„Regressionstesten“

Ausarbeitung zum
Vortrag im Hauptseminar

**„Software Test:
Verfahren und Werkzeuge“**

im SS 2005
am 12. Juli 2005
von Dominik Schindler

Inhaltsverzeichnis

1. Einleitung ... 3
1.1. Definition: Regressionstest ... 3
1.2. Zeitpunkt ... 3
1.3. Motivation ... 3
1.4. Probleme ... 4
2. Reduzierung des Testaufwands ... 5
2.1. Motivation ... 5
2.2. Testfälle reduzieren ... 5
2.3. Testfälle auswählen ... 5
2.3.1. TestTube von Yih-Farn Chen et al, für prozedurale Programmiersprachen ... 6
2.3.2. „Graph Traversal Algorithm“ von Rothermel & Harrold für C++ ... 7
2.3.3. „RETEST“ von Harrold et al, für Java und andere OO-Sprachen ... 10
3. Fazit ... 15
4. Werkzeuge ... 16
4.1. Mercury: WinRunner: ... 16
4.2. AutomatedQA: TestComplete ... 16
4.3. Lehrstuhl Informatik 11: aRTS – a Regression Test Selection Tool ... 16

Abbildungsverzeichnis

Abbildung 1: Fehlerklassifikation bei Änderungen von SW ... 4
Abbildung 2: Illustration der Idee von TestTube ... 7
Abbildung 4: grafische Darstellung der Technik von Rothermel und Harrold ... 8
Abbildung 4: Quelltext der Beispielfunktion `avg` und geänderte Version `avg'` ... 8
Abbildung 5: Kontrollflussgraph für `avg` (links) und `avg'` (rechts) ... 9
Abbildung 6: Darstellung interne Methode (links) und externe Methode (rechts) ... 11
Abbildung 7: Beispieldarstellung eines internen Methodenaufrufs ... 12
Abbildung 8: Beispiel eines externen Methodenaufrufs ... 13
Abbildung 9: Exceptions im JIG ... 14
Abbildung 10: Prozentsatz d. ausgewählten Testfälle für die verschiedenen Versionen ... 16

Tabellenverzeichnis

Tabelle 1: Testpaket für avg ... 10
Tabelle 2: Kantenüberdeckungsmatrix vom Testpaket für avg. ... 10
Tabelle 3: Testobjekte zum Testen des Selektionsverfahrens ... 15

1. Einleitung

1.1. Definition: Regressionstest

Der Begriff Regressionstest wird auf zwei verschiedene Arten verwendet. Beiden gemeinsam ist die Wiederverwendung bereits existierender Testfälle aus vorangehenden Tests[1]:

- Einerseits bezeichnet Regressionstest die Wiederholung von Tests auf Teile der Software, die korrigiert wurden. Es werden also genau die Tests wiederholt, die den Fehler aufgedeckt haben um somit sicherzustellen, dass der Fehler behoben wurde.
- Regressionstest ist andererseits auch die Wiederholung von Tests um auszuschließen, dass die Änderung irgendwelche Auswirkungen auf andere Teile der Software hat. Es wird also die Integrität der Software getestet und nicht der Erfolg des Bugfixes selbst.

Spricht man einfach von Regressionstest meint man fast immer beide oben genannten Definitionen. Als Grundlage für den Regressionstest kommen üblicherweise Black-Box und White-Box Tests zum Einsatz. Folgende zwei Arten von Regressionstests werden unterschieden[2]:

- **progressiver Regressionstest („progressive regression test“):** Bei geänderter Spezifikation aufgrund →„adaptive maintenance“ oder →„perfective maintenance“ modifiziertes Programm gegen modifizierte Spezifikation testen ⇒ es müssen geeignete Testfälle hinzugefügt werden
- **korrigierender Regressionstest („corrective regression test“):** Bei dieser Art von Regressionstest ist die Spezifikation unverändert, evtl. kann sich der Entwurf ändern ⇒ es müssen nur einige Zeilen Code im Programm mit bestehen Testfällen gestestet werden.

1.2. Zeitpunkt

Der Regressionstest wird immer dann durchgeführt, wenn Software geändert wurde. Im gesamten Softwarelebenszyklus (-entwicklungszyklus) kann dies schon bei der Entdeckung und Behebung von Fehlern notwendig sein, zum Beispiel beim Modul-, Integrations- oder Systemtesten. Größtenteils erfolgen Regressionstests aber in der Wartungsphase[2].
In dieser Phase sind Regressionstests nach folgenden Arten von Softwareänderungen notwendig[2,3]:

- **Entfernung aktueller Fehler („corrective maintenance“):** Behebt Software- und Implementationsfehler, damit das System korrekt funktioniert. Das Entfernen von Fehlern wird normalerweise bereits im gesamten Softwarelebenszyklus durchgeführt.
- **Anpassung an neue Benutzeranforderungen („perfective maintenance“):** z.B. das Hinzufügen neuer Funktionalität, Verbesserung der Performance, …
- **vorbeugende Maßnahmen („preventive maintenance“):** z.B. Verbesserung der Softwarequalität (z. B. Wartbarkeit und Struktur), Anpassung der Dokumentation, …
- **Anpassung an neue Umgebung („adaptive maintenance“):** z.B. neuer Übersetzer, neue Hardware, neues Betriebssystem, neue Systemumgebung (z.B. länderspezifische Besonderheiten), …

1.3. Motivation

Regressionstests werden hauptsächlich durchführt, um Vertrauen in die Software nach deren Modifikation zu gewinnen. Betrachtet man das Hauptproblem bei der Software-Wartung, nämlich das Erzeugen von Fehlern durch Änderungen, Erweiterungen und Fehlerkorrekturen[4], erkennt man, das nur durch den wiederholten Test der Software dieses Vertrauen (zurück)gewonnen werden kann.

Laut Sharp[5] besteht eine Wahrscheinlichkeit von 20% - 50%, dass neue Fehler (so genannte „bad fixes“) hinzugefügt werden. Marciniak erwähnt in seiner Enzyklopädie[2] eine Studie von Hetzel (1984), die in machen Fällen sogar von einer Wahrscheinlichkeit von 50%-80% ausgeht. Wallmüller[4] hat die Wahrscheinlichkeit von Fehlern durch Änderungen an Software nach Art der Veränderung klassifiziert und ist zu folgendem Ergebnis gekommen:

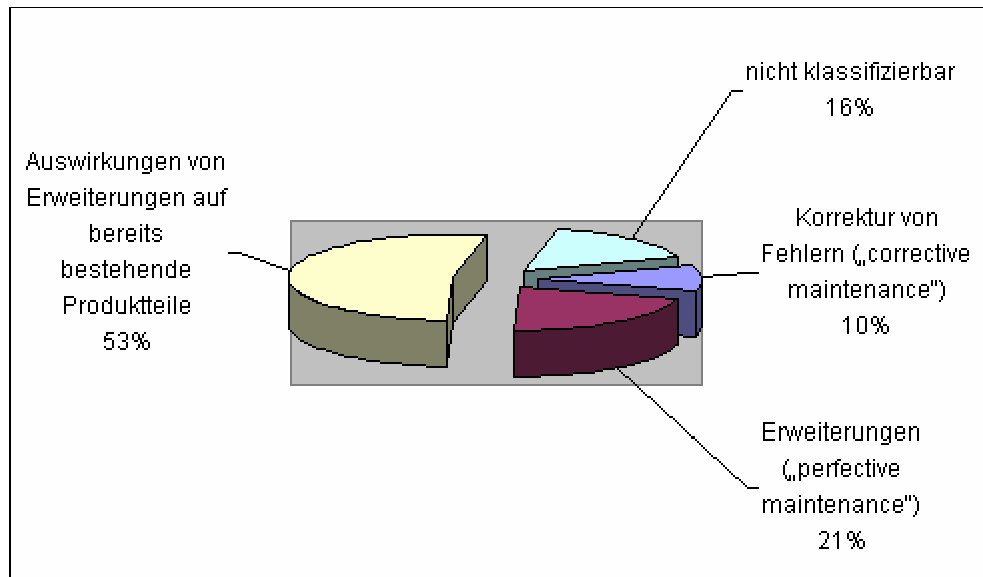

Abbildung 1: Fehlerklassifikation bei Änderungen von SW
Quelle: Wallmüller 1990

Wie in Abbildung 1 zu sehen ist, sind über die Hälfte der Fehler die Auswirkungen durch Erweiterungen. Das bedeutet, dass Regressionstests wichtig sind, um eventuelle Seiteneffekte durch Veränderungen auszuschließen. Solche Seiteneffekte bzw. „bad fixes“ können im schlimmsten Fall schwerwiegender sein als der Fehler, der zur Korrektur der Software geführt hat.

Ein weiterer entscheidender Grund für Regressionstest ist das Problem, dass die Korrektur eines Fehlers nur in 50% der Fälle, bei einer Anzahl der Statements ≤ 10, das erste mal überhaupt den Fehler beseitigt. Bei 50 oder mehr Anweisungen beträgt die Wahrscheinlichkeit nur noch 20%[[6]].

Die hier beschriebenen Probleme bekräftigen die Notwendigkeit von Regressionstest. Nicht umsonst werden diese ebenfalls in vielen (Qualitäts)Standards der Software-Entwicklung gefordert[[7]].

1.4. Probleme

Bei der Wiederverwendung von Testfällen im Regressionstest wäre es Unsinn, alle Testfälle zu benutzen. Deshalb tritt häufig das Problem auf, dass entschieden werden muss, welche Testfälle zum Testen der geänderten Version ausgewählt werden sollen (engl. „regression-test-selection problem“). Für dieses Problem gibt es bereits viele Techniken für verschiedene Programmiersprachen/-paradigmen, die mehr oder weniger sicher bzw. effizient sind. Eine Auswahl und Bewertung solcher Techniken werden im Kapitel 2 detailliert vorgestellt.

Das größere Problem betrifft das Hinzufügen von neuer Funktionalität, also das Problem, welche Testfälle man zum Testpaket hinzufügen muss, um die Erweiterungen zu testen (engl. „test-suite augmentation problem“). Die hier vorgestellten Selektionstechniken können nicht überdeckte Kanten erkennen und den Tester darüber informieren, dass für diese Kanten Testfälle fehlen. Die Testfälle selbst müssen aber noch per „Handarbeit“ hinzugefügt werden. Die in diesem Seminar vorgestellten Verfahren und Werkzeuge können dabei helfen.

Da die Durchführung des Regressionstests immer die wiederholte Abarbeitung stets gleich bleibender Schritte ist, bietet es sich an, die Durchführung durch Werkzeuge zu automatisieren. Würde man den Regressionstest manuell durchführen, wäre die Wahrscheinlichkeit Fehler zu übersehen aufgrund des monotonen Testverlaufs ziemlich groß. Der immer gleiche Ablauf, also das Eingeben der Testdaten gefolgt von dem Vergleich der Ausgaben, führt oft zu Ermüdungserscheinungen, sodass Diskrepanzen nicht erkannt werden. Deshalb sind manuelle Regressionstests in der Regel wirtschaftlicher Unsinn und sollten deshalb automatisiert werden. Die Werkzeuge, die dafür benötigt werden, sind aber meistens recht teuer und bedürfen nicht selten auch einer langen Einarbeitungszeit.

2. Reduzierung des Testaufwands

2.1. Motivation

Da der Regressionstest im Allgemeinen nichts anderes ist als die Wiederholung bereits durchgeführter Tests mit vorhandenen Testfällen (aus dem Modultest), könnte man im einfachsten Fall alle Tests auf die korrigierte Version der Software loslassen. Aufgrund der Anzahl der Testfälle komplexer Software wäre diese Art des Vorgehens wirtschaftlicher Unsinn und bei konsequenter Durchführung wäre bei den meisten Projekten kein Ende in Sicht.
Würde man andrerseits zu wenige Testfälle hernehmen, kann unter Umständen die Software zu wenig getestet werden. So kann zum Beispiel der Quelltext so verändert worden sein, dass mindestens eine Anweisung nicht durch die ursprünglichen Testfälle abgedeckt wird.

Die hier vorgestellten Techniken beschäftigen sich mit dem Entfernen bzw. Auswählen von Testfällen, um den Aufwand des wiederholten Testens zu verringern. Bei den Techniken zur Selektion von Testfällen wurden nur Techniken berücksichtigt, die auch als sicher gelten (vgl. Kapitel 2.3, „Inclusiveness“). Außerdem werden keine datenflussorientierten Verfahren[8] berücksichtig, da die Datenflussanalyse insbesondre bei großen Softwaresystemen sehr viel Zeit in Anspruch nehmen kann und es auch wenige Tools zur Zeit gibt, die die Datenflussanalyse unterstützen. Dadurch können diese Verfahren noch ineffizienter sein als gleich alle Testfälle wieder zu verwenden[9].

2.2. Testfälle reduzieren

Bevor die ausgefeilten Selektions-Verfahren vorgestellt werden, möchte ich auf andere Vorgehensweisen eingehen, mit denen man bereits ohne größeren Aufwand die Anzahl der Testfälle verringern kann[10,11].

Eine Möglichkeit Testfälle zu verringern ist die Entfernung so genannter „obsolete test cases“. Diese Testfälle wurden bei vorausgehenden progressiven Regressionstests hinzugefügt, um speziell Änderungen/Erweiterungen der Software zu testen. Da diese Änderungen/Erweiterungen aber nach mehreren Testläufen bereits ausreichend gestestet wurden, sind diese Testfälle - selbstverständlich unter der Voraussetzung, dass keine weiteren Fehler mehr gefunden wurden - für andere Regressionstests überflüssig und können deshalb entfernt werden.
Die Anzahl der Testfälle kann weiterhin durch Weglassen von redundanten Testfällen („redundant test cases“) verringert werden. Als redundante Testfälle bezeichnet man mehrere Testfälle, die dieselben Teile der Software testen. Das kann zum Beispiel auftreten wenn mehrere Tester Testfälle erstellen.
Ferner kann man evtl. semantisch gleichartige Testfälle zu einem großen Testfall kombinieren, so dass dieser Testfall nur einmal bei einem Test zur Anwendung kommt.

2.3. Testfälle auswählen

Allen Selektions-Verfahren gemeinsam ist die statische Analyse des Quelltextes, um die Änderungen zu erkennen und eine dynamische Analyse, um die von den Testfällen überdeckten Anweisungen/Methoden bzw. Kanten zu ermitteln.

Bevor die Selektions-Verfahren vorgestellt werden, möchte ich zuerst einige wichtige Definitionen zum besseren Verständnis der nachfolgenden Bewertungskriterien und Techniken geben:

Definitionen:[8]

- `P` : Programm/-teil, das getestet wird
- `T` : eine Menge von Testfällen (sog. Testpaket, engl. „Testsuite“)
- `P(i)`: Ausführung des Programms `P` mit `i` als Eingabe,
- `P’` : geändertes Programm `P`
- `t` : ein Testfall aus der Menge der Testfälle `T`, $t \in T$, $t = (i, o)$ mit o = erwartetes Ergebnis

Um die Selektionstechniken vergleichen zu können, möchte ich außerdem auf die vier Möglichkeiten der Bewertung dieser Verfahren eingehen. Die Definitionen wurden aus der Studienarbeit am Lehrstuhl[12] übernommen.

Definition „modifikation-traversierend“: Einen Testfall t bezeichnet man als modifikation-traversierend („modification-traversing“), wenn er neuen oder geänderten Code in P’ ausführt, bzw. wenn er Code in P ausführte, der in P’ gelöscht wurde.

Definition „Inclusiveness“: Das „Inclusiveness“-Kriterium misst die Anzahl der selektierten modifikations-traversierende Testfälle. Angenommen, T enthält genau n modifikations-traversierende Testfälle für P und P’, und ein Verfahren V selektiert m von n Testfällen, dann ist die „Inclusiveness“ von V bzgl. P, P’ und T: $I = \frac{m}{n} \cdot 100$ für n > 0, sonst 100%. Ein Verfahren V mit einer „Inclusiveness“ von 100% bezeichnet man als **sicher**,
Beispiel: Angenommen T besteht aus 40 Testfällen, wovon 10 modifikations-traversierend für P und P’ sind, und ein Verfahren V selektiert insgesamt 15 Testfälle, darunter 9 modifikations-traversierende. Die „Inclusiveness“ von V bzgl. P, P’ und T ist dann $I = \frac{9}{10} \cdot 100 = 90\%$

Definition „Precision“: Das „Precision“-Kriterium misst die Anzahl der nicht modifikations-traversierenden Testfälle, die weggelassen werden (d.h. nicht selektiert werden). Angenommen T enthält genau n nicht modifikations-traversierende Testfälle für P und P’, und ein Verfahren V selektiert m von den n Testfällen nicht, dann ist die „Precision“ von V bzgl. P, P’ und T: $P = \frac{m}{n} \cdot 100$, für n > 0, sonst 100%.
Beispiel: Man betrachte erneut die beim Beispiel für „Inclusiveness“ beschriebene Situation. T enthielt 30 nicht modifikations-traversierende Testfälle für P und P’, und das Verfahren V selektiert 6 nicht modifikation-traversierende Testfälle, d.h. 24 wurde nicht selektiert. Die „Precision“ von V bzgl. P, P’ und T ist dann: $P = \frac{24}{30} \cdot 100 = 80\%$.

Definition „Efficiency“: Das „Efficiency“-Kriterium betrachtet den Zeit- und Speicherplatzbedarf der Selektionsverfahren.

Definition „Generality“: Das „Generality“-Kriterium betrachtet die Breite der Anwendbarkeit eines Verfahrens. Zum Beispiel bewertet dieses Kriterium ein Verfahren, das nicht alle Konstrukte einer Sprache berücksichtigt, oder nicht mit realistischen Arten von Modifikationen umgehen kann, als schlecht. Auch Verfahren, die abhängig von einer bestimmten Test- und Wartungsumgebung sind, oder von bestimmten Analysewerkzeugen abhängen, bewertet dieses Kriterium als schlecht.

2.3.1. TestTube von Yih-Farn Chen et al, für prozedurale Programmiersprachen

Test Tube[[13]] ist eine von vielen Selektionstechniken, die für prozedurale Programmiersprachen, im speziellen C, entwickelt wurden. Die Firewall-Technik[[14]] von White und Abdullah ähnelt diesem Verfahren, wurde aber auf das Klassenkonzept übertragen. Da dieses Programmierparadigma aber zunehmend an Bedeutung aufgrund der objektorientierten Programmierparadigmen verloren hat, gehe ich auf diese Technik nur ganz kurz ein.

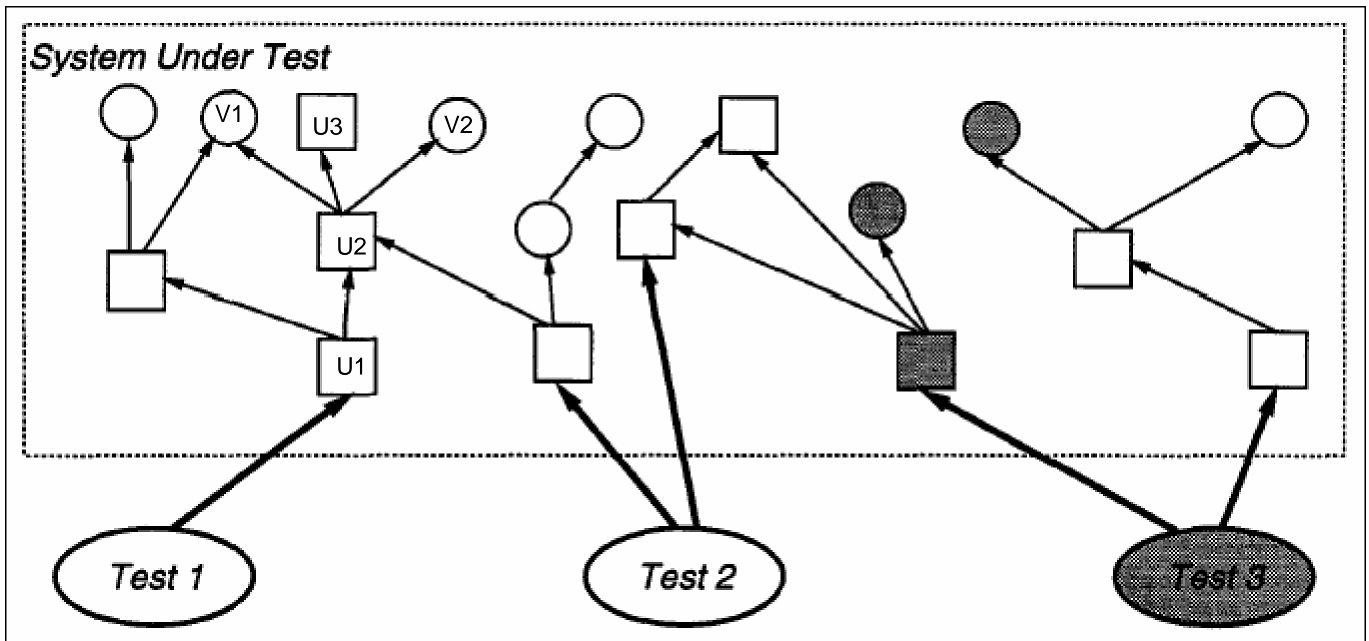

Abbildung 2: Illustration der Idee von TestTube
Quelle: Yih-Farn Chen et al, 1994

Die Idee, die hinter Test Tube steckt, ist in Abbildung 2 dargestellt. Die Kästchen stellen Unterprogramme (Prozeduren) dar, die Kreise Variablen und die Kanten statische (z.B. ein Aufruf) und dynamische (z.B. Benutzung einer Variablen) Abhängigkeiten dar. Die Pfeilspitze zeigt dabei auf das/die Unterprogramm/Variable, von dem das/die Unterprogramm/Variable am Pfeilende abhängt. So hängt z.B. das Unterprogramm U1 vom Unterprogramm U2 und dieses wiederum von den Variablen V1 und V2, sowie vom Unterprogramm U3 ab. Angenommen, die grau markierten Teile werden verändert, sieht man sehr schnell, dass nur der Testfall 3 wiederholt werden muss und die anderen nicht von den geänderten Teilen abhängen.
Das Verfahren gliedert sich in zwei Teile: Teil 1 beschäftigt sich mit der statischen Analyse der Beziehungen untereinander und dadurch auch der Anhängigkeiten von Variablen und Unterprozeduren. Der 2. Teil bestimmt durch dynamische Analyse die Überdeckung der einzelnen Testfälle. Durch die ermittelten Anhängigkeiten und der Überdeckung kann das Verfahren feststellen, welcher Testfall durch eine Änderung wieder angewendet werden muss. Da keine speziellen Features wie z.B. Polymorphismus, dynamisches Binden, usw. bei prozeduralen Programmiersprachen vorhanden sind, ist dieses Verfahren, trotz seiner Einfachheit, besonders effektiv und skaliert auch bei größeren Programmen/-teilen sehr gut.

Bewertung:

- **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
- **„Precision“:** 100%
- **„Efficiency“:** gut, wg. die Einfachheit des Verfahrens
- **„Generality“:** schlecht, weil es nicht für „moderne“ Programmiersprachen, wie z.B. objektorientierte Sprachen geeignet ist

2.3.2. „Graph Traversal Algorithm“ von Rothermel & Harrold für C++

Die Technik von Rothermel & Harrold[[15]] bestimmt die Menge `T'`, einer Teilmenge von `T`, um damit `P'` zu testen. Diese Menge besteht nach Anwendung des Verfahrens nur noch aus den Testfällen, die gezielt die Änderungen in `P'` testen. Um die Testfallmenge `T'` bestimmen zu können, benutzt diese Technik eine statische (den Kontrollflussgraphen) und dynamische Analyse (die Überdeckung) des Quelltextes von `P` und `P'`.

Zuerst wird bei jedem Test eine Überdeckungsmatrix erstellt, welche jedem Testfall `t` die entsprechenden besuchten Kanten zuordnet. Danach werden anhand der beiden Kontrollflussgraphen für `P` und `P'` die gefährlichen Kanten bestimmt. Zum Schluss wählt die „Select-Tests“-Komponente durch den Abgleich der Kanten aus der Überdeckungsmatrix mit der Menge der gefährlichen Kanten

die Testfälle für `P'` aus. Die folgende Abbildung stellt den Zusammenhang der einzelnen Komponenten grafisch dar:

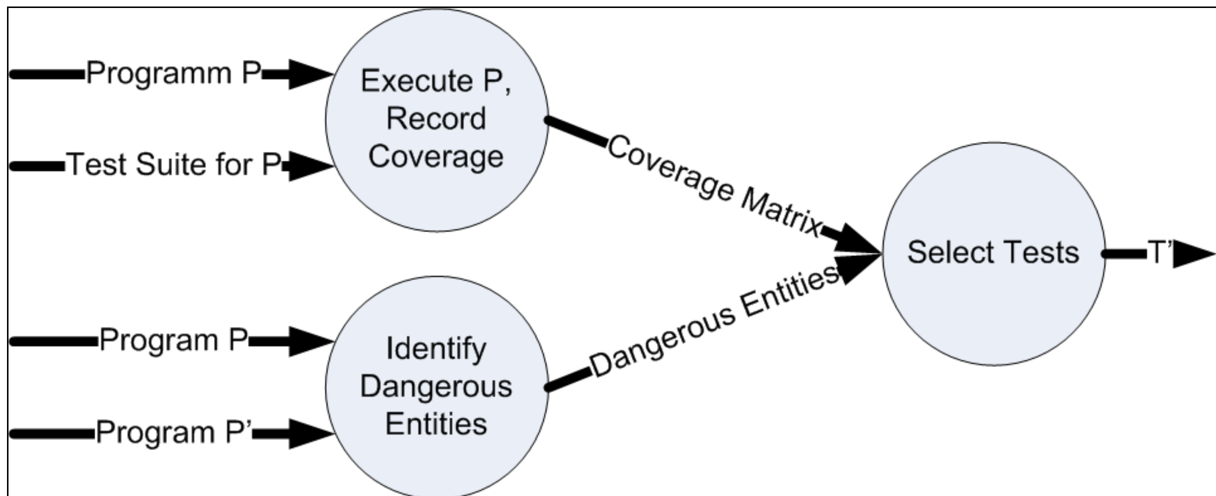

Abbildung 4: grafische Darstellung der Technik von Rothermel und Harrold
Quelle: Harrold et al., 2001

Definition gefährliche Kante: Eine gefährliche Kante `e` ist eine Kante, sodass für jede Eingabe `i` die Anweisung `e` von `P` überdeckt wird und `P(i)` und `P'(i)` sich unterschiedlich verhalten. Dieses unterschiedliche Verhalten kann entweder durch eine Kante hervorgerufen werden, die in `P'` auf einen anderen Knoten zeigt als in `P`, oder in `P'` nicht mehr vorhanden ist.

Das Vorgehen der Technik möchte ich anhand des folgenden kleinen Beispiels demonstrieren. Die Abbildung 4 zeigt den Quelltext der Beispiel-Funktion `avg` und der geänderten Version `avg'`. Die dazugehörenden Kontrollflussgraphen für avg und avg' zeigt Abbildung 5.

```
public static float avg(Float[] floats) {

(1)   int count = 0, s = 0;

(2)   if (floats.length == 0) {
(3)         return 0;
      }

(4)   if (floats.length > 100) {
(5)         System.out.println("Zu viele
                     Elemente im Array!");
(6)         return -1;
      }

(7)   while (count < floats.length) {
(8)         s += floats[count].intValue();
(9)         count += 1;
      }

(10)  return s / count;
}
```

```
public static float avg'(Float[] floats) {

(1)   int count = 0, s = 0;

(2)   if (floats.length == 0) {
(3)         return 0;
      }

(4)   if (floats.length > 100) {
(5)

(6)         return -1;
      }

(7)   while (count < floats.length) {
(8)         s += floats[count].intValue();
(9)         count++;
      }

(10)  return s / count;
}
```

Abbildung 4: Quelltext der Beispielfunktion `avg` und geänderte Version `avg'`

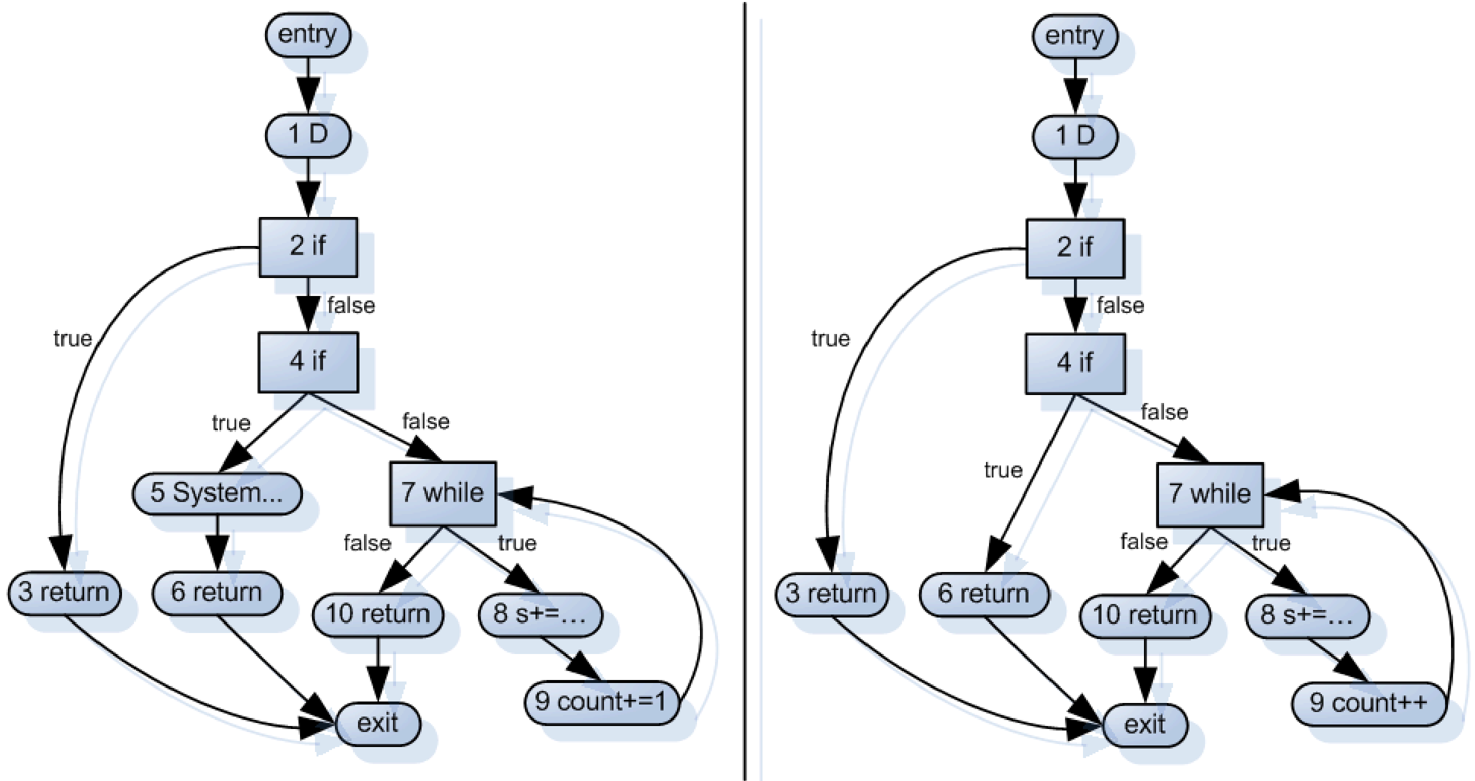

Abbildung 5: Kontrollflussgraph für `avg` (links) und `avg'` (rechts)

Wie man am Quelltext und am Kontrollflussgraphen erkennen kann, wurde in der Funktion `avg'` die Anweisung in Zeile 5 gelöscht und die Anweisung in Zeile 9 verändert.

Der Algorithmus beginnt den Durchlauf beim `entry`-Knoten von `avg` und `avg'` und durchläuft parallel alle Pfade in avg und avg'. Stellt er dabei fest, dass bei einem Knoten eine ausgehende Kante in `avg'` auf einen anderen Zielknoten zeigt als in `avg`, fügt der die Kante zum Knoten zu den gefährlichen Kanten hinzu. Der Test auf Gleichheit erfolgt anhand der Beschriftung der Konten. Im Beispiel-Kontrollflussgraphen in Abbildung 5 existieren zwei Knoten, von denen eine ausgehende Kante auf unterschiedliche Knoten zeigt.
Der erste Knoten ist Knoten 4, bei dem in `avg` die `true`-Kante auf Knoten 5 zeigt, die gleiche Kante zeigt aber in `avg'` auf Knoten 6, da Knoten 5 gelöscht wurde. Deshalb wird die Kante (4,5) zur Menge der gefährlichen Kanten hinzugefügt. Der Algorithmus stoppt hier die Traversierung dieses Pfades, da alle Testfälle, die Pfade nach dieser Kante überdecken würden, ebenfalls diese Kante überdecken müssten, und diese Testfälle würden dadurch ebenfalls ausgewählt werden.
Der zweite Knoten ist Knoten 9, bei dem sich die Beschriftung in `avg` und `avg'` unterscheiden. Durch diesen Unterschied wird die Kante (8,9) zu den gefährlichen Kanten hinzugefügt und die weitere Traversierung dieses Pfades ebenfalls abgebrochen.
Der Algorithmus stoppt die Traversierung des Kontrollflussgraphen, wenn er entweder die beiden `exit`-Knoten erreicht hat, oder es keine weiteren Pfade zum Traversieren gibt, auf denen nicht schon gestoppt wurde.

Ein besonderer Knoten stellt der Knoten `D` dar. In diesem Knoten befinden sind alle Definitionen der Funktion, um auch Änderungen in den Variablendefinitionen erkennen zu können. Eine Änderung des Typs, z.B. von `int` nach `long`, würde im Kontrollflussgraphen untergehen. Durch diesen speziellen Knoten `D` erkennt der Algorithmus eine eventuelle Änderung und würde die Kante (`entry`, `D`) zu den gefährlichen Kanten hinzufügen.

Nachdem alle gefährlichen Kanten gefunden wurden, verwendet die „Select-Tests"-Komponente (siehe Abbildung 4) die Menge der gefährlichen Kanten und die Überdeckungsmatrix, um Testfälle für `T'` aus der Menge der Testfälle `T` auszuwählen. Die Tabelle 1 zeigt ein mögliches Testpaket `T` für `avg`:

Testfall	Eingabe	Erwartete Ausgabe
1	[]	0
2	[0,1,2,…,99,100]	„Zu viele Elemente im Array!“, -1
3	[1]	1.0
4	[1, 2]	1.5

Tabelle 1: Testpaket für avg

Wenn das Programm ausgeführt wird, wird zu jedem Testfall `t` die Menge der überdeckten Kanten mit gespeichert. Im obigen Beispiel überdecken Testfall 1, 2, 3 und 4 die Kante (`entry`, 1), Testfall 1 die Kanten (1,2), (2,3) (3, `exit`), usw. Die Tabelle 2 zeigt die vollständige Kantenüberdeckungsmatrix für das Testpaket `T` von `avg`:

Testfall	Testfall
1	(`entry`,1), (1,2), (2,3), (3,`exit`)
2	(`entry`,1), (1,2), (2,4), **(4,5)**, (5,6), (6,`exit`)
3	(`entry`,1), (1,2), (2,4), (4,7), (7,8), **(8,9)**, (9,7), (7,10), (10, `exit`)
4	(`entry`,1), (1,2), (2,4), (4,7), [(7,8), **(8,9)**, (9,7)]², (7,10), (10, `exit`)

Tabelle 2: Kantenüberdeckungsmatrix vom Testpaket für avg.

Die „Select-Tests“-Komponente nimmt die Menge der gefährlichen Kanten und sieht in der Kantenüberdeckungsmatrix nach, welche Testfälle diese Kanten überdecken (in Tabelle 2 **rot** dargestellt). In unserem Beispiel waren die gefährlichen Kanten die Kante (4, 5) und (8, 9), und deshalb müssen die Testfälle 2, 3, 4 wiederholt werden.

Bewertung:

- **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
- **„Precision“:** schlecht, da externe Klassen (z.B. Bibliotheken) ebenfalls analysiert werden müssen
- **„Efficiency“:** gut
- **„Generality“:** schlecht, da das Verfahren keine Exceptions unterstützt und nur für einen Teil des Sprachumfangs von C++ implementiert wurde

2.3.3. „RETEST“ von Harrold et al, für Java und andere OO-Sprachen

Dieses Verfahren ist nicht nur eines der ersten für Java, sondern auch eines der präzisesten für objektorientierte Software, nicht nur aufgrund der Tatsache, dass alle Konstrukte moderner objektorientierter Sprachen unterstützt werden. Deshalb wird dieses Verfahren im Rahmen dieser Ausarbeitung detailliert behandelt.

Aufgrund der in Kapitel 2.3.2. gezeigten schlechten Bewertung des „Precision“ und „Generality“-Kriteriums wurde dieses Verfahren für die Programmiersprache Java entwickelt. Das Verfahren ist aber so allgemein gehalten, dass es auch für andere objektorientierte Programmiersprachen angewendet werden kann.
Um das „Generality“-Kriterium des o.g. Verfahrens zu verbessern und um alle Javakonstrukte darstellen zu können, wurde der sog. „Java Interclass Graph“ (JIG) als Erweiterung des Kontrollflussgraphen entwickelt. Dieser Graph berücksichtigt nicht nur den Kontrollflussgraphen innerhalb einer Methode, sondern auch den Kontrollfluss über Methodengrenzen hinweg, sowie andere objektorientierte Features wie Vererbung, Polymorphismus, dynamisches Binden und Exceptions. Das Selektionsverfahren ist das Gleiche wie unter 2.3.2., es werden aber anstatt die Kontrollflussgraphen die JIGs verglichen. Im Folgenden werden die Erweiterungen beschrieben und anhand eines Beispiels demonstriert.

Variablen und Objekttypen
Das in 2.3.2 vorgestellte Verfahren verwendete einen Deklarationsknoten (im Beispiel `D`) um Deklarationen im Programm/-teil abzubilden. Findet in diesem Knoten eine Änderung statt, erkennt der Selektionsalgorithmus die Änderung zwar, leider führt die anschließende Auswahl der Kante (`entry`, `D`) dazu, dass ausnahmslos alle Testfälle ausgewählt werden.
Um dieses Manko zu verbessern, wird jeder Variable primitiven Typs der Typ an den Variablennamen angehängt. So wird z.B. aus `a: long` die Darstellung a_long. Dadurch wird die Erkennung der

Änderung des Variablentyps an die Stelle verschoben, an der die Variable benutzt wird. Somit wird die Selektion wesentlich präziser als bei der Verwendung des Deklarationsknotens.
Weiterhin wird bei diesem Verfahren bei jeder Instantiierung einer Variablen (z.B. mittels `new()`) die komplette Klassenhierarchie von der Wurzel an mit dargestellt. Die Darstellung der Klassenhierarchie erfolgt mittels eines global klassifizierenden Klassennamen. Zusätzlich zur Klassenhierarchie werden außerdem noch die implementierten Schnittstellen in alphabetischer Reihenfolge mit berücksichtigt. Als Beispiel wäre eine Klasse `B` aus dem Paket `foo` genannt, die von der Klasse `A` im gleichen Paket erbt und Klasse `A` implementiert Schnittstelle `I` aus dem Paket `bar`. Für dieses Beispiel würde sich der global klassifizierende Klassenname `java.lang.Object:bar.I:foo.A:foo.B` ergeben. Diese Darstellung erlaubt es dem Selektionsalgorithmus, Änderungen in der Klassenhierarchie zu erkennen. Außerdem wird die Änderung an die Stelle verschoben, an der die Klasse instantiiert wird. Somit ist eine präzisere Auswahl der Testfälle möglich.

Interne und externe Methoden

Im JIG wird jede interne Methode, die analysiert werden soll, durch einen Kontrollflussgraphen dargestellt. Der Kontrollflussgraph auf der Aufruferseite wird durch einen `call`-Knoten und einem `return`-Knoten erweitert. Zwischen den beiden neuen Konten wird eine Pfad-Kante gezogen, welche den Pfad durch die Methode darstellt (siehe Abbildung 6).

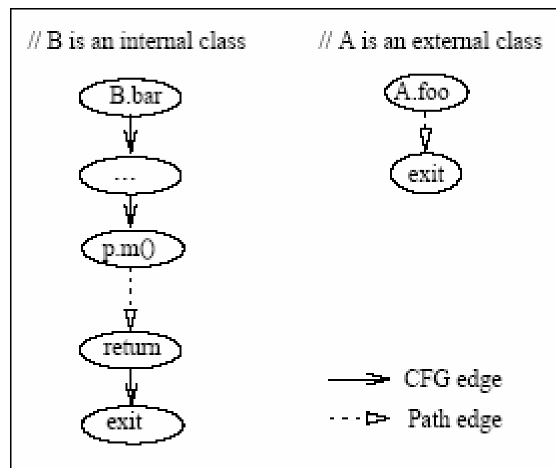

Abbildung 6: Darstellung interne Methode (links) und externe Methode (rechts)
Quelle: Harrold et al, 2001

Der Knoten `p.m()` auf der linken Seite ist der `call`-Knoten, welcher durch eine Pfad-Kante mit dem `return`-Knoten verbunden ist.
Der JIG stellt jede externe Methode, die von einer internen Methode aufgerufen wird, durch einen zusammengeklappten Kontrollflussgraphen dar. Normalerweise ist der Quelltext externer Klassen nicht verfügbar. Auch wenn er verfügbar ist, möchten wir ihn nicht analysieren, da das Verfahren davon ausgeht, dass externe Klassen sich nicht ändern. Deshalb besteht der zusammengeklappte Kontrollflussgraph einer externen Methode nur aus einem Eingangsknoten und einem Ausgangs-Knoten, die mit einer Pfadkante verbunden sind. Diese Pfadkante fasst den Weg durch die externe Methode zusammen. Die Abbildung 6 zeigt auf der rechten Seite die Darstellung einer externen Methode.

Interprozedurale Interaktionen zwischen internen Methoden

Wie bereits beschrieben wurde, wird jeder Methodenaufruf durch einen `call`-Knoten und einem `return`-Knoten, beide verbunden durch eine Pfadkante, dargestellt. Der `call`-Knoten ist außerdem mit dem Eingangsknoten der aufgerufenen Methode durch eine `call`-Kante verbunden.
Das Beispiel in Abbildung 7(a) zeigt die drei Klassen `A` (extern), `B` und `C`, die Methode `bar()` und die Vererbungshierarchie. So erbt die Klasse `B` von `A`. `C` erbt außerdem von `B` und überschreibt die Methode `m()` von `A`. Die Methode `bar()` ruft die statische Methode `A.foo()` auf und abhängig vom Typ von `p` die Methode `m()`.

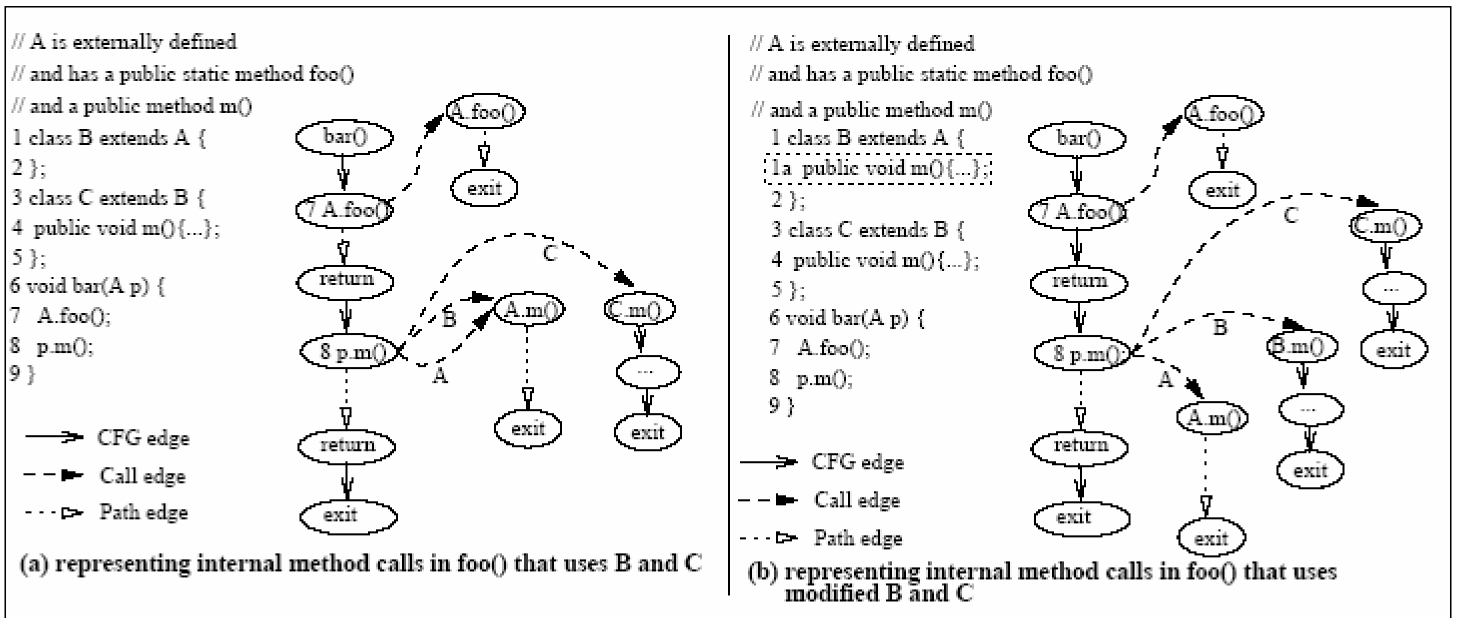

Abbildung 7: Beispieldarstellung eines internen Methodenaufrufs
Quelle: Harrold et al, 2001

Wenn der Methodenaufruf nicht virtuell ist, hat der aufrufende Knoten nur eine ausgehende `call`-Kante. In der Abbildung 7(a) ist das z.B. der Aufruf der Methode `A.foo()`. Da diese statisch ist und nicht dynamisch gebunden wird, existiert nur eine `call`-Kante von der Aufrufstelle zum Eingangsknoten der Methode `A.foo()`.
Ist der Methodenaufruf dagegen virtuell, d.h., wird die aufzurufende Methode erst zur Laufzeit bestimmt, ist der aufrufende Knoten und der Eingangsknoten aller in Frage kommender Methoden über eine `call`-Kante verbunden. Im JIG wird zum besseren Erkennen, welcher dynamische Typ angenommen werden muss, damit die Methode aufgerufen wird, der Typ an der `call`-Kante notiert. Ist zum Beispiel bei `p.m()` der Typ von `p` `A` oder `B` so wird `A.m()` aufgerufen, andernfalls beim Typ `C` die Methode `C.m()`. Deshalb hat der Konten `p.m()` im Beispiel 7(a) drei ausgehende `call`-Kanten. Um die in Frage kommenden Methoden zu ermitteln, gibt es verschiedene Ansätze. Der gängigste Algorithmus ist die Klassenhierarchieanalyse[[16]] welcher hier aber nicht weiter vorgestellt werden soll.
Abbildung 7(b) zeigt eine geänderte Version, in der die Klasse `B` die Methode `m()` von `A` überschreibt. Zu sehen ist, dass die `call`-Kante mit dem Typ `B` nicht mehr auf die Methode `A.m()` zeigt, sondern auf den Eingangsknoten der neu hinzugekommenen Methode `B.m()`. Dadurch erkennt der Algorithmus, dass sich das Ziel der ausgehenden `call`-Kante `B` geändert hat und fügt diese Kante zu den gefährlichen Kanten hinzu.

Interprozedurale Interaktionen durch externe Methoden
Der vorherige Abschnitt behandelte die Methodenaufrufe von innen, dieser Abschnitt behandelt nun die Methodenaufrufe von außen.

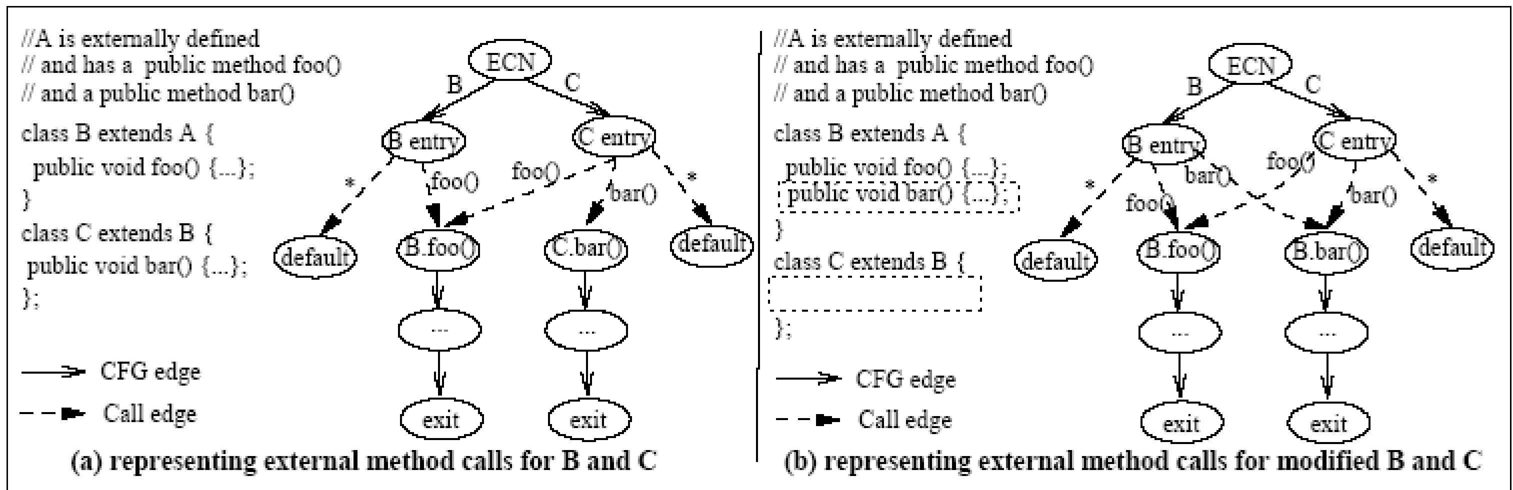

Abbildung 8: Beispiel eines externen Methodenaufrufs
Quelle: Harrold et al, 2001

Die aufrufende externe Methode wird durch einen „External-Code"-Knoten (`ECN`) dargestellt, wie in Abbildung 8 zu sehen ist. Für jede von einer äußeren Klasse zugängliche interne Klasse (im Beispiel die Klassen `B` und `C`) wird eine Kontrollflusskante vom `ECN`-Knoten zu allen zugänglichen Klasseneingangsknoten gezeichnet. Ein Klasseneingangsknoten wird zwischen dem `ECN`-Knoten und den Eingangsknoten jeder von außen zugänglichen Methode geschaltet, damit entschieden werden kann, ob eine Methode überschieben wurde oder nicht. Im ersten Fall wird die überschriebene Methode aufgerufen (`call`-Kante zur Methode), im zweiten Fall die Methode der Oberklasse, dargestellt durch die `call`-Kante mit „*" zum `default`-Knoten. Ein `default`-Knoten steht z.B. für alle Methoden der Klasse `A`, die durch ein Objekt `A` aufgerufen werden können, aber extern definiert sind. Hätte z.B. die Klasse `A` zusätzlich die Methode `x()`, erben alle Klassen diese Methode. Wird jetzt `B.x()` von außen aufgerufen, führt dies zur Methode `A.x()` über den Standardknoten, da diese nicht in `B` überschrieben wurde.
Das bedeutet, dass ein Klasseneintrittsknoten für jede Klasse erstellt werden muss, die mindestens eine externe Methode überschreibt. Das gleich muss für jede Klasse, die eine Methode erbt, die eine externe Methode überschreibt, gemacht werden. Im Abbildung 8(a) überschreibt z.B. die Klasse `B` die externe Methode `A.foo()` und benötigt somit einen Klasseneingangskoten. Dass gleiche gilt für Klasse `C`, die die Methode `A.bar()` überschreibt und `B.foo()` erbt, welche wiederum `A.foo()` überschreibt.

Die einzigen internen Methoden, die von Außen aus aufgerufen werden können, sind Methoden, die Methoden externer Klassen überschreiben. Diese Einschränkung (näheres dazu siehe Unterabschnitt Annahmen) muss gelten, da sie die Anzahl der möglichen Interaktionen, die betrachtet werden müssen, erheblich einschränkt. In den meisten Fällen ist diese Einschränkung aber nicht so schlimm, da externe Klassen meistens Bibliotheken-Klassen sind, die unabhängig sind und meistens vor der Entwicklung fertig gestellt wurden.

Durch dieses Vorgehen erkennt der Algorithmus externe Aufrufe von Methoden, die geändert wurden. Im Beispiel in Abbildung 8(b) wurde die Methode `foo()` von der Klasse `C` in die Klasse `B` verschoben. Der Algorithmus erkennt, dass sich das Ziel der `bar()`-Kante geändert hat und fügt diese zu den gefährlichen Kanten hinzu.

Exceptions
Eine weitere Ergänzung des „Graph Traversal"-Algorithmus ist die Behandlung von Exceptions, die in fast jeder modernen objektorientierten Programmiersprache vorkommen. Im JIG werden der `try`-Block, die `catch`-Blöcke und der `finally`-Block durch einen extra Knoten dargestellt. Abbildung 9(a) zeigt ein Beispiel, wie Exceptions im JIG behandelt werden.

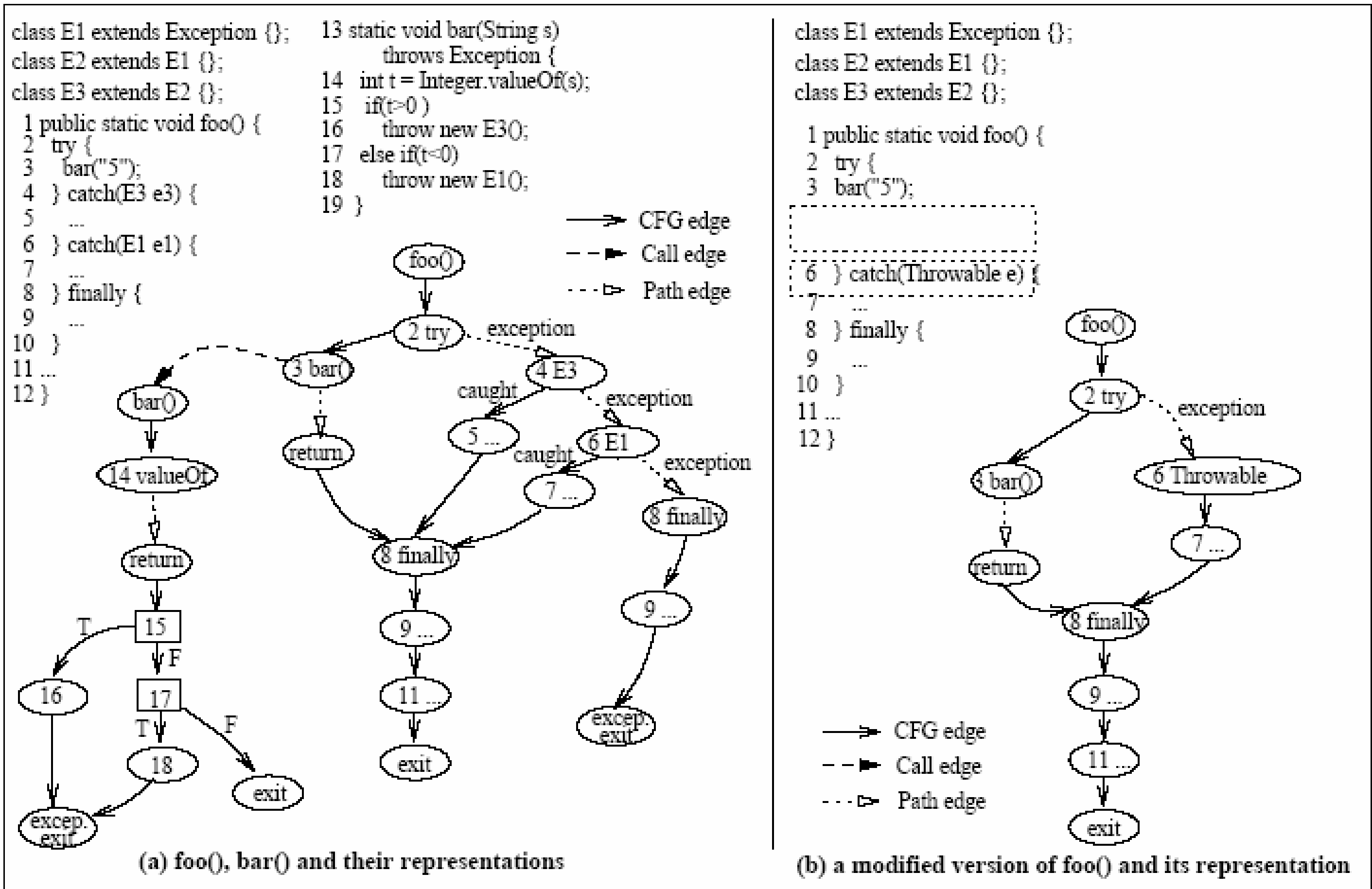

Abbildung 9: Exceptions im JIG
Quelle: Harrold et al, 2001

Für jede `try`-Anweisung wird ein `try`-Knoten im Kontrollflussgraphen erstellt. Die Anweisungen zwischen dem `try`-Block werden ebenfalls durch einen Kontrollflussgraphen dargestellt, der mit dem Knoten für die Anweisung zwischen dem `try`-Block über eine Kontrollflusskante verbunden ist.
Für jede `catch`-Anweisung wird ebenfalls ein eigner Knoten mit dem Typ der Exception als Name erstellt. Dieser Knoten wird mit einer Kontrollflusskante mit dem Namen `caught` mit dem Eingangsknoten des Kontrollflussgraphen verbunden, der die Anweisung(en) darstellt, die beim Auslösen der Exception ausgeführt werden soll(en). Die Pfadkante vom `try`-Knoten zum ersten `exception`-Knoten, die bis zum `finally`-Knoten weitergeht, stellt die Weiterreichung der Exception dar. Zum Beispiel die Pfadkante (2,4) in Abbildung 9(a) wird immer im Falle einer Exception in Zeile 16 und 18 überdeckt. Die Pfadkante (6, 8) wird nur überdeckt, wenn eine Exception ausgelöst wird, die von keinem `catch`-Block behandelt wird. Da der `finally`-Block immer ausgeführt wird, z.B., wenn eine Exception behandelt wurde, sind die `catch`-Knoten ebenfalls über eine Kontrollflusskante mit dem `finally`-Knoten verbunden. Der `finally`-Knoten ist auch mit dem Kontrollflussgraphen verbunden, der die Anweisungen nach dem `try`-Block darstellt, genauso wie mit dem Kontrollflussgraphen, der die Anweisungen zwischen dem `try`-Block darstellt.
Der `exception-exit`-Knoten stellt den Fall dar, dass die Exception nicht behandelt wurde und somit die Methode umgehend verlassen wird. Wäre in diesem Beispiel der `finally`-Block nicht vorhanden, wäre der letzte `catch`-Knoten mit dem `exception-exit`-Knoten über eine Pfadkante verbunden.
Durch diese Repräsentation der Exceptions kann der Algorithmus einfach Änderungen in der Exceptionbehandlung erkennen. Betrachtet man das Beispiel in Abbildung 9(b) erkennt man, dass in der Methode `foo()` alle `catch`-Anweisungen gelöscht worden sind und ein neuer `catch`–Knoten hinzugefügt wurde, der alle Exceptions abfängt. Durch den Vergleich der ausgehenden Kante des `try`-Knotens erkennt der Algorithmus, dass sich der Zielknoten geändert hat und fügt diese Kante zu den gefährlichen Kanten hinzu.

Annahmen: Damit das Verfahren effizient arbeiten kann, müssen folgende Annahmen erfüllt sein. Diese Annahmen schränken die Anwendbarkeit des Verfahrens aber nur in sehr bestimmten Fällen ein.

- **keine Verwendung von Reflection:** Durch Reflection erhält man Zugriff auf Felder, Methoden und Konstruktoren geladener Klassen und kann diese manipulieren. Solche Änderungen überall zu finden bedarf eines riesigen Aufwands und würde den Algorithmus ineffizient machen.
- **unabhängige externe Klassen:** Der Algorithmus geht davon aus, das externe Klassen ohne interne Klassen kompiliert werden können und das externe Klassen keine internen Klassen explizit durch einen Klassenlader laden. Dadurch wird sichergestellt, dass externe Klassen nur über definierte virtuelle Methoden mit internen Klassen interagieren. Außerdem wird dadurch die Anzahl der Interaktionstypen beschränkt, die der Algorithmus analysieren muss (siehe z.B. „Interprozedurale Interaktionen durch externe Methoden“).
- **deterministische Testläufe:** Jeder unveränderte Testlauf muss deterministisch sein, d.h. das unter gleichen Vorbedingungen die gleichen Ergebnisse erzielt werden müssen. Diese Einschränkung stellt sicher, dass die Überdeckungsinformationen nicht von einem spezifischen Testlauf abhängen und dass das Ergebnis bei der Ausführung eines Testfalls, der modifizierte Stellen im Code nicht überdeckt, beim Original und der veränderten Version gleich ist.

Bewertung:

- **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
- **„Precision“:** gut
- **„Efficiency“:** gut (bei o.g. Annahmen), da externe Klassen (z.B. Bibliotheken) nicht analysiert werden müssen
- **„Generality“:** gut, das Verfahren unterstützt alle Konstrukte von objektorientierten Programmiersprachen (Exceptions, Polymorphie, usw.) und ist nicht nur für Java anwendbar

3. Fazit

Um exemplarisch zu zeigen, in welchem Ausmaß durch die Selektion von Testfällen der Testaufwand reduziert werden kann, möchte ich eine empirische Studie für das Verfahren von Harrold et al[[17]] vorstellen. Die Technik wurde an verschiedenen Versionen von SIENA (internetbasiertes Ereigniserinnerungssystem), JEdit (ein Text-Editor geschrieben in Java), Apache JMeter (Javaanwendung für den Lasttest von Webanwendungen) und RegExp (eine GNU-Bibliothek für reguläre Ausdrücke) angewendet. Tabelle 3 zeigt für jedes Testobjekt die Anzahl der Methoden, die Anzahl verschiedener Versionen und die Anzahl der Testfälle vor den Änderungen. Die verschiedenen Versionen wurden entweder von den Entwicklern selbst geliefert oder künstlich durch gezielte Änderungen erzeugt.

Programm	Beschreibung	#Methoden	#Versionen	#Testfälle	Methoden-überdeckung
Siena	Internetbasiertes Ereigniserinnerungssystem	185	7	138	70%
JEdit	Texteditor	3495	11	189	75%
JMeter	Web-Anwendungen testen	109	5	50	67%
RegExp	Bibliothek für reguläre Ausdrücke	168	9	66	46%

Tabelle 3: Testobjekte zum Testen des Selektionsverfahrens
Quelle: Harrold et al, 2001

Die folgende Abbildung zeigt das Ergebnis der Studie. In den Fällen, in denen relativ wenige Testfälle ausgewählt wurden (z.B. V6 bei Siena), waren die Änderungen relativ gering, sodass nur wenige Methoden betroffen waren und zum wiederholten Testen wenige Testfälle benötigt wurden.

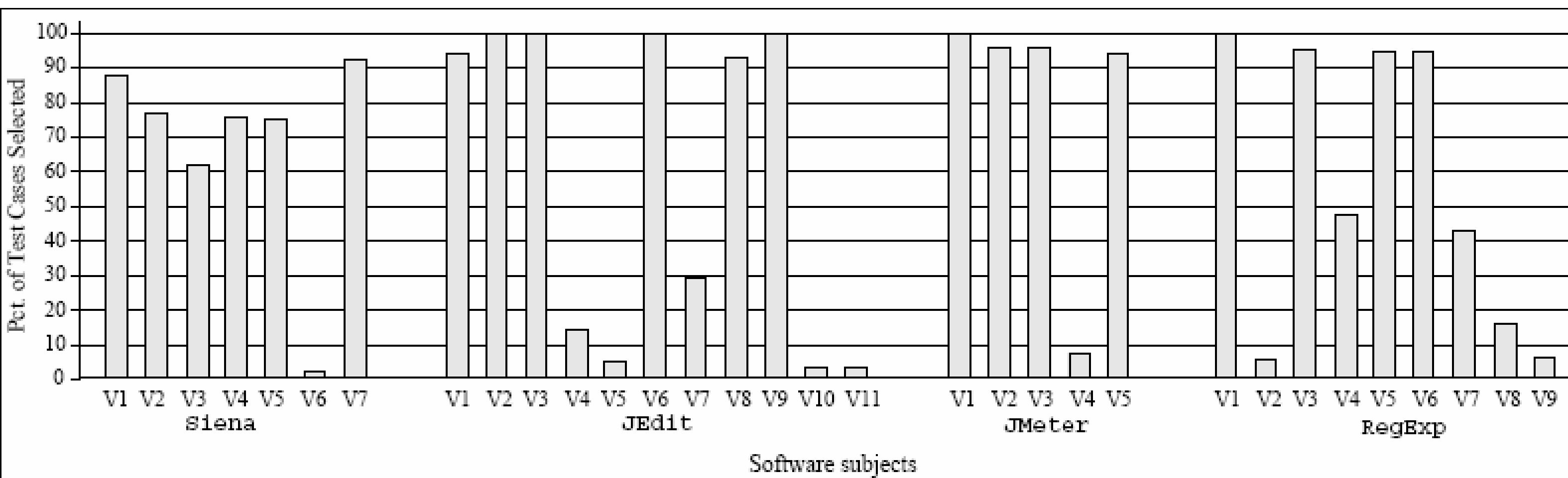

Abbildung 10: Prozentsatz d. ausgewählten Testfälle für die verschiedenen Versionen
Quelle: Harrold et al, 2001

Die Abbildung zeigt außerdem, dass die Anzahl der Testfälle, die ausgewählt wurden, sehr stark innerhalb des einzelnen Testobjekts, sowie über alle Testobjekte variieren kann. Durch das Verfahren ergab sich gemittelt eine Ersparnis von 33% für Siena, 41,6% für JEdit, 21,8% für JMeter und 43,9% für RegExp. Wie diese Studie zeigt, können Selektionsalgorithmen zwar nicht massiv (im Sinne von 75% und mehr) Testfälle verringern, in Umgebungen, in denen die Kosten für Testfälle sehr hoch sind, z.B. das Testen von Software in der Luftfahrt, kann schon das Weglassen eines einzelnen Testfalles bereits Tausende von Euro sparen.
Es sollte jedem klar sein, dass solche empirischen Studien nicht für alle Programme und Programmänderungen gelten. Vielmehr hängt die Effizienz solcher Selektionstechniken überwiegend vom Ort und dem Ausmaß der gemachten Änderungen ab. Bei einer Änderung einer Anweisung ganz am Anfang einer Methode, durch die alle Testfälle zwangsläufig durchmüssen, gibt es keine andere Möglichkeit als alle Testfälle auszuwählen (siehe in Abbildung 10, z.B. V2 und V3 von JEdit). Im Gegensatz dazu kann die Änderung einer Methode in der Mitte bzw. am Ende der Methode durch relativ wenige Testfälle abgedeckt werden (siehe Abbildung 10, z.B. V6 bei Siena und V10, V11 bei JEdit).

4. Werkzeuge

4.1. Mercury: WinRunner:

- Bietet funktionale Tests und Regressionstests
- Kann Aktionen aufzeichnen und abspielen
- Unterstützt die Sprachen Visual Basic, Java, Delphi
- WinRunner QuickStart Standard Seminar: 9900€ für 5 Tage

4.2. AutomatedQA: TestComplete

- Bietet automatisierte funktionale Tests (UI), Modul-, Regressions-, verteilte Tests und HTTP-Last- und Stresstests, Datenbanktest und „Data-Driven“ Test in einem Werkzeug
- Unterstützt die Sprachen .NET, Java, Visual Basic, C++, Delphi und Web
- Für die gebotene Funktionalität recht günstig: ab 350$
- Die Firma AutomatedQA hat für ihre Tools viele Auszeichnungen bekommen

4.3. Lehrstuhl Informatik 11: aRTS – a Regression Test Selection Tool

- Implementiert das Selektionsverfahren von Harrold et al für Java
- Reines Selektionstool

Literaturverzeichnis

[1] Cem Karner, Jack Falk, Hung Quoc Nguyen, „Testing Computer Software“, S. 49-50, 1993
[2] John J. Marciniak, „Encyclopedia of Software Engineering“, S. 1039-1040, 1994
[3] Francesca Saglietti, „Grundlagen des Software Engineering“, Teil XXI, S. 21, 2004
[4] Ernest Wallmüller, „Software Qualitätssicherung“, S. 211, 1990
[5] Alec Sharp, „Software Quality and Productivity“, S. 258, 1993
[6] Cem Karner, Jack Falk, Hung Quoc Nguyen, „Testing Computer Software“, S. 50, 1993
[7] Peter Liggesmeyer, „Software Qualität“, S. 187-189, 2002
[8] Thomas J. Ostrand, Elaine J. Weyuker, „Using data flow analysis for regression testing“, 1988 - Mary Jean Harrold, Rajiv Gupta, Mary Lou Soffa „A methodology for controlling the size of a test suite“ s. 270–285, 1993 - Mary Jean Harrold, Mary Lou Soffa, „An incremental approach to unit testing during maintenance“, S. 362–367, 1988
[9] Yih-Farn Chen, David S. Rosenblum, Kiem-Phong Vo, „TestTube: a system for selective regression testing“, S. 2, 1994
[10] John J. Marciniak, „Encyclopaedia of Software Engineering“, S. 1041, 1994
[11] Cem Karner, Jack Falk, Hung Quoc Nguyen, „Testing Computer Software“, S. 140-141, 1993
[12] Benjamin Polak, „Regressionstest für Java-Software“, S. 3-6, 2004
[13] Yih-Farn Chen, David S. Rosenblum, Kiem-Phong Vo, „TestTube: a system for selective regression testing“, 1994
[14] Lee White, Khalil Abdullah, „A Firewall Approach for the Regression Testing of Object-Oriented Software”, 1997
[15] Gregg Rothermel, Mary Jean Harrold, „Regression Test Selection for C++ Software”, 2000
[16] Jeffrey Dean, David Grove, and Craig Chambers, „Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis“, S. 77-101, 1995
[17] Marry Jean Harrold et al, „Regression Test Selection for Java Software“, S. 11-13, 2001