Seminar
Testen von Software
Sommersemester 2001

Betreuer: Dr.-Ing. Matthias Müller, Dr.-Ing. Frank Padberg

Inhaltsverzeichnis:

Automatische Generierung von Testdaten 1
Rolf Kampffmeyer

Program Slicing 10
Felix Hupfeld

Software Zuverlässigkeit 22
Andreas Hoffmann

Testen objektorientierter Software 35
Ulf Krum

Automatische Generierung von Testdaten

Rolf Kampffmeyer – s_kampfm@ira.uka.de

1 Ziel

Testdaten dienen dazu, vorhandene Fehler im zu testenden Programm aufzudecken. Unter der Voraussetzung, daß das Testverfahren gut ist, deuten eine kleine Zahl an entdeckten Fehlern darauf hin, daß tatsächlich wenige vorhanden sind, und die Qualität des Programmes hoch ist.

Natürlich dient das Testen nicht zur Einschätzung der Programmqualität, sondern auch dazu, die Fehleranzahl zu reduzieren.

Testdaten finden ist von Hand mühsam, und soll deswegen möglichst automatisiert werden.

2 Ansätze

Grundsätzlich unterscheidet man zwischen funktionalen Testverfahren (Blackboxing) und strukturellen Verfahren (Whiteboxing).

Beim Funktionalen Ansatz betrachtet man ein Softwaremodul nur von Außen. D.h. man kennt die Spezifikation: Den Eingabebereich und erwartete Ergebnisse / Nebenwirkungen.

Ob die Ausgabe und Nebenwirkung des Moduls korrekt ist, läßt sich mit Hilfe von Vor- und Nachbedingungen, oder mit einem Orakel feststellen. Ein Orakel wird grundsätzlich auch bei strukturellen Testverfahren benötigt, und bekommt daher einen eigenen Abschnitt <3>.

Ein Problem beim Funktionalen Ansatz sind Bibliotheksfunktionen, auf denen das zu testende Programm basiert, sie lassen sich nur schwer in eine Spezifikation einbinden.

Der eigentliche Anwendungsbereich für automatisches Generieren von Testdaten sind Strukturelle Testansätze. Dort wird die innere Struktur eines Programms analysiert, um fehlerhafte Stellen zu entdecken. Strukturelle Ansätze lassen sich zum einen in zielorientierte und in pfadorientierte unterteilen, und zum andern in statische Ansätze und dynamische. Alle arbeiten mit dem sogenannten Kontrollflussgraph.

Ein Kontrollflussgraph eines Programms enthält als Knoten sogenannte „Basisblöcke“ bzw. einzelne Anweisungen, und als Kanten dazwischen die möglichen

Abarbeitungsreihenfolgen. Gibt es von Knoten A nach Knoten B eine Kante, so heißt das, daß bei einem konkreten Programmablauf zuerst A und dann B ausgeführt werden kann (Es ist auch möglich, daß nach A ein C ausgeführt wird anstatt B, sofern es auch von A nach C eine Kante gibt). Ein Basisblock ist eine Folge von Anweisungen, die immer als ganzes oder gar nicht ausgeführt wird, und daher wie eine einzelne Anweisung behandelt werden kann.

2.1 Der Kombinatorische Ansatz (funktional)

Eine naive Möglichkeit, Eingaben zu erhalten wäre die Zufällige Wahl innerhalb des erlaubten Eingabebereichs. Zusätzlich sollte man Randwerte des Eingabebereichs in allen Kombinationen testen. Eine geschicktere Wahl der Eingabedaten wird durch den kombinatorischen Ansatz erreicht [CDPP96] und [CDFP97]. Dabei werden je zwei Variablen betrachtet, und alle Kombinationen aus ihren erlaubten Bereichen als Eingabe herangezogen. Wenn es zwei Variablen sind, mit jeweils 1000 verschiedenen erlaubten Werten, so erhält man 1000 * 1000 Eingaben. Der Wert aller übrigen Variablen ist beliebig. Um einen Testsatz für das Kriterium „2-fach kombinatorisch" zu erhalten generiert man nun für alle Paare aus Eingabevariablen diese Testfälle. Statt zwei lassen sich auch jeweils drei Variablen betrachten; dabei steigt die Anzahl der Testfälle natürlich exponentiell. Der Extremfall wäre n-fach kombinatorische Überdeckung. Damit würde man den gesamten Eingaberaum abdecken, und das ist ja gerade, was man reduzieren möchte.

Um durch kombinatorisches Testen eine bessere Abdeckung zu erhalten, kann man das Programm in Module unterteilen, und kleine Module mit höherer Kombinationszahl testen.

Hinter dem kombinatorischen Testen steckt die Annahme, daß ein Fehler im Programm sich aus der Kombination von 2 oder 3 Eingabevariablen ergibt. Das macht auch Sinn, denn an einer Stelle im Programm werden selten alle Eingaben verwendet.

Eine Untersuchung des kombinatorischen Ansatzes zeigte folgende Ergebnisse [CDFP97]:

	Anzahl Testfälle	Anweisung	Verzweigung	Prädikat-Nutzungen	Rechen-Nutzungen
2 – fach	200	92 %	85 %	49 %	72 %
∞ - fach	436	92 %	85 %	49 %	72 %
Zufall	300	67 %	58 %	36 %	55 %

Die Zeile „2-fach" gibt die Anweisungs-, Verzweigungs- , Prädikatnutzungs- und Rechennutzungsüberdeckung an, die mit 2-fach kombinatorischem Testen erreicht wurde, die Zeile „∞ - fach" gibt die Überdeckungen für erschöpfende Kombination aller Variablen an, und die Zeile „Zufall" gibt die Überdeckungen an für Eingabedaten, die per Zufallsgenerator generiert wurden. Man sieht, dass durch eine höhere Kombinationszahl

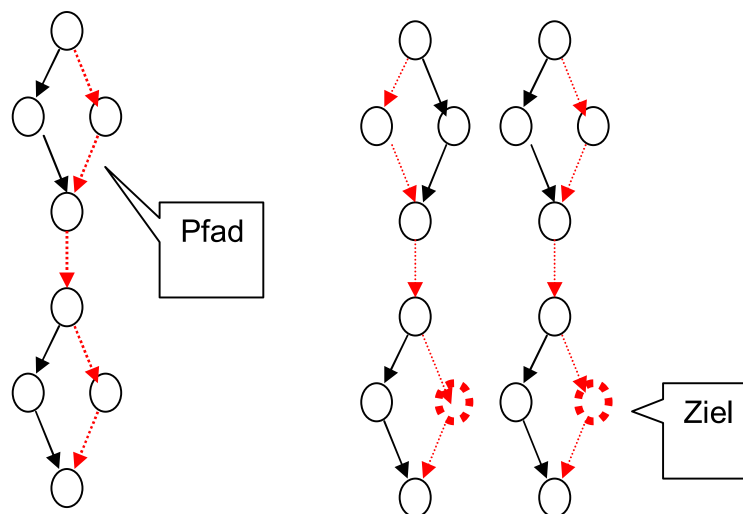
keine bessere Anweisungsüberdeckung erreicht wurde, und auch keine verbesserte Verzweigungsüberdeckung, Prädikatnutzungsüberdeckung oder Rechennutzungsüberdeckung. Die Eingaben, die Zufällig gewählt wurden erreichten schlechtere Überdeckungen obwohl 300 statt 200 Testfälle benutzt wurden.

2.2 Eigenschaften struktureller Ansätze

2.2.1 Pfadorientiert / Zielorientiert

Ein pfadorientierter Ansatz bekommt einen Pfad durch den Kontrollflussgraph vorgegeben und versucht Eingaben für das Programm zu finden, so daß genau dieser Pfad durchlaufen wird.

Ein zielorientierter Ansatz bekommt nur ein Ziel im Kontrollflussgraph, eine einzelne Anweisung. Er versucht Eingaben für das Programm zu finden, so daß dieser Zielknoten erreicht wird. Auf welchem Weg ist dabei egal.



2.2.2 Statisch / Dynamisch

Ein statisches Testverfahren analysiert aufgrund des Programmtextes die Abhängigkeiten der Variablen von anderen Variablen und von Anweisungen. Anstatt den tatsächlichen Wert der Variable zu berechnen wird eine Zeichenfolge aufgebaut (z.B: „a=a+2“). Das wird *Symbolische Ausführung* genannt. Der Wert der Variablen bestimmt ja letztendlich, welche Verzweigung bei einer if-Anweisung genommen wird. Statische Verfahren haben Schwierigkeiten mit dynamischen Datenstrukturen, mit großen Arrays und mit Bibliotheksfunktionen (da ihre Wirkung nicht leicht modelliert werden kann) und mit Goto- und Break-anweisungen.

Ein dynamisches Testverfahren arbeitet das Programm tatsächlich ab (in einem Interpreter) und beobachtet, wo der Programmfluss entlang geht. Das Programm kann natürlich vor dem Interpretieren mit Code in der selben Programmiersprache

instrumentiert werden. Geht der Programmfluss nicht, wie gewünscht, so sucht das Verfahren bessere Eingaben und startet das Programm von neuem. Das Programm wird sehr oft ausgeführt, bis Eingaben gefunden werden, durch die die gewünschten Pfade im Programmflussgraphen abgedeckt werden.

2.3 Strukturelle Ansätze

2.3.1 Der Randbedingungsansatz

Der Randbedingungsansatz (von Gotlieb, Botella und Rueher [GBR98]) ist ein statischer und pfadorientierter Ansatz. Statisch heißt, das Programm wird symbolisch analysiert; die Werte von Variablen werden nicht berechnet, vielmehr wird protokolliert, woraus die Werte entstehen. Das wichtigste Werkzeug, das verwendet wird, ist die sogenannte „Static Single Assignment“ Form des Programmes. Das ist eine Umbenennung der Variablen, so daß jede Variable nur ein einziges mal im Programm definiert wird

Originalcode:	Single Static:
a = Eingabe;	a_1 = Eingabe;
if (a > 7)	if ($a_1 > 7$)
a = 0;	$a_2 = 0$;
print a;	$a_3 = \Phi(a_1, a_2)$
	print a_3;

(definieren im Sinne von Wert zuweisen). Dadurch läßt sich eine eindeutige Abhängigkeit der Variablen voneinander modellieren.

Aus dem Programm in SSA Form wird ein Gleichungssystem gewonnen, dessen Lösungen angeben, wie die Eingabevariablen belegt sein müssen, damit ein bestimmter Pfad im Kontrollflussgraph durchlaufen wird. Die Gleichungen und Ungleichungen werden folgendermaßen gewonnen: Zunächst erhält man für jede Zuweisung „A = f(V1, ..., Vn)“ eine Gleichung „A == f(V1, ..., Vn)“. Gleichungen aus dem selben Basisblock werden UND-verknüpft. Diese Gleichungen modellieren das Programm. Um einen bestimmten Pfad durch den Kontrollflussgraph zu durchlaufen nimmt man aus allen if-Anweisungen Gleichungen hinzu, z.B. aus „if (A < B)“ die Gleichung „A < B“.

Dieses Gleichungssystem löst man mit einem beliebigen Gleichungslöser, den man aus anderen Forschungsarbeiten wählt. Findet dieser eine Lösung, so sind die Parameter, die zur Lösung führen auch die Parameter, die dazu führen, daß das zu testende Programm den gewünschten Pfad entlangläuft.

2.3.2 Simulated Annealing

Dies ist ein Dynamischer Ansatz von Gotlieb, Botella und Rueher [TCM98]. Er dient nur zum Testen einer einzelnen Funktion und setzt voraus, daß die Funktion Vor- und Nachbedingungen enthält. Die Vorbedingungen werden in Disjunktive Normalform

gebracht, die negierten Nachbedingungen ebenfalls. Anschließend werden Eingabedaten gesucht, für die bei der Ausführung des Programms beide Formeln wahr werden. Somit sind die Vorbedingungen der Funktion erfüllt, die Nachbedingungen aber nicht, und das bedeutet, daß die Funktion einen Fehler enthält. Die Formeln werden mit Simulated Annealing gelöst. Man kann die Formeln auch mit einem Gleichungssystemlöser lösen, dann würde man das Verfahren natürlich nicht mehr „Simulated Annealing“ nennen.

2.3.3 Der Verkettungsansatz

Der Verkettungsansatz ist ein strukturelles, zielorientiertes, dynamisches Verfahren, d.h.

Er benutzt den Kontrollflussgraphen des Programms.
Er bekommt eine Ziel-Anweisung im Kontrollflussgraph, die er erreichen soll und nicht einen ganzen Pfad, den er ablaufen muß.
Das Programm wird nicht als ganzes analysiert, sondern es wird interpretiert und immer dann unterbrochen, wenn der Programmfluss nicht so läuft, wie gewünscht.

Die beiden Komponenten des Verfahrens sind der Pfadgenerator und der Eingabegenerator. Diese beiden Programme werden bei der Suche von geeigneten Testeingaben ständig abwechselnd ausgeführt.

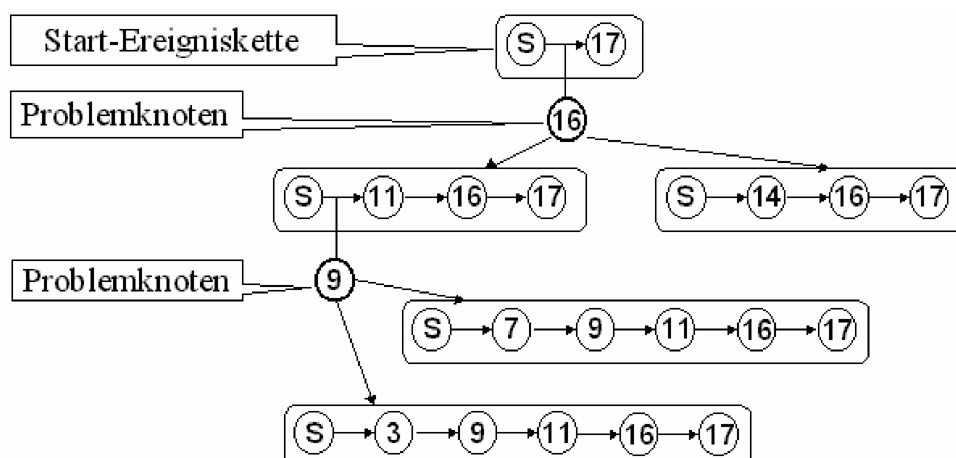
2.3.3.1 Der Pfadgenerator

Der Pfadgenerator steuert den Prozeß, indem er passende Ereignisketten erzeugt (solche, die den Suchprozess zu einer Lösung führen). Eine Ereigniskette ist eine Folge von Ereignissen. Ein Ereignis besteht aus einem Knoten des Kontrollflussgraphen (eine einzelne Anweisung), und einer Bedingungsmenge. Die Bedingungsmenge enthält Variablen, die nicht mehr neu definiert werden dürfen, und die den Kontrollfluss an if-Anweisungen steuern. Die i. Bedingungsmenge in der Ereigniskette entsteht aus der (i-1)., indem jene Variablen dazugenommen werden, die im Ereignis i definiert werden.

Die Ereigniskette, mit der das Verfahren startet besteht nur aus dem Startknoten, und dem Zielknoten. Der Pfadgenerator gibt die neu gebaute Ereigniskette dem Eingabegenerator in der Hoffnung, daß dieser Eingaben findet, die diese Ereigniskette durchlaufen.

Gelingt dem Eingabegenerator das nicht, dann liefert er dem Pfadgenerator einen „Problemknoten“, an dem er nicht in eine geeignete Richtung verzweigen konnte. Der Problemknoten ist eine Verzweigungsanweisung.

Der Pfadgenerator sucht die Menge aller Knoten, in denen eine Variable definiert wird,



die im Problemknoten verwendet wird. Aus dieser Menge und der bestehenden Ereigniskette generiert er neue Ereignisketten, indem er die definierenden Knoten irgendwo vor dem Problemknoten einfügt.

Der Sinn der Ereigniskette ist es, den Suchraum einzuschränken. Es werden Variablen festgehalten, mit Werten, die sie tatsächlich bei der Programmausführung auch annehmen. Der Eingabegenerator hat so wesentlich bessere Chancen, passende Eingaben zu finden.

2.3.3.2 Der Eingabegenerator

Der Eingabegenerator bekommt vom Pfadgenerator eine Ereigniskette geliefert und versucht, Eingaben zu finden, so daß diese Ereigniskette abgelaufen wird. Welchen Weg der Programmfluss außerhalb der Ereigniskette nimmt, ist egal. Hauptsache, die Knoten der Ereigniskette werden in der gegebenen Reihenfolge besucht.

Nimmt der Programmfluss eine Verzweigung, die nicht zum nächsten Knoten in der Ereigniskette führen kann, ohne Variablen neu zu definieren, die in vorherigen Knoten aus der Ereigniskette definiert wurden, so wird der Programmfluss unterbrochen und neue Eingaben werden gesucht.
Die Variablen, die auf einem Knoten definiert werden, der in der Ereigniskette enthalten ist, werden nur ein einziges mal an dieser Stelle definiert. Variablen, die in Knoten definiert werden, die nicht zur Ereigniskette gehoeren, koennen beliebig oft definiert werden.

Findet der Eingabegenerator keine Eingaben, die den Programmfluss an dieser Stelle ändern, so ist dieser Knoten ein „Problemknoten", und der Pfadgenerator muß weiterhelfen.

Das geht solange, bis Eingaben gefunden werden, bei denen der Zielknoten erreicht wird, oder bis ein Zeitlimit erreicht wurde. Schließlich ist nicht sichergestellt, ob der Zielknoten überhaupt erreichbar ist. Beispiel:

```
if (false) {
   fail()
}
```

fail() wird hier nie ausgeführt.

Wenn der Zielknoten erreichbar ist, dann ist leider immernoch nicht sicher, ob wir eine entsprechende Eingabe finden. Das liegt daran, dass nicht der ganze Eingaberaum abgesucht wird, sondern nur „vielversprechende" Teile daraus.

2.4 Randbedingungen versus Genetische Optimierer

Beim Vergleich von statischen mit dynamischen Testdatengenerierungsverfahren ist mir aufgefallen, daß beidemal Optimierungstechniken verwendet werden. Bei den Statischen Verfahren sind das Gleichungssysteme mit Randbedingungen. Das ist der Versuch, eine Aufgabe genau darzustellen, und deterministisch zu lösen. Bei den dynamischen Verfahren werden evolutionäre Optimierungsstrategien verwendet (auch im Verkettungsverfahren steckt ein evolutionärer Optimierer; man kann einen passenden auswählen). Diese Optimierer versuchen nicht, die Aufgabe im ganzen zu erfassen, sondern sie versuchen durch lokale Änderung der Eingabevariablen die Funktion schrittweise zu optimieren. Die Hoffnung dabei ist, daß man im Endeffekt eine gültige Lösung findet, die allen Einschränkungen gerecht wird.

Die beiden grundsätzlichen Optimierungsparadigmen findet man wieder in der Testdatengenerierung.

3 Orakel

Ein Orakel im umgangssprachlichen Sinn ist eine Entität, von der man wahre Aussagen über die Zukunft erlangt. In der Informatik ist ein Orakel ein Programm, das für theoretisch unlösbare Probleme die Lösung angibt, und zwar deterministisch und in endlicher Zeit und mit endlich großem Speicher.

Wenn man dem Programm die automatisch generierten Eingaben gegeben hat, muß man prüfen können, ob die Ausgaben korrekt sind. Sind sie es nicht, so hat man einen Fehler im Programm gefunden; das Testen war erfolgreich.

Wüßte man, ob das Ergebnis korrekt wäre, so hieße das, daß man beim Prüfen des Ergebnis von irgendwoher schon ein korrektes Ergebnis hätte, mit dem man verglichen hätte. Das wiederum heißt, daß das Programm mit dem man das andere Ergebnis berechnet hat korrekt gewesen sein muß. Und das ist ja gerade, was man insgesamt Testen will. Eine Ausnahme von dieser Folgerungskette sind Programme, deren Lösung spezifiziert ist, nach dem Schema „Suche eine Zahl, für die das und das gilt“; wenn „das und das“ gilt, so ist die Ausgabe des Programmes natürlich korrekt, und man braucht sie mit nichts anderem vergleichen.

In der Praxis hat man keine Orakel, die unfehlbar das richtige Ergebnis liefern, aber man hat oftmals Eingabe-Ausgabe-Paare, von denen man Korrektheit erwarten kann. Das ist dann der Fall, wenn das entwickelte Programm ein schon vorhandenes ersetzen soll, d.h. wenn schon einmal ein korrektes Programm existierte, ein sogenanntes „Gold-Programm“. Das neue Programm braucht man deshalb, weil für das alte Programm einer der folgenden Punkte gilt: Es ist nicht mehr wartbar; Es wurde in einer Umgebung implementiert, die für den Gebrauch einfach zu langsam ist; Es wird in einem größeren Kontext benötigt, als Teilprogramm für ein komplettes Anwendungssystem; Es ist auf ein

bestimmtes Betriebssystem (bzw. für eine bestimmte Hardware) zugeschnitten, und wird nun auf einem anderen Betriebssystem (bzw. Hardware) benötigt;

Hat man kein Goldprogramm, so kann man die Ausgaben des zu testenden Programmes immernoch daraufhin prüfen, ob sie innerhalb von „sinnvollen" Grenzen liegen. Beispiel: Ein Programm, das eine Klappe eines Flugzeugs steuert darf die Klappe niemals mehr als 100% auslenken.

3.1 Zusicherungen

Zusicherungen (auf Englisch Assertions) helfen dem Programmierer beim Verständnis des Programms. Sie dienen dazu, erfaßte Zusammenhänge auszudrücken, und werden wie normaler Code ins Programm geschrieben. Die Konsistenz mit der Spezifikation, und die Konsistenz innerhalb einer Implementierung läßt sich so gewährleisten. Eine Zusicherung wird zur Laufzeit überprüft. Ist sie nicht erfüllt, so bricht das Programm mit einer Fehlermeldung ab, die üblicherweise die Quellcode-Datei angibt, die Zeile, in der die Zusicherung steht, sowie die genaue Bedingung der Zusicherung. Eine spezielle Verwendungsmöglichkeit für Zusicherungen ist die Überprüfung von Vorbedingungen und Nachbedingungen in Funktionen. In der Programmiersprache C zum Beispiel sieht eine Zusicherung so aus:

```
assert(hoehe == breite);
```

Eine Zusicherung kann man insofern auch als Orakel ansehen, als sie eine Instanz ist, die uns sagt, ob das Programm korrekt ist oder nicht.

3.1.1 Zusicherungen nutzen, um Testdaten zu generieren

Jede Zusicherung der obigen Art läßt sich darstellen als eine if-Anweisung, und eine „Fail"-Anweisung, die die Fehlermeldung ausgibt und das Programm terminiert. Man kann auch ein ganzes Code-Stück (geschrieben in der jeweiligen Programmiersprache) als Zusicherung ansehen, sofern es irgendwo im Stil einer Zusicherung eine fail-Anweisung hat. Ziel des Testens ist es nun, diese Fail-Anweisung zur Ausführung zu bringen. Gelingt dies, so bedeutet das, daß entweder die Zusicherung nicht stimmt, oder daß das Programm irgendwo inkonsistent ist; jedenfalls enthält dann das Gesamtprogramm einen Fehler. Und das ist genau, was wir mit dem Testen aufdecken wollen.

Um Zusicherungen zum Testen auszunutzen können wir ein beliebiges Strukturelles Verfahren benutzen, welches Eingabedaten generiert, die zur Ausführung einer speziellen Anweisung im Programm führen: Der Fail-Anweisung (bzw. einer Fail-Anweisung von mehreren, die es normalerweise gibt).

Das Verfahren, das Korel und Al-Yami in ihrem Paper „Assertion-Oriented Automated Test Data Generation" verwenden, ist das Verkettungsverfahren von Korel und Ferguson (Englisch: „The Chaining Approach for Software Test Data Generation").

4 Literatur

[GBR98] Arnaud Gotlieb, Bernard Botella, Michel Rueher. Automatic test data generation using constraint solving techniques. In ACM Sigsoft International Symposium on Software Testing and Analysis 1998, volume 23,2 of ACM Software Engineering Notes, Seiten 53-62, New York, März 1998

[TCM98] Nigel J. Tracey, John Clark, Keith Mander. Automated program flaw finding using simulated annealing. In ACM SIGSOFT International Symposium on Software Testing and analysis 1998, volume 23,2 of ACM Software Engineering Notes, Seiten 73-81, New York, März 1998.

[DJKLLPH99] Dalal, Jain, Karunanithi, Leaton, Lott, Patton, Horowitz. Model-Based Testing inPractice. Interantaional Conference on Software Engineering ICSE 21 (1999), Seiten 285-294.

[DESMI97] Dunietz, Ehrlich, Szablak, Mallows, Iannino. Applying Design of Experiments to Software Testing. International Conference on Software Engineering ICSE 19, 1997, Seiten 205-215.

[CDFP97] Cohen, Dalal, Fredman, Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Transactions on Software Engineering TSE 23, 7, 1997, Seiten 437-444.

[CDPP96] Cohen, Dalal, Parelius, Patton. The Combinatorial Design Approach to Automatic Test Geneeration. IEEE Software 13,5, 1997, Seiten 83-88.

[KA96] Korel, Al-Yami. Assertion-Oriente Automated Test Data Generation. International Conference on Software Engineering ICSE 188, 1996, Seiten 71-80.

[KF96] Korel, Ferguson. The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology, Ausgabe 5, Januar 1996, Seiten 63-86.

[Ros92] Rosenblum. Toward a Method of Programming with Assertions. Proceeings of the Interanational Conference on Software Engineering. 1992, Seiten 92-104.

Program Slicing

Felix Hupfeld - Felix.Hupfeld@bigfoot.de

1 Einleitung

Die vorliegende Ausarbeitung beschäftigt sich mit einer Reihe von Analysetechniken, die dem Software-Entwickler bei der Entdeckung von Strukturen im Programmcode behilflich sind. Dieses Wissen kann ihn bei seinen Aufgaben unterstützen, sei es bei der Fehlersuche, beim Testen oder beim Reverse Engineering.
Der Schwerpunkt liegt auf der Technik des sogenannten *Program Slicing*. Das Programm wird hierbei in *Slices* (Scheiben) aufgeschnitten, die den Teil des Programms enthalten, der Einfluß auf die Daten an einer bestimmten Stelle des Programms hat. Der Programmierer kann so bei der Fehlersuche im Fehlerfall zurückverfolgen, welche Teile des Programms das fehlerhafte Datum geändert haben könnten.

Anschließend wird die Analyse der Nebeneffekte von Programmänderungen (MOD-Analyse) betrachtet, die in etwa den komplementären Fall zum *Program Slice* behandelt. Man ist hier an der Menge der Variablen interessiert, die durch die Ausführung eines gegebenen Programmteils beeinflusst werden.

Im Weiteren wird ein Verfahren vorgestellt, mit Hilfe dessen sich Invarianten in Programmen dynamisch berechnen lassen. Invarianten bewahren den Entwickler vor falschen Annahmen über das vorliegende Programm und beugen so Fehlern vor.
Abschließend beschäftigt sich die vorliegende Arbeit mit Extraktoren für Aufruf-Graphen, also Verfahren, die herausfinden, welche Teile eines Programms welchen anderen Teil aufrufen. Es wird auf eine Studie eingegangen, die 5 Implementierungen davon vergleicht. Diese Aufruf-Graphen werden von Compilern während der Optimierung benötigt, sollen aber auch Programmierern bei dem Verstehen von Programmen unterstützen.
Im Anhang befindet sich noch eine kleine Übersicht über einige Kernbegriffe des *Program Slicing*.

2 Program Slicing

2.1 Einführung und Definition

Der Begriff des *Program Slicing* wurde 1979 in der Dissertation von Mark Weiser [M79] geprägt. Mark Weiser kam später noch durch die Erfindung des *Ubiquitous Computing* Paradigma zu weiterem Ruhm und verstarb 2000 bei einem Unfall. Mehrere Publikationen [M81, M84] von ihm beschreiben das Verfahren des *Program Slicing* und weisen unter anderem experimentell nach, daß es eine Methode ist, die erfahrene Programmierer von sich aus unbewußt zur Fehlersuche anwenden [W82]. In einer späteren Studie [NKI98] konnte mit Hilfe eines Experimentes nachgewiesen werden, daß

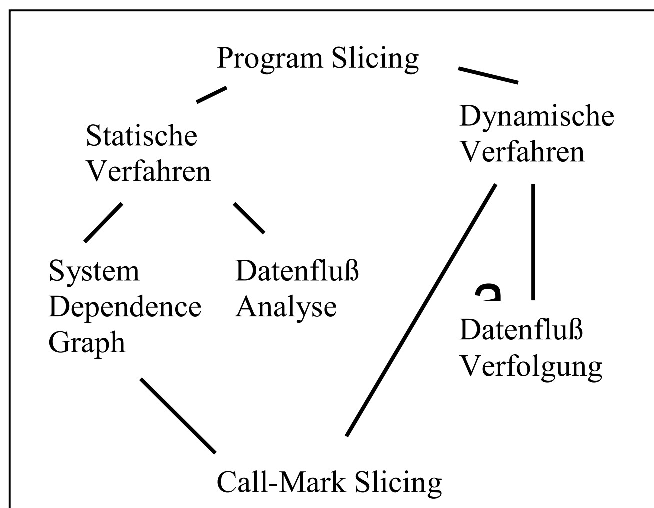
die Unterstützung durch Program Slicing die Zeit reduziert, die gebraucht wird, um Fehler in einem Programm zu lokalisieren.

In der allgemeinen Definition wird ein *Program Slice* durch einen Programmpunkt *p* und eine Variable *x* beschrieben und enthält den Teil des Programms, der den Wert der Variable *x* am Programmpunkt *p* beeinflussen kann. Ein Programmpunkt entspricht einem Ausdruck im Quelltext des Programms, also praktisch meist einer Zeile. Es gibt immer mindestens einen *Program Slice*, das Programm selbst und ein *Program Slice* ist immer ausführbar. Das Tupel (*p*, *x*) wird auch als *Slice* Kriterium bezeichnet.
Das Program wird also quasi in Scheiben zerschnitten. Die so gewonnen Teilprogramme sind für den Wert einer Variable an einer bestimmten Stelle verantwortlich, sprich, sie können ihn im Laufe der Programmausführung auf irgendeine Weise beeinflußen. So können bei der Fehlersuche die Teile des Programms, die sicher nichts mit dem fehlerhaften Wert an einer bestimmten Stelle des Programms zu tun haben, einfach ausgeblendet werden.

Beispiel 1

```
1:void P( int a, int b)
2:{
3:  int c = a + b;
4:  int k = Math.PI;
5:  int t = k * 3;
6:  int d = c * 5;
7:  int e = t/d;
8:}
```

Zeilen 3 und 6 sind der zum Slice-Kriterium (7,{d}) gehörende Programm Slice.



Auch ist es in manchen Fällen interessant, alle Programmzeilen zu finden, die durch eine Variable x am Programmpunkt p beeinflusst werden und nicht umgekehrt, wie in der klassischen Definition. Man spricht dabei von einem *forward Program Slice*. Gerade bei der Abschätzung von Auswirkungen von Änderungen kann das interessant sein.
Im Zuge der Weiterentwicklung des Konzepts wurde der Begriff des *Program Slicing* verfeinert und zahlreiche Differenzierungen wurden eingeführt.

2.2 Statisches und dynamisches Slicing

In seiner klassischen Version spricht man von einem s*tatischen Program Slice*, da die Definition nur die statische Struktur des Programms berücksichtigt. Dynamische Aspekte, wie z.B. die Frage, welche Teile des Programms tatsächlich ausgeführt wurden und somit die betrachtete Variable tatsächlich beeinflusst haben, bleiben unberücksichtigt. Erst seit den Arbeiten von Korel [KL88] wird *dynamisches Program Slicing* untersucht. Dabei wird während der Programmausführung mitprotokolliert, welche Teile des Programms ausgeführt worden sind und wie sich die Variablen

untereinander beeinflußt haben (Datenabhängigkeiten). Diese Information wird dann beim *Slicing* dazu verwendet, auch Teile des Programms wegzulassen, die praktisch keinen Einfluß auf die *Slicing*-Variable hatten. Das sind genau die Teile des statischen *Slice*, die vorher nicht ausgeführt worden sind und somit nur theoretisch die Variable hätten ändern können, es praktisch aber nicht getan haben.
In der praktischen Anwendung haben statische Slices oft das Problem, daß sie zu groß sind, dh. daß sie zu viele Teile des Programms zum Slice hinzufügen und damit ihre Nützlichkeit für den Entwickler stark einschränken. Die Berechnung von dynamischen Slices hingegen ist sehr resourcenintensiv, da die dynamischen Variablenabhängigkeiten während der Ausführung mitverfolgt werden müssen. Es hat aber den Vorteil, daß die so berechneten Slices kleiner werden, da es nur die tatsächlich ausgeführten Teile des Programms berücksichtigt.

2.3 Besonderheiten prozeduralen Slicens

Im praktischen Einsatz eines *Program Slicers* ist man daran interessiert, große Programmsysteme zu bearbeiten, die natürgemäß aus vielen Prozeduren bestehen. In den ursprünglichen Arbeiten wurde nur der intraprozedurale Fall betrachtet, d.h. das *Slicing* einer einzelnen Prozedur oder eines Programms ohne lokale Variablen. Programme die nur globale Variablen haben, sind praktisch mit einer einzelnen Prozedur äquivalent, da die Probleme der Variablenumbenennung durch Parameterübergabe und der Sichtbarkeit nicht vorkommen. Mittlerweile beherrscht man auch *interprozedurales Program Slicing*.

Einige Sprachen weichen das Konzept des prozeduralen Programmierens auf.. So unterstützt z.B. die Sprache C++ die Ausnahmebehandlung oder C das *exit*() und *goto*. Man spricht hier von *arbitrary* oder *unstructured interprocedural control flow*, also einem Kontrollfluß zwischen den Prozeduren, der sich nicht mehr nur an die Strukturen der *call-return* Semantik hält und damit nicht mehr unbedingt dahin zurückkehrt, wo er ursprünglich her kam. Bei einem *exit()* stoppt der Kontrolfluß einfach, ein *throw()* springt an das zugehörige *catch()* einer Prozedur oder Methode, die unter Umständen viel weiter oben in der Aufrufkette zu finden ist. Das *goto* kann als die allgemeinste Form von unstrukturiertem Kontrollfluß betrachtet werden.

2.4 System Dependence Graph vs. Datenfluß-Analyse

Die bisher entwickelten Verfahren zur Berechnung von *Program Slices* lassen sich in zwei Gruppen einteilen.

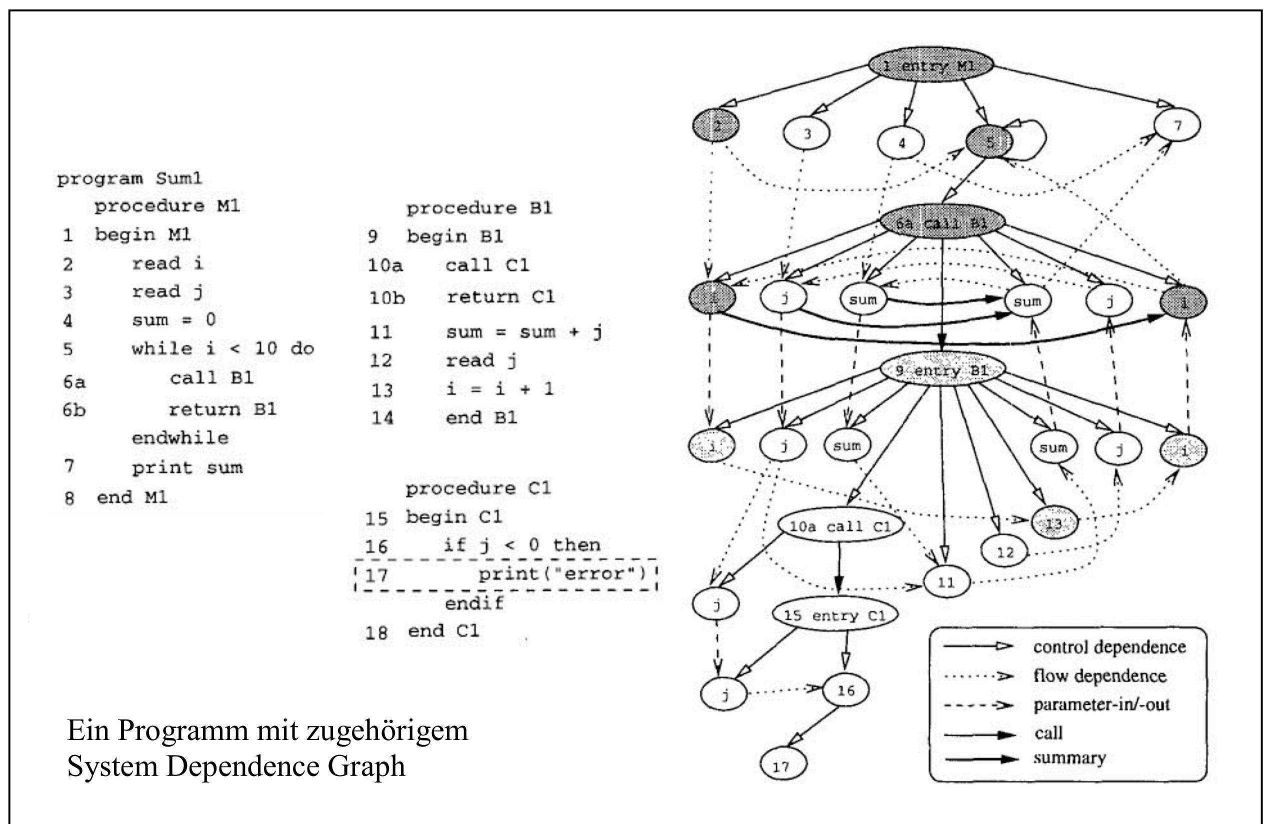
Verfahren die auf dem *System Dependence Graph* (SDG) arbeiten, berechnen mit zum Teil großem Aufwand den Datenfluß und andere Informationen des kompletten Programms um den SDG zu konstruieren. Mit dessen Hilfe können die *Program Slices* durch einen einfachen Graph-Erreichbarkeits-Algorithmus berechnet werden. Dieser skaliert linear mit der Größe des SDGs. Der Algorithmus wurde als erstes von Horwitz, Reps und Brinkley [HRB90] beschrieben und wird auch manchmal als HRB-SDG Algorithmus referenziert. Er arbeitet in zwei Phasen und findet, von dem Knoten ausgehend, der das Slice Kriterium repräsentiert, alle über bestimmte Arten von Kanten erreichbare Knoten. Diese sind dann der zugehörige *Program Slice*.

Die andere Gruppe der Verfahren beruht auf der Datenfluß-Analyse und berechnet die nötigen Informationen während des *Slice*-Prozesses. Sie benötigen weniger Berechnungen im Voraus, sind dafür während der eigentlichen Berechnung aufwendiger. Dadurch haben beide Gruppen von Verfahren ihre Anwendungsgebiete. Will man mehrere *Slices* berechnen kann man den SDG mehrfach verwenden. Dessen einmaliger Berechnungsaufwand amortisiert sich dann für die verschiedenen Slices. Bei einem einzigen *Slice* hingegen ist die aufwendige Berechnung des SDG nicht rentabel und die Verwendung eines Datenfluß-basierten Algorithmus bietet sich an.

2.4.1 Procedure Dependence Graph (PDG)

Ein Procedure Dependence Graph (PDG) stellt die Daten- und Kontrollabhängigkeiten innerhalb einer Prozedur dar. Die Knoten sind hierbei die Darstellung der Ausdrücke und Anweisungen der Prozedur, die gerichteten Kanten entsprechen den Daten- und Kontrollabhängigkeiten. Zusätzlich werden noch Knoten für den Eintritts- und die Austrittspunkte der Prozedur eingefügt. Außerdem sind zu jedem Knoten noch die Mengen *DEF* und *REF* definiert, die die an dem Knoten definierten oder benutzten Variablen beinhalten.

Flußabhängigkeiten werden weiter in *loop-carried* und *loop-independent* unterteilt, je nachdem ob sie Ergebnis eines Schleifendurchlaufs sind.



Ein Programm mit zugehörigem System Dependence Graph

Ein *Slice*-Kriterium entspricht hier einem Knoten im PDG und den dort definierten oder benutzen Variablen.

2.4.2 System Dependence Graph (SDG)

Der System Dependence Graph (SDG) erweitert das Konzept des PDG auf ganze Programme. Er besteht aus den PDGs für die einzelnen Prozeduren und einem Program Dependence Graph für *main*. Zusätzlich werden noch weitere Knoten und Kanten eingefügt um die Parameter Übergabe an die Prozeduren darzustellen.
Die Parameter Übergabe wird in mehrere Teilschritte zerlegt:

1. Die aufrufende Prozedur kopiert ihre tatsächlichen Parameter in temporäre Variablen.
2. Die formalen Parameter der aufgerufenen Prozedur werden mit den entsprechenden temporären Variablen initialisiert.
3. Bei der Rückkehr werden die Resultatswerte in temporäre Variablen kopiert und dann
4. die tatsächlichen Variablen mit den Werten der temporären Variablen überschrieben.

Für jeden Prozeduraufruf wird in den SDG ein *call-site* Knoten (Knoten 6a im Beispiel) mit
actual-in (i, j, sum links darunter) und *actual-out* Knoten (sum, j, i rechts) für die Parameter eingefügt. Die *actual-** Knoten sind dabei kontrollabhängig vom *call-site* Knoten. Die PDGs der Prozeduren selbst werden mit einem Eingangsknoten und *formal-in* (i, j, sum unterhalb von Knoten 9) und *formal-out* (sum, j, i rechts davon) Knoten erweitert, die beide Kontrollabhängig zum Eingangsknoten sind.
Zusätzlich werden noch Kontrollabhänigkeitskanten zwischen den *call-sites* und den Eingangsknoten der Prozeduren und *parameter-in* bzw. *parameter-out* Kanten zwischen den entsprechenden *formal-** und *actual-** Knoten eingefügt.

2.4.3 HRB-SDG Algorithmus

Der *Slicing* Algorithmus von Horwitz, Reps und Binkley [HRB90] arbeitet auf dem SDG in zwei Phasen um den *Slice* zu berechnen.

In der ersten Phase läuft er von dem Knoten, der das Slice Kriterium repräsentiert zurück über alle außer *parameter-out* Kanten und markiert auf dem Weg die Knoten. Das sind alle Knoten die von s aus erreicht werden können, ohne in die Prozeduraufrufe hinabzusteigen. In der zweiten Phase verfolgt er von den bisher markierten Knoten aus alle außer *call* und *parameter-in* Kanten und markiert die dann gefundenen Knoten. Dabei geht der Algorithmus also in alle vorher liegengelassenen Prozeduren und berechnet die restlichen Knoten des Slice. Der *Slice* ist dann die Menge aller markierten Knoten.

2.5 Unstruktierierter Kontrollfluß

Für das *Slicen* von Programmen in Programmiersprachen mit Konstrukten, die nicht der *call-return* Semantik gehorchen (*unstructured interprocedural control flow)* existieren zwei Verfahren. Das Verfahren von Sinha, Harrold und Rothermel [SHR99] baut einen erweiterten SDG auf und benutzt dann den zweiphasigen HRB-SDG Algorithmus um den *Slice* zu berechnen, das Verfahren von Harrold und Ci [HC98] arbeitet hingegen Datenfluß-basiert und kann dabei noch alte berechnete Slices in die Berechnung mit

einbeziehen um sie so zu beschleunigen. Wie oben beschreiben haben beide Arten ihre Anwendungsfälle (*SDG* vs. *data-flow*). [SHR99] erweitern den SDG hauptsächlich durch Einführung der sogenannten *return* und *exit* Knoten, mit denen der Tatsache Rechnung getragen wird, daß die Prozedur auch über andere als die der normalen *call-return* Semantik entsprechenden Wege verlassen werden kann. Solche Prozeduraufrufe werden als *potentially non-returning call sites* (PNRCs) bezeichnet, also als Aufrufe, von denen möglicherweiße nicht mehr an die aufrufende Stelle zurückgekehrt wird.

2.6 Slicing objektorientierter Programme

Die Sprachkonstrukte objektorientierter Sprachen wie Klassen und Polymorphismus müssen beim Slicen gesondert betrachtet werden. Zuerst stellt sich natürlich die Frage wie mit Konstruktoren, polymorphen Funktionsaufrufen und abgeleiteten Methoden umgegangen werden muß. Mit der Klasse wird aber auch eine neue Einheit eingeführt, die man gerne isoliert oder in unvollständigen Verbünden *slicen* würde.
Larsen und Harrold [LH96] beschreiben hierzu, wie man den SDG Graphen für Basisklassen, abgeleitete Klassen, Instanziierungen und polymorphe Methodenaufrufe aufbaut. Unvollständige Systeme, wie einzelne Klassen oder Klassenbibliotheken werden mit Hilfe eines sogenannten *frames* behandelt, das als eine Art *main*() für das unvollständige System dient und nach Instanziierung jede öffentliche Methode der Klasse in einer Schleife aufruft und dann das erzeugte Objekt wieder löscht.

Für objektorientierte Software muß weiterhin das Slice Kriterium erweitert werden. Wo beim prozeduralen Slicen noch das *Slice* Kriterium (Programmpunkt *p*, Variable *x*) genügte, werden beim objektorientierten Programmieren Variablenzugriffe oft durch Methodenaufrufe realisiert. Das *Slice* Kriterium wird dahingehend verändert, daß *x* sowohl eine Variable als auch Methodenaufruf sein kann.
Aufgrund der Polymorphie kann dieser Methodenaufruf desweiteren dynamisch verschiedene Klassen ansprechen. Es wird also wichtig, welche Klassen durch einen solchen Aufruf gemeint sein können um den Slice entsprechend zu erweitern.
Ein wichtiges Hilfsmittel hierzu ist die sogenannte *points-to* Analyse (PTA). Sie liefert für jeden polymorphen Methodenaufruf die Menge der aufrufbaren Methoden.
Tonella, Antoniol, Fiutem und Merlo [TAFM97] stellen hierzu ein Verfahren vor, das für C++ Programme diese Analyse vornimmt. Das Verfahren arbeitet ohne Rücksicht auf den Kontrollfluß (*flow insensitive*), d.h. der Programmquelltext wird einfach sequentiell von vorne nach hinten durchgegangen, wobei auf Ausdrücke geachtet wird, die das Ziel von Pointern verändern. Dadurch kann es insofern zu Ungenauigkeiten kommen, als daß zu viele points-to Relationen eingefügt werden. Weggelassen werden dadurch aber keine.
Während des Durchlaufs wird ein sogenannter *storage shape* Graph (SSG) aufgebaut aus dem dann abgeleitet wird, welche Ausdrücke auf eine feste Speicherstelle zeigen (*fixed location*) oder deren Ziel sich abhängig von Variablen ändern kann (*variable location*), was dann vom Slicer berücksichtigt werden kann.

2.7 Slicing mehrfädiger Programme

Bisher wurde nur das Slicen einfädiger (*single-threaded*) Programme betrachtet. Das Hauptproblem beim Slicen mehrfädiger (*multi-threaded*) Programme ist die Behandlung von Datenabhängigkeiten zwischen verschiedenen Threads.
Nanda und Ramesh [NR00] beschreiben hierzu, wie man dazu einen erweiterten Kontrollflußgraphen (TCFG) aufbaut und dann mit Hilfe eines vorgestellten Algorithmus zum Slice kommt. Ihr Verfahren produziert bewiesen korrekte Slices, setzt aber voraus, daß Threads sich nicht explizit synchronisieren, also z.B. keine Semaphoren o.ä. verwenden. Ihre Implementierung kann Java Programm slicen, die *synchronized* Methoden enthalten.

2.8 Call-Mark Slicing

Das *call-mark slicing* von Nishimatsu, Jihira, Kusumoto und Inoue [NJKI99] ist eine Kombination von statischer und dynamischer Analyse. Vor der Ausführung des Programms werden die statische Daten-, Kontroll und sogenannte Ausführungsabhängigkeiten (*execution dependence*) berechnet. Bei der Ausführung des Programms wird dann nur noch mitprotokolliert, welche Prozeduren tatsächlich aufgerufen worden sind, d.h. im SDG werden die ausgeführten *call* Knoten markiert (*call marking*). Der daraus resultierende *call-mark slice* ist als ein Teil des statischen *program slice* und enthält die tatsächlich ausgeführten Teile des statischen Slice.
Die Praxis zeigt, daß die mit Hilfe des *call-mark slicing* gewonnen Slices kleiner sind, die Analyse und die Ausführung des Programmes aber nicht viel länger dauert.

3 Interprozedurale Analyse von Änderungs-Nebeneffekten

Während man beim *Program Slicing* an der Frage interessiert ist, welche Programmteile den Wert einer bestimmten Variable an einer bestimmten Stelle beeinflussen, will man bei der sogenannten MOD-Analyse (*interprocedural modficiation side effects analysis*) von Ryder, Landi und Zhang [LRZ93] herausfinden, welche Variablen von einer bestimmten Prozedur beeinflußt werden. Die MOD-Analyse ist damit eine Art Gegenstück zum Program Slicing und leistet wertvolle Hilfe im Softwareentwicklungsprozeß. Man spricht in der Literatur auch vom MOD Problem, dessen Lösung eben die Menge der geänderten Variablen ist.
MOD kann während des Übersetzungsvorgangs erzeugt werden. Diese Information aber während der weiteren Systementwicklung dadurch aktuell zu halten wäre ein teurer Prozeß. Man ist daher daran interessiert, MOD inkrementell mit den Änderungen des Systems auf dem neuesten Stand zu halten. Jede einfache Änderung im Quelltext soll also eine einfache

Aufteilung von MOD(P) in Teilprobleme:

1. Seiteneffekte der Ausführung von Zuweisung n (incl. Variablenumbennenungen) im Aufruf-Kontext RA: CondLMOD(n, RA), die Menge der geänderten Variablen
2. Seiteneffekte aller Zuweisungen in Prozedur P im Kontext RA, CondIMOD(P, RA)
3. Interprozedurale Seiteneffekte von P im Kontext RA, PMOD(P, RA)
4. Interprozedurale Seiteneffekte von P, MOD(P)

Änderung in den Datenstrukturen zum Erzeugen von MOD nach sich ziehen.

Die vorliegende Publikation von Yur, Ryder und Landi [YRLS97] faktorisiert hierfür das MOD Problem in Teilprobleme und leitet dann einen Algorithmus ab, mit Hilfe dessen Änderungen im Quelltext direkt in die MOD Berechnungen eingehen. Der Algorithmus arbeitet auf einem sogenannten *call-RA* Graphen, der einem Aufruf-Graphen ähnelt und für jede Prozedur pro Aufrufmöglichkeit (*calling context* bzw. *reaching alias* (RA)) einen Knoten enthält.
Die für die MOD Berechnung wichtigsten Ausdrücke sind Zuweisungen und Funktionsaufrufe, wobei nochmal unterschieden wird zwischen Zeiger-Zuweisungen und Nicht-Zeiger-Zuweisungen. Einsetzen, ändern oder entfernen eines solchen Ausdrucks induziert jeweils eine Änderung im call-RA Graphen, die entweder strukturell (Form des Graphen wird verändert) oder nicht-strukturell sein können.
Wenn z.B. eine Zuweisung geändert wird, die ohne Pointer eine lokale Variable ändert, dann betrifft dies nur die MOD Lösung der beinhaltenden Prozedur. Wenn hingegen ein globale Variable in diese Zuweisung geändert wird, kann das Auswirkungen auf die MOD Lösungen aller direkt und indirekt aufrufenden Prozeduren haben.

4 Dynamisches Extrahieren von Programm Invarianten

Programm Invarianten sind Aussagen über Variablen an einem bestimmten Programmpunkt. So kann z.B. innerhalb einer leeren Zählschleife von eins bis hundert die Aussage gemacht werden, daß die Zählvariable nur Werte zwischen eins und hundert annimmt. Läuft die Schleife immer bis zum Ende, d.h. enthält sie keine weiteren Abbruch-Kriterien, so ist der Wert der Zählvariable nach der Schleife immer 101.
Solche Invarianten können helfen, korrekte Programme zu scheiben und können dynamisch (*assertions*, *assert*()) oder statisch (z.B. Typüberprüfung beim Übersetzen des Programms) überprüft werden.

```
void Prod( int a, int b)
{
  k = 5;
  for( int i = 0; i < 100; i++ )
     k++;
  ;
}
```

5 <= k < 105; 0 <= i < 100

k = 105; i = 100

Die vorliegenden Arbeiten von Ernst, Cockrell, Griswold und Notkin [ECGN99, ECGN00] konzentrieren sich auf das dynamische Entdecken von Invarianten, d.h. die Invarianten werden während des Programmlaufs extrahiert. Diese können dann den Entwickler zusammen mit dem Quelltext beim Verstehen des Programms unterstützen und ihn so vor falschen Annahmen schützen die zu Fehlern bei Änderungen führen können.
Der Weg vom Quelltext zu den Invarianten erfolgt über mehrere Zwischenschritte:

1. Das Programm wird mit Instruktionen angereichert, die den Wert der Variablen an bestimmten Programmpunkten in einem Logfile mitschreiben (*Instrumentation*).
2. Das so erweiterte Programm wird übersetzt und

3. mit einer Reihe von Eingabedaten (*Test suite*) gestartet.
4. Aus dem dabei generierten Logfile werden die Invarianten extrahiert.

Interessant sind hierbei die Eintritts- und Austrittspunkte von Prozeduren und die Schleifenköpfe. Für jeden solchen Programmpunkt enthält dann das Logfile die dort vorkommenden Variablen und ihre Werte. Auch abgeleitete Variablen, wie z.B. `array[i]` in einer Schleife oder die Größe von Sequenzen werden mitgeschnitten.

Die *Dynamic Invariant Detection Engine* überprüft dann für jede Invariante, ob sie gilt und wieviele sie unterstützende Meßwerte oder Wertpaare gefunden werden können. Für jede Variable wird dazu jede unäre Variante überprüft, für binäre Invarianten müssen alle Variablenkombinationen untersucht werden.

Die überprüften Invarianten sind hierbei:

- Unäre Invarianten, z.B. ob die Variable konstant ist oder ob sie nur wenige Werte annimmt, etc.
- Unäre Invarianten über numerische Variablen, z.B. Aussagen über ihr Werteintervall, ob sie ungleich Null ist, Modulos, etc.
- Binäre Invarianten, wie z.B. lineare Abhängigkeiten, Ordnungsrelationen, et.c
- Ternäre Invarianten, wie z.B. lineare Abhängigkeiten, Funktionen, etc.
- Invarianten über Sequenzen allgemein, wie deren minimale und maximale Werte, die Ordnung ihrer Elemente, Invarianten die für alle ihre Elemente gelten, etc.
- Invarianten über zwei Sequenzen, wie z.B. Relationen zw. Ihren Elementen, etc.
- Invarianten über eine Sequenz und eine Variable, wie z.B. Enthaltensein

Sobald ein einer Invariante widersprechender Meßwert gefunden wurde, wird diese nicht weiter überprüft. Wenn genügend Meßwerte gefunden worden sind, die die Invariante unterstützen und die Wahrscheinlichkeit für eine zufällige Meßfolge klein genug ist, wird die Invariante ausgegeben.

Das System wird desweiteren von einem statischen Typanalysesystem unterstützt, das die Vergleichbarkeit von Variablen aus dem Programmtext extrahiert. Mit dessen Hilfe kann die Anzahl der vorzunehmenden Überprüfungen von Invarianten zwischen Variablen stark eingeschränkt werden. Dies führt gleichzeitig zu einer Geschwindigkeitsverbesserung und zu einer erhöhten Genauigkeit bei der Invariantenentdeckung.

5 Statische Extraktoren für Aufruf-Graphen

Ein Aufruf-Graph (*call graph*) ist eine Relation über bestimmte Programmeinheiten, z.B. über Dateien, Module oder Prozeduren. Eigentlich sollte der Aufruf-Graph genau alle Aufrufe von einer Einheit zur anderen in jeder möglichen Ausführung des Programms enthalten. Leider ist dieses Problem unentscheidbar, d.h. man muß sich mit einer Annäherung an diese Menge begnügen. Die berechnete Menge Extraktoren von Auruf-Graphen bewegt sich also zwischen der exakt definierten (untere Schranke) und dem Kreuzprodukt der Menge der untersuchten Einheiten mit sich selbst (obere Schranke).

Während Übersetzer eine konservative Annäherung an diese Menge für die Untersuchung von möglichen Optimierungen benötigen (keine Aufruf darf ausgelassen werden), können

Aufruf-Graphen, die zum besseren Programmverständnis generiert wurden, etwas ungenauer sein.
Programme, die solche Graphen extrahieren, können auf unterschiedliche Art gebaut werden. Als erstes unterscheiden sie sich in der Eingabe: Die Einen arbeiten direkt auf dem Quelltext, die Anderen ziehen ihre Informationen aus dem mit Debug-Informationen angereicherten Programm.
Einen wichtigen Einfluß auf das Ergebnis hat auch die Art wie Makros behandelt werden. Während manche Extraktoren auf den bereits durch den Präprozessor behandelten Quellcode arbeiten, benutzen andere den Originalquelltext. Soll der erzeugte Aufruf-Graph für das bessere Programmverständnis herangezogen werden, sollte er natürlich die Makros in ihrer ursprünglichen Form belassen. Auch müssen Aufrufe über Funktionspointer berücksichtigt werden.

Die vorliegende Studie von Murphy, Notkin und Lan [MNL96] untersucht fünf Extraktoren für Aufruf-Graphen. Diese mußten jeweils drei größere Programme bearbeiten und ihre Ergebnisse wurden dann paarweise verglichen hinsichtlich der Anteile an Aufrufen, die das jeweils andere Tool nicht fand und hinsichtlich der Überdeckung der gefunden Aufrufmengen.
Über die Arbeitsweise der verwendeten Algorithmen ist laut der Autoren leider nichts bekannt, auch sei in der Literatur nicht mehr zu erfahren.
Interessant war hierbei, das keine zwei Tools die gleiche Aufrufmenge extrahierten, teilweise waren die Mengen sogar disjunkt. Auch konnte kein eindeutiger Sieger ausgemacht werden: Die Reihenfolge variierte mit dem zu untersuchenden Programmen.

6 Literatur

[KL88] B. Korel. J. Laski. Dynamic Program Slicing. Information Processing Letters. Vol. 29. No. 10 (1988), 155-163.

[HRB90] S. Horwitz, T. Reps, D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1). January 1990, 26-60.

[NKI98] A. Nishimatsu, S. Kusumoto, K. Inoue. An Experimental Evaluation of Program Slicing on Fault Localization Process. *Technical Report of IEICE Japan*. SS98-3. 1998, 17-24.

[W79] M. Weiser. Program slices: formal, psychological and practical investigations of an automatic program abstraction method. *Ph.D. thesis, University of Michigan, Ann Arbor*. 1979

[W81] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (ICSE ’81). IEEE Press, Piscataway, NJ, 439-449.

[W82] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM*. Vol.25. July 1982. pp. 446-452

[W84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352-357, July 1984

[T97] F.Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages* Vol.3, No.3, pp.121-189, September, 1995

[LRZ93] W. Landi, B. Ryder, S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *In SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56--67, June 1993. SIGPLAN Notices 28(6).

[NR00] M. Nanda and S. Ramesh. Slicing concurrent programs. In *Papers of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2000* (Portland, Oregon, August 2000). ACM Press, New York, NY, 180-190.

[ECGN00] Ernst, Czeisler, Griswold, Notkin. Quickly Detecting Relevant Program Invariants. *In Proceedings of the 22nd International Conference on Software Engineering* (ICSE '00, May). IEEE Press, Piscataway, NJ, 449-458.

[ECGN99] Ernst, Cockrell, Griswold, Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE '99, May). IEEE Press, Piscataway, NJ, 213-224.

[NJKI99] Nishimatsu, Jihira, Kusumoto, Inoue. Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE '99, May). IEEE Press, Piscataway, NJ, 422-431

[SHR99] Sinha, Harrold, Rothermel. System-Dependance-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE '99, May). IEEE Press, Piscataway, NJ, 432–441.

[HC98] Harrold, Ci. Reuse-Driven Interprocedural Slicing. In *Proceedings of the 20th International Conference on Software Engineering* (ICSE '98, Kyoto, Japan, Apr.). IEEE Press, Piscataway, NJ, 74–83.

[YRLS97] Yur, Ryder, Landi. Incremental Analysis of Side Effects for C Software Systems. In *Proceedings of the 19th International Conference on Software Engineering* (ICSE '97). IEEE Computer Society Press, Los Alamitos, CA, 422-432

[TAFM97] Tonella, Antoniol, Fiutem, Merlo. Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing. In *Proceedings of the 19th*

International Conference on Software Engineering (ICSE ‘97). IEEE Computer Society Press, Los Alamitos, CA, 433-443.

[MNL96] Murphy, Notkin, Lan. An Empirical Study of Static Call Graph Extractors. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE ’96, Berlin, Germany, Mar. 25–29), H. D. Rombach, Chair. IEEE Computer Society Press, Los Alamitos, CA, 90-99.

[LH96] Larsen, Harrold. Slicing Object-Oriented Software. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE ’96, Berlin, Germany, Mar. 25–29), H. D. Rombach, Chair. IEEE Computer Society Press, Los Alamitos, CA, 495-505.

Software Zuverlässigkeit

Andreas Hoffmann - uds6@rz.uni-karlsruhe.de

1 Definition der Zuverlässigkeit

Bei der Untersuchung der Zuverlässigkeit eines Systems wird das System als BlackBox betrachtet. Es spielt also keine Rolle, wie das Programm intern aufgebaut ist, lediglich das nach außen sichtbare Verhalten ist interessant.
In der Praxis ist man im allgemeinen bestrebt, zuverlässige Produkte zu entwickeln. Zuverlässigkeit ist ein wichtiges Qualitätsmerkmal geworden, insbesondere für komplexe Produkte wie Software-Systeme. Um Fragen, zum Beispiel nach dem richtigen Auslieferungs-Zeitpunkt, beantworten zu können, benötigt man mathematische Modelle, welche quantitative Aussagen über die Zuverlässigkeit ermöglichen.

1.1 Hardware-Zuverlässigkeit

Die Untersuchung der Zuverlässigkeit von Systemen hat eine lange Tradition. Dabei ging es vorerst nur um die Zuverlässigkeit von Hardware Systemen, wobei unter Hardware hier nicht Computer zu verstehen sind, sondern auch nicht-elektronische Komponenten wie Schrauben. Dabei ging man davon aus, dass das System bei Inbetriebnahme fehlerfrei ist. Fehler beim Design oder bei der Herstellung wurden nicht in Betracht gezogen.
Hardware unterliegt dem physikalischen und chemischen Verschleiß. Bei andauernder Benutzung wird irgendwann der Ausfall des Systems auftreten. Bedienfehler und Wartungsfehler werden nicht berücksichtigt.

1.2 Software-Zuverlässigkeit

Wir gehen im folgenden von einem mehrstufigen Software-Prozess aus. Zuerst erfolgt die Spezifikation, die als fehlerfrei angesehen wird. Danach wird die Spezifikation von Programmierern umgesetzt. An dieser Stelle werden Fehler in das Software-Produkt eingebracht. Zum Schluß wird ein iterativer Test-And-Debug-Prozess aufgenommen, in welchem versucht wird, möglichst viele Fehler aus dem Programm zu entfernen.

Bei der Software-Zuverlässigkeit können Grundannahmen der Hardware-Zuverlässigkeit nicht verwendet werden. Ein Software-System unterliegt keinem physikalischen Verschleiß, lediglich die Hardware, auf der die Software läuft. Zu einem Ausfall des Systems können nur logische Fehler führen, die schon von Anfang an in der Software enthalten waren. Dies führt zu folgender Definition:

Software-Zuverlässigkeit ist die Fähigkeit eines Software-Systems, für eine gegebene Zeit korrekt zu arbeiten.

1.3 Messung der Zuverlässigkeit

Nun stellt sich die Frage, wie diese Fähigkeit quantitativ messbar ist. Natürlich wäre es schön zu wissen, wie viele Fehler im Programm zu einem bestimmten Zeitpunkt t enthalten sind. Diese Größe ist allerdings nicht messbar, da Programme (immer noch) von Menschen entwickelt werden und man Fehler leider nicht ausschliessen kann. Könnte man die Anzahl der Fehler messen, so müßte man ein Programm nur so lange testen, bis alle Fehler gefunden worden sind. Das Programm wäre nach Behebung dieser Fehler fehlerfrei.
Messbar ist allerdings ein Programmfehler, zum Beispiel wenn das Programm abstürzt oder falsche Werte ausgibt. Demzufolge kann man die Fehlerrate, also die Anzahl der Fehler pro Zeiteinheit, messen. Andere Messungen beziehen sich auf den Zeitabstand bis zum nächsten Fehler. Die Größen "Time Between Failure" oder "Mean Time To Failure" sind lediglich zwei Beispiele dafür.

Die vorgestellten Meßgrößen enthalten alle eine zeitliche Komponente. Es können verschiedene Zeiteinheiten benutzt werden. Zum Beispiel:
Execution Time
Kalender-Zeit
Anzahl der Transaktionen.

Bei der Execution Time kann nochmals zwischen der Programm-Zeit, also die Zeit, in der das Programm wirklich Instruktionen ausführt, und Betriebs-Zeit unterschieden werden. Die Betriebs-Zeit ist die Zeit, in der das Programm auf einem Rechner läuft, aber unter Umständen keine Programm-Instruktionen ausführt (idle-Zustand).

Als Schlussfolgerung ergibt sich: Je schneller die Computer werden, also je mehr Instruktionen sie pro Zeiteinheit ausführen können, desto schlechter wird die Zuverlässigkeit der Programme.

Aus den obigen Überlegungen resultieren jetzt zwei Kern-Fragestellungen bezüglich der Zuverlässigkeit, auf die im folgenden versucht wird, eine Antwort zu geben.

Wie viele Fehler sind zu einem bestimmten Zeitpunkt im Programm enthalten?
Wann wird der nächste Fehler auftreten?

1.4 Programm-Profile

Ob ein Fehler auftreten wird, hängt natürlich stark von der Benutzung des Programms ab. Es kann also passieren, dass noch sehr viele Fehler im Programm enthalten sind, diese aber aufgrund des Verhaltens der Benutzer so gut wie nie auftreten. Zur Analyse eines Programms erstellt man deshalb sogenannte Programm-Profile. Man unterscheidet zwischen dem operationalen und dem funktionalen Profil.

Das operationale Profil gibt die relative Häufigkeit bestimmter Eingabewerte an. Ein Programm besitzt einen Eingaberaum. Der Eingaberaum ist die Menge aller möglichen Eingabekombinationen eines Benutzers. Jedem Element dieses Raumes kann dann eine relative Benutzungs-Häufigkeit `h` zugeordnet werden. Die Summe der relativen Häufigkeiten über alle Elemente muss zusammen natürlich Eins ergeben.

Beispiel:
Gegeben sei ein Programm mit drei booleschen Eingabevariablen. Der Eingaberaum `I` ist demnach die Menge $I = \{true, false\}^3$

Jedem Tupel aus dem Eingaberaum wird nach einer bestimmten Anzahl von Programmdurchläufen seine Auftrittshäufigkeit h_r zugeordnet:

$$h_r(X = (false, false, false) = 0.2$$

Es kann auch passieren, dass einige Eingabe-Kombinationen gar nicht auftreten. Diese bekommen dann die Häufigkeit Null zugeordnet. Wenn der Eingaberaum sehr groß ist, verwendet man anstelle von Tupeln Wertebereiche.

Bei der Definition des funktionalen Profils wird die Sichtweise des Programms als BlackBox verworfen. Stattdessen modelliert man das Programm als Menge von Funktionen. Das funktionale Profil gibt dann die relative Aufrufhäufigkeit dieser Funktionen bei der Ausführung des Programms an. Normalerweise hängt das funktionale Profil vom operationalen Profil ab. Deshalb verzichtet man in der Praxis häufig auf die Erstellung funktionaler Profile.

2 Modelle

Im folgenden werden die fünf wichtigsten Modelle zur Vorhersage der Zuverlässigkeit eines Software-Systems vorgestellt. Wie im vorigen Abschnitt erwähnt, stehen dabei zwei Fragestellungen im Mittelpunkt:
die Abschätzung der Anzahl der Fehler
die Vorhersage der Fehlerrate.

Die folgende Tabelle gibt einen Überblick über die betrachteten Modelle:

Modell	**Abschätzung (Fehleranzahl)**	**Vorhersage (Fehlerrate)**
Function-Point-Methode	X	
Exponentielles Modell	X	X
Littlewoodsches Modell	X	X
Logarithmisches Modell		X
Hypergeometrisches Modell	X	X

2.1 Die Function-Point-Methode

Das einfachste Verfahren, um die Anzahl der Fehler in einem Programm zu schätzen, ist der Vergleich mit historischen Daten ähnlich großer Software-Systeme. Dabei ermittelt man zunächst die Anzahl der Function Points und bestimmt dann mit Hilfe von Tabellen die Anzahl der Fehler, die nach der Auslieferung entdeckt werden.

Als Tabelle bieten sich entweder hausinterne Erfahrungswerte oder Branchenstatistiken an. Die folgende Tabelle zeigt die Branchentabelle für das Jahr 1995.

Branche	**Durchschnittliche Fehleranzahl pro Function Point**
System Software	0.4
Commercial Software	0.5
Information Software	1.2
Military Software	0.65

Diese Zahlen sind für eine genaue Schätzung natürlich viel zu grob und können nur Tendenzen ausdrücken. Auch hausinterne Werte liefern meist keine bessere Schätzung, da die Anzahl der Fehler nicht nur von der Komplexität des Programms abhängt, sondern auch von der Qualität des Test-And-Debug-Prozesses.

Um den Test-And-Debug-Prozess besser modellieren zu können, greift man auf feinere stochastische Modelle zurück, die im folgenden vorgestellt werden.

2.2 Definitionen

Bisher wurde nur allgemein von Fehlern im Programm gesprochen. Wir haben bereits kennengelernt, dass das Auftreten eines Fehlers vom Profil der Benutzung abhängt. Doch was genau ist ein Fehler?

Beim Umsetzen der Spezifikation macht der Programmierer logische Fehler, die vom Ziel der Spezifikation abweichen. Diese Fehler werden **Faults** oder auch **Bugs** genannt und sind die potentielle Quelle eines Ausfalls. Ein Ausfall oder **Failure**, den der Benutzer

oder Tester beobachten kann, ist das fehlerhafte Verhalten eines Programms, also eine Abweichung zwischen dem beobachteten und dem spezifizierten Verhalten. Ein **Defect** ist ein allgemeiner Produktfehler, der nicht unbedingt zum Ausfall des Programms führen muss. Hierunter werden auch Architekturfehler oder Dokumentationsfehler zusammengefaßt.

Der Tester eines Programms interessiert sich demnach für die Ausfälle, die in einem Programm auftreten können. Wie die Ursache eines Ausfalls zu finden ist, wird hier nicht weiter betrachtet.

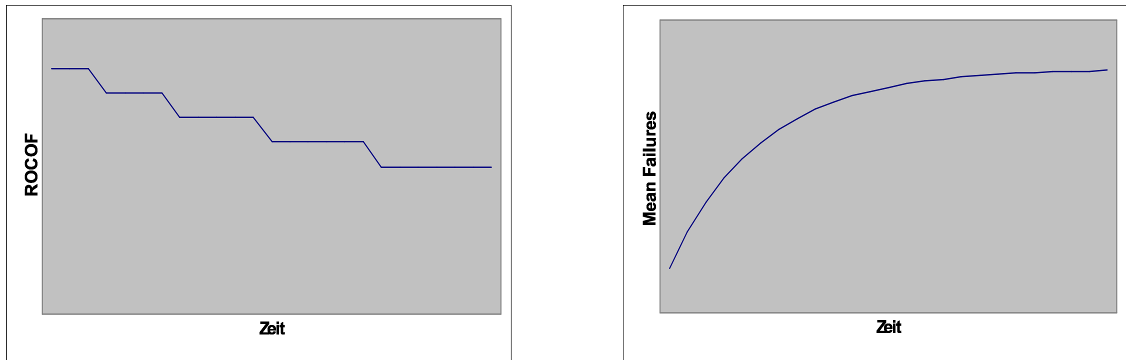
Aus dieser Ausfall-Definition lassen sich verschiedene andere Maße ableiten. Die Rate Of Occurence Of Failures (**ROCOF**), also die Ausfallrate, ist die erste Ableitung der Ausfall-Funktion, welche die Anzahl der Ausfälle über die Zeit beschreibt, sofern diese differenzierbar ist.

Mean Failures ist die durchschnittliche kumulierte Anzahl der Ausfälle. Hier wird lediglich der Durchschnitt betrachtet, da das Auftreten von Fehlern vom operationalen Profil eines Programms abhängt. Testen zum Beispiel zehn Benutzer das Programm eine ganze Woche lang, so wird die Anzahl der entdeckten Fehler pro Benutzer unterschiedlich sein.

Die Mean Time To Failure (**MTTF**) beschreibt die durchschnittliche Zeit, die zwischen zwei Ausfällen vergeht. Damit kann insbesondere die erwartete Zeit bis zum nächsten Ausfall vorhergesagt werden. Dieses Kriterium wird meist im Projektmanagement zur Festlegung des Auslieferungs-Zeitpunktes verwendet. Für den Benutzer ist es nämlich unerheblich, wie viele Fehler er entdeckt. Ihn interessiert vielmehr, in welchen Abständen Ausfälle bei ihm auftreten.

2.3 Das Exponentielle Modell

Das exponentielle Modell ist das intuitivste Modell. Es orientiert sich noch stark an der Hardware-Zuverlässigkeit. Dabei wird angenommen, dass das Programm zu Beginn des Test-And-Debug-Prozesses N Fehler besitzt. Tritt ein Fehler während des Testens auf, so wird die Ursache des Fehlers gesucht und der Fehler perfekt behoben. Man spricht von einer perfekten Fehlerbehebung, wenn durch das Debuggen keine neuen Fehler in das Programm eingebracht werden. Ist der Fehler beseitigt, wird mit dem Testen fortgefahren. Dabei wird angenommen, dass jeder Fehler einen konstanten Beitrag zur Ausfallrate leistet. Wird ein Fehler beseitigt, nimmt demnach die Ausfallrate um eine Konstante Φ ab. Dies impliziert die Annahme, dass alle Fehler gleichwahrscheinlich und unabhängig voneinander sind.



a) die ROCOF-Kurve b) die Mean-Failures-Kurve

In Abbildung a) sieht man sehr deutlich, dass die Fehlerrate immer um einen konstanten Betrag reduziert wird. Die Zeitabstände zwischen dem Auftreten der Fehler sind nicht konstant. Sie hängen von den Fähigkeiten der Tester ab.

Betrachtet man als nächstes Abbildung b), so erkennt man, warum das exponentielle Modell seinen Namen trägt. Durch die konstante Abnahme der Fehlerrate ergibt sich ein exponentieller Verlauf der durchschnittlichen Ausfall-Anzahl. Dabei steigt die Kurve asymptotisch bis zur oberen Fehlerschranke `N` an.

Formel: $E(t) = N(1 - e^{-\Phi t})$

`E(t)` gibt die durchschnittliche Anzahl der Fehler bis zum Zeitpunkt `t` an. Die Parameter `N` und Φ werden aufgrund anfänglicher Messungen geschätzt.

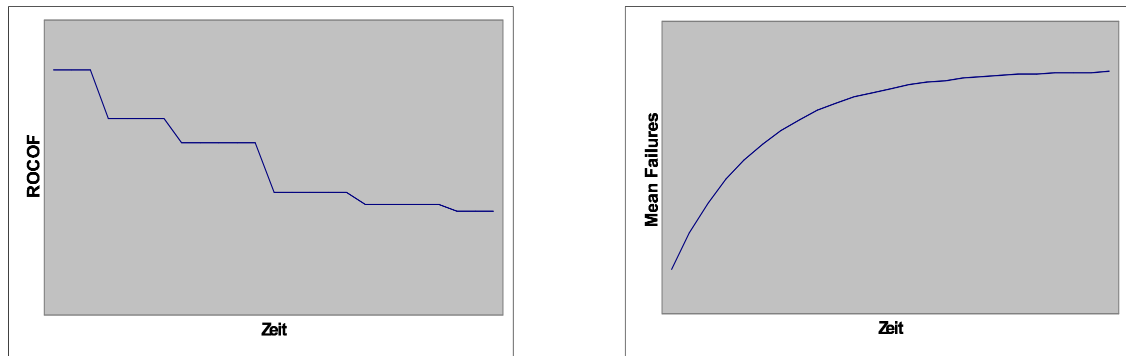
Die Kritik am Modell bezieht sich auf die Grundannahmen. Meist hängen Fehler voneinander ab. Vom operationalen Profil eines Programms folgt, dass das Auftreten von Fehlern nicht gleichwahrscheinlich ist, sondern abhängig vom Test-Prozess. Desweiteren ist eine Fehlerbehebung erfahrungsgemäß niemals perfekt.

Das Modell besticht durch seine Einfachheit und wurde deshalb des öfteren in der Praxis auch eingesetzt. Allerdings sind die Prognosen meist zu optimistisch, da gerade am Anfang mehr Fehler entdeckt werden. Die daraus resultierende Schätzung der Parameter `N` und Φ ist demnach zu ungenau.

2.4 Das Littlewoodsche Modell

Auch hier setzt man voraus, dass `N` Fehler im Programm enthalten sind. Tritt ein Fehler auf, so wird dieser Fehler erst behoben und danach wird mit dem Testen fortgefahren. Es wird eine perfekte Fehlerbehebung angenommen.
Im Gegensatz zum Exponentiellen Modell hat in diesem Modell jede Fehlerbehebung unterschiedlich starken Einfluss auf die Ausfallrate. Diese Tatsache wird mit Hilfe einer Gamma-verteilten Zufallsvariable modelliert.

Gamma-Verteilungen besitzen zwei Parameter α und ν, von denen der erste den Maßstab angibt ("Skalenparameter") und der zweite die Struktur ("Formparameter"). Für ν=1 ergibt sich die Exponential-Verteilung.



a) die ROCOF-Kurve b) die Mean-Failure-Kurve

In Abbildung a) sieht man sehr deutlich, dass nach jeder Fehlerbehebung die Ausfallrate unterschiedlich stark sinkt. Je nach Wahl der Parameter der Gamma-Verteilung wird die Reduktion $\hat{\Phi}$ der Ausfallrate unterschiedlich ausfallen.

Für die Darstellung der Mean-Value-Function in Abbildung b) wird der Erwartungswert der Gamma-Verteilung bestimmt. Es ergibt sich eine Kurve, die fast identisch mit der Kurve im Exponentiellen Modell ist.

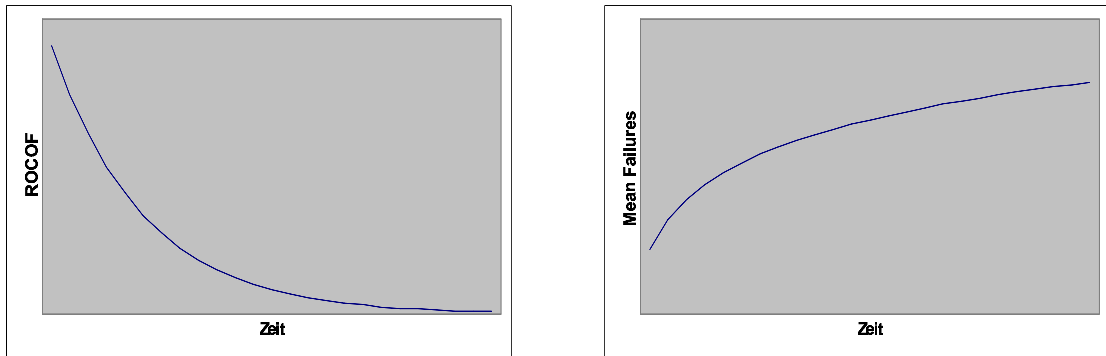
Formel: $$E(t)=N(1-e^{-\hat{\Phi}t})$$

Das Problem dieses Modells ist die vergleichsweise große Anzahl der zu schätzenden Parameter. Zum einem muss die initiale Anzahl N der Fehler geschätzt werden, zum anderen müssen die beiden Parameter α und ν für die Gamma-Verteilung bestimmt werden. Außerdem ist es nicht realistisch, dass das Testen und Debuggen abwechselnd erfolgt. Aus betriebswirtschaftlichen Gründen wird man versuchen, gleichzeitig zu testen und zu debuggen.

2.5 Das Logarithmische Modell

Das Logarithmische Modell beschreibt eine völlig andere Sichtweise als die bisher vorgestellten Modelle. Hier wird nicht von einer bestimmten Anzahl von Fehlern im Programm ausgegangen, sondern lediglich eine bestimmte Ausfallrate λ vorausgesetzt. Durch jede Fehlerbehebung können gefundene Fehler behoben aber auch neue Fehler in das Programm eingebracht werden. Durch diese Art der Fehlerbehebung ändert sich die Ausfallrate des Programms um einen bestimmten Betrag θ. Es wird jedoch angenommen, dass der Effekt des Debuggens in der Summe positiv ist; die Ausfallrate des Programms sinkt also stetig.

Testen und Debuggen verlaufen parallel, deshalb sind die Kurven in den Abbildungen auch kontinuierlich gezeichnet.



a) die ROCOF-Kurve b) die Mean-Failure-Kurve

In Abbildung a) sieht man, wie die Ausfallrate exponentiell abnimmt. Dies liegt an der relativen Änderung der Ausfallrate um dem Parameter θ. Im Unterschied zu den beiden bereits vorgestellten Modellen kann hier die Ausfallrate niemals ganz auf Null sinken, da sich die Kurve nur asymptotisch der x-Achse nähert. D.h. ein Programm kann niemals zu 100% fehlerfrei werden, was auch der intuitiven Vorstellung jedes Programmierers entspricht.

In Abbildung b) bemerkt man eine logarithmische Ausprägung der kumulierten Ausfall-Anzahl. Es ist deutlich zu sehen, dass die Kurve nicht nach oben beschränkt wie bei den beiden anderen Modellen. Dort war die Anzahl `N` der Fehler im Programm eine obere Schranke für die kumulierten Ausfall-Anzahl.

Formel: $$E(t) = \frac{1}{\theta} \ln(\lambda \theta t + 1)$$

Als Kritik könnte man anführen, dass man mit diesem Modell nicht die Anzahl der Fehler im Programm schätzen kann. Dies ist aber kein echter Kritikpunkt, da das Modell die zukünftige Ausfallrate vorhersagen kann. Dem Endbenutzer kann es egal sein, wie viele Fehler im Programm sind, er registriert lediglich den Effekt der Ausfallrate.

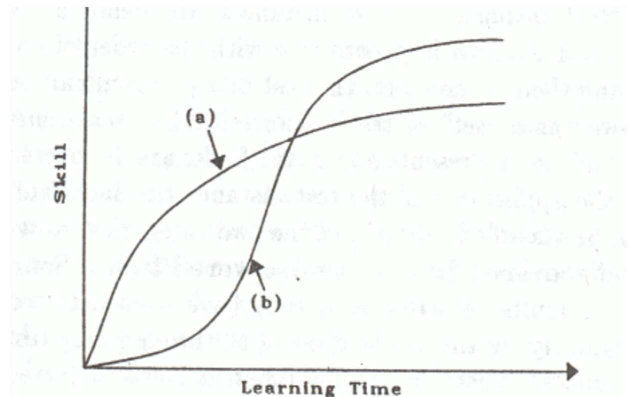
2.6 Das Hypergeometrische Modell

Zu Beginn des Test-And-Debug-Prozesses enthält das Programm `N` Fehler. Um die Realität genauer zu modellieren, geht man hier von sog. Test-Instanzen aus. In jeder Test-Instanz `i` werden `w(i)` Fehler entdeckt. Dabei setzt sich `w(i)` aus neu entdeckten und bereits gefundenen Fehlern zusammen. Dies liegt daran, dass Testen und Debuggen parallel verlaufen und Fehler meistens nicht sofort behoben werden können. Auch hier wird eine perfekte Fehlerbehebung angenommen.

Dieses Modell kann man auch mit dem Urnenmodell vergleichen. Eine Urne enthält vorerst nur `N` weiße Kugeln. Bei der Ziehung `i` werden `w(i)` Kugeln entnommen. Alle weißen Kugeln werden nun rot gefärbt und wieder zusammen mit den roten Kugeln in die Urne zurückgelegt. Das Testen kann dann als Ziehung der roten und weißen Kugeln angesehen werden. Nur die weißen Kugeln entsprechen neu entdeckten Fehlern.

Das Besondere an diesem Modell ist die Schätzung der `w(i)`. Hier wird nicht versucht aufgrund anfänglicher Beobachtungen `w(i)` zu bestimmen, sondern mit Hilfe sog. Lernkurven den Testverlauf zu modellieren. Grundannahme ist dabei, dass die Tester einen Lernprozess durchlaufen und Fehler immer schneller entdecken werden.

Beispiele verschiedener Lernkurven:



a) exponentiell

$$w_i = Np_{LT}(1-e^{-ai})$$

b) logarithmisch

$$w_i = Np_{LT}\frac{1}{1+be^{-ai}}$$

Der Parameter p_{LT} beschränkt den Lernerfolg auf ein Maximum. Damit kann in jeder Testinstanz nur maximal ein Bruchteil aller Fehler `N` gefunden werden.

Es gibt auch Lernkurven, die gänzlich ohne die Parameter `N` und `p` auskommen. Man nimmt einfach an, dass der Testerfolg, also die Anzahl der gefundenen Fehler, lediglich von der Anzahl der Tester `u(i)` abhängt.
Beispiel mit einer linearen Lernkurve: $w_i = u_i(ai+b)$

Für die Abschätzung der Mean-Failures ergibt sich dann folgende Formel:

$$E(i)=N\left[1-\prod_{j=1}^{i}(1-\frac{w_j}{N})\right]$$

Die Mean-Failures-Kurve hängt also von der gewählten Lernkurve ab.

Als Kritik kann man anführen, dass die Schätzung der Parameter bei diesem Modell ungewöhnlich schwierig ist.

2.7 Schätzung der Parameter

Bei allen Modellen müssen mindestens zwei Parameter geschätzt werden. Dies erfolgt meistens aufgrund anfänglicher Meßwerte für das Auftreten von Ausfällen. Aus der Stochastik werden zwei Verfahren zum zuverlässigen Schätzen verwendet.

a) Der Maximum-Likelihood-Schätzer

Der Maximum-Likelihood-Schätzer maximiert die sog. Likelihood-Funktion. Es seinen R Beobachtungen $(t_1, ..., t_R)$ gegeben, dann ist

$$L(N \mid t_1, t_2, \quad t_R) = \prod_{i=1}^{R} f(N \mid t_1, \quad t_i)$$

die Likelihood-Funktion der Variable N. Anstatt eine feste Wahrscheinlichkeitsverteilung für verschiedene Beobachtungen zu betrachten, halten wir jetzt die Beobachtungen fest und untersuchen die Wahrscheinlichkeit des Auftretens von $(t_1, ..., t_R)$ unter verschiedenen durch die Variable N parametrisierten Modellen.
Die Funktion f(N) stellt die Wahrscheinlichkeits-Verteilungs-Funktion dar. Sie hängt dabei von allen vorhergehenden Beobachtungen ab. Das Produkt dieser Wahrscheinlichkeiten ergibt die Wahrscheinlichkeit für diesen Pfad; also die Wahrscheinlichkeit, dass die Beobachtungen in dieser Reihe auftraten. Nun wird versucht, diese Wahrscheinlichkeit zu maximieren.

Nimmt man beispielsweise das exponentielle Modell, so ergibt sich die Liklihood-Funktion:

$$L(N \mid t_1, t_2, \quad t_R) = \prod_{i=1}^{R} N e^{-N t_i}$$

Da beim exponentiellen Modell die Fehler laut Voraussetzungen unabhängig voneinander sind, hängt die Wahrscheinlichkeits-Verteilungs-Funktion nur vom Zeitpunkt der Messung t_i ab. Ableiten und Nullsetzen liefert dann das gewünschte Ergebnis für den Parameter N:

$$N = \sum_{i=1}^{R} t_i$$

b) Der Least-Squares-Schätzer

Der Least-Square-Schätzer bestimmt den minimalen euklidschen Abstand der vorhergesagten Werte zu den tatsächlichen Werten in Abhängigkeit des zu bestimmenden Parameters. Die Square-Funktion ist definiert als Summe der euklidschen Abstände:

$$S(N \mid t_1, t_2, \quad t_R) = \sum_{i=1}^{R} \left[t_i - f(N \mid t_1, \quad t_i) \right]^2$$

Für das exponentielle Modell ergibt sich folgende Square-Funktion:

$$S(N \mid t_1, t_2, \quad t_R) = \sum_{i=1}^{R} \left[t_i - N e^{-N t_i} \right]^2$$

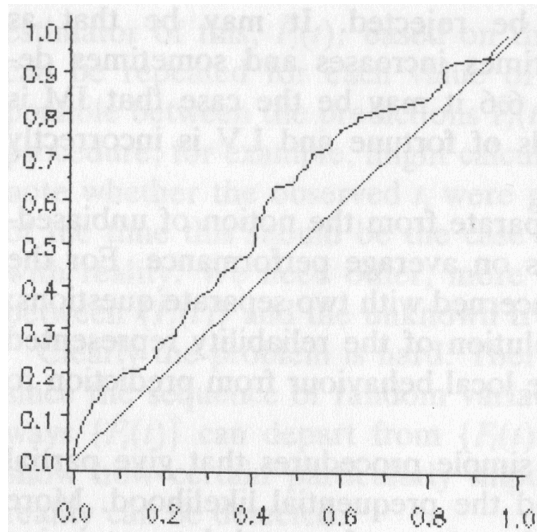
Die Auflösung dieses nichtlinearen Gleichungssystems bedarf weiterer mathematischer Schritte, die hier nicht erläutert werden können.

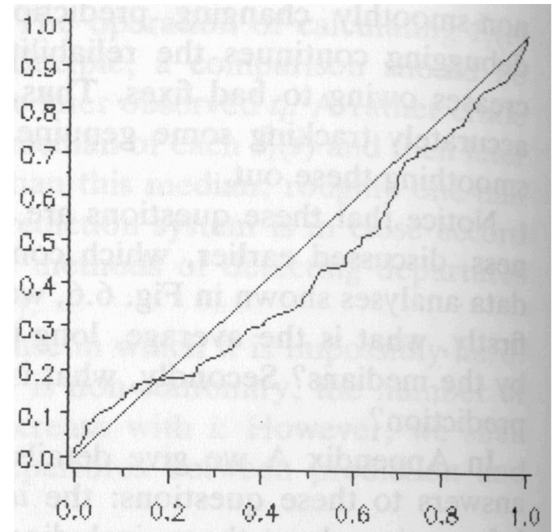
3 Bewertung

Welches Modell ist nun das beste?
Um die Modelle miteinander zu vergleichen, versucht man mit Hilfe von Simulationen realistische Zeitreihen aufzustellen. Danach kann man die Vorhersage-Qualität der Modelle mit dem sog. u-Plot vergleichen. Dabei wird ein Verfahren aus der Statistik zur Normierung der Vorhersagen der einzelnen Modelle angewendet.

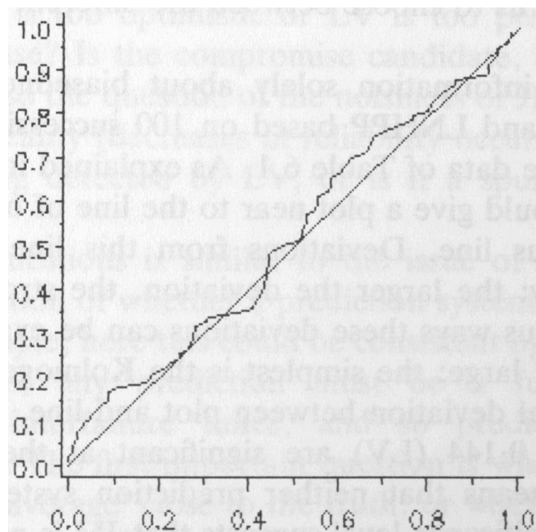
3.1 Vergleich der Modelle

Exemplarisch sollen hier drei Modelle miteinander verglichen werden.





a) Exponentielles Modell b) Littlewoodsches Modell



c) Logarithmisches Modell

Das exponentielle Modell ist, wie erwartet, bei dieser Zeitreihe zu optimistisch. Das Littlewoodsche Modell sieht zwar am Anfang recht gut aus, ist aber im gesamten Verlauf gesehen zu pessimistisch. Das logarithmische Modell paßt für diese Zeitreihe am besten.

3.2 Fazit

Es gibt sehr viele Modelle zur Vorhersage der Software-Zuverlässigkeit. In der Praxis kommen aber nur einige wenige Modelle zum Einsatz, da die meisten Modelle zwar stochastisch korrekt aber in der praktischen Anwendung zu kompliziert sind. Außerdem liefern die Modelle schlechte oder gar widersprüchliche Aussagen. Ein weiteres Problem ist die mitunter schwierige Schätzung der Parameter.

Das meistbenutzte Modell ist eine Variante des logarithmischen Modells. Es wird vornehmlich im militärischen Bereich und bei der Entwicklung von System-Software verwendet. Als Folgerung verbleibt:

Der Einsatz von Software-Zuverlässigkeitsmodellen lohnt sich nur bei sehr großen Software-Projekten oder dort, wo Unzuverlässigkeit Leben kostet.

4 Literatur

[1] John D. Musa, Anthony Iannino, Kazuhira Okumoto – *Software Reliability*. McGraw-Hill, 1990.

[2] Paul Rook – *Software Reliability Handbook*. Elsevier Science, 1990.

[3] Yashwant Malaiya, Jason Denton – *What Do the Software Reliability Growth Model Parameters Represent*. Technical Report Colorado State University CS-97-115, 1997.

[4] Rong-Huei Hou, Sy-Yen Kuo, Yi-Ping Chang – *Applying Various Learning Curves to Hyper-Geometric Distribution Software Reliabilitity Growth Model*. Proceedings 5. International Symposium on Software Reliabilitiy Engineering, 1994.

[5] Timm Gramms – *Reliability Growth Models Criticized.* http://www.fh-fulda.de/~fd9006/RGM/RGMcriticized.html, 1999.

[6] Michael Lyu - *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

Testen objektorientierter Software

Ulf Krum - ulf@oocs.de

1 Einleitung

Die objektorientierte Softwareentwicklung bringt nicht nur die bekannten Vorteile, wie bessere Wiederverwendbarkeit, sondern birgt auch neue Probleme und Fehlerquellen in sich (Abschnitt 2). Diese Probleme finden sich in allen Phasen der Softwareentwicklung und treten insbesondere bei verschiedenen Qualitätssicherungsverfahren auf.
Daneben bieten die Methoden und Modelle der objektorientierten Softwareentwicklung neue Möglichkeiten für Qualitätssicherungsverfahren, z.B. bei der automatischen Testfallgenerierung.
In dieser Arbeit werden einige Techniken zur Qualitätssicherung unter dem Aspekt der objektorientierten Softwareentwicklung untersucht.
Es war ursprünglich vorgesehen die Probleme und Fehler zu analysieren, die speziell bei der objektorientierten Softwareentwicklung auftreten. Um dann die Methoden, die zur Vermeidung oder Identifizierung dieser Probleme entwickelt wurden, zu ordnen, miteinander vergleichen und bewerten zu können.
Dies gestaltet sich aus verschiedenen Gründen als schwierig:

- Oft werden traditionelle Methoden für die objektorientierte Softwareentwicklung angewandt, ohne neue Aspekte einzubringen.
- Die Methoden betreffen verschiedene Phasen der Softwareentwicklung.
- Die Methoden berücksichtigen verschiedene Probleme.

Stattdessen werden, um eine Übersicht zu bekommen, im folgenden Kapitel Probleme beschrieben, die durch das objektorientierte Paradigma bei der Softwareentwicklung auftreten. In Abschnitt 3 werden einzelne Arbeiten und Methoden zur Qualitätssicherung in der objektorientierten Softwareentwicklung vorgestellt und auf folgende Fragen hin beurteilt:

Frage 1: *Deckt die betrachtete Technik spezielle OO-Probleme ab?*

Frage 2: *Kann die traditionelle, evtl. modifizierte Technik eine andere Art von Problemen identifizieren oder bearbeiten?*

Abschließend werden die Ergebnisse in Abschnitt 4 zusammenfaßt.

2 Spezielle Fehlerquellen und Probleme

2.1 Delokalisierung von Code

Bei der objektorientierten Programmierung werden Teile von Algorithmen und Funktionalitäten durch dynamisches Binden realisiert.
Beispielsweise können lange Codestücke in den Zweigen einer switch-Anweisung vermieden werden, indem man das Strategie-Pattern anwendet [GHJV99]. Dabei befinden sich in den Zweigen

nur noch Anweisungen zur Objekterzeugung. Die Algorithmen sind dann in den Objekten gekapselt.

Die Methoden sind daher im Allgemeinen kleiner und einfacher als Prozeduren und Funktionen imperativer Sprachen. Betrachtet man eine Klasse, so sieht man, daß sie meistens sehr viele Methoden zur Verfügung stellt. Jede einzelne manipuliert oft nur einen kleinen Teil des Objektzustandes. Einzelne Algorithmen und Funktionen verteilen sich auf mehrere Klassen und viele Methoden. In diesem Zusammenhang wird von Delokalisierung des Codes gesprochen, da zur Lösung einer Aufgabe nicht mehr nur ein zusammenhängendes Programmstück zuständig ist.

Es ist also notwendig zum Verständnis des Codes viele Zeilen zu verstehen, die wiederum auf verschiedene Stellen über den gesamten Quellcode verteilt sind.

Genaue Kenntnis über die benutzten Klassen und Methoden sind für eine korrekte Implementierung notwendig. Dies schließt die Beziehungen zwischen den einzelnen Klassen mit ein.

Delokalisierung ist nichts Neues. [SPLLL88] haben sich schon damit beschäftigt und Code imperativer Sprachen untersucht. Sie nannten Algorithmen und Funktionen, die im Programmcode durch nicht zusammenhängende Zeilen realisiert sind *delocalised plans.*

Aber die komplexen Beziehungen zwischen den Klassen bei der objektorientierten Programmierung verschärfen das Problem erheblich.

2.2 Der Objektzustand

Während man bei der imperativen Programmierung stets bemüht ist, Seiteneffekte zu vermeiden, lebt die objektorientierte Programmierung davon. Durch Methoden werden Attribute gelesen und manipuliert. Rückgabewerte und Auswirkungen von Methoden hängen vom aktuellen Zustand des Objektes ab.

Wie bei der Delokalisierung (Abschnitt 2.1) schon erwähnt, sind die Methoden einfach von ihrer Struktur und meist nur wenige Zeilen lang. Sie bestehen aus keiner oder nur wenigen Verzweigungen. Die durchschnittliche Anzahl der Verzweigungen innerhalb einer Methode liegt nach [SP00] zwischen 0 und 3. Algorithmen und Funktionalitäten spiegeln sich in der Manipulation von Objektzuständen wider.

Daß eine einzelne Methode nicht sehr komplex ist, zeigt auch ein Report über eine Restrukturierung bei der BASF [GE00]. Diese Restrukturierung wurde im Rahmen einer Plattform-Migration des Opal-Systems durchgeführt. Dabei wurden die Anzahl der Klassen und Methoden sowohl vor als auch nach der Restrukturierung gezählt (Tabelle 1).

	vor der Migration	nach der Migration
Methoden	54159	98540
Klassen	2865	5240
Pakete	595	965
Anweisungen pro Methode	16,7	8,1

Tabelle 1: *Opal-System der BASF*

Wenn man davon ausgeht, daß der Code nach der Restrukturierung besserem Design genügt, folgt aus Tabelle 1, daß sich besseres objektorientiertes Design durch kleinere Methoden und einer größeren Zahl von Klassen und Methoden auszeichnet.

2.3 Polymorphie

Eines der wichtigsten Konzepte der objektorientierten Softwareentwicklung ist die Polymorphie. Sie wird durch sogenanntes Late-Binding oder dynamisches Binden ermöglicht. Dabei wird der tatsächlich auszuführende Code zu einem Methodenaufruf erst dynamisch zur Laufzeit bestimmt. Unterklassen können daher Attribute (und damit den Objektzustand) der Oberklasse auf unvorhergesehene Weise ändern.

2.4 Unbekannte Server- und Klientenklassen

Die objektorientierte Softwareentwicklung fördert bei gutem Design eine einfache Wiederverwendung von übersetztem Code. Dies wird durch zwei Konzepte ermöglicht:

Getrenntes Übersetzen: Klassen können getrennt voneinander übersetzt werden. Wird in einer zu übersetzenden Klasse eine fremde Klasse verwendet, reicht es aus, daß der Programmcode dieser fremden Klasse vorhanden ist.

Dynamisches Laden: Zur Laufzeit werden die Klassen eines Programmes geladen, sobald sie benötigt werden. Insbesondere können neue, übersetzte Klassen zur Laufzeit nachgeladen werden.

Klassen müssen nicht aus dem eigenen Hause stammen, daher ist eventuell der Quellcode nicht verfügbar. Oft weiß man auch heute noch nicht, wie die eine oder andere Klasse künftig eingesetzt wird. Ein entsprechendes, gezieltes Testen ist daher nur schwer oder gar nicht möglich. Damit ist auch schwer vorherzusagen, wie sich eine Klasse im Zusammenspiel mit einer anderen verhält.

3 Qualitätssicherungsverfahren

3.1 Review objektorientierten Quellcodes

In [DRW00] behandeln die Autoren das Problem der Delokalisierung bei Inspektionen objektorientierten Codes.

3.1.1 Motivation

Bei Reviews können bestimmte Fehler nur erkannt werden, wenn eine gute Kenntnis über Konzepte und Funktionen der benutzten Klassen und Methoden vorhanden ist. Um eine Zeile im Quellcode auf ihre Korrektheit hin überprüfen zu können, müssen viele weitere Zeilen verstanden werden, die über mehrere Methoden und Klassen verteilt sein können.
Will man die Verhaltensweise einer Methode m verstehen, muß man:

- die Verhaltensweise aller Methoden, die in m aufgerufen werden, verstehen und
- den Kontext von m verstehen, d.h. die Aufrufkette zurückverfolgen.

Dabei muß man Ketten von Methodenaufrufen verfolgen, indem man den Vererbungsbaum in beide Richtungen (Generalisierung und Spezialisierung) traversiert (Abbildung 1).

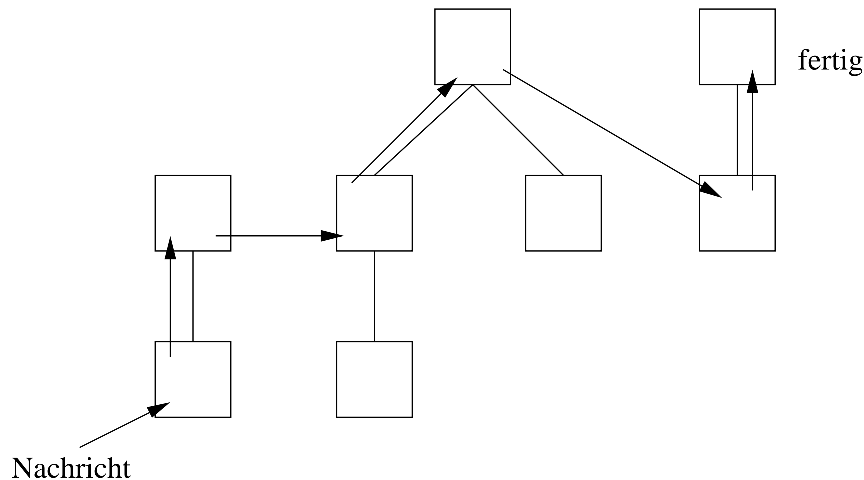


Abbildung 1: *Kette von Methodenaufrufen [DRW00]*

3.1.2 Ergebnisse

Die Autoren belegen in [DRW00], daß die Delokalisierung in objektorientiertem Code ein großes Problem bei herkömmlichen Verfahren des Code-Review ist.
Die Autoren begründen ihre Auffassung durch Literaturrecherche, einem Experiment und einer Umfrage in der Wirtschaft.
Bei dem Experiment handelte es sich um eine Code-Inspektion mit unerfahrenen Probanden (*ad-hoc approach*). Der zu inspizierende Code wurde auf Papier geliefert und enthielt kontrolliert eingebrachte Fehler verschiedener Typen. Es wurde sichergestellt, daß er sonst keine Fehler enthielt. Jeder Proband hatte die Zeit, zu der er einen bestimmten Fehler fand, zu notieren. Um beurteilen zu können, ob der Zeitpunkt des Auffindens eines Fehlers tatsächlich mit der Delokalisierung zusammenhängt, hatten nicht alle Fehler "delokalisierten" Charakter. Das Experiment zeigt, daß Delokalisierung eine Rolle bei der Identifizierung von Fehlern spielt.
Die Umfrage wurde per Email, nach dem Experiment, in mehreren Software-Unternehmen durchgeführt, um herauszufinden wie in der Industrie Defekte in OO-Code entfernt werden, wie Inspektionen auf OO-Code durchgeführt werden und wie die Fehler in ihrer Beschaffenheit sind, die den Entwicklern Probleme bereiten. Sie unterstrich die Ergebnisse des Experimentes.
Des weiteren begründen die Autoren, warum traditionelle Methoden des Reviews nicht geeignet sind. Dabei diskutieren sie folgende Methoden:

Ad-hoc Ansatz: Der Inspekteur geht nach eigenem Ermessen und Erfahrung vor. Es gibt keine Leitfäden oder sonstige Hilfsmittel zur Unterstützung.

Checkliste: Es wird eine Checkliste als Leitfaden und zur Kontrolle herangezogen.

PBR (Perspective Based Review): Jeder Inspekteur erhält eine bestimmte Rolle, die verschieden von den Rollen der anderen ist. Die vergebenen Rollen entsprechen verschiedenen Interessenten am Code (Entwickler, Tester, Kunde, etc.) und dienen dazu, den Code aus unterschiedlichen Perspektiven zu betrachten. Dadurch erhofft man sich mehr Fehler ausfindig machen zu können.

Sie weisen darauf hin, daß keine dieser Methoden den Inspekteur im Hinblick auf Delokalisierung unterstützt oder hilft, die Menge des zu inspizierenden Codes zu reduzieren. Diese Methoden

gehen aber alle davon aus, daß eine angemessen handhabbare Menge von Quelltext (z.B. 100 LOC / h) einfach isoliert werden kann, aber

> *"An inspection on 200 lines of OO code could easily swell by an order of magnitude due to inter-class dependencies." [DRW00]*

Die Autoren halten fest, daß folgende Punkte weiter erforscht werden müssen:

- Wie muß zu inspizierender Quellcode partitioniert werden? (Was ist zu lesen?)
- Wie muß Quellcode gelesen werden?
- Wie kann man ein Verständnis dafür fördern, was nicht gelesen wurde? (Lokalisiere Delokalisierung!)

3.1.3 Beurteilung

Eine Beurteilung nach Frage 1 (Spezielle OO-Probleme) und Frage 2 (Modifizierte Technik) ist hier natürlich nicht möglich.
Festzustellen ist jedoch, daß die Schlüsseleigenschaften der objektorientierten Programmierung erheblich zu einer hohen Delokalisierung beitragen. Hier besteht also noch Bedarf zur Erforschung dieses Phänomens.

3.2 Algebraisches Testen

In [HS96] stellen die Autoren ein Werkzeug vor, mit dem man Klassen objektorientierter Programme algebraisch testen kann.

3.2.1 Motivation

Die funktionale Notation algebraischer Spezifikationen kann bei objektorientierten Programmen nicht verwendet werden, da in der objektorientierten Programmierung Zustände von Objekten durch Seiteneffekte verändert werden.

3.2.2 Ergebnisse

Das Werkzeug, mit Namen Daistish, eignet sich zum systematischen, algebraischen Testen von ADT's. Dabei handelt es sich um ein Perl-Skript, welches aus einer formalen Spezifikation und dem implemetierenden Code eines ADT's einen Testtreiber generiert. Daistish wurde für Eiffel entwickelt und um ein C++ Backend erweitert.
Das Ziel der Autoren war es, eine ähnliche Notation wie sie bei der formalen Spezifikation für imperative Sprachen üblich ist, auch für die objektorientierte Programmierung verwenden zu können. Um dies zu erreichen, übergeben sie den Objektzustand den Funktionen der Spezifikation als Argument. Als Rückgabewert erhalten sie einen neuen Objektzustand.
Die folgenden Zeilen stellen ein Beispiel einer Spezifikation dar.

```
# Aliases
  -- alias   real name
  stack      stack [id]
  intToId    ()
```

```
  printBool  io.putbool
  ...
# Signatures
  -- type    name        arguments
  {stack}    create
  {stack}    push        ({stack}, id)
  {stack}    pop         ({stack})
  ...
  -- user-defined for testing
  boolean    equals      ({*}, *)
  boolean    notEquals   ({*}, *)
  boolean    bigger      ({*}, *)
  ...
# Axioms
  -- name    rule
  PopNew     equals(pop(create),create)
  PopPush    equals(pop(push(stack,element)),stack)
  TopNew     top(create) = nullId
  ...
# Vectors
  -- name    value
  id0        intToId(0)
  ...
```

Die Autoren vergleichen ihr Werkzeug mit ASTOOT [DF94]. ASTOOT generiert automatisch Testtreiber für Testfälle. Die Testfälle werden entweder von Hand geschrieben oder mit Hilfe einer Spezifikation generiert. ASTOOT verwendet dafür eine eigene Spezifikationssprache namens LOBAS. Außerdem kann man in LOBAS Ungleichheiten zwischen linken und rechten Seiten der Axiome definieren.

3.2.3 Beurteilung

Frage 1 (Spezielle OO-Probleme) kann deutlich mit *Nein* beantwortet werden. Denn die Spezifikation muß von Hand geschrieben werden. Dabei besteht keinerlei Unterstützung im Hinblick auf spezielle OO-Probleme, wie sie in Abschnitt 2 beschrieben sind.
Die Betrachtung des Objektzustandes als Argument und Rückgabewert ermöglicht nur die ähnliche Syntax der Spezifikation.
Frage 2 (Modifizierte Technik) kann ebenso mit *Nein* beantwortet werden. Denn abgesehen vom Objektzustand wird keines der Probleme, die in Abschnitt 2 diskutiert werden, berücksichtigt. Das Problem des Objektzustandes wird nur implizit durch den Autor der Spezifikation berücksichtigt. Dem Autor obliegt es, die entsprechenden Axiome zu definieren.
Ein großer Nachteil besteht außerdem darin, daß dieses Verfahren nur dazu geeignet ist, ADT's zu testen.

3.3 Strukturiertes Testen

[SP00] stellt einen Ansatz zum strukturierten Testen objektorientierter Software vor. Die Autoren nennen diesen Ansatz **OMEN**, da er auf der Analyse von **O***bject* **M***anipulations* und **E***scape i***N***formation* beruht.

3.3.1 Motivation

Die Effektivität von kontroll- und datenflußorientierten Testverfahren ist für objektorientierte Programme unklar, denn diese Testverfahren wurden ursprünglich für imperative Sprachen entwickelt. Objektorientierte Programme unterscheiden sich aber in ihrer Struktur signifikant von imperativen Programmen.
Wie schon gezeigt, mündet objektorientiertes Design in viele kleine Methoden. Diese Methoden haben sehr einfachen interprozeduralen Kontrollfluß mit wenigen Anweisungen (Abschnitt 2.2).
Anhand eines einführenden Beispiels zeigen die Autoren, daß bisher bekannte Analysemethoden zur Identifizierung von def-use Beziehungen in objektorientierten Programmen unzureichend sind.
Bis auf zwei berücksichtigen die meisten Methoden nicht die Beziehungen zwischen Objekten. Diese zwei Methoden berücksichtigen aber auch nicht die Möglichkeit der mehrfachen Referenzierung. Ebenso wird dem Tester keine Information darüber gegeben, ob Teile des zu testenden Codes nicht verfügbar sind, da es sich dabei um Bibliotheken oder um Code eines anderen Herstellers handelt.

3.3.2 Ergebnisse

Der OMEN-Ansatz konzentriert sich auf die Manipulation von Objekten, und damit auf Daten, nicht auf die Prozeduren, die diese Daten manipulieren.
Um def-use Beziehungen analysieren und Testtupel erzeugen zu können, wird ein Programm durch verschiedene Graphen repräsentiert. Der Kontrollfluß jeder Methode wird durch einen Kontrollflußgraphen, die Methodenaufrufe durch einen Aufrufgraphen repräsentiert. Die Beziehungen der Objekte untereinander modellieren die Autoren durch einen so genannten *ape graph*. Dieser wurde aus dem Points-To Escape Graphen von Whaley und Rinard [WR99] entwickelt. Der Name *ape graph* steht für*annotated points-to escape graph*. Ein *ape graph* faßt mehrere Points-To Escape Graphen zusammen, um Speicherplatz zu sparen.
Ein Points-To Escape Graph beinhaltet zum einen Informationen darüber, welche Referenzen zwischen Objekten bestehen und zum anderen, welche Objekte und Referenzen innerhalb bzw. außerhalb des analysierten Quellcode-Bereiches erzeugt wurden.
Abbildung 2 oben zeigt zwei Points-To Escape Graphen. Der erste stellt eine Situation vor der Anweisung r1 = r2.f dar. Nach der Anweisung wird der Points-To Escape Graph so transformiert, daß er die resultierende Situation widerspiegelt. Hier handelt es sich um Referenzen und Objekte, die alle innerhalb des untersuchten Codes erzeugt werden.
In Abbildung 2 unten ist die gleiche Situation gezeigt, mit dem Unterschied, daß das Objekt, welches durch r1 referenziert wird, außerhalb des untersuchten Bereiches erzeugt wurde.
In [WR99] wird die Points-To Escape Analyse dazu verwendet um festzustellen, für welche Objekte und Referenzen vollständige Informationen vorliegen, d.h. welche Objekte verbleiben vollständig im untersuchten Bereich und welche "verlassen" diesen. Dies geschieht zum Beispiel durch Parameterübergabe an Methoden, die außerhalb des untersuchten Bereiches liegen.
Dadurch könnte man beispielsweise unnötige Synchronisationsmechanismen bei der Nebenläufigen Programmierung vermeiden [WR99].
Um einen Points-To Escape Graphen zu berechnen, wird für eine Methode ein initialer Points-To Escape Graph zum Zeitpunkt der ersten Anweisung erstellt. Daraufhin wird entlang des zur Methode gehörigen Kontrollflußgraphen für jede weitere Anweisung der Points-To Escape Graph entsprechend transformiert. Dies geschieht bis das Ende der Methode erreicht ist.
Für die Analyse in [SP00] wird ebenso verfahren, jedoch muß für jede Anweisung, die Objek-

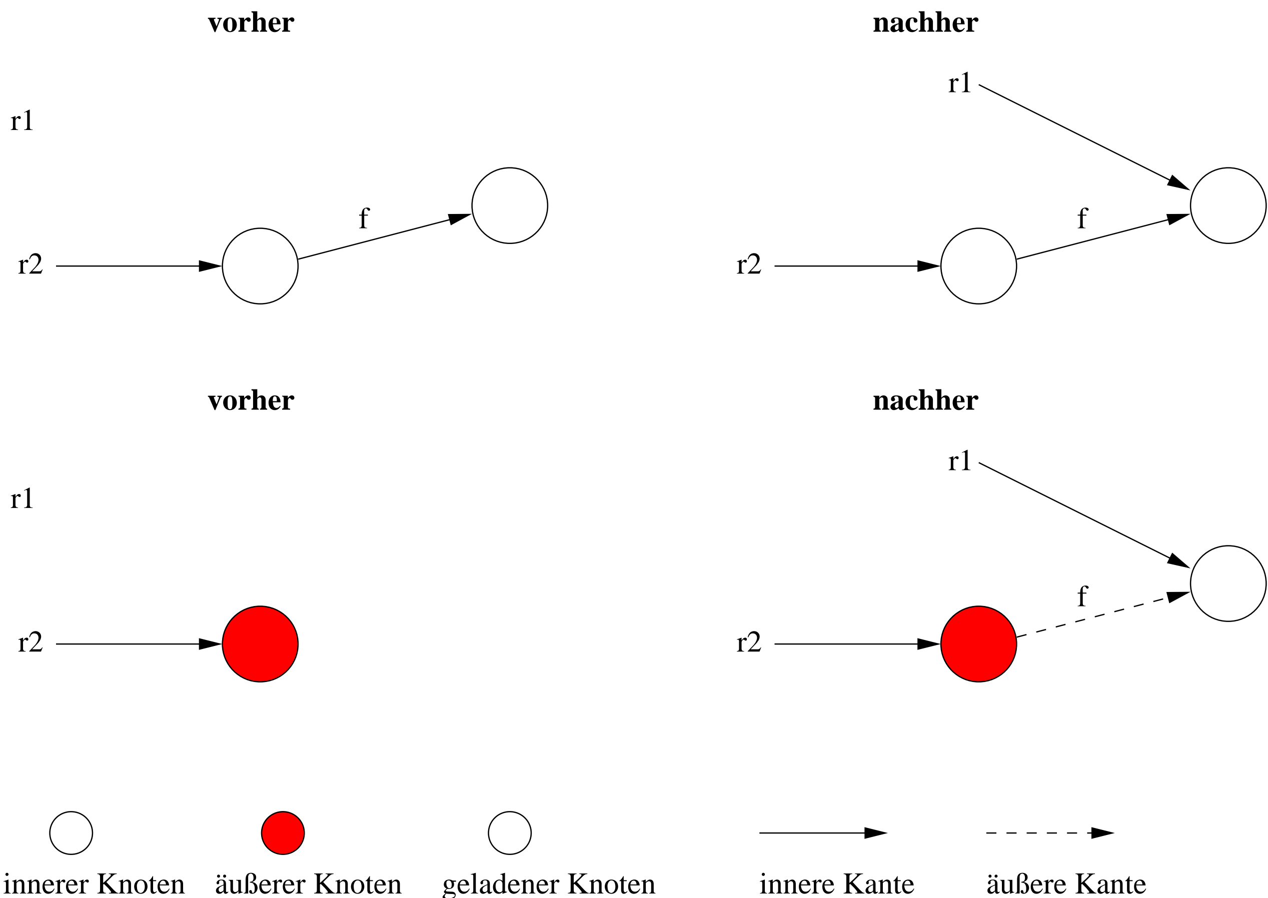


Abbildung 2: *Points-to Escape Graph*

te manipuliert oder liest, in der betrachteten Methode ein Points-To Escape Graph erstellt werden. Mit anderen Worten: es werden, angefangen vom initialen Points-To Escape Graphen, alle Points-To Escape Graphen benötigt, die durch Transformationen entlang des Kontrollflußgraphen entstanden sind. Denn die Analyse soll Fragen beantworten können, wie: *Welche Referenzen könnten an einem bestimmten Programmpunkt auf ein bestimmtes Objekt zeigen? Welche Objekte könnten an diesem Punkt gelesen werden? An welchen Punkten kann in dieses Objekt geschrieben werden?*

Der Aufwand für jede Anweisung, die in einem Zusammenhang mit einem Objekt steht, einen Points-To Escape Graphen zu speichern ist zu groß. Daher entwickelten [SP00] den ape graph.

Beim ape graph werden keine Kanten entfernt, die durch eine Schreibanweisung gelöscht würden, weil die entsprechende Referenz auf ein neues Objekt zeigt. Stattdessen wird bei jeder Transformation ein Vermerk (Annotation) zu jeder Kante gemacht, bei welcher Anweisung sie beispielsweise erzeugt oder gelöscht wurde.

Der Algorithmus, der die Testtupel basierend auf def-use Beziehungen generiert, erhält als Eingabe eine Reihe von Aufrufgraphen für die zu testende Komponente (CUT, component under test), sowie einen ape graph pro Methode in CUT. Der Algorithmus traversiert jeden Aufrufgraphen von der Wurzel an in einer topologischen Reihenfolge. Die zu einem Knoten im Aufrufgraphen gehörige Methode wird bearbeitet, in dem der ape graph analysiert wird.

Als Ausgabe produziert der Algorithmus eine Reihe von Testtupeln und Hinweise auf mögliche Einflüsse, die außerhalb von CUT auftreten könnten.

3.3.3 Beurteilung

Frage 1 (Spezielle OO-Probleme) kann mit *Ja* beantwortet werden. Es werden Einflüsse unbekannter Client- und Serverklassen berücksichtigt. Außerdem wird die mehrfache Referenzierung von Objekten beachtet. Die Analyse deckt def-use Beziehungen auf, die mit normalen Methoden des Strukturierten Testens nicht erkannt werden, da sie durch objektorientierte Konzepte, wie in Abschnitt 2 beschrieben, verborgen bleiben.

3.4 Testreihenfolge von Klassen

In [LTWD00] wird ein Algorithmus vorgestellt, mit dessen Hilfe man eine optimale Testreihenfolge für Klassen findet.

3.4.1 Motivation

Bei der Erzeugung von Testtreibern besteht ein Problem darin, die Testreihenfolge der zu testenden Klassen so festzulegen, daß nur eine minimale Anzahl von Stummel erzeugt werden muß. Stummel sind minimale, notwendige Programme oder Klassen, ohne deren Hilfe die zu testende Klasse nicht ausführbar ist.
Die Lösung besteht darin, eine Testreihenfolge zu suchen, die alle Abhängigkeiten der Klassen untereinander berücksichtigt.

3.4.2 Ergebnisse

Die Klassen werden so in verschiedene Level partitioniert, daß Klassen des i-ten Levels nur von Klassen des i-1-ten Levels abhängen. In [KGH95] wird ein *"test order finding algorithm"* definiert, der genau dies leistet. Allerdings berücksichtigt dieser Algorithmus nur statische Abhängigkeiten. Solche, die durch spätes Binden entstehen könnten, werden nicht beachtet.
In [LTWD00] werden die Grundlagen dieses Algorithmus erläutert. Er basiert auf einem Klassenmodell, das ORD (Object Relation Diagram) heißt. Dabei handelt es sich um einen Graphen, in dem die Beziehungen der Klassen untereinander (Vererbung, Assotiation, Aggregation) modelliert sind. In [KGH95] wird dieses ORD durch reverse engineering aus dem Quellcode gewonnen.
In Abbildung 3 ist ein Beispiel eines solchen ORD zu sehen. Die mit I bezeichneten Kanten stellen Vererbungsbeziehungen dar. Dabei erweitert eine Klasse von der eine Kante ausgeht, diejenige Klasse, in der die Kante endet. Die Klasse E erweitert also die Klasse A. Die mit Ag und As bezeichneten Kanten sind jeweils gerichtete *Aggregationen* bzw. *Assotiationen.*
Mittels des ORD kann nun für jede Klasse X die Menge aller Klassen $CFW(X)$ (class firewall) berechnet werden. $CFW(X)$ enthält alle Klassen, die durch eine Änderung von X betroffen sein könnten und erneut getestet werden müssen. Tabelle 2 zeigt die zum Beispiel in Abbildung 3 korrespondierenden Mengen $CFW(X)$.
Aus den Mengen lassen sich dann die in Tabelle 3 gezeigten Testlevel berechnen.
Da das ORD ähnlich dem Klassendiagramm der UML ist, muß man es nicht notwendigerweise durch Reverse Engineering erzeugen, sondern kann es u.U. aus Dokumenten der Designphase gewinnen.

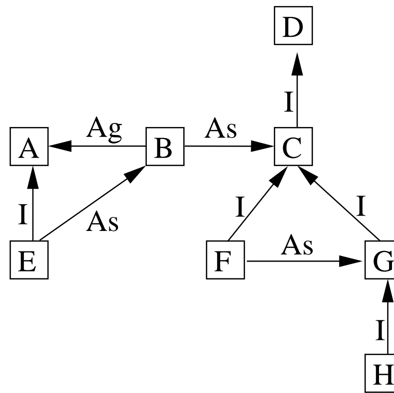


Abbildung 3: *ORD [LTWD00]*

Im Unterschied zu [KGH95] wird in der analysierten Arbeit Polymorphie und mögliches dynamisches Binden berücksichtigt.
In dem Beispiel sorgen die Assotiation zwischen den Klassen *B* und *C*, sowie der Vererbungsbaum, dessen Wurzel die Klasse *C* ist, für zusätzliche Abhängigkeiten der Klasse *B* von den Klassen *F*, *G* und *H*. Diese Abhängigkeiten sind dynamisch, da erst bei Programmablauf feststeht, zwischen welchen Klassen die Assotiation tatsächlich besteht. Beispielsweise könnte die Klasse *B* eine Assotiation mit der Klasse *F* anstatt der Klasse *C*, wegen der Vererbungsbeziehung zwischen *C* und *F*, eingehen.
Berücksichtigt man neben den statischen Abhängigkeiten auch die dynamischen, erhält man den *C-ORD (Completed ORD)*, der in Abbildung 4 zu sehen ist.
Manche Test-Level sind außerdem teilweise oder ganz unmöglich, da sie abstrakte Klassen beinhalten. Von abstrakten Klassen können keine Objekte instantiiert werden, und deren Tests müssen auf einen späteren Zeitpunkt verschoben werden. Der betreffende Test-Level wird dann unter Umständen aufgelöst.
Die vorgelegte Methode beinhaltet fünf Schritte:

1. Statische und dynamische Abhängigkeiten der Klassen werden analysiert.

2. Statische und dynamische Testlevel werden definiert, je nachdem ob es sich um statische oder dynamische Abhängigkeiten handelt.

3. Mit einer partiell geordneten Relation wird die Testreihenfolge der einzelnen Klassen ermittelt. Klassen, die dynamisch von einander abhängen, werden zuerst getestet, um

Class X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	∅
F	∅
G	F, H
H	∅

Tabelle 2: *CFW(X)*

Testing Level	Class(es)
1	A, D
2	C
3	B, G
4	E, F, H

Tabelle 3: *Testing Level*

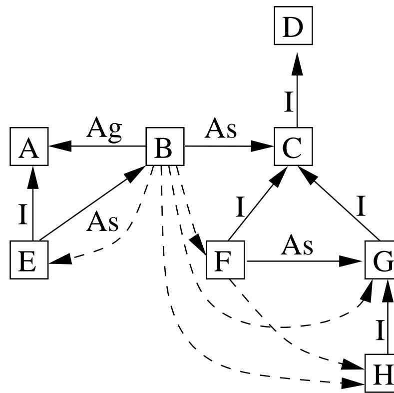


Abbildung 4: *C-ORD [LTWD00]*

statische Probleme zuerst lösen zu können. Dies erleichtert das Lösen von Problemen, die auf Grund von Polymorpie entstanden sind.

4. Weitere Informationen werden den Testlevel hinzugefügt, z.B. welche Klasse getestet wird.

5. Unmögliche Testlevel werden entfernt und abstrakte Klassen in spätere Level verschoben.

Der Ansatz wurde in einem Softwaretool mit dem Namen TOONS (Testing level generator for Object-OrieNted Software) implementiert.

3.4.3 Beurteilung

Frage 1 (Spezielle OO-Probleme) kann mit *ja* beantwortet werden. Es werden Objektbeziehungen und dynamisches Binden berücksichtigt.

Frage 2 (Modifizierte Technik) kann ebenfalls mit *ja* beantwortet werden.

3.5 Automatisches Generieren von Testfällen

[Buy00] stellt einen Algorithmus vor, mit dessen Hilfe Testfälle generiert werden können, die Objektzustände berücksichtigen.

3.5.1 Motivation

Wie in Abschnitt 2.2 genannt, ist das Ergebnis eines Methodenaufrufes vom aktuellen Zustand eines Objektes abhängig. Beim Testen einzelner Klassen können Fehler unentdeckt bleiben, weil ein kritischer Zustand nicht beachtet wird oder gar bekannt ist.
Von entscheidender Bedeutung ist die Reihenfolge der Methodenaufrufe seit der Erzeugung eines Objektes. Also ist eine Kette von Methodenaufrufen interessant, die das zu testende Objekt in einen kritischen Zustand bringt und zum Schluß diejenige Methode aufruft, die einen Fehler verursacht.

3.5.2 Ergebnisse

Der Ansatz betrachtet eine einzelne, isolierte Klasse (CUT). Es werden Testfälle generiert, die jeweils aus einer Kette von Methodenaufrufen bestehen. Test-Orakel, die die zu den jeweiligen Testfällen erwarteten Ergebnisse liefern, werden durch den vorgestellten Algorithmus nicht erzeugt.
Es werden Methodenpaare selektiert, die auf dasselbe Attribut lesend bzw. schreibend zugreifen. Jedes Paar hat die folgenden Eigenschaften:

- Es gibt eine Methode m_d, die eine Definition s_d eines Attributes v enthält
- Eine Methode m_u enthält eine Anweisung s_u, welche v liest.

Für jedes dieser Methodenpaare wird nun eine Kette von Methodenaufrufen gesucht, die m_d und m_u in der richtigen Reihenfolge enthalten und folgende Eigenschaften besitzen:

- Die erste Methode ist ein Konstruktor für CUT.
- s_d wird mindestens einmal ausgeführt.
- Es existiert ein definitionsfreier Pfad zwischen s_d und s_u.
- m_u ist die letzte aufgerufene Methode.

Abbildung 5 verdeutlicht dies.

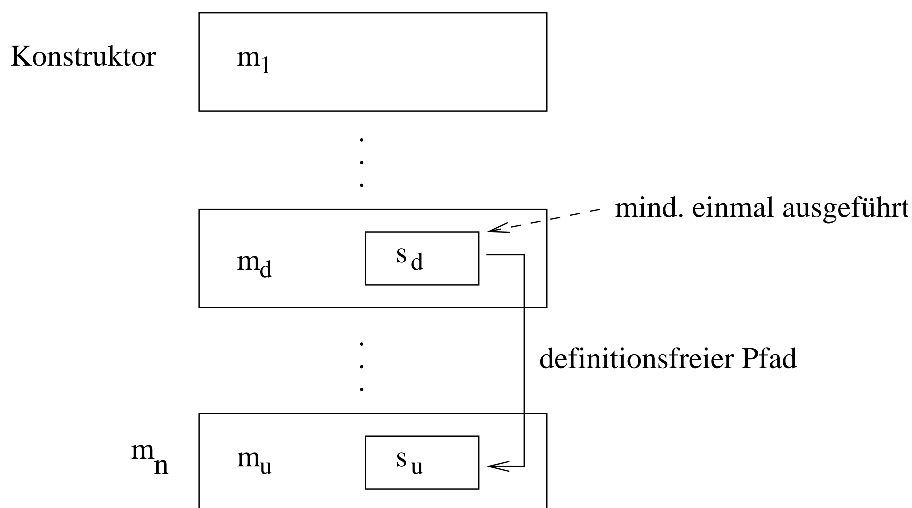


Abbildung 5: *Kette von Methodenaufrufen.*

Die Berechnung der Aufrufketten, also der Testfälle, ist in drei Phasen unterteilt:

Datenflußanalyse: Die Datenflußanalyse berechnet alle möglichen def-use Paare $d = \langle s_d, s_u, v \rangle$.

Symbolische Ausführung: Für jedes def-use Paar d berechnet die symbolische Ausführung einen möglichen Pfad, falls einer existiert, durch eine Reihe von Methoden. Darin müssen insbesondere die Methoden m_d und m_u enthalten sein. Ausserdem dürfen zwischen s_d und s_u keine weiteren Definitionen, d.h. Schreibzugriffe, auf v stattfinden.

Automatische Deduktion: In dieser letzten Phase werden die Ergebnisse der vorangehenden Phasen dazu verwendet, um letztendlich die Methodenaufrufketten zu berechnen. Dabei ist vor allem die korrekte Reihenfolge der jeweiligen Methodenaufrufe innerhalb einer Kette zu finden.

3.5.3 Beurteilung

Der Ansatz deckt Fehler auf, die durch einen Methodenaufruf hervorgerufen werden, während sich das betreffende Objekt in einem ganz bestimmten Zustand befindet. Daher kann Frage 1 (Spezielle OO-Probleme) positiv beantwortet werden.
Es werden traditionelle Methoden wie Datenflußanalyse und Symbolische Ausführung verwendet, um Testfälle zu erzeugen. Auch Frage 2 (Modifizierte Technik) kann mit *ja* beantwortet werden.
Der Ansatz ist mit Einschränkungen verbunden: Jede Klasse wird für sich in Isolation analysiert, daher können Aufrufe fremder Objekte oder dynamische Effekte nicht berücksichtigt werden. Des weiteren sollten die Klassen Attribute einfacher Datentypen, d.h. keine Objekte, besitzen.

4 Zusammenfassung

In Tabelle 4 sind die betrachteten Methoden, zur Übersicht, den Problemen gegenübergestellt. Dabei wird Frage 1 (Spezielle OO-Probleme) und Frage 2 (Modifizierte Technik) jeweils beantwortet.

Verfahren	Delokalisierung	Objektzustand	Polymorphie	unbek. Klassen
Review von OO-Code	wurde untersucht	–	–	–
Algebraisches Testen	–	nein (beide)	nein (beide)	nein (beide)
Strukturiertes Testen	–	ja (nur 1)	nein (beide)	ja (beide)
Testlevel	–	nein (beide)	ja (nur 1)	ja (nur 1)
autom. gen. v. Testfällen	–	ja (beide)	nein (beide)	nein (beide)

Tabelle 4: *Zusammenfassung*

Literatur

[Buy00] U. Buy. Automated testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis 2000*, pages 39–48, Portland, Oregon, August 2000.

[HIM00] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis 2000*, pages 60–70, Portland, Oregon, August 2000.

[HS96] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *ACM SIGSOFT International Symposium on Software Testing and Analysis 1996*, pages 53–61, 1996.

[SP00] A. Souter and L. Pollock. OMEN: A strategy for testing object-oriented software. In *ACM SIGSOFT International Symposium on Software Testing and Analysis 2000*, pages 39–59, Portland, Oregon, August 2000.

[LTWD00] Labiche, Thevenod-Fosse, Waeselynck, Durand. Testing Levels for Object-Oriented Software. International Conference on Software Engineering ICSE 22 (2000) 136-145

[DRW00] Dunsmore, Roper, Wood. Object-Oriented Inspection in the Face of Delocalisation. *International Conference on Software Engineering ICSE 22 (2000)* 467-476

[DF94] Doong, Frankl. The ASTOOT Approach to Testing Object-Oriented Programms. In *ACM Transactions on Software Enigineering and Methodology*, Vol. 3, No 2, April 1994, 101-130.

[SPLLL88] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Liiman, Robin Lampert. Designing Documentation to Compensate for Delocalized Plans. In *Communications* of the ACM, Vol. 31, No 11, pp. 1259-1267, 1988.

[KGH95] D. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima. Class Firewall, test order and regression testing of object-oriented programs. In *Journal of Object-Oriented Programming*, vol. 8 (2) 1995, Seiten 51-65.

[GE00] Matthias Grund, Herbert Ehrlich. Smalltalk und XP im "Corporate Development": ien Erfahrungsbericht. In *OBJEKTSpektrum* 3/2000, Seiten 66-73.

[WR99] John Whaley and Martin Rinard. Composital Pointer and Escape Analysis for Java Programs. *Conference on Object-Oriented Programming, Systems, Languages and Applications 1999* pages 187-206, Denver, Colorado, November 1999.

[GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley