschilderten Ansätze zusammen, und Abschnitt 4.10 versucht, eine kurze Einschätzung der Ansätze zu geben. Referenzen zu weiterführender Literatur sind in Kapitel 5 aufgeführt.

2 Zweck des Reports

2.1 Überblick über aktuelle Forschungsansätze

Da die Aufwände für den Softwaretest in der industriellen Praxis mit zunehmender Größe der entwickelten Systeme weiter wachsen, ist das Interesse an automatisierten Softwaretests groß. Dementsprechend gibt es eine unüberschaubare Vielzahl von Forschungsansätzen und –projekten in dieser Richtung.

Daher versucht dieser Report, die vorhandenen Ansätze nach Phasen des Softwaretests zu gliedern und darzustellen. Aufgrund der schieren Anzahl ist es nicht möglich, alle Ansätze zu erfassen und zu beschreiben. Stattdessen werden jeweils einige ausgewählte Ansätze stellvertretend für ihre Klasse ausführlicher analysiert und vorgestellt.

Diese Vorgehensweise hat den Vorteil, dass sie sowohl einen Überblick gibt, in welchem Bereich sich die Forschung in einer Phase des Softwaretests bewegt, als auch Einblicke in die auftretenden Probleme und zu erwartenden Schwierigkeiten bei der Umsetzung in die Praxis. Bestimmte Details wurden dabei vereinfacht oder vernachlässigt, für weitergehende Informationen sei daher auf die angegebenen Literaturreferenzen verwiesen.

2.2 Zusammenfassen vorhandener Informationen

Ein weiteres Ziel dieses Reports ist, eine Zusammenfassung über Informationen im Bereich Testen und Testautomatisierung zu liefern. Alle Informationen sind selbstverständlich über Veröffentlichungen frei verfügbar, es ist jedoch mit immensem Aufwand verbunden, sie zu sammeln und zu aggregieren.

Daher werden in Kapitel 4 verschiedene Forschungsansätze in einem Detailgrad beschrieben, der die grundsätzliche Idee des jeweiligen Ansatzes und auftretende Probleme hinreichend genau darstellt. Durch diesen Verzicht auf weniger wichtige Details ist es möglich, Informationen über einzelne Ansätze zusammenzufassen, sodass sie einen Überblick über aktuelle Forschungsthemen in einer bestimmten Phase des Softwaretests geben.

3 Software-Testen

3.1 Testen von Software

Systemfehler in industriellen Softwaresystemen können sehr teuer werden: So verursachte die Sprengung der Ariane 5-Rakete [Ber04] 1996 einen Schaden von 800 Millionen Euro, davon 450 alleine durch die zerstörte Nutzlast. Auch Personenschäden können durch Softwarefehler verursacht werden, so geschehen beispielsweise beim Unfall eines Airbus A320-211 im Jahre 1993 in Warschau .

Ein Weg, um die Verlässlichkeit von Softwaresystemen zu erhöhen (und damit die Gefahr, dass Unglücksfälle wie die oben angeführten auftreten, zu senken), ist, das jeweilige Stück Software eingehend zu testen und die dadurch gefundenen Fehler zu korrigieren. Testen umfasst hauptsächlich folgende Aktivitäten:

- Testfälle entwerfen,
- die zu testende Software mit diesen Testfällen ausführen, und
- die Resultate auf aufgetretene Fehler untersuchen.

Verschiedene Studien weisen darauf hin, dass in der industriellen Praxis teilweise über 50% der Kosten für die Softwareentwicklung durch Testen verursacht werden. Dieser Prozentsatz könnte sogar noch zunehmen, da Software in immer weitere Bereiche vordringt und daher auch immer höheren Qualitätsanforderungen entsprechen muss. Damit steigen auch die absoluten Testkosten; es ist offensichtlich, dass Automatisierung hier große Einsparpotenziale birgt.

3.2 Vorgehensweisen beim Testen

Grundsätzlich unterscheidet man zwei Arten von Testfällen, deren Anwendung davon abhängt, welches Wissen man über die zu testende Software hat.

- Weiß man, welche Funktionen die zu testende Software ausführen soll, so können Tests entworfen werden, die genau diese Funktionen ausführen und damit überprüfen, ob sie die erwarteten Resultate liefern. Diese Art von Testen nennt man *Black-Box-Testen* oder *funktionales Testen.*
- Kennt man die interne Funktionsweise der zu testenden Software, so können Tests so entworfen werden, dass sie überprüfen, ob alle internen Ope-

rationen der Spezifikation entsprechen. Diese Art von Testen nennt man *White-Box-Testen* oder *strukturelles Testen.*

Beim Black-Box-Testen werden üblicherweise die funktionalen und nichtfunktionalen Anforderungen an die zu testende Software herangezogen, um Testfälle abzuleiten. Beim White-Box-Testen analysiert man die Kontrollstruktur des zu testenden Programms, um Testfälle abzuleiten, die bestimmte Kriterien der Abdeckung erfüllen.

Die Abdeckungskriterien beim White-Box-Testen fokussieren auf verschiedene Eigenschaften der internen Struktur der zu testenden Software:

- *Anweisungsabdeckung:* Hierbei werden Testfälle so angelegt, dass jede Anweisung mindestens einmal ausgeführt wird. Dies führt zu mäßigem Aufwand, da analysiert werden muss, wie welche Entscheidungen im Programmablauf zu fällen sind, um alle Anweisungen zu erreichen, und die anschließende Ausführung der Testfälle durchaus etwas dauern kann.

- *Pfadabdeckung:* Hierbei werden Testfälle so angelegt, dass jeder Pfad, der im Kontrollfluss der Software möglich ist, mindestens einmal durchlaufen wird. Dies ist meist recht aufwendig, da oft sehr viele Entscheidungen und damit Pfade in einem Programm vorhanden sind und damit sehr viele Testfälle entstehen.

Beim Black-Box-Testen unterscheidet man auch mehrere Arten, wie die Anforderungen verarbeitet werden:

- *Äquivalenzklassen:* Die Unterteilung in Äquivalenzklassen geht von der Annahme aus, das sich die Inputs und Outputs eines Programms in Klassen einteilen lassen, sodass sich alle Elemente, die einer Klasse zugeordnet sind, gleichartig verhalten. Daraufhin werden ein oder mehrere Repräsentanten der Klasse getestet und davon auf das Verhalten der ganzen Klasse geschlossen. Die Schwierigkeit liegt hierbei, die Klassen zu bestimmen und repräsentative Werte zu wählen, die getestet werden.

- *Grenzwertanalyse:* Die Grenzwertanalyse kann als ein Spezialfall der Äquivalenzklassentestens gesehen werden. Die zugrunde liegende Annahme der Klassen ist die selbe, anstatt jedoch Repräsentanten aus der Klasse zu wählen, konzentriert man sich bei der Grenzwertanalyse auf Werte direkt auf den Grenzen der einzelnen Klassen, sowie an Werte dicht auf beiden Seiten der Grenze.

- *Zufallstesten:* Bei dieser Testmethode werden die Testdaten zufällig erzeugt und das zu testende Programm damit beschickt. Die Testergebnisse werden aufgezeichnet und können dann inspiziert werden.

- *Entscheidungstabellen:* In Entscheidungstabellen werden Variablen mit ihren möglichen Belegungen gelistet sowie die jeweils erwartete Entscheidung. Diese Vorgehensweise wird beispielsweise in Situationen verwendet, bei denen abhängig von mehreren logischen Eingangswerten verschiedene Aktionen ausgeführt werden sollen. Entscheidungstabellen erzwingen durch ihren Aufbau die logische Vollständigkeit des Tests. Sie können direkt erstellt werden oder indirekt, beispielsweise ermittelt aus Ursache-Wirkungs-Graphen.

Beide Testarten (Black-Box, White-Box) schließen sich nicht aus, sie sind im Gegenteil als komplementär zu betrachten, da sie unterschiedliche Arten von Fehlern aufdecken.

3.3 Testebenen

Software wird üblicherweise auf mehreren Ebenen getestet. Begonnen wird auf einer sehr feingranularen Ebene, dann werden schrittweise die getesteten Objekte vergrößert, bis schließlich das gesamte System getestet wird. Die Ebenen sind im Einzelnen:

- *Unit-Ebene:* Eine Unit ist die kleinste Einheit der zu testenden Software, also beispielsweise eine einzelne Klasse. Obwohl grundsätzlich alle Testarten denkbar sind, finden Unit-Tests meist als White-Box-Tests statt. Oft werden Unit-Tests direkt von den Entwicklern durchgeführt, sodass das dafür erforderliche Wissen über die Unit-Interna vorhanden ist.

- *Integrationsebene:* Das korrekte Funktionieren aller Units garantiert in keinster Weise, dass die Units auch im Zusammenspiel wie vorgesehen funktionieren. Dies wird beim Integrationstesten überprüft. Hierzu werden mehrere Units zusammengesetzt und ihre Interaktion getestet. Während dem Integrationstesten wird schrittweise das gesamte Programm zusammengesetzt, während gleichzeitig Tests ausgeführt werden, die beispielsweise Fehler in den Schnittstellen der schon erfolgreich einzeln getesteten Units aufdecken sollen. Auch Integrationstests werden aufgrund der zu testenden Objekte (Schnittstellen) meist als White-Box-Tests ausgeführt.

- *Systemebene:* Das Ziel eines Systemtests ist es, eine gewisse Sicherheit herzustellen in Bezug auf das spezifikationsgemäße Funktionieren des Gesamtsystems. Insofern stellt der Systemtest eine Vorbereitung auf den Akzeptanztest durch den Benutzer dar, da die vom Benutzer spezifizierte und erwartete Leistung (sowohl funktional als auch nicht-funktional) im Ganzen getestet wird. Bei Tests auf Systemebene herrschen Black-Box-Techniken vor, da das Hauptaugenmerk auf der Erfüllung der Anforderungen liegt, nicht auf der Implementierung.

- *Akzeptanzebene:* Bei diesen Tests überprüfen typischerweise Repräsentanten des endgültigen Benutzerkreises die Funktionsweise des Systems, d.h. sie führen am laufenden System Arbeiten aus, wie sie typisch für die spätere Benutzung sind. Der Akzeptanztest stellt den letzten Test vor der formalen Auslieferung und Inbetriebnahme beim Kunden dar.

3.4 Terminologie

Generell wird beim Testen unterschieden zwischen Fehlverhalten *(failure)*, Fehlern *(fault)* und Fehlerursachen *(error)*. Unter einer Fehlerursache versteht man eine menschliche Handlung, aufgrund derer ein Fehler entsteht. Ein Beispiel für eine Fehlerursache wäre ein Denkfehler eines Programmierers, der zu einem Fehler im Programm in Form einer falschen Anweisung/Codezeile führt. Dieser Fehler ist von außen nicht sichtbar, selbst mit diesem Fehler kann sich das Programm noch korrekt verhalten, etwa falls die fehlerhafte Zeile nicht zur Ausführung kommt. Wird sie ausgeführt, so resultiert daraus ein Fehlverhalten des Programms, welches von außen beobachtet werden kann.

Aus dieser Definition folgt, dass Fehler nicht direkt zu den immensen Kosten führen, die in Abschnitt 3.1 aufgeführt werden. Kosten werden immer durch Fehlverhalten von Software verursacht, welches durch Fehler verursacht wird. Das bedeutet, dass ein aufgetretenes Fehlverhalten immer auf die Anwesenheit von Fehlern hinweist, die Anwesenheit von Fehlern jedoch nicht zwingend zu Fehlverhalten führt. Das hat ganz praktische Auswirkungen für den Test: Wenn im Test kein Fehlverhalten der Software beobachtet wird, dann können trotzdem Fehler enthalten sein, die einfach kein beobachtbares Fehlverhalten der Software provoziert haben!

Im weiteren Verlauf dieses Reports wird der Begriff „Fehler“ synonym für die hier aufgeführten Definitionen „Fehler“ und „Fehlverhalten“ benutzt, da eine Unterscheidung nicht wirklich notwendig ist.

3.5 Herausforderungen beim Testen

Da selbst fehlerhafte Software für viele Eingabewerte das erwartete Verhalten zeigen kann, sind gute Testfälle mit einem hohen Potential, Fehler aufzudecken, entscheidend. Meist können nur Fehler, die auch zu einem von außen beobachtbaren Fehlverhalten des Programms führen, durch Testfälle entdeckt werden. Daher hängt die Güte eines Tests stark von seinen Testfällen ab.

Im Idealfall würden Testfälle so gewählt werden, dass die erfolgreiche Ausführung der Testfälle implizieren würde, dass keine Fehler mehr im Programm vorhanden sind. Dies ist jedoch aufgrund theoretischer und praktischer Einschränkungen nicht möglich. Theoretische Beschränkungen besagen einerseits, dass die vollständige Abwesenheit von Fehlern mittels Tests nicht bewie-

sen werden kann. Andererseits könnte ja das Programm mit allen möglichen Kombinationen von Inputdaten, die auftreten können, getestet werden; tritt hierbei kein Fehler auf, so wäre das Programm zumindest für den vorgesehenen Anwendungsbereich frei von Fehlverhalten. Dies ist jedoch wiederum praktisch nicht möglich, da die möglichen Inputdaten oftmals unendlich große Mengen darstellen.

Ein weiterer Freiheitsgrad ist der jeweilige Zustand des Systems. Wenn ein System in einem Zustand A einen bestimmten Eingabestrom E korrekt verarbeitet, so kann es durchaus denselben Eingabestrom E in einem von A unterschiedlichen Zustand B fehlerhaft verarbeiten. Dieses Verhalten ist verwandt mit dem Verhalten von Objekten, siehe auch Abschnitt 4.8.

Gleichzeitig kostet die Entwicklung jedes Testfalles Zeit, die Ausführung verbraucht Maschinenzeit, und weitere Zeit wird benötigt zur Aufzeichnung und Auswertung der Ergebnisse. Also sollte die Anzahl der Testfälle minimiert werden. Dies führt zu zwei Hauptzielen bei praktischen Software-Tests:

- Die Anzahl der entdeckten Fehler soll maximiert werden.

- Die Anzahl der Testfälle soll minimiert werden.

Da sich diese beiden Ziele widersprechen, ist eine sinnvolle Auswahl der Testfälle essentiell. Damit stellt sich die Frage, wann denn „genug“ getestet wurde. In der Praxis wird dieses „genug“ oft durch zeitliche Beschränkungen vorgegeben (Release-Datum), doch eine allgemeine Antwort kann es nicht geben. Mit der Testdauer eng zusammen hängt auch die Restfehlerzahl, es hängt stark vom System ab, wie viele in der Software verbleibende Fehler akzeptabel sind. Ist bei einer Decodiersoftware für Videoströme ein Pixelfehler aufgrund falscher Berechnungen pro Sekunde nicht weiter schlimm (pro Sekunde werden 25 Einzelbilder angezeigt, d.h. ein falsches Pixel in einem der Bilder fällt nicht weiter auf), so wird ein Rechenfehler pro Sekunde in einem Flugkontrollmodul an Bord eines Flugzeuges nicht toleriert werden.

3.6 Automatisierung

Unter Automatisierung versteht man im Zusammenhang mit Softwaretests, dass einige oder alle der Schritte Testfallerstellung, Testdatengewinnung, Testtreiber- und Stuberstellung, Testausführung und Testauswertung nicht mehr von Menschen ausgeführt werden, sondern ohne menschliches Zutun von Maschinen. Sind an bestimmten Stellen Eingriffe von Menschen notwendig, so spricht man von Teilautomatisierung.

Zumindest die in Abschnitt 3.5 erhobene Forderung nach der Minimierung der Anzahl der Testfälle lässt sich durch Automatisierung etwas abschwächen.

Meist ist Maschinenzeit deutlich billiger als menschliche Arbeitszeit, sodass mittels automatisch generierter Testsuiten effektiv Kosten gespart werden. Die möglicherweise ineffizienter und damit größer werdenden Testsuiten erzeugen zwar höhere Maschinenkosten, dies wird jedoch meist durch die Reduzierung des Personalaufwands mehr als ausgeglichen. Weiterhin lässt sich durch logische Analysen der zu testenden Software auch eine effizientere Pfadabdeckung als durch manuelle Testfallerstellung erreichen, sodass auch – bei gleich bleibender Anzahl der Testfälle – die Anzahl der gefundenen Fehler steigt.

Zur Automatisierung von Softwaretests existiert eine Vielzahl von Forschungsansätzen. Einige davon werden, stellvertretend für ihre jeweilige Klasse, in Kapitel 4 vorgestellt.

4 Aktuelle Forschungsansätze

4.1 Vollautomatische Testfallgenerierung aus formalen Anforderungen

Während bei der automatisierten Ausführung von Tests schon verschiedene praktisch einsetzbare Methoden und Werkzeuge existieren (z.B. xUnit), so wird die Erstellung von Testfällen für diese Werkzeuge meist noch in Handarbeit betrieben. Dies hat mehrere Nachteile:

- Die manuelle Codierung von Testfällen ist zeitraubend. Der Eindruck der „verlorenen Zeit“ wird noch zusätzlich dadurch verstärkt, dass das Schreiben von Testfällen meist als unproduktive Arbeit angesehen wird, da ja „nur“ Testfälle erstellt und keine Features des Produkts implementiert werden. Maße für den Fortschritt der Arbeit, wie z.B. Function Points, orientieren sich meist nur am die Anforderungen erfüllenden Softwareprodukt, nicht jedoch an Begleitwerk wie Testfällen. Unter diesem Gesichtspunkt raubt die Testfallerstellung gerade in zeitlich knappen Projekten wertvolle Zeit, die eigentlich für die Implementierung von Features genutzt werden könnte.

- Die codierten Testfälle stellen wieder Software dar, die eigentlich mit den gleichen Maßnahmen der Qualitätssicherung behandelt werden müsste wie das Hauptprodukt. Das bedeutet vor allem, dass auch Testfälle selbst durchaus fehlerhaft sein können: Einerseits decken sie unter Umständen nicht alle Fehlerfälle ab, sind also unvollständig, andererseits können sie auch im Falles eines tatsächlich auftretenden Fehlers diesen unter Umständen nicht anzeigen, bzw. fälschlicherweise einen Fehler melden, obwohl keiner vorliegt. In diesem Fall ist das Verhalten der Testfälle nicht korrekt. Beides führt zu Verzögerungen bei der Entwicklung des Hauptproduktes bzw. zu unentdeckten Fehlern. Implizit wird meist davon ausgegangen, dass Testfälle korrekt sind, die Vollständigkeit kann mittlerweile durch manche Werkzeuge, die beispielsweise die Testabdeckung ermitteln, zumindest rudimentär überprüft werden. Um Vollständigkeit und Korrektheit echt zu gewährleisten, müssten die Testfälle gegenüber den Anforderungen verifiziert werden, was nach heutigem Stand der Forschung sehr schwierig bis unmöglich ist.

- Von Hand erstellte Testfälle sind – genau wie andere Software auch – ohne spezielle Maßnahmen nur schwer zu warten oder an neue Versionen des zu testenden Produkts anzupassen. Vor allem bei Software, die über viele Generationen gewachsen ist, nehmen die Testsuiten oft Ausmaße an, die denen des Hauptproduktes gleichkommen oder diese sogar überschreiten. Die

Wartung dieser Testmonster ist extrem zeitaufwendig und fehlerträchtig, und auch die Ausführung aller Tests kann sehr lange dauern (vgl. auch Abschnitte 4.6 und 4.7).

Aus diesen Gründen wird seit einigen Jahren bereits rege an der automatischen Testfallableitung aus Anforderungen geforscht. Diese würde fast alle der o.g. Probleme der manuellen Testfallerstellung lösen: Eine automatische Erstellung würde den Programmierer keine Zeit kosten, die Testfälle wären gleichzeitig vollständig und korrekt in Bezug auf die Anforderungen (wenn der Ableitungsalgorithmus dies ist), und die Wartung bei geänderten Anforderungen würde sich auf eine Neugenerierung beschränken.

Da es jedoch in den meisten Fällen entweder extremen Aufwand erfordert, alle Anforderungen formal zu spezifizieren, oder dies nach dem aktuellen Stand der Forschung nicht möglich ist, sind über die reine Theorie herausgehende Ansätze in Bereichen angesiedelt, die das Kriterium der Formalität der Anforderungen abschwächen. Ein Ansatz, der auch (zumindest in begrenztem Umfang) praktisch einsetzbar ist, beschreibt ein Framework für automatisches Black-Box Testen von komponentenbasierter Software .

Der Ansatz ist dreigeteilt:

(1) Automatische Generierung von Komponenten-Testtreibern,

(2) automatische Generierung von Testdaten, und

(3) (semi-) automatische Generierung von Wrappern, die als Testorakel dienen.

Die Orientierung an Komponenten rührt daher, dass Software in zunehmendem Maße verteilt entwickelt wird, und daher das globale Wissen über das gesamte zu entwickelnde System nicht bei jedem Entwickler komplett vorhanden ist. Um dies zu kompensieren, bietet sich der Komponentenansatz an. Ein weiterer Grund ist die zunehmende Komplexität von Software, die auch nur mittels Teile-und-Herrsche überblickt werden kann. Auch hierfür bieten sich Komponenten an. In beiden Fällen ist die frühzeitige Fehlerfindung (deutlich vor der Integration des Gesamtsystems) wichtig.

Die einzige prinzipielle Voraussetzung für die Anwendung des Frameworks liegt darin, dass das Verhalten der Komponenten formal spezifiziert sein muss. Prototypische Entwicklungen für die Generierung von Testtreibern, -daten und Wrappern basieren auf der Verhaltens-Spezifikationssprache RESOLVE, sind aber prinzipiell auch für andere Sprachen, die explizite Vor- und Nachbedingungen unterstützen, realisierbar.

Beispielhaft sei hier die Spezifikation einer einfach verketteten Liste in RESOLVE erläutert. Die Liste besteht aus einer Menge einfach und ohne Zyklen miteinander verketteter Elemente. Es gibt kein Konzept eines Cursors o.ä., sondern das aktuell gewählte Element ist implizit durch eine Partitionierung der Liste in den Teil, der links vom aktuellen Element liegt, und den Teil, der rechts davon liegt,

```
concept One_Way_List
context
  global context
    facility Standard_Boolean_Facility
  parametric context
    type Item
interface
  type List is modeled by (
    left  : string of math[Item]
    right : string of math[Item]
  )
  exemplar s
  initialization
    ensures s = (empty_string, empty_string)

  operation Move_to_Start (alters s: List)
    ensures s = (empty_string, #s.left * #s.right)

  operation Move_to_Finish (alters s: List)
    ensures s = (#s.left * #s.right, empty_string)

  operation Advance (alters s : List)
    requires s.right /= empty_string
    ensures there exists x : Item
      (s.left = #s.left * <x> and #s.right = <x> * s.right)

  operation Add_Right (alters s : List, consumes x : Item)
    ensures s = (#s.left, <#x> * #s.right

  operation Remove_Right (alters  s : List, produces x : Item
    requires s.right /= empty_string
    ensures s.left = #s.left and  #s.right = <x> * s.right

  operation Swap_Rights (alters s1 : List, alters s2 : List)
    ensures s1.left = #s1.left and s1.right = #s2.right
        and s2.left = #s2.left and s2.right = #s1.right

  operation At_Start (preserves s : List) : Boolean
    ensures At_Start iff s.left = empty_string

  operation At_Finish (preserves s : List) : Boolean
    ensures At_Finish iff s.right = empty_string

end One_Way_List
```

Abbildung 1: RESOLVE-Spezifikation einer verketteten Liste

bestimmt. Weiterhin gibt es Operationen, um zum Anfang und zum Ende der Liste zu springen, ein Element weiter zu springen, und Elemente hinzuzufügen und zu entfernen. Schließlich lassen sich die Endelemente zweier Listen vertauschen und es lässt sich feststellen, ob das aktuelle Element am Anfang oder am Ende einer Liste ist. Diese Eigenschaften sind in Abbildung 1 in RESOLVE beschrieben.

Das Schlüsselwort **`requires`** beschreibt hier eine Vorbedingung einer Operation, Nachbedingungen werden durch **`ensures`** symbolisiert.

Die formale Spezifikation von Vor- und Nachbedingungen ist deutlich einfacher als eine komplette Verhaltensspezifikation, und daher auch realistischerweise für mehr als einige wenige akademische Fälle zu erstellen. Aufgrund dieser Spezifikation der Vor- und Nachbedingungen lassen sich nun automatisch Testtreiber und –daten generieren, und im Idealfall auch Wrapper. Dies wird im folgenden beschrieben.

Dreh- und Angelpunkt des Frameworks sind *Wrapper*, die den Komponenten eingebaute Testfähigkeiten verleihen (Abbildung 2). Diese Wrapper kapseln die Komponente, sind jedoch völlig transparent für Clients, die auf die Komponente zugreifen. Damit können Clients ohne jegliche Änderungen auf solchermaßen verpackte Komponenten zugreifen. Gleichzeitig ermöglicht der Wrapper, den internen Zustand der Komponente zu beobachten, und Vor- und Nachbedingungen sowie Invarianten zu überprüfen. Sollten Auffälligkeiten auftreten, so wird dies entdeckt, bevor andere Komponenten davon betroffen sind, d.h. falls gewünscht können Korrekturen vorgenommen werden.

Man unterscheidet in diesem Zusammenhang Ein- und Zweiwegwrapper. Einwegwrapper überprüfen nur, ob beim Aufruf einer Operation/Methode alle Vorbedingungen erfüllt sind, wohingegen ein Zweiwegwrapper auch die Einhaltung aller Nachbedingungen überprüft. Sollten Vor- oder Nachbedingungen verletzt sein, so sind mehrere Verhaltensweisen denkbar: Die Komponente könnte trotzdem benutzt werden, als ob die Bedingungen korrekt wären, und nur eine Nachricht nach außen gegeben, was genau verletzt wurde, oder der Aufruf wird abgelehnt, da ein Clientfehler entdeckt wurde.

Es ist wichtig, dass der Wrapper die ummantelte Komponente nicht beeinflusst. Deshalb werden alle Operationen, die der Wrapper ausführt, nicht auf der Komponente selbst ausgeführt, sondern auf einer Kopie des abstrakten Zustands der Komponente. Der dadurch entstehende Overhead ist natürlich während dem Normalbetrieb der Komponente unerwünscht, weshalb der Wrapper mittels Deklarationsänderungen ein- und ausgeschaltet werden kann.

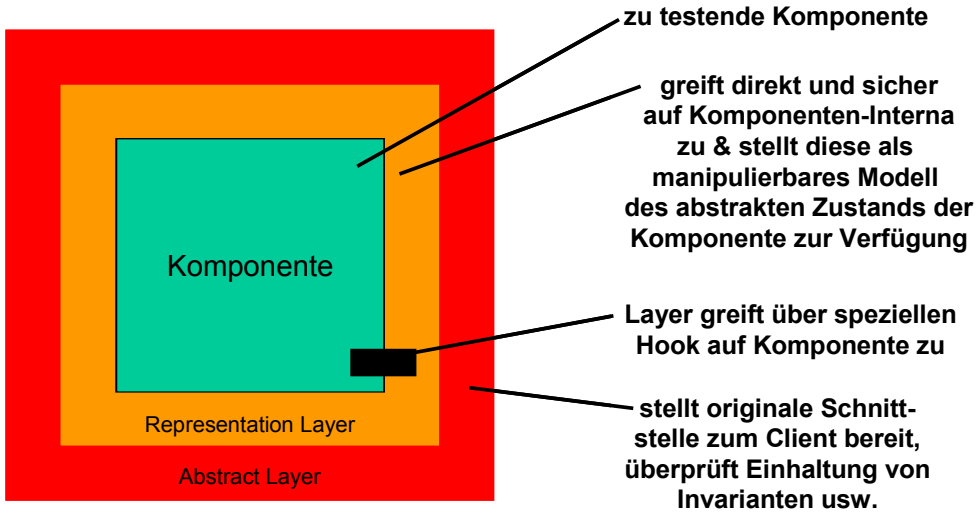

Abbildung 2: Komponentenwrapper

Für jeden ausgeführten Testfall führt nun der Wrapper zusätzlich eine Menge von internen Konsistenzchecks durch, die oftmals Fehler aufdecken, die durch reine Outputkontrolle nicht aufgefallen wären. Wenn Fehler gefunden werden, dann ist sofort auch der komplette interne Zustand des getesteten Objekts verfügbar, was die Fehlersuche deutlich vereinfachen kann. Mittels des Wrappers kann dieser interne Zustand sogar modifiziert werden, etwa für Debugging-Zwecke.

Die Konstruktion eines Wrappers ist im Idealfall komplett automatisierbar. Das Interface kann von der Komponente übernommen werden, lediglich die Überprüfung der Vor- und Nachbedingungen ist manchmal problematisch. Mittels des in [ESS+98] beschriebenen Ansatzes können viele Vor- und Nachbedingungsanforderungen automatisch in Code umgesetzt werden. Vor allem Bedingungen mit Quantoren sind jedoch nicht mechanisch in Code umsetzbar. Für dieses Problem bieten sich drei Lösungen an:

(1) Halbautomatische Generierung. Für alle Bedingungen, wo es klappt, wird automatisch Code erzeugt, und die restlichen müssen von Hand verarbeitet werden. Dies erfordert wenig methodischen Aufwand und führt immer noch zu Zeitersparnissen.

(2) Dynamische Überprüfung der Bedingungen. Die momentan verfügbaren Verifizierungstools haben meist Probleme mit Quantifizierungen und deren statischer Verifikation. Zur Laufzeit jedoch, wenn alle Variablen Werte haben, ist das Problem deutlich einfacher, sodass Wrapper denkbar sind, die eine solche Verifikationsengine einbinden und zur Laufzeit aufrufen.

(3) „Gepanzerte“ Komponenten. Hier überprüft der Wrapper zur Laufzeit das Verhalten der Komponente mit dem einer bekannt korrekten Implementierung. Diese Vorgehensweise ermöglicht es z.B. auch, bei auftretenden Fehlern in der zu testenden Komponente den Testlauf trotzdem korrekt weiterlaufen zu lassen, indem die korrekten Daten der Referenzimplementierung übernommen oder sogar „gewaltsam“ in die Komponente eingebracht werden.

Testtreiber können gesehen werden als Interpreter, die Testfälle in Aktionen auf der zu testenden Komponente umsetzen. Dieser Auslegung folgend, wird der Interpreter konstruiert, indem das Komponenteninterface geparst und die Operationen identifiziert werden. Sollten dabei ungültige Methodenaufrufe herauskommen, so filtert dies der Wrapper weg, sodass hier keine Probleme zu erwarten sind.

Basierend auf dieser Strategie wurde ein Testtreiber-Generator entwickelt, der auf RESOLVE-Spezifikationen aufbaut und Testtreiber in C++ produziert. Die Testfälle selbst werden daher in einer C++-ähnlichen Sprache spezifiziert. Erste Erkenntnisse haben gezeigt, dass der Interpreteransatz bei großen Testsets zu signifikanten Zeiteinsparungen führen kann. Probleme können jedoch auftreten bei Inversion-of-control Situationen oder bei Komponenten, die erhebliche Anteile menschlicher Interaktion besitzen.

Zur Generierung von Black-Box *Testdaten* aus Verhaltensspezifikationen von Komponenten existieren verschiedene Strategien [Bei95]. Eine dieser Strategien beschreibt die Konstruktion von Flussgraphen aus Verhaltensspezifikationen. Basierend auf den so generierten Flussgraphen können dann angepasste White-Box Teststrategien verwendet werden, und somit auch deren Lösungen auf entstehende Fragen wie „Welche Kanten sollten verfolgt werden“ und „Wie sollten Testdaten selektiert werden“. Heuristiken haben sich hier als praktikabel erwiesen [Edw00]. In Kombination mit Wrappern können auch zu optimistische Heuristiken eingesetzt werden, da der Wrapper ungültige Testfälle und –daten aussortiert.

Erste Experimente mit den beschriebenen Techniken sind ermutigend. Als Testkomponenten dienten C++-Implementierungen eines *Stack*, einer *Queue*, der schon beschriebenen *One-Way-List*, und einer *Partial Map*. Fehler wurden in diese Komponenten mittels *Expression-Selective Mutation* eingebracht. Dabei entstandene äquivalente Mutanten wurden von Hand identifiziert und entfernt, sodass alle Mutanten garantiert unterschiedlich waren. Tests haben ergeben, dass das Black-Box Analogon zum *All Uses*-Kriterium extrem effektiv Fehler findet: In drei der vier getesteten Komponenten wurden auf diese Weise sämtliche eingebrachten Fehler gefunden, in der Partial Map immerhin noch 87,3%.

Weiterhin wurden die Fehler, die durch Beobachtung des Komponentenoutputs gefunden wurden, separat von denen, die durch die Beobachtung der Komponenteninterna per Wrapper entdeckt wurden, gezählt. Die Wrapper wurden jedoch manuell erstellt. Es zeigte sich, dass die Benutzung von Zweiwegwrappern zu 8%-200% mehr gefundenen Mutanten gegenüber reiner Outputbeobachtung führte. Der größte Zuwachs ergab sich in den schwächsten Testsets, z.B. in einem Testset für die *Queue* stieg die Detektionsrate von 18,5% ohne Wrapper auf 55,6% mit Zweiwegwrapper. Außerdem hätten die Testsets, die sämtliche Fehler detektierten, dieses Ergebnis ohne Wrapper nicht erreicht, da nicht alle Fehler durch Beobachtung des Komponentenoutputs zu erkennen waren.

4.2 Testfallerstellung aus halbformalen Anforderungen (UML-basiertes Testen)

Die Unified Modeling Language (UML) erfreut sich in der Software-Entwicklung großer Beliebtheit. Der Markt hält viele UML-Werkzeuge zur visuellen Modellierung bereit, die als Basis für Design und Implementierung von Software oder Komponenten dienen. Ein nahe liegender Gedanke ist nun, die auf diese Weise erstellten Modelle auch zum Testen der erstellten Software heranzuziehen.

In [AO99] wird eine generelle Vorgehensweise zur Generierung von Testfällen aus Zustandsdiagrammen dargestellt. Sie wird anhand von Beispielen in SCR- und UML-Notation vorgeführt. Die generelle Vorgehensweise ist wie folgt:

- Bedingungen für Zustandsübergänge feststellen. Diese sind in UML direkt in den Statecharts codiert.

- Anforderungen für die Abdeckung der Zustandsübergänge feststellen.

- Anforderungen für die *Full-Predicate* Abdeckung feststellen. Ein *Predicate* besteht aus *Clauses*, die *Full-Predicate* Abdeckung ist definiert als der Zustand, in dem eine *Clause* die Werte „wahr“ und „falsch“ annimmt und die anderen *Clauses* solche Werte haben, dass die Werte der betrachteten *Clause* mit denen des gesamten *Predicates* übereinstimmt. Man kann die Anforderungen mittels Wahrheitstabellen oder Ausdrucksbäumen erstellen.

- Anforderungen für Zustandsübergang-Paare feststellen. Ein Zustandsübergang-Paar ist ein geordnetes Paar von Bedingungswerten, die je einen Input und einen Output eines Zustands darstellen. Zuerst werden alle diese Paare identifiziert, und dann durch die *Predicates* aus dem Spezifikationsgraphen ersetzt.

- Komplette Anforderungen für die Testsequenzen feststellen. Dazu werden alle Zustände identifiziert und dann transformiert in Sequenzen von Bedin-

gungen, die zu diesen Zuständen führen. Damit sind alle Testanforderungen bestimmt.

- Aus den Testanforderungen werden Testspezifikationen erstellt. Dazu gehören Präfix-Werte, Testfallwerte, Verifizierungsbedingungen, Exit-Bedingungen und erwartete Outputs.

- Aus den Testspezifikationen werden Testskripte erstellt. Dieser Schritt ist der einzige, der Kenntnis über die Implementierung der zu testenden Software verlangt.

Grundsätzlich ist die beschriebene Vorgehensweise automatisierbar, da die Informationen aus den UML-Statecharts gezogen werden können und die verwendeten Algorithmen implementierbar sind.

Eine mögliche Realisierung von UML-basiertem Testen ist in [HIM00] beschrieben. Mit dem Fokus auf Integrationstesten beschreiben die Autoren die Integration der Generierung von Komponenten-Testfällen sowie deren Ausführung mit kommerziell erhältlichen UML-Werkzeugen wie Rational Rose. Das dynamische Verhalten der Komponenten wird in UML beschrieben, beispielsweise in Sequenzdiagrammen. Um ein dynamisches Verhalten exakt zu beschreiben, sind jedoch unter Umständen viele solche Sequenzdiagramme nötig, sodass oft auf Statecharts zurückgegriffen wird. Diese Statecharts sind auch die Basis für das beschriebene Vorgehen.

Zuerst wird das dynamische Verhalten der Komponente in Statecharts beschrieben. Dabei kann ein Statechart pro Komponente anfallen, aber auch, bei komplexeren Systemen, mehrere. Sind bei der Kommunikation mittels eines bestimmten Protokolls beispielsweise mehrere Komponenten beteiligt, so wird für jede davon ein Statechart erstellt. Außerdem können die Entwickler über die Statecharts und deren Interaktion hinaus noch spezielle Testanforderungen definieren, die Größe und Komplexität der zu erstellenden Testsuite beeinflussen.

Anschließend werden die einzelnen Statecharts zu einem globalen Verhaltensmodell aggregiert. Dabei bleibt das Verhalten der einzeln beschriebenen Komponenten erhalten. Um die Komplexität dieses globalen Verhaltensmodells zu kontrollieren, können einzelne Komponenten in Subsysteme gruppiert werden. Um das globale Modell zu erhalten, werden die einzelnen Statecharts als endliche Mealy-Automaten angesehen, deren Produkt ein zusammengesetzter Automat ist. Dieses Produkt enthält das Verhalten aller Statecharts und reagiert auch auf Stimulation entsprechend der Spezifikation seiner Komponenten.

Aus dem globalen Verhaltensmodell des zu testenden Systems können nun (Integrations-) Testfälle abgeleitet werden. Hierbei wird angenommen, dass mittels Unit-Tests sichergestellt ist, dass sich die zu integrierenden Komponenten

gemäß ihrer Spezifikation verhalten. Die erstellten Tests stellen wiederum sicher, dass sich das *System* gemäß seiner Spezifikation verhält. Als zugrunde liegende Technologie wird ein selbst entwickeltes *Test Development Environment* (TDE) verwendet, welches ein in einer *Test Specification Language* (TSL) erstelltes Test-Design verarbeitet. Die TSL basiert auf Kategorien und Partitionen von verhaltensmäßigen Äquivalenzklassen.

Ein TSL-Design wird nun abgeleitet aus dem globalen Verhaltensmodell mittels Abbildung der Zustände und Zustandsübergänge auf TSL-Kategorien, TSL-Partitionen und Auswahl. Hierbei symbolisieren Zustände die Äquivalenzklassen und werden deshalb auf Partitionen abgebildet. Zustandsübergänge werden als Auswahl der Kategorie/Partition abgebildet. Jeder Zustandsübergang definiert eine Auswahl für den aktuellen Zustand und kombiniert einen Testdatenstring (die Anmerkungen zum gesendeten und empfangenen Event) mit einer Referenz zum nächsten Zustand. Ein Endzustand wird durch einen leeren Testdatenstring dargestellt.

Die Generierung von Testfällen erfolgt mittels rekursiver, gerichteter Graphen. Abdeckung kann mittels zwei Arten von Anforderungen beeinflusst werden: Generative Anforderungen beschreiben, welche Testfälle (Wege durch den Erreichbarkeitsbaum des Graphen) instanziiert werden, und einschränkende Anforderungen bewirken, dass bestimmte Testfälle wieder fallen gelassen werden. Die Generierung funktioniert sowohl für Unittests (einzelne Komponente, Standard-Abdeckungskriterium: Alle Pfade müssen mindestens einmal durchlaufen werden) als auch für Integrationstests (mehrere Komponenten, Standard-Abdeckungskriterium: Alle Zustandsübergänge, die Komponenteninteraktion beinhalten, müssen mindestens einmal durchlaufen werden). Aus den solchermaßen erstellten Testfällen können direkt Testtreiber abgeleitet werden. Ein Testfall besteht aus einer Sequenz von SEND- und RECEIVE-Events, die auf entsprechende Objekte abgebildet werden können.

Die beschriebenen Mechanismen wurden in einer Testumgebung bei Siemens Corporate Research implementiert und werden genutzt, um UML-basiert Testfälle zu erstellen und als COM- bzw. CORBA-Objekte bereitzustellen.

4.3 Testdatengewinnung

Die Testdatengewinnung lässt sich in verschiedene Klassen einteilen, abhängig davon, wie viel Wissen über die zu testende Software vorhanden ist. Wenn nichts über die Implementierung bekannt ist, so fällt es schwer, gezielt und sinnvoll Testdaten so auszuwählen, dass sie „kritische“ Bereiche besonders intensiv testen und andere Bereiche weniger eingängig. Allenfalls kann noch anhand der Spezifikation (falls vorhanden) ermittelt werden, welche Grenzbereiche möglicherweise problematisch werden könnten, jedoch ohne die Gewissheit, dass dies tatsächlich so ist, oder dass die kritischen Stellen nicht

in ganz anderen Teilen des Programms stecken. In diesem Falle wird oft das so genannte Zufallstesten eingesetzt, welches Testdaten zufällig generiert und diese dann in einem Black-Box Verfahren für die Tests verwendet.

Ist hingegen die Implementierung der Software bekannt, kann eine gezielte Auswahl von Testdaten anhand der Testkriterien und der Programmstruktur stattfinden. Auf diese Weise kann sichergestellt werden, dass alle kritischen Elemente der Software getestet werden, und auch der Testaufwand kann gesteuert werden, z.B. indem wichtige Teile intensiver getestet werden als andere (White-Box Verfahren).

4.3.1 Zufallstesten

Beim Zufallstesten werden die Daten, die als Input für die zu testende Komponente verwendet werden, zufällig gewonnen, d.h. entweder echt zufällig, beispielsweise auf Basis des Rauschens kosmischer Strahlung, oder Pseudo-zufällig anhand von Algorithmen. Gute Algorithmen erreichen fast die gleiche Zufälligkeitsgüte wie rauschbasierte Ansätze, haben jedoch den Vorteil, dass bei gleichen Ausgangsbedingungen („Seed“) die gleiche Folge von Zahlen herauskommt: Wenn Fehler aufgetreten sind, so können (identische) Tests wiederholt werden. Ein Nachteil ist inhärent: Die Unvorhersagbarkeit der Methode. Da es sich um zufällig erzeugte Daten handelt, lässt sich prinzipiell nichts Genaues über Abdeckung u.ä. aussagen. Statistische Aussagen sind selbstverständlich möglich, diese sind jedoch immer mit einer gewissen Unsicherheit behaftet.

Zufallstesten muss nicht immer in seiner Reinform angewandt werden. Es ist durchaus möglich, Tests deterministisch anzulegen und nur in bestimmten Teilen auf Zufallsdaten zurückzugreifen, beispielsweise in bestimmten Teilen eines ansonsten fest vorgegebenen Eingabestroms.

Die Forschung in diesem Gebiet beschränkt sich auf die Gewinnung zufälliger Daten. Da meist die Güte von bekannten Zufallszahlen-Algorithmen ausreicht, ist es nicht schwer, hier praktikable Lösungen zu finden bzw. zu entwickeln. Deshalb gehen wir an dieser Stelle nicht weiter auf Zufallstesten ein.

4.3.2 Gezielte Testdatenauswahl

Bei der gezielten Testdatenauswahl ist entscheiden, welche Art von Tests man durchführen möchte. Bei White-Box-Tests müssen die Testdaten so selektiert werden, dass die internen Strukturen der zu testenden Software so gut wie möglich überprüft werden, wohingegen bei Black-Box-Tests die Testdaten so ausgewählt werden müssen, dass die Anforderungen möglichst gut überprüfen. Im Folgenden werden Ansätze für sowohl für White-Box-Tests als auch für Black-Box-Tests vorgestellt.

White-Box-Tests

Die gezielte Testdatenauswahl für White-Box-Tests setzt Kenntnisse über die zu testende Software voraus. Testdaten können so selektiert werden, dass z.B. jeder Pfad mindestens einmal ausgeführt wird *(Path Coverage)*, oder jede Anweisung *(Statement Coverage).* Dies wird allerdings mit zunehmender Programmgröße sehr aufwendig, sodass es irgendwann nicht mehr praktikabel ist, diese Auswahl manuell zu treffen. Eine automatisierte Lösung würde hier viel Zeit sparen.

Leider ist es nicht trivial, automatisch Testdaten so auszuwählen, dass sie beispielsweise jeden Pfad testen. Eine Möglichkeit besteht darin, die symbolische Ausführung von Programmen mit C*onstraint Logic Programming* (CLP) zu kombinieren. Dieser Ansatz wird im Folgenden vorgestellt.

Bei der symbolischen Ausführung von Programmen handelt es sich um eine automatische, statische Analyse des Sourcecodes von Programmen, die von den Programminputs und Unterprogrammaufrufparametern abstrahiert und stattdessen Symbole verwendet. Damit kann die gesamte Semantik des Programms in symbolischen Ausdrücken dargestellt werden. Dies ist trivial für einfache Zuweisungen von Werten an Variablen, wird jedoch komplizierter, sobald Bedingungen den Programmfluss verzweigen lassen. Beispielsweise werden aufgrund einer *if…then…else*-Anweisung zwei Pfade angelegt, die abhängig von der Bedingung durchlaufen werden.

Dies führt bei größeren Programmen zu einer unüberschaubaren Anzahl verschiedener Pfade nebst anderen technischen Problemen. Es hat sich jedoch gezeigt, dass der größte Teil der technischen Probleme mittels *CLP* adressiert werden kann. Die große Anzahl der generierten Pfade kann (und muss) verringert werden, um sinnvoll behandelt werden zu können. Das Problem liegt ursprünglich darin, dass es schwierig ist, die Zulässigkeit von ermittelten Pfaden zu bestimmen. Meist sind die Bedingungen, die bei der symbolischen Ausführung für die Pfadwahl erzeugt werden, jedoch so komplex, dass die darunter liegende Programmiersprache stark vereinfacht werden muss, um die Komplexität in verarbeitbaren Grenzen zu halten.

Der hier vorgestellte Ansatz versucht nun, die üblicherweise exponentiell steigende Anzahl von Zuständen zu begrenzen und gleichzeitig das Subset der Programmiersprache, das verwendet werden kann, zu vergrößern. Dazu werden die traditionell separat gehaltenen Teile eines Testdatengenerators integriert: die symbolische Ausführung, der Pfadselektor und die Zulässigkeitsanalyse.

Algebraisch gesehen reduziert sich die Frage nach der Zulässigkeit eines ermittelten Pfades auf die Frage, ob es eine Belegung der Variablen in der Bedin-

gung gibt, für die diese Bedingung wahr wird. Dabei werden die Wertebereiche der Variablen eingeschränkt auf Mengen zulässiger Werte; ist eine dieser Mengen leer, so kann die Bedingung nicht erfüllt werden, und der Pfad ist unzulässig, weil nicht erreichbar. So kann also die algebraische Bedingung gesehen werden als ein System von Einschränkungen seiner Variablen, und damit kann das Problem der Erreichbarkeit von Pfaden reduziert werden auf eine Suche nach einer Belegung von Variablen in deren Wertebereich, die ein vorgegebenes, endliches System von Einschränkungen erfüllt (CSP, *Constraint Satisfaction Problem*).

Leider führt diese Reduzierung auf ein CSP nicht direkt zu praktischen Vorteilen: CSPs sind üblicherweise NP-vollständig, sodass ein einfacher „generieren-und-überprüfen"-Ansatz, bei dem eine mögliche Lösung generiert und dann gegen das System von Einschränkungen geprüft wird, nicht praktikabel ist. Es gibt allerdings auch effizientere Lösungsverfahren für diese Art von Problemen, eines der bekannteren dürfte die Simplex-Methode sein.

Dennoch würde mittels der vorhandenen Algorithmen der Aufwand sehr hoch bleiben. Erleichterung verschafft hier die Sprache Prolog, die auf Prädikatenlogik erster Ordnung basiert. Statt sich wie in herkömmlichen Programmiersprachen auf das „wie" zu konzentrieren, ist die Frage in Prolog eher „was". Sobald dieses „was" in Prolog beschrieben ist, kann der eingebaute Berechnungsmechanismus die Lösung berechnen.

Verschiedene Nachteile von Prolog erfordern weitere Arbeiten, etwa um den Suchraum für Lösungen einzuschränken oder um momentan unlösbare Gleichungen zu verschieben, sodass sie erst später wieder angefasst werden. Man fasst diese Erweiterungen zusammen unter dem Namen C*onstraint Logic Programming* (CLP). CLP ist ideal, um einen Gleichungslöser zu implementieren, der unerreichbare Pfade, die bei der symbolischen Programmausführung generiert wurden, entdeckt.

Es existiert ein prototypisches Testtool, das dem beschriebenen Ansatz folgt: ATGen (*An Automated Test Data Generator)*. Er wird benutzt, um vollautomatisch Testdaten zu erzeugen, die zu vollständiger Entscheidungsabdeckung beim Testen von SPARK Ada-Programmen führen. SPARK Ada entspricht vom Kern her Ada, allerdings ohne die Konzepte Nebenläufigkeit, dynamische Speicherallokation, Zeiger, Rekursion und Interrupts.

Beim Parsing wird zuerst auf syntaktischer Basis der SPARK Ada-Code in eine Liste von Prolog-Ausdrücken gewandelt. ATGen generiert inter-prozedurale Testdaten, sodass alle Entscheidungen abgedeckt werden. Dazu werden die Prolog-Terme in eine flache Struktur verwandelt. Das bedeutet beispielsweise, dass Subroutinen extrahiert und gesondert aufgeführt werden. Um die Semantik des ursprünglichen Programms beizubehalten, werden Parameter hinzuge-

fügt, die die Variablensichtbarkeit beschreiben, und Subroutinenaufrufe entsprechend modifiziert, um diese Parameter zu berücksichtigen.

Während der symbolischen Ausführung des entstandenen Prolog-Programms werden nun die zu verfolgenden Pfade selektiert. In der initialen Version von ATGen wird nur festgestellt, ob ein Pfad bereits von einem anderen Test abgedeckt wird. Daraus folgt unter anderem, dass Endlosschleifen entstehen können, die abgefangen werden müssen (manuell oder automatisch, *„future work"*). Die Bedingungen von Verzweigungen werden der Menge der Einschränkungen hinzugefügt. Endet ein Pfad, so geht die symbolische Ausführung im Entscheidungsbaum zurück zum letzten Entscheidungspunkt und setzt von dort aus die Ausführung fort (*„backtracking"*). Sind keine Entscheidungspunkte mehr übrig, so endet die symbolische Ausführung.

Die Menge von Einschränkungen wird dann einem Solver zugeführt, der Lösungen sucht und den Testdatenreport erstellt. Der Testdatenreport enthält die generierten Testdaten, den dadurch abgedeckten Pfad, die erwarteten Resultate, und Input- und Outputvariablen. Weiterhin wird die Gesamtabdeckung berechnet.

Erste Experimente mit einer SPARK Ada-Repräsentation von *Insertion Sort* verliefen positiv: ATGen generierte Testdaten, die zu vollständiger Entscheidungsabdeckung führten, in weniger als 10 Sekunden auf einem handelsüblichen PC mit einem 450 MHz Pentium III Prozessor. Da jedoch die Suche nach bisher nicht abgedeckten Entscheidungen zufällig erfolgt, können große Variationen in der Laufzeit auftreten. Weiterhin werden mehr Entscheidungspunkte festgehalten als nötig, was zu unnötigem Backtracking führt und somit die Effizienz unnötig senkt.

ATGen wird erweitert, um C-Code verarbeiten zu können.

Black-Box-Tests

Da bei Black-Box-Tests üblicherweise die funktionale Spezifikation der zu testenden Software zugrunde gelegt wird, muss sich auch die Testdatengenerierung an dieser Spezifikation orientieren. Schon 1993 wurde deshalb das Verfahren der *Classification-Tree Method* (CTM) entwickelt . Das Verfahren erzeugt Testfälle für Black-Box-Testing durch Konstruktion eines Klassifikationsbaumes. Das Verfahren war sehr viel versprechend, allerdings enthielt es keinen Algorithmus zur systematischen Erzeugung des Klassifikationsbaumes. Dieser musste manuell, nur Anhand von vagen Anweisungen, erzeugt werden, was für komplizierte Spezifikationen schwierig und fehleranfällig war. Fehlerhafte Klassifikationsbäume hatten natürlich auch negative Auswirkungen auf die erzeugten Testfälle, sodass eine systematischere Vorgehensweise angezeigt war.

Diese Systematik wurde im Jahre 2000 erstellt und als *Integrated Classification-Tree Method* (ICTM) bekannt [CPT00]. Der vorgestellte Algorithmus erlaubt es, die Klassifikationsbäume systematisch zu erzeugen. 2003 wurde ein prototypisches System zur Erzeugung von Testdaten, basierend auf ICTM, entwickelt [CCG+03]. ICTM selbst besteht aus sieben Schritten:

(1) Die Spezifikation wird in funktionale Einheiten zerlegt, die unabhängig voneinander getestet werden können. Für jede dieser Einheiten werden die nachfolgenden Schritte durchgeführt.

(2) Es werden Klassifikationen und ihre zugehörigen Klassen identifiziert. Unter Klassifikationen versteht man unterschiedliche Kriterien, den Wertebereich der Eingaben einer funktionalen Einheit zu partitionieren. Klassen sind disjunkte Untermengen von Werten für jede Klassifikation. Für jede Klassifikation sollten die assoziierten Klassen den Wertebereich dieser Klassifikation vollständig partitionieren. Hierbei zeigt die Gruppierung bestimmter Werte in einer Klasse den Glauben an die Äquivalenz dieser Werte bzgl. Testfällen an: Man geht davon aus, dass ein Testfall mit einem Wert der Klasse im Grunde genauso gut ist wie ein Testfall mit einem beliebigen anderen Wert der Klasse.

(3) Anschließend wird für jede funktionale Einheit eine Klassifikationshierarchietabelle konstruiert, die die hierarchische Relation jedes Klassifikationspaares festhält.

(4) Aus der Klassifikationshierarchietabelle wird ein Klassifikationsbaum erstellt.

(5) Aus dem Klassifikationsbaum wird eine Kombinationstabelle erstellt. Aus dieser Tabelle können dann aufgrund vordefinierter Auswahlregeln verschiedene Kombinationen von Klassen ausgewählt werden. Jede dieser Kombinationen von Klassen nennt man einen potentiellen Testrahmen.

(6) Alle potentiellen Testrahmen werden gegen ihre jeweilige funktionale Einheit getestet, um zu bestimmen, ob sie vollständig sind oder nicht. Die lässt sich wie folgt überprüfen: Es sei ein potentieller Testrahmen gegeben. Lässt sich ein Input für die funktionale Einheit finden durch selektieren eines Elements aus jeder Klasse des Testrahmens, so ist der Testrahmen vollständig. Falls dies nicht so ist, dann ist der Testrahmen unvollständig und daher nutzlos für den Softwaretest; er sollte verworfen werden.

(7) Aus jedem vollständigen Testrahmen wird ein Testfall konstruiert, indem ein Element aus jeder Klasse selektiert wird.

In [CCG+03] wird ein prototypisches System namens ADDICT (**A**utomate**D** test **D**ata generation using the **I**ntegrated **C**lassification-**T**ree methodology), das die

Schritte (2) bis (5) von ICTM umfasst. Die Dekomposition in funktionale Einheiten (Schritt 1) wird vorerst außer Acht gelassen, d.h. sie muss manuell durchgeführt werden. Die Identifikation von Klassifikationen und ihren zugehörigen Klassen (Schritt 2) geschieht werkzeugunterstützt, jedoch nicht automatisch: Das Werkzeug hilft mittels einer grafischen Oberfläche bei der Erfassung, jedoch nicht bei der Auswahl.

Sobald die Klassifikationen und ihre Klassen in ADDICT eingegeben sind wird die Klassifikationshierarchietabelle konstruiert (Schritt 3). Es gibt vier Typen von Relationen, die vollständig und disjunkt sind, daher ist die Relation zweier Klassifikationen wohldefiniert. ADDICT überprüft bei der Eingabe Einschränkungen von ICTM und informiert den Benutzer über die grafische Oberfläche, wenn diese Einschränkungen verletzt werden. Weiterhin existieren drei Eigenschaften hierarchischer Relationen [CPT00], die ADDICT dazu benutzt, einen gewissen Grad an automatischer Deduktion und Konsistenzprüfung zu erreichen.

Sobald die Klassifikationshierarchietabelle vollständig eingegeben ist, kann ADDICT basierend auf einem in [CPT00] angegebenen Algorithmus den dazu gehörenden Klassifikationsbaum automatisch konstruieren (Schritt 4). Hierzu ist allerdings folgendes zu bemerken: Die Anzahl der später als unvollständig zu klassifizierenden Testrahmen wird während der Konstruktion des Baumes möglichst niedrig gehalten, um die Effektivität (definiert als Anzahl der vollständigen Testrahmen geteilt durch die Gesamtzahl der Testrahmen) möglichst hoch zu halten.

Um nun die Effektivität hoch zu halten, sollten möglichst wenige unvollständige Testrahmen erzeugt werden. Chen et al. [CPT00] führen aus, dass ein Hauptgrund für niedrige Effektivität Duplikate in Teilbäumen des erzeugten Klassifikationsbaumes sind. ADDICT überprüft während der Konstruktion des Baumes, ob Duplikate entstehen, und wendet gegebenenfalls eine ebenfalls in [CPT00] vorgestellte Restrukturierungstechnik auf den Baum an, die doppelte Teilbäume eliminiert, dabei aber vollständige Testrahmen erhält. Die gesamte Konstruktion des Klassifikationsbaumes läuft vollautomatisch ab.

Die Erzeugung der Kombinationstabelle und die Auswahl potentieller Testrahmen (Schritt 5) verläuft ebenfalls vollautomatisch anhand einiger in [CPT00] vorgegebener Regeln.

Die Schritte (6) und (7) müssen wieder manuell durchgeführt werden: Der Tester überprüft zunächst die von ADDICT erzeugten potentiellen Testrahmen auf Vollständigkeit (Schritt 6) und wählt dann für jeden Testrahmen ein Element aus jeder enthaltenen Klasse für einen Testfall aus.

Die Schritte (1), (6) und (7) sind nicht in ADDICT berücksichtigt. Für die Schritte (1) und (6) ist dies auch zunächst nicht geplant, da sie nur schwer zu automati-

sieren sind. Schritt (7) kann jedoch automatisiert werden, indem für jeden Testrahmen zufällig ein Element jeder Klasse selektiert wird, um daraus einen Testfall zu generieren. Weiterhin können aus jedem Testrahmen natürlich mehrere Testfälle generiert werden, um die Gesamtzahl von Testfällen zu erhöhen.

4.4 Testdurchführung

Unterstützung bei der Testausführung ist mit am leichtesten, da keine aufwendigen Analysen notwendig sind. Die einfachste Automatisierung wären beispielsweise manuell erstellte Skripte, die Testfälle mit den entsprechenden Testdaten aufrufen, sodass die Tests nur durch den Aufruf des Skripts angestoßen werden müssen und dann automatisch ablaufen. Weitergehende Automatisierung kann die automatische Skripterstellung und –wartung umfassen sowie konditionale Skripte, die abhängig von verschiedenen Bedingungen verschiedene Teile der Testsuite abfahren. Aspekte wie Konfigurationsmanagement der Testsuiten, Planung der Ausführung (beispielsweise automatisch jede Nacht) oder Fortschrittsberichte erleichtern die Testarbeit weiter.

All diese Punkte werden (in unterschiedlichem Maße) von etlichen kommerziell erhältlichen Werkzeugen abgedeckt. Die Herausforderung besteht darin, die Lösungen zu integrieren, um ein hochwertiges Testmanagement zu ermöglichen und stupide, wiederkehrende Arbeiten zu reduzieren. Oft sind auch die Testumgebungen historisch gewachsen und mittels einer Vielzahl von selbst geschriebenen Skripten „integriert“, sodass Wartung und Veränderung aufwendig sind. Diese Schwierigkeiten liegen jedoch außerhalb des Fokus' dieses Reports, sodass wir an dieser Stelle nicht weiter darauf eingehen.

4.5 Testauswertung

Nachdem die Tests durchgeführt sind, müssen sie ausgewertet werden. Das bedeutet, dass die tatsächlichen Ergebnisse des Testdurchlaufs mit den erwarteten Ergebnissen (Orakel) verglichen werden müssen. Abweichungen müssen aufgezeichnet und dem Tester in verständlicher Form dargeboten werden. Weiterhin kann eine Historie der Testdurchläufe hilfreich sein, um beispielsweise große Abweichungen vom Projektplan vorhersehen zu können.

Die meisten der genannten Punkte werden in den gängigen kommerziell erhältlichen Testwerkzeugen adressiert. In sämtlichen für diesen Report begutachteten Verfahren wurden die erwarteten Testergebnisse (Orakel) gleich mit in die Testfälle eingebaut. Das macht auch mehr Sinn als sie einzeln zu verwahren und später zuzuordnen, oder nur die Testergebnisse zu protokollieren und manuell auf Korrektheit zu prüfen. Die Anzeige der Testergebnisse nutzt dabei die gesamte Bandbreite der Darstellungsformen: Grafisch, in Farbe, in Tabellenform, …

Es gibt daher kaum Forschung, die sich ausschließlich mit der Auswertung von Tests beschäftigt. Dadurch, dass das Testorakel direkt mit dem Testfall gespeichert wird, beschränkt sich die Auswertung auch auf einen simplen Vergleich des Orakels mit den tatsächlichen Ergebnissen und einer übersichtlichen Anzeige des Vergleichsergebnisses. Bei Letzterem spielen Aspekte der Benutzerfreundlichkeit eine große Rolle, dies liegt jedoch außerhalb des Fokus' dieses Reports.

4.6 Wiederverwendung von Testfällen

Testfälle zu erstellen ist eine langwierige und teure Arbeit. Abhängig von der Art der durchzuführenden Tests sind Kenntnisse über Ein- und Ausgaben oder die interne Struktur der zu testenden Software nötig, um hochwertige Testfälle zu spezifizieren.

Darüber hinaus müssen aus der Spezifikation ausführbare Testfälle erstellt werden. Diese sind üblicherweise wieder in einer Programmier- oder Skriptsprache verfasst, und stellen damit selbst Software dar, für die, wenn auch in kleinerem Umfang, ein kompletter Entwicklungszyklus notwendig ist, inklusive Design, Implementierung, Debugging und Testen.

Um eine Automatisierung des Testvorgangs zu erreichen, müssen die Testfälle so in Programme umgesetzt werden, dass sie die zu testende Software mit passenden Daten aufrufen und die Ausgabe mit dem erwarteten Ergebnis vergleichen. Offensichtlich sind diese Programme anfällig für Änderungen in der zu testenden Software. Der Wartungsaufwand für die Testfälle kann auf diese Weise erheblich werden, speziell bei Änderungen spät im Entwicklungszyklus.

Ein weiteres Problem stellen Portierungen auf andere Plattformen dar. Viele Testwerkzeuge unterstützen nur eine Plattform, die wenigsten sind crossplattformtauglich oder für mehrere Plattformen verfügbar. Bei selbst geschriebenen Testfällen reicht es auch meist nicht aus, den Code auf der neuen Plattform zu übersetzen, weil fast immer plattformspezifische Features verwendet werden, sowohl im Testcode als auch im zu testenden Programm.

Daher bietet es sich an, die Testfälle in einer plattformunabhängigen Sprache zu spezifizieren und dies dann in einem zweiten Schritt in ausführbaren Code zu übersetzen. Sinnvollerweise sollte dies automatisch erfolgen, sodass nur die Testfälle noch von Hand erstellt werden müssen. Dazu muss ein Parser die Testbeschreibungen lesen und dann plattform- und programmiersprachenspezifischen Code generieren. Dieses Vorgehen erleichtert die Portierung von Anwendungen erheblich, da nur der Generator neu programmiert werden muss, nicht mehr alle Testfälle. Weiterhin lassen sich in einer dedizierten Test-Sprache Konzepte wie Eingabedaten und Testorakel leichter ausdrücken als in normalen Programmiersprachen.

TestTalk [Liu00] ist eine solche Test-Sprache. Die Testfälle werden in einer XML-artigen Syntax spezifiziert, die mittels Transformationsregeln in echten Code umgesetzt werden können. Dieser Ansatz ist sehr flexibel: Testfälle für Webanwendungen können genauso spezifiziert werden wie Tests von sicherheitskritischen Systemen auf bekannte und unbekannte Schwachstellen oder herkömmliche Softwaretests. Bindeglied ist in allen Fällen der Parser, der den eigentlichen Testcode erzeugt. Die steuernden XML-Testfälle und der Parser sind erheblich einfacher zu warten als fest in einer Programmiersprache codierte Tests.

Ein anderer Ansatz [San01] kombiniert die Spezifikation der Testfälle in Tabellenform mit automatischen Abdeckungsanalysen und der Generierung von Code für Treiber und Stubs. Die Abdeckung wird ermittelt mittels automatisch in den Code eingefügter Hilfsanweisungen, anhand derer das Werkzeug die ausgeführten Pfade bestimmt. Sämtliche Skripte in dem Toolset sind für Ada-Code entwickelt worden, mit dem Fokus auf Zeitersparnis beim Testen und der Dokumentation der Testabdeckung. Durch die Testfallspezifikation in Tabellenform können auch in Ada weniger erfahrene Tester Testfälle erstellen. Außerdem werden die erwarteten Testresultate gleich mit angegeben, sodass ein spezielles Analysewerkzeug entfällt. Um die Testfallerstellung zu erleichtern, analysiert ein weiteres Werkzeug den zu testenden Code auf Ein- und Ausgabevariablen sowie referenzierte globale Variablen und Subprogramme.

4.7 Optimierung von Testsuiten für Regressionstests

Ein großes Problem beim Testen von Software ist das Wachstum der Testsuiten. Wird eine Software über mehrere Generationen entwickelt und gewartet, so wächst die Menge der durchzuführenden Tests mit an. Dies allein führt zu längerer Laufzeit der Tests und somit zu Verzögerungen. Oft entstehen auch mehrere Varianten der Software, immer zumindest mehrere Releases, die separat gewartet und getestet werden müssen. Hier spielt das Konfigurationsmanagement der Testsuiten eine große Rolle.

Um Testsuiten nicht ausufern zu lassen, muss versucht werden, sie immer wieder zu reduzieren. Im Laufe des Software-Lebenszyklus' wird immer wieder Funktionalität hinzugefügt oder geändert, und dafür neue Tests geschrieben. Dabei werden möglicherweise Programmteile mehrfach getestet, sodass manche Testfälle eigentlich überflüssig wären. Für die Reduzierung der Testsuiten untersucht man daher, wie diese mehrfach getesteten Programmteile identifiziert und die überflüssigen Testfälle aus den Testsuiten entfernt werden können, ohne die Fähigkeit der Testsuite, Fehler zu entdecken, signifikant zu reduzieren.

Es ist natürlich wirtschaftlich von erheblicher Bedeutung, wie sich das Kosten-Nutzen-Verhältnis der Testsuite dabei entwickelt: Die Kosten für die Verkleine-

rung der Testsuite und die (zusätzlichen) Kosten, die aufgrund nicht mehr gefundener Fehler entstehen, sollten kleiner sein als der Nutzen, der aus der Reduzierung der Größe der Testsuite gezogen wird. Zu diesem Thema existieren verschiedene Experimente und Studien, die jedoch leider zu keinem eindeutigen Schluss kommen.

[WHL+95], [WHL+98] und [WHM+97] schließen aus ihren Studien, dass Testsuiten im Umfang reduziert werden können, ohne signifikant weniger effektiv Fehler zu finden. In [WHL+98] untersuchen die Autoren anhand von zehn Unix-Hilfsprogrammen (in C programmiert), wie effektiv verkleinerte Testsuiten Fehler finden. Neun Programme waren zwischen 90 und 289 Zeilen lang, eines 842. Es wurden zufällig Testfälle erzeugt, die wiederum zufällig zu Testsuiten hinzugefügt wurden. Damit wurden Abdeckungen zwischen 50 und 95% erreicht. Die Testprogramme wurden ähnlich Mutationen mit Fehlern versehen und dann mit den Testsuiten getestet.

Danach wurden sowohl Testsuiten als auch die gesamten Pools der Testfälle mit dem Werkzeug ATAC [HL92] unter dem Gesichtspunkt der Erhaltung der Abdeckung reduziert. Unter anderem zogen die Autoren folgende Schlüsse:

- Mit zunehmender Abdeckung der ursprünglichen Testsuiten steigen auch die Einsparungen bei deren Reduktion.

- Mit zunehmender Abdeckung der ursprünglichen Testsuiten nehmen auch die Verluste in der Effektivität bei der Fehlerfindung zu, wenn sie reduziert werden. Dies geschieht allerdings auf niedrigem Niveau: die meisten Verluste lagen unter 5%.

- Die Schwierigkeit, die Fehler zu finden (definiert durch die Anzahl der Testfälle, die einen Fehler finden), hat Einfluss auf die Effektivität der Fehlerfindung.

- Die mittels ATAC reduzierten Testsuiten haben einen Kosten-Nutzen-Vorteil gegenüber zufällig reduzierten.

Die Autoren schließen daraus, dass die Effektivität der Fehlerfindung kaum beeinträchtigt wird, solange bei der Reduktion der Testsuiten deren Abdeckung erhalten bleibt.

Eine zweite Studie [WHM+97] untersuchte ein einziges Programm aus dem Bereich der Steuerung großer Antennenverbünde mit 9564 Zeilen Quellcode. Testsuiten wurden auf zwei Arten erzeugt: bis eine vorgegebene Größe (Anzahl an Testfällen) erreicht war, oder bis eine vorgegebene Abdeckung erreicht war. Die eingesetzten Fehler waren im Laufe der Entwicklung tatsächlich aufgetrete-

ne (und korrigierte) Fehler. Die Testsuiten wurden wieder mittels ATAC [HL92] reduziert. Unter anderem zogen die Autoren folgende Schlüsse:

- Die Testsuiten ließen sich deutlich verkleinern.

- Da die Verluste bei der Effektivität der Fehlerfindung kleiner waren als die Reduktion der Testsuitegröße war das Kosten-Nutzen-Verhältnis ebenfalls positiv.

- Auch hier haben die mittels ATAC reduzierten Testsuiten einen Vorteil gegenüber zufällig reduzierten.

Die beiden genannten Studien sehen also insgesamt Vorteile in der Reduktion von Testsuiten. Andere Studien kommen jedoch zu gegenteiligen Ergebnissen. In [RHR+02] untersuchen die Autoren kleinere Programme ähnlich denen in [WHL+95]. Ihre Ergebnisse waren:

- In Bezug auf die Größenreduzierungen stimmen sie [WHL+98] und [WHM+97] zu: Es gibt signifikante Einsparungen, die mit steigender Größe der Testsuiten zunehmen. Dabei lassen sich die Testsuiten jeweils auf eine annähernd konstante Größe reduzieren.

- Sie kommen jedoch bei der Effektivität der Fehlerfindung zu anderen Ergebnissen. In ihrer Studie kam es durchaus zu erheblichen Beeinträchtigungen bei abdeckungsäquivalenten Testsuiten, die sogar mit zunehmender Größe der Testsuiten noch weiter steigen.

- Die zufällig reduzierten Testsuiten schnitten dabei schlechter ab als die abdeckungsäquivalenten.

Eine andere Studie [RHR+02] untersucht erneut das bereits in [WHM+97] untersuchte große Programm. Auch hier wurden die Testsuiten einmal zufällig reduziert und einmal abdeckungsäquivalent. Die Studie ergab im Einzelnen:

- Wie bei der ersten Studie ließen sich die Testsuiten zu einer annähernd konstanten Größe reduzieren.

- Wie bei der ersten Studie ergaben sich signifikante Verluste bei der Effektivität der Fehlerfindung

- Die zufällig reduzierten Testsuiten schnitten auch hier durchweg schlechter ab als die abdeckungsäquivalenten.

Ausgehend von den aufgeführten Studien ist es somit nicht möglich, vorherzusagen, ob die Reduzierung von Testsuiten ein positives oder ein negatives Kosten-Nutzen-Verhältnis hat. Auch wenn die Studien nicht direkt vergleichbar

sind (anderer Reduktionsalgorithmus, andere Testprogramme, andere Testsuiten), so zeigen sie auf, dass die Reduktion von Testsuiten zu signifikanten Beeinträchtigungen bei der Effektivität der Fehlerfindung führen kann – *oder auch nicht.* Der Mechanismus aus Programmcharakteristika, Fehlern und Testfällen ist noch zu wenig verstanden, um zweifelsfreie Aussagen machen zu können.

4.8 Teststrategien für objektorientierte Software

Objektorientierte Software bietet viele Vorteile, die man in der modernen Softwareentwicklung nicht mehr missen möchte (Kapselung, definierte Zugriffsmöglichkeiten, Information hiding,…). Diese Vorteile der Objektorientierung machen die Entwicklung großer Systeme einfacher oder ermöglichen sie überhaupt.

Die Objektorientierung führt zu der paradoxen Situation, dass objektorientierte gleichzeitig leichter und schwerer zu testen sind. Leichter, da die gekapselten Objekte als Einheiten besser zu testen sind als eine Vielzahl miteinander verwobener Funktionen mit gemeinsamen Variablen. Mittels Objektorientierung wird schon während der Implementierung der Grundstein für leichteres Testen gelegt, indem beispielsweise nur genau definierter Zugriff auf Variablen möglich ist. Dies schließt schon viele Fehlerquellen aus, die andernfalls mühsam abgetestet werden müssten.

Leider handelt man sich mit einer objektorientierten Softwarestruktur nicht nur Vorteile ein. Die Objektstruktur führt aber auch dazu, dass Testen schwerer wird: Das Verhalten der Objekte hängt entscheidend von dem Zustand ab, in dem sie sich zum Zeitpunkt des entsprechenden Methodenaufrufs befinden. Daher muss beim Softwaretest sichergestellt werden, dass Methoden mehrfach getestet werden, in Abhängigkeit vom Zustand des Objektes. Dieser Zustand wird bestimmt von der Anzahl und Reihenfolge der Nachrichten, die das Objekt bis zum Zeitpunkt des Tests erhalten hat. Testwerkzeuge, die Objekte testen, müssen daher gezielt Nachrichten erzeugen können, um Objekte in einen definierten Zustand zu bringen. Eine Automatisierung im Bereich objektorientierter Programme muss also die möglichen Zustände des zu testenden Objekts unterstützen und auch hervorrufen, um Fehler zu finden, die zustandsabhängig sind.

4.8.1 Codebasiertes Vorgehen

Ein codebasierter Ansatz ist in [BOP00] beschrieben. Dazu werden basierend auf Analysen des Quellcodes Sequenzen von Methodenaufrufen generiert, die das zu testende Objekt in verschiedene Zustände bringt, in denen dann der eigentliche Aufruf der zu testenden Methode erfolgt.

Die zugrunde liegende Idee ist die folgende: Die Ausführung der Methode ist von ihren Eingabeparametern und den verwendeten Instanzvariablen abhängig.

Die Eingabeparameter werden durch den Aufruf bestimmt, also bleiben die Instanzvariablen übrig. Diese werden wieder durch Methoden bestimmt und verändert. Um nun zustandsabhängige Fehler zu finden, identifiziert der Ansatz Methodenpaare, die eine Instanzvariable zuerst definieren und dann benutzen. Ist ein solches Paar gefunden, wird versucht, eine Sequenz von Methodenaufrufen zu finden, die die beiden Methoden des identifizierten Paares in der richtigen Reihenfolge aufruft. Jede identifizierte Sequenz bildet einen Testfall des Objektes, da sie das Objekt in unterschiedliche Zustände bringt.

Die dazu verwendete Technik ermittelt zuerst mittels Datenflussanalyse so genannte du-Paare von Anweisungen, wobei die erste Anweisung eine Instanzvariable definiert und die zweite sie benutzt. Mittels symbolischer Ausführung werden dann Bedingungen bzgl. Pfadausführungen und Variablendefinitionen gesammelt, um schließlich mittels automatischer Deduktion Sequenzen von Methodenaufrufen zu finden, die die im ersten Schritt gefundenen du-Paare beinhalten. Somit führt, ausgehend von einem initialen Zustand, jede Sequenz zur Definition einer Instanzvariablen und ihrer darauf folgenden Benutzung.

Können alle drei Phasen (Datenflussanalyse, symbolische Ausführung, automatische Deduktion) erfolgreich durchgeführt werden, so werden alle zulässigen und unzulässigen du-Paare identifiziert. Sollte die symbolische Ausführung für manche Methoden scheitern, so wird immer noch eine Menge von du-Paaren für das zu testende Objekt erzeugt, wenn auch nicht für alle Methoden. Die fehlenden können beispielsweise von Hand nachgerüstet werden.

Sollte die symbolische Ausführung erfolgreich sein, jedoch die automatische Deduktion für ein du-Paar scheitern, so können die Aufrufsequenzen unter Zuhilfenahme von Vor- und Nachbedingungen der jeweiligen Methoden von Hand erstellt werden. Die Datenflussanalyse funktioniert meist nur für einfache Variablen. Komplexe Instanzvariablen (arrays, structures, rekursive Typen) können entweder bei der Analyse übersprungen werden, oder die Tester können von Hand Anweisungen identifizieren, die diese Variablen definieren und benutzen.

Die aufgeführten Probleme können lediglich dazu führen, dass weniger Anweisungsfolgen automatisch generiert werden, nicht jedoch zum Totalversagen der gesamten Methode. Ein Toolset zur Unterstützung der Methode befindet sich in der Entwicklung; erste Anwendungen der Methode haben die Erwartungen bestätigt, dass zustandsabhängige Fehler mittels automatisch generierter Anweisungsfolgen entdeckt werden können.

4.8.2 Spezifikationsbasiertes Vorgehen

Eine ähnliche Vorgehensweise ist in beschrieben. Hier wird allerdings nicht der Quellcode analysiert, sondern Testfälle und –treiber werden anhand von alge-

braischen Spezifikationen der getesteten abstrakten Datentypen erstellt. Eine solche algebraische Spezifikation besteht aus einem syntaktischen und einem semantischen Teil. Im syntaktischen Teil werden die Methoden und ihre Signaturen beschrieben (sog. Funktionen), der semantische Teil besteht aus einer Menge von Axiomen, die die Beziehungen der Funktionen untereinander beschreiben. Ein wichtiger Begriff in diesem Zusammenhang ist der Begriff der *Äquivalenz:* Zwei Sequenzen von Funktionen (Methodenaufrufen) heißen *äquivalent*, wenn die eine Sequenz mittels Anwendung der Axiome als Transformationsregeln in die andere Sequenz überführt werden kann.

LOBAS [Doo93] ist eine Sprache, um algebraische Spezifikationen auszudrücken, die sich am objektorientierten Paradigma orientiert. Eine Klasse C ist eine korrekte Implementierung eines (beispielsweise in LOBAS spezifizierten) abstrakten Datentyps T, wenn die abstrakten Zustände von C und T eine 1:1-Beziehung haben. ASTOOT ist ein Set von Werkzeugen, welches die algebraische Spezifikation und die Implementierung einer Komponente als Input verarbeitet und Testfälle und –daten erzeugt, die Testfälle ausführt und die Resultate überprüft. Eine Übersicht ist in Abbildung 3 gegeben. Sollte keine algebraische Spezifikation der Komponente verfügbar sein, so können die generierten Testtreiber immer noch ausgeführt werden anhand von manuell erstellten Testfällen.

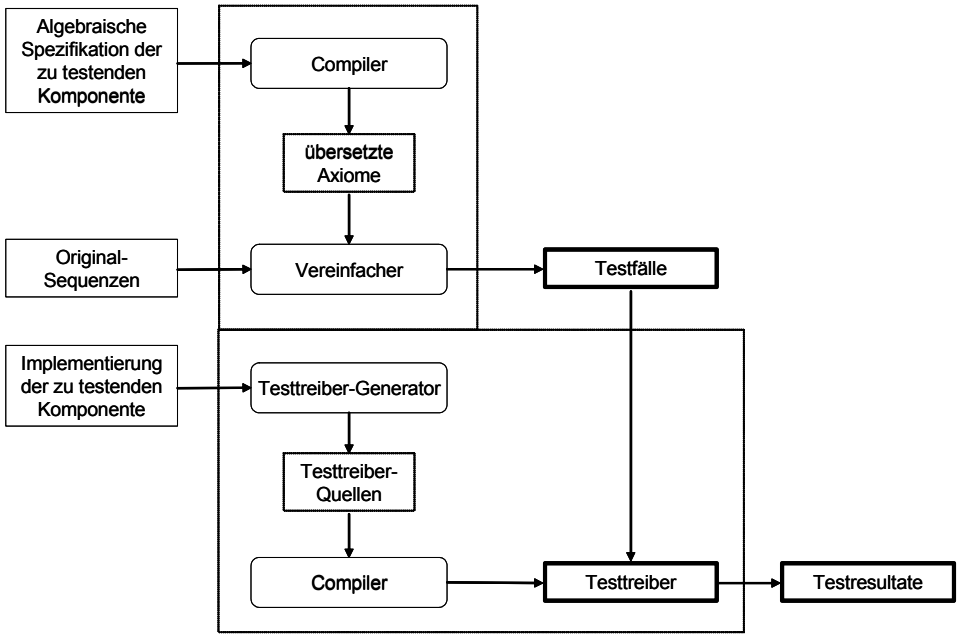

Abbildung 3: Die Komponenten von ASTOOT

Compiler und Vereinfacher verwandeln die algebraische Spezifikation und die (vom Benutzer bereitzustellenden) Original-Sequenzen über den Zwischenschritt der Baumdarstellung in Testfälle. Der erstellte Baum kann jedoch sehr groß werden (exponentielles Wachstum), sodass manuelle Eingriffe in den Erstellungsprozess nötig werden können. Weiterhin wird meist eine große Anzahl von Testfällen erstellt, die dann noch manuell gefiltert werden müssen. Der Testtreiber-Generator erzeugt anhand der Signatur der zu testenden Komponente Quellcode für Testtreiber, welcher dann von einem Compiler der jeweiligen Sprache in ausführbare Testtreiber übersetzt wird. Mit den Testtreibern werden die vorher erstellten Testfälle ausgeführt.

Es bleiben einige Fragen offen: Wie sollten die Original-Sequenzen beschaffen sein, um möglichst „gute“ Testfälle zu erzeugen? Und welche Pfade durch den Baum sollten verfolgt werden? Erste Fallstudien haben ergeben, dass längere Sequenzen bessere Ergebnisse erzielen als kürzere, falls der Wertebereich der Parameter ausreichend groß ist. Außerdem sollten die Parameter und die Pfade durch den Baum möglichst weit verteilt sein. Weiterhin sollten, falls in den Axiomen Vergleichsoperatoren vorkommen, viele verschiedene Testfälle mit variierenden Parametern benutzt werden.

Prototypische Implementierungen für ASTOOT sind für die Programmiersprache Eiffel verfügbar, die Konzepte sind jedoch grundsätzlich auch auf andere Sprachen übertragbar.

4.9 Zusammenfassung

Mit Ausnahme der Testdurchführung und –auswertung ist in allen Bereichen des Software-Testens ein deutlicher Abstand zwischen Forschung und Praxis festzustellen. Im Bereich der vollautomatischen Testfallgenerierung sind einzelne Ansätze anzutreffen, die nach bestimmten Vorarbeiten aus formalen Anforderungen in der Tat komplett automatisiert Testfälle erzeugen können. Verbreiteter sind jedoch Ansätze, die auf halbformalen Anforderungen arbeiten. Einen großen Stellenwert nimmt hier UML als sehr verbreitete Form der Anforderungsspezifikation ein. Daher konzentrieren sich auch viele Forschungsarbeiten auf diese Sprache.

Bei der Testdatengewinnung hängt die Vorgehensweise stark davon ab, wie gut man die Implementierung der zu testenden Komponente kennt. Ohne genaue Kenntnisse der internen Abläufe können nur sehr begrenzt sinnvoll Testdaten ausgewählt werden, sodass hier zufällige Auswahlen dominieren. Bei genauerer Kenntnis der Programminterna ist eine gezielte Auswahl für White-Box-Tests möglich, allerdings treten dabei einige nicht-triviale logische Probleme auf, die die praktische Anwendbarkeit wieder einschränken. Für spezifikationsbasierte Tests (Black-Box) existiert eine systematische Methodik, die sogar schon teilweise automatisiert ist.

Die Wiederverwendung von Testfällen spielt in der Industrie eine zunehmend größere Rolle, und ist daher auch forschungsrelevant. Hier kristallisiert sich der Weg heraus, die Testfälle in einer plattformunabhängigen Sprache zu spezifizieren und dann nur nach Bedarf tatsächlich ausführbaren Code zu erstellen.

Bei der Wiederverwendung von Testsuiten ist deren Optimierung für Regressionstests ein weiteres Thema, allerdings sind die Forschungsergebnisse in diesem Bereich noch nicht eindeutig. Manche Studien finden Kosten-Nutzen-Vorteile bei der Reduktion von Testsuiten, andere das Gegenteil, sodass keine klare Aussage möglich ist. Dabei wird klar, dass etliche Faktoren noch zu wenig untersucht sind, und weitere Arbeit nötig ist.

Schließlich werden spezielle Teststrategien für objektorientierte Software entwickelt, da klassische Vorgehensweisen bei Objekten viele Fehler gar nicht finden können. Auch hier gibt es verschiedene Vorgehensweisen: Ist der Quellcode verfügbar, so kann dieser für Analysen genutzt werden, um Testfälle zu erstellen. Eine andere Vorgehensweise nutzt eine algebraische Spezifikation der zu testenden Komponenten.

4.10 Einschätzung der Ansätze

Viele Forschungsansätze im Bereich Testen und Testautomatisierung weisen in interessante Richtungen. Eine vollautomatische Testfallgenerierung beispielsweise ist jedoch in absehbarere Zeit für mittlere bis größere Systeme nicht zu erwarten. Generell könnte man sagen, dass der Grad der Automatisierung proportional zum Grad der Formalität ist. Hat man formal spezifizierte Anforderungen, so ist es durchaus möglich, darauf aufbauend Testfälle generieren zu lassen. Für die weitaus meisten Systeme ist es jedoch praktisch nicht möglich, sie formal zu spezifizieren. Dies kann unterschiedliche Gründe haben (zu teuer/nicht ausreichendes Know-How der Entwickler/Kunden, zu komplexe Systeme,…), führt jedoch immer dazu, dass Abstriche an der Automatisierbarkeit in Kauf genommen werden müssen.

Ähnliches gilt für die Testdatengewinnung, da bei gezielter Auswahl auch logische Probleme auftreten, die praktisch nicht mit vertretbarem Aufwand zu lösen sind. Eine Teilautomatisierung ist hier immerhin bei Black-Box-Testen möglich.

Oft werden Algorithmen eingesetzt, die die auftretenden Probleme nicht optimal lösen, sondern nur annähernde Lösungen anbieten, dies dann aber deutlich schneller. Jedoch sind auch zufallsbasierte Verfahren nicht von Anfang an auszuschließen.

Der Ansatz, für die Wiederverwertung von Testfällen auf eine plattformunabhängige Sprache zur Testfallspezifikation zu setzen, ist eine ziemlich geradlinige

und einfache Lösung, die das Problem effektiv angeht. Im Zweifelsfall muss nur der Codegenerator neu geschrieben werden, nicht jedoch alle Testfälle.

Die Objektorientierung fügt dem Thema „Software-Testen“ noch einmal eine gehörige Portion Komplexität hinzu. Auch hier treten wieder logische Probleme auf, die praktisch nicht ganz einfach zu lösen sind, und die Anforderungen an die Anforderungen steigen ganz erheblich. Dennoch können mehr Fehler gefunden werden als mit klassischen Testverfahren, sodass die praktische Anwendung wieder auf eine Kosten-Nutzen-Rechnung herausläuft.

5 Referenzen

[AO99] A. Abdurazik, J. Offutt: *Generating Test Cases from UML Specifications*. Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), Fort Collins, Colorado, USA, 1999.

[BOP00] U. Buy, A. Orso, M. Pezzè: *Automated Testing of Classes*. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Portland, Oregon, USA, 2000.

[Bei95] B. Beizer: *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons 1995.

[Ber04] Bernd Leitenberger: *Die Ariane 5*. URL: http://www.bernd-leitenberger.de/text/ariane5.html, last visited: 2004-05-13.

[CCG+03] A. Cain, T. Y. Chen, D. Grant, P. Poon, S. Tang, T. Tse: *An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology*. Proceedings of the First International Conference on Software Engineering Research and Applications (SERA), San Francisco, California, USA, 2003.

[CPT00] T. Y. Chen, P. Poon, T. Tse: An Integrated Classification-Tree Methodology for Test Case Generation, In: *International Journal of Software Engineering and Knowledge Engineering* 10 (6), S.647-679, 2000.

[Doo93] R. Doong: *An Approach to Testing Object-Oriented Programs* 1993.

[ESS+98] S. H. Edwards, G. Shakir, M. Sitaraman, B. W. Weide, J. Hollingsworth: *A Framework for Detecting Interface Violations in Component-Based Software*. Proceedings of the 5th International Conference on Software Reuse, Victoria, British Columbia, Canada, 1998.

[Edw00] S. H. Edwards: Black-Box Testing Using Flowgraphs: An Experimental Assessment of Effectiveness and Automation Potential, In: *Software Testing, Verification and Validation* 10 (4), S.249-262, 2000.

[HIM00] J. Hartmann, C. Imoberdorf, M. Meisinger: *UML-Based Integration Testing*. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Portland, Oregon, USA, 2000.

[HL92] J. R. Horgan, S. A. London: *A Data Flow Coverage Testing Tool for C*. Proceedings of the Symposium on Assessment of Quality Software Development Tools, New Orleans, Louisiana, USA, 1992.

[Liu00] C. Liu: *Platform-independent and Tool-neutral Test Descriptions for Automated Software Testing*. 22nd International Conference on Software Engineering, 2000.

[RHR+02] G. Rothermel, M. J. Harrold, J. v. Ronne, C. Hong: Empirical Studies of Test-Suite Reduction, In: *Journal of Software Testing, Verification, and Reliability* 12 (4), S.219-249, 2002.

[San01] U. Santhanam: *Automating Software Module Testing for FAA Certification*. 2001 Annual ACM SIGAda International Conference on Ada, Bloomington, Minnesota, USA, 2001.

[WHL+95] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur: *Effect of Test Set Minimization on Fault Detection Effectiveness*. Proceedings of the 17th International Conference on Software Engineering (ICSE), Seattle, Washington, USA, 1995.

[WHL+98] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur: Effect of Test Set Minimization on Fault Detection Effectiveness, In: *Software - Practice and Experience* 28 (4), S.347-369, 1998.

[WHM+97] W. E. Wong, J. R. Horgan, A. P. Mathur, A. Pasquini: *Test Case Minimization and Fault Detection Effectiveness: A Case Study in a Space Application*. Proceedings of the 21st Annual International Computer Software & Applications Conference (COMPSAC), Washington, DC, USA, 1997.

Dokumenten Information

Titel:	Stand der Forschung von Software-Tests und deren Automatisierung
Datum:	18 Juni, 2004
Report:	IESE-068.04/D
Status:	Final
Klassifikation:	Öffentlich