Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Diplomarbeit

Ein Werkzeug zur Überdeckungsmessung für kontrollflussbezogene Testverfahren

eingereicht von Hendrik Seffler

Berlin, den 14. Dezember 2004

Betreuer: Prof. Dr. Klaus Bothe

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den 14. Dezember 2004 Hendrik Seffler

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 14. Dezember 2004 Hendrik Seffler

Zusammenfassung

Die vorliegende Diplomarbeit ist im Rahmen des Projektseminars *Software-Sanierung* am Lehrstuhl Software-Technik des Instituts für Informatik der Humboldt-Universität zu Berlin entstanden.

Das Projektseminar hat ein umfangreiches Reengineering-Vorhaben der Software *XCTL* zur Untersuchung von Halbleiterstrukturen zum Thema. Aufgabe der Software bei dieser Untersuchung ist die Steuerung externer Geräte, die Messungen durchführen, und die Auswertung der gemessenen Werte. Das Projekt entstand in Folge einer Anfrage des Instituts für Physik im Jahr 1998.

Im Rahmen dieses Projektes sind bisher umfangreiche Reengineering-Maßnahmen an dem beschriebenen Softwaresystem vorgenommen sowie zusätzliche Funktionalität hinzugefügt worden. Es entstand im Laufe dieser Tätigkeiten der Bedarf nach einer Lösung, die den Test des Gesamtsystems und von Komponenten erleichtert und unterstützt, um sicherzustellen, dass Änderungen an Teilsystemen oder Ergänzung neuer Subsysteme die bestehende Funktionalität nicht beeinträchtigen.

Aufgrund dieser Anforderungen entstand bereits das Testsystem *ATOS*, das funktionsbasierte Tests auf der Basis von Spezifikationen zu automatisieren hilft. ATOS bietet dabei sowohl Unterstützung bei der Erstellung als auch bei der Auswertung von Tests, indem es XCTL kontrolliert ausführt, Eingaben vorgibt und die Ausgaben aufzeichnet und auswertet.

Im Rahmen dieser Arbeit ist ein System erstellt worden, das ATOS ergänzt und es erlaubt, den Umfang der Tests auszuweiten. ATOS untersucht nur nach außen hin sichtbares Verhalten eines Softwaresystems. Ziel dieser Arbeit ist die Erstellung eines Testsystems, dass Aussagen über die Struktur des zu untersuchenden Softwaresystems zulässt und die Testfälle, die für ATOS erstellt wurden, auf verschiedene Hinlänglichkeitskriterien überprüft.

Dazu wird die zu untersuchende Software auf Quellcodeebene geparst und ausgewertet und um zusätzliche Befehle ergänzt, die ein beobachten des Verhaltens zur Laufzeit ermöglichen (*Instrumentierung*).

Schwerpunkt der Darstellung ist vor allem die technische Umsetzung dieses Systems. Dies umfasst eine Beschreibung der verschiedenen Phasen der Erstellung inklusive Entwurf, Implementierung und Test. Zusätzlich werden die theoretischen Grundlagen, nach denen Testfälle und deren Kombinationen bewertet werden, aufgearbeitet und, wo nötig, den Erfordernissen angepasst bzw. entsprechend erweitert. Dargestellt wird auch, wie sich der funktionsorientierte Test, wie ihn ATOS leistet, mit den Ansätzen des strukturorientierten Tests kombinieren lässt, um weitergehende Aussagen zu erzielen.

Dies umfasst natürlich einen Test des Systems, um die korrekte Funktion des Gesamtsystems zu überprüfen. Besonderes Augenmerk wird dabei auf den Vergleich mit bekannten Referenz-Daten gelegt. Zu einem vollständigen System gehört ebenfalls Dokumentation für spätere Nutzer, wobei Tester im Rahmen komplexer Projekte, die lange mit dem System arbeiten, von den Nutzern abzugrenzen sind, die die Software im Rahmen etwa einer universitären Lehrveranstaltung kennen lernen.

Den Abschluss der Arbeit bildet eine Bewertung des erstellten Softwaresystems sowie eine Einschätzung der Möglichkeiten der Erweiterung und der Verbesserung.

Überblick über die Kapitel

Das erste Kapitel gibt einen Überblick über das Software-Sanierungsprojekt, in dessen Rahmen die Idee und die Grundlagen für diese Arbeit gelegt wurden. Dazu werden unter anderen das System XCTL, mit dem sich das Sanierungsprojekt beschäftigt, und die eigenentwickelte Testumgebung ATOS vorgestellt.

Das zweite Kapitel ordnet ATOS und das für diese Arbeit erstellte Testsystem SBTWAN in den größeren Zusammenhang des Softwaretests ein. Die gängigen und wesentlichen Ansätze, Software zu testen, werden kurz vorgestellt und in eine Systematik eingeordnet. Black- und White-Box-Test werden gegenüber gestellt und ihre mögliche Kombination erläutert.

Im dritten Kapitel werden die theoretischen Grundlagen für die Betrachtung der Struktur von Modulen in Softwaresystemen gelegt, indem das Konzept der Kontrollflussgraphen eingeführt wird und wesentliche, für die spätere Auswertung wichtige, Eigenschaften vorgestellt werden. Auf Basis dieser Definitionen lassen sich Komplexitätsmaße zur Beurteilung von Softwaremodulen ableiten.

Dynamische Maße werden im vierten Kapitel erläutert. Im Gegensatz zu den Maßen des dritten Kapitels, die statisch ermittelt werden, erfordern diese eine Instrumentierung und anschließende Ausführung des zu untersuchenden Systems. Es werden verschiedene Überdeckungsmaße vorgestellt, einschließlich derer zur Bedingungs- und Pfadüberdeckung.

Das fünfte Kapitel fasst die funktionalen und nicht-funktionalen Anforderungen an die Software zusammen und erläutert Details der gestellten Anforderungen. Diese werden im Pflichtenheft ausführlich dargelegt und nach dem Use-Case-Ansatz strukturiert.

Im sechsten Kapitel wird das dem zu entwickelnden System zugrundeliegende Design dargestellt und die Grundsätze der Architektur vorgestellt. Die in der Anforderungsdefinition gestellten Anforderungen werden von der Seite ihrer technischen Realisierbarkeit betrachtet und mögliche Implementierungsansätze vorgestellt.

Das siebente Kapitel beschreibt das Vorgehen bei der Implementation und die Probleme, die sich dabei ergaben. Dies umfasst auch eine Betrachtung der verwendeten Entwicklungsumgebung und weiterer Werkzeuge, die zum Einsatz kamen.

Das achte Kapitel beschreibt den Test des Systems und wie dieser durchgeführt wurde. Die verwendeten Testansätze werden erläutert und deren Ergebnisse bewertet, um schließlich zu Aussagen über das Gesamtsystem zu kommen.

Im neunten Kapitel findet sich eine Dokumentation für Nutzer von SBTWAN, die alle wesentlichen Aspekte der Nutzung von der Projekterstellung über die Konfiguration bis zur Auswertung der Daten abdeckt und darstellt. Wo es nötig ist, werden die Erklärungen durch Bespiele ergänzt.

Das zehnte Kapitel schließlich gibt eine Beurteilung dessen, was in der Diplomarbeit erreicht wurde und macht Vorschläge und führt Ideen aus, wie man auf der bestehenden Basis weitere Funktionalität realisieren könnte.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation: Das Softwaresanierungs-Projekt

Die Motivation zur Entwicklung des in dieser Arbeit dokumentierten Softwaresystems zum strukturorientierten Softwaretest entstand aus dem Umfeld des Softwaresanierungs-Projekts, das seit dem Sommer 1998 am Lehrstuhl *Softwaretechnik* von Prof. Bothe am Institut für Informatik durchgeführt wird. Dieses Projekt befasst sich mit dem Erweiterung, Wartung, Restrukturierung und Dokumentation einer Steuersoftware zur Röntgendiffraktometrie.

Diese Software namens XCTL wird am Institut für Physik in der AG *Röntgenbeugung an dünnen Schichten* eingesetzt, die unter der Leitung von Prof. Köhler steht. Dort wird die Software zur Steuerung eines speziellen Versuchsaufbaus genutzt, der der Untersuchung von kristallinen Halbleiterstrukturen dient.

Ursprünglich wurde diese Software durch einen ehemaligen Mitarbeiter dieser Arbeitsgruppe erstellt, wobei grundlegende Ansätze des Software-Engineering keine Verwendung fanden. Mit zunehmendem Alter der Software und dem Verlangen, auch neuere Messapparaturen in das bestehende System einzubinden, entstand der Bedarf, das bestehende System zu erweitern.

Aus dieser Situation entstand das Softwaresanierungs-Projekt unter Prof. Bothe, das rasch über seinen kleinen Beginn hinauswuchs, als der Zustand des ursprünglichen Softwaresystems klar wurde. Es entstand ein umfangreiches Reengineering-Projekt, in dessen Verlauf die Software in großen Teilen umstrukturiert und dokumentiert wurde. Zusätzlich wurden neue Funktionen eingefügt, wie etwa die automatische Justage [SF01].

In Laufe dieser Maßnahmen wurde tief in die ursprüngliche Struktur des bestehenden Systems eingegriffen. Vordringlicher Wunsch der Nutzer am Institut für Physik war dabei natürlich, dass die bestehende Funktionalität erhalten blieb und keine neuen Fehler entstanden. Die Entwicklung eines entsprechenden Testregimes wurde dadurch notwendig.

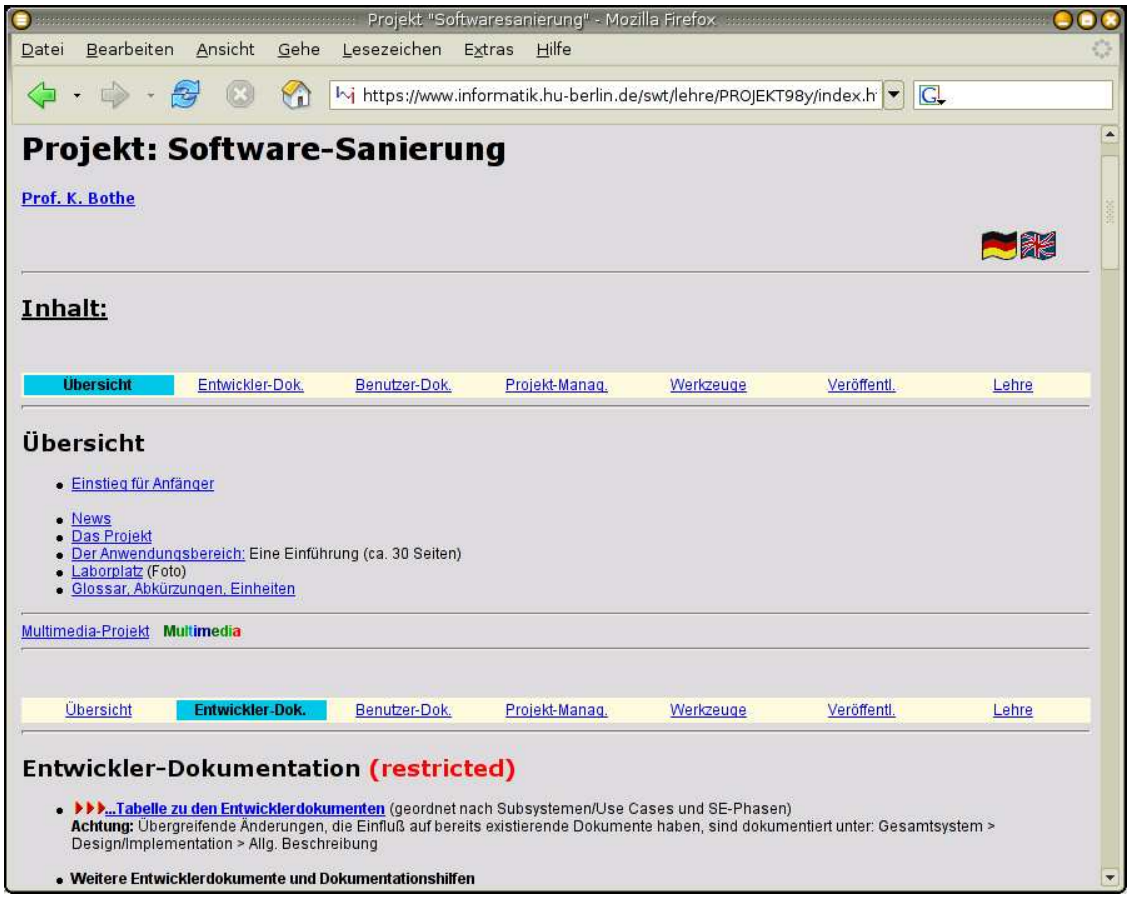


Abbildung 1.1: Die Web-Seite des Softwaresanierungs-Projekts

1.2 Ziele der Diplomarbeit

Aus diesem Anforderungen erwuchs der Wunsch, ein umfassendes Testsystem für den Test des Systems XCTL zu entwickeln, das wichtige Elemente des Softwaretests beinhaltet.

Im Rahmen dieser Arbeit wird die Entwicklung des Softwaresystems *SBTWAN* beschrieben, das die Durchführung von nicht-funktionalen Tests unterstützt, indem es die Ausführung eines Programms nach verschiedenen Kriterien überwacht und bewertet, um Aussagen über die Vollständigkeit von Tests im Bezug auf die Programmstruktur zu gewinnen. Darüber hinaus erlaubt SBTWAN, die Komplexität des vorliegenden Quellcodes eines Programms nach verschiedenen Kriterien zu beurteilen, um potentiell fehleranfällige Programmabschnitte aufzufinden.

Dazu befasst sich ein großer Teil dieser Arbeit mit den theoretischen Grundlagen dieser Komplexitätsmaße und der Art und Weise, wie man die Vollständigkeit von Tests im Bezug auf die Programmstruktur messen und bewerten kann. Es werden entsprechende Maße vorgestellt, sogenannte Überdeckungsmaße.

Als Sprachen werden Java und C++ unterstützt werden. Um ein reibungsfreies Funktionieren zu gewährleisten, muss ein Weg gefunden werden, um die Syntax und Struktur beider Sprache in eine möglichst einheitliche Struktur abzubilden, um daraus weitere Informationen gewinnen zu können.

Da nicht-funktionale Softwaretests nur einen Teil der sinnvollen Testabläufe abbilden, ist eine Kombination eines entsprechenden Systems mit einem funktionsorientierten System sinnvoll, dass nicht die Struktur bewertet, sondern auch die Einhaltung von Spezifikationen und Funktionsbeschreibungen überprüft.

1.2.1 ATOS

Aus diesem Erwägungen entstand bereits vor dieser Arbeit ein funktionsorientiertes Testsystem mit dem Namen *ATOS* (Automatisierter Test oberflächenbasierter Systeme) [JH02], dass eine weitgehende Automatisierung des Testablaufs gestattet.

ATOS ist ein Regressionstestsystem, das genutzt werden kann, um ein sich während der Entwicklung änderndes System auf Korrektheit und Funktionalität zu überprüfen.

ATOS wertet die Komponenten einer grafischen Oberfläche wie der von XCTL aus und ermöglicht über eine eigene Skript-Sprache die Steuerung des ablaufenden Programms, so dass einzelne Testläufe beliebig wiederholbar sind. XCTL legt während seiner Arbeit Protokoll-Dateien seines Ablaufes an und erzeugt ggf. grafische Darstellungen.

ATOS ist nun in der Lage mittels seiner Komponente *DataDiff* die Aufzeichnungen von XCTL mit vorher abgelegten Referenzen zu vergleichen, wobei für Übereinstimmungen auch gewisse Toleranz-Bereiche festgelegt werden können (Schwankungen der Messwerte sind im konkreten Fall unvermeidbar), um korrekte Funktion zu überprüfen. Grafische Darstellungen müssen dabei manuell verglichen werden und Übereinstimmung dem ATOS-System mitgeteilt werden.

ATOS wird im Rahmen des Softwaresanierungs-Projektes regelmäßig eingesetzt und liefert wertvolle Erkenntnisse zur Behebung von Fehlern. Darüber hinaus er-

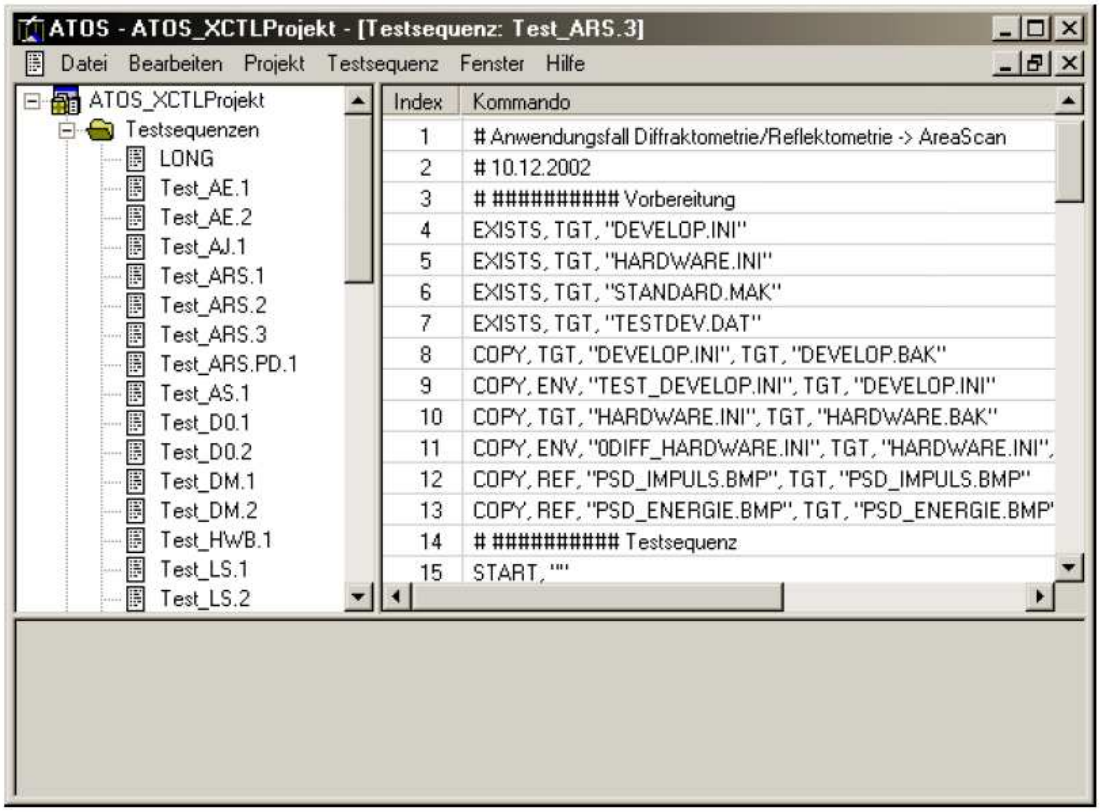


Abbildung 1.2: Das Hauptfenster von ATOS

möglicht es auch, die Spezifikation, aus der die Testfälle gewonnen werden, auf ihre Genauigkeit zu überprüfen.

1.2.2 Einsatz in der Lehre

Neben dem Einsatz im Rahmen des XCTL-Projektes ist auch ein Einsatz des Programms in der Lehre an der Humboldt-Universität beabsichtigt, etwa bei Vorlesungen zum Thema Software-Engineering durch die praktische Ermittlung der im weiteren Verlauf dieser Arbeit erläuterten Komplexitätsmaßzahlen und Überdeckungsmaße, um ein besseres Verständnis für die Materie zu entwickeln.

Im Rahmen des Projekt-Einsatzes wird die Untersuchung eines Softwaresystems über einen längeren Zeitraum erfolgen, so dass Zeit besteht sich mit den Werkzeugen, die dafür zum Einsatz kommen, umfassend auseinander zu setzen. Eines dieser Werkzeuge kann das Tool zur Überdeckungsmessung sein, das im Rahmen dieser

Arbeit entsteht. Dabei steht nicht nur eine kurze Einarbeitungszeit im Vordergrund, sondern vor allem der gründliche Test des betrachteten Softwaresystems.

Beim Einsatz in der Lehre, etwa in einer Einführungsveranstaltung, die viele Themenbereiche umfasst, sind andere Schwerpunkte zu setzen. Das Erlernen und Verstehen von Metriken und Überdeckungskriterien stehen im Vordergrund und Werkzeuge sind vor allem Hilfsmittel, die einfach zu verstehen und zu handhaben sein müssen, um nicht vom eigentlichen Gegenstand der Betrachtung abzulenken.

Einfache, intuitive Bedienung, die die wesentlichen Arbeitsprozesse einfach, nachvollziehbar und übersichtlich gestaltet, ist daher neben einem vollständigen Abdecken der funktionalen Anforderungen von besonderer Bedeutung.

1.3 Überblick über die Ergebnisse der Arbeit

Durch das System SBTWAN ist eine Grundlage entstanden, um die Struktur von Softwaresystemen zu untersuchen. Durch den verwendeten Parser, der verschiedene Sprachen auf *eine* Datenstruktur abbildet, ist eine Erweiterung einfach möglich.

Basierend auf dieser Datenstruktur sind Auswertungsalgorithmen entstanden, die statische und dynamische Programmauswertung erlauben. Dabei wurden drei Schwerpunkte gesetzt. Für die anschauliche Darstellung und auch die interne Darstellung erstellt SBTWAN Graphen, die Kontrollflüsse repräsentieren. Diese müssen aus der Darstellung der Programmsyntax gewonnen werden. Auf dieser Basis können Komplexitätsmaße ermittelt werden, wobei verschiedene Graph-Algorithmen zum Einsatz kommen, um die gewünschten Ergebnisse zu erzielen.

In der abstrakten Darstellung der Programmstruktur sind darüber hinaus Informationen eingebettet, die es möglich machen, einen konkreten Programmdurchlauf nachzuvollziehen. Davon ausgehend gestattet es SBTWAN, diese Aufzeichnungen in verschiedener Art und Weise darzustellen und auszuwerten.

Dabei wird durch ein überschaubares Design die Möglichkeit eröffnet, in Zukunft weitere Analysen in das SBTWAN-System zu integrieren.

2 Qualitätssicherung bei Softwaresystemen

2.1 Entstehung des Softwaretests

Die Entwicklung der Disziplin des Softwaretests hat sich zusammen mit der Entstehung des Software-Engineering, also der Systematisierung der Softwareentwicklung, vollzogen. Mit der Entwicklung hin zu programmierbaren Computersystemen, die jeweils wechselnden Aufgaben angepasst werden konnten, entstand auch die Software-Entwicklung. Zu Beginn wurden zunächst kleine Programme erstellt, die in ihrer Funktion und in ihrem Umfang eng gefasst waren und spezielle, gut überschaubare Aufgaben erfüllten. Mit dem Anstieg der Leistungsfähigkeit der verwendeten Rechnersysteme wuchs die Software heraus aus der Rolle einer reinen Beigabe zur Hardware und wurde als eigenständiges Produkt wahrgenommen.

Mit der zunehmenden Wichtigkeit der Software unabhängig von der Hardware und deren größerer finanzieller Bedeutung wuchs auch das Risiko, das mit der Entwicklung neuer Softwaresysteme eingegangen werden musste. War Softwareentwicklung zunächst etwas, das als eine Art Kunst aufgefasst wurde, in deren Entstehensprozess aus nicht näher festgelegten Anforderungen ein fertiges Programm entstand, so wandelte sich dieses Verständnis grundlegend. Die Anforderungen und damit der Aufwand, der getrieben werden musste, wuchsen stetig. Immer größere Projekte mit größerer Zahl an Entwicklern verlangten nach einer Festlegung eines Prozesses, in dessen Rahmen Softwareentwicklung stattzufinden hatte.

Mit der wachsenden Bedeutung und Standardisierung des Computereinsatzes entwickelte sich das Verlangen, die Softwareentwicklung zu einem ingenieursmäßigen Vorgang zu machen, d.h. zu einem Vorgang, der systematisch, quantifizierbar und wiederholbar ist.

Eine verwirrende Fülle von Vorschlägen zur Gestaltung des Prozesses der Softwareentwicklung entstand. Diesen Prozessmodellen gemein ist insbesondere, dass sich das Augenmerk von der Implementierung der Software in einer speziellen technis-

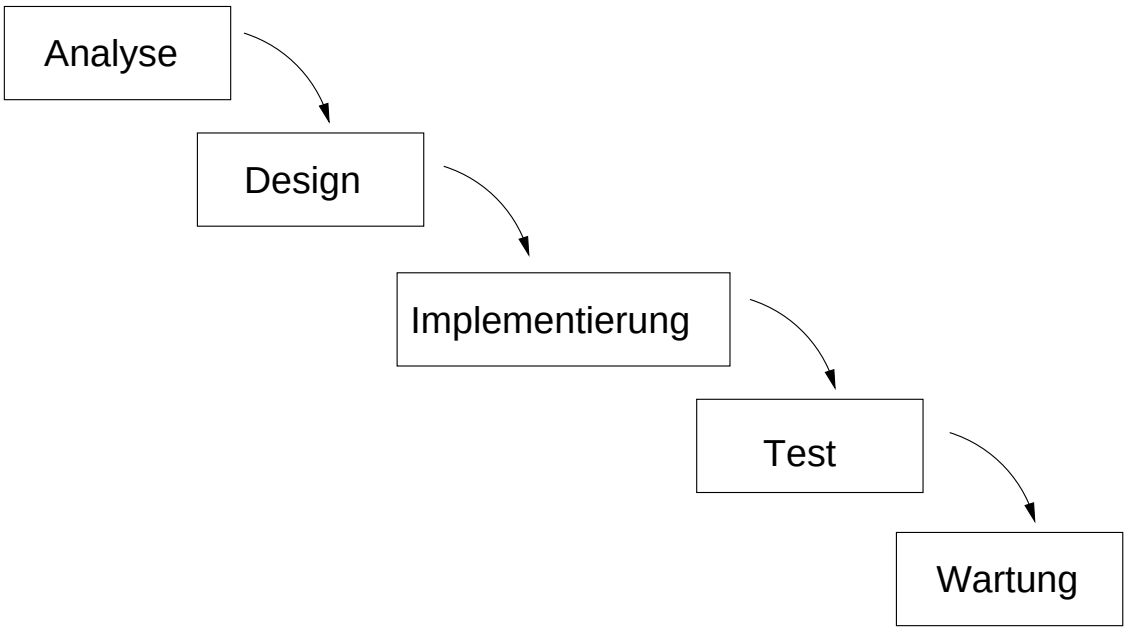


Abbildung 2.1: Das klassische Prozessmodell: das Wasserfallmodell

chen Umgebung auf die vor- und nachgelagerten Vorgänge verschob. Das klassische Modell der Softwareentwicklung etwa, das Wasserfallmodell (siehe Abbildung 2.1), setzt vor die Implementierung Vorgänge zur Problemdefinition und zum Entwurf des Softwaresystems. Je umfangreicher diese Modelle und Dokumente sind, desto dringlicher ist der Bedarf nach einer Methodik, die überprüft, ob die spezifizierten Ansprüche an ein Programm auch tatsächlich erfüllt werden. Vollständigkeit, Robustheit und Korrektheit müssen folglich in einer Testphase überprüft werden.

Es bleibt die Frage, wie man diesen Softwaretest durchführen kann und welche Methodenkombination sicherstellt, dass Entwurf und Programm übereinstimmen und vorher festgelegte Qualitätsanforderungen erfüllt werden.

2.2 Systematik gängiger Qualitätssicherungsverfahren

Prozessmodelle sollen auch die Menge von Fehlern, die in der Testphase gefunden werden müssen, minimieren. Durch einen festgelegten Entwurfsprozess wird versucht, die Entwicklung eines Softwaresystems einer Systematik zu unterwerfen.

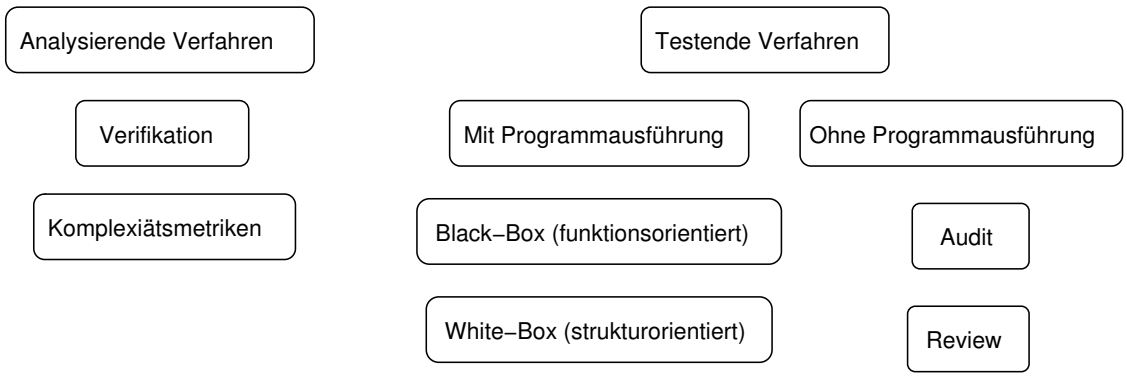


Abbildung 2.2: Übersicht über wesentliche Verfahren der Qualitätssicherung

Dabei soll durch konstruktive Maßnahmen von vorne herein versucht werden, Fehler zu vermeiden und Fehlerquellen möglichst frühzeitig zu erkennen und zu umgehen.

Mögliche Verfahren, die im Folgenden vorgestellt werden, sind in Abbildung 2.2 zu einer Übersicht zusammengestellt.

Der Schwerpunkt dieser Arbeit liegt auf der *nach* der Implementierung von Teilsystemen bzw. *nach* der des Gesamtsystems zu erfolgenden analytischen Bewertung des entstandenen Systems. Die Systematik der hier vorgestellten Verfahren ist an [Bal98] orientiert.

2.3 Analysierende Qualitätssicherung

Die abstrakteste Herangehensweise ist die Verifikation eines Softwaresystems oder von Teilen eines Systems gegen eine formale Spezifikation. Durch ein mathematisch präzise beschriebenes Verfahren, dessen Ergebnisse formal beweisbar sind, wird hier eine hohe Qualität der Ergebnisse erreicht. Formale Verifikation erfordert allerdings bei der Erstellung der Spezifikation und bei der Bewertung der Einhaltung dieser Spezifikation einen sehr hohen Aufwand, der für ein großes, komplexes Softwaresystem nur selten angemessen ist. Je nach Einsatzbereich kann es aber sinnvoll sein, besonders kritische Teile bzw. zentrale Kontrollstrukturen mit solchen Mitteln zu überprüfen. Als Beispiele für diese Herangehensweise sind etwa die Spezifikationssprache *Z* [Spi92] oder TLA [Lam03] zu nennen.

Im Allgemeinen wird man sich auf weniger präzise, weniger formale Testmethoden beschränken müssen, die sich mit weniger Aufwand durchführen lassen, aber dennoch angemessen gründlichen Test sicherstellen.

Viele der Schwächen und Fehler, die bei der Entwicklung von Softwaresystemen auftreten, wiederholen sich und lassen sich durch Analyse des Quellcodes verdeutlichen.

Bei solchen Verfahren wird das zu untersuchende Programm nach einem bestimmten Kriterium aufbereitet, um bestimmte Merkmale oder Strukturen, die fehleranfällig sind, passend hervorzuheben. Exemplarisch dafür sind Metriken, die Programme bzw. einzelne Funktionen auf Maßzahlen verdichten, die jeweils ein einzelnes Kriterium ausdrücken und dann mit einer Interpretation belegt werden.

Einfache Verfahren dieser Art sind rein zählende Verfahren, angefangen bei Lines-Of-Code (LOC)-Metriken, die schlicht die Zahl von Zeilen eines Programms messen, über Metriken, die die Mengen bestimmter Konstrukte erfassen und bewerten. Solche vergleichsweise wenig abstrahierenden Methoden hängen allerdings stark von Programmiergewohnheiten und anderen wechselnden, subjektiven Faktoren ab, so dass eine Interpretation schwierig und stark auf den jeweiligen Anwendungsbereich bezogen erfolgen muss.

Metriken, die auf höherem Abstraktionsniveau arbeiten, versuchen, sich von solchen Einflüssen zu befreien. Hier sind etwa Metriken zu nennen, die auf einer Aufwertung des Kontrollflusses basieren. Kontrollflüsse von Funktionen stellen Befehle als Knoten eines Graphen und mögliche Programmabläufe als Kanten zwischen diesen Knoten dar (siehe Abbildung 2.3).

Auf diesen Graphen lassen sich diverse Metriken definieren, etwa die Zählung der Zahl der möglichen Durchläufe durch eine Funktion, die der Zahl der Pfade durch den Kontrollflussgraphen entspricht oder eine Bewertung im Bezug auf die Unstrukturiertheit eines Programms. Solche Maße dienen der Bewertung der Komplexität einer Funktion. Ziel solch eines Vorgehens ist es, die Teile eines Softwaresystems zu identifizieren, die besonders komplex und damit besonders fehleranfällig bzw. auch wartungsintensiv sind.

2.4 Eigentliche Testverfahren

Unter den eigentlichen Testverfahren versteht man die Verfahren, bei denen nicht eine formale Spezifikation oder eine abstrakte Darstellung der Programmlogik einer

```
void example(boolean a, boolean b)
{
    if (a)
        do();
    if(b)
        doAlso();
}
```

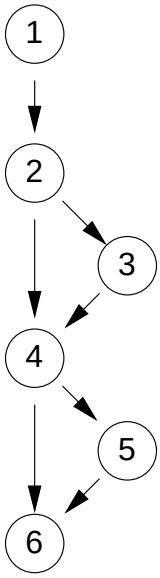


Abbildung 2.3: Beispiel eines einfachen Kontrollflussgraphen

Bewertung unterzogen werden, sondern das Softwaresystem nach seiner Implementierung in einer Programmiersprache.

Zu unterscheiden sind Vorgehensweisen, die ohne tatsächliche Ausführung des Softwaresystems (also statisch) arbeiten, von denen, die das Programm tatsächlich zur Ausführung bringen und unter bestimmten Rahmenbedingungen durchlaufen.

Unter den statischen Vorgehensweisen finden sich insbesondere Review- bzw. Audit-Vorgänge, bei denen der Quellcode systematisch überprüft und durchgesehen wird, um Fehlerfreiheit an sich und auch Vorhandensein und Korrektheit von geforderten Funktionen zu überprüfen.

Bei denjenigen Testverfahren, die auf tatsächlicher Ausführung des Softwaresystems basieren, lassen sich im wesentlichen zwei grundsätzliche Herangehensweisen unterscheiden: Black und White Box-Tests.

2.5 Black Box-Test

In einem Prozessmodell geht der Implementierung eine lange Modellierungs- und Entwurfsphase voraus, in der Anforderungen und Spezifikationen erarbeitet und verfeinert werden. Insbesondere sind die dabei entstehenden Dokumente oft auch

Grundlage für Verträge über Softwareentwicklung und die Einhaltung der Spezifikation somit wichtig für den wirtschaftlichen Erfolg.

Man nennt Softwaretest, der auf diesen Dokumenten, wie etwa einem Pflichtenheft, basiert, auch den funktionsorientierten Softwaretest. Ausgehend von Funktionsbeschreibungen, die in der Regel nicht sehr formal und in natürlicher Sprache abgefasst sind, werden Testfälle entwickelt, die den Funktionsumfang abdecken und sicherstellen, dass die Anforderungen an die Funktion erfüllt werden.

Der Testling, d.h. das zu untersuchende Softwaresystem, wird dabei als Einheit betrachtet, das ausschließlich im Bezug auf sein nach Außen beobachtbares Verhalten untersucht wird. Vorhandenes Wissen über Details der Implementierung wird nicht berücksichtigt. Die Entwicklung der Testfälle erfolgt ausschließlich ausgehend von der Spezifikation. Dabei wird das Ergebnis der Ausführung mit der Ausgabe verglichen, die nach der Spezifikation vorgesehen ist. Aus dieser Vorgehensweise rührt der Name *Black Box*-Test her.

Es existieren eine Reihe von Verfahren, um die Erstellung von Testfällen zu systematisieren und gleichzeitig deren Zahl auf ein realistisches Maß zu beschränken. Exemplarisch für dieses Vorgehen ist etwa die *Klassifikationsbaummethode* [uKG93], die Eingaben eines Programms in logische Klassen unterteilt und für diese typische Eingaben ermittelt, die jeweils Gruppen von tatsächlichen Eingaben repräsentieren. Besondere Berücksichtigung finden dabei in der Regel die Grenzfälle zwischen den Klassen, da bei Grenzfällen oft Unterscheidungen zwischen den zu verwendenden Vorgehensweisen zur Erstellung der Programmausgabe erfolgen müssen, die korrekt zu treffen sind und je nach Natur der Grenzfälle fehleranfällig sind.

2.6 White Box-Test

Im Gegensatz zum Black Box-Test ist beim White Box-Test die interne Struktur des Testlings bekannt und dieses Wissen findet bei der Testfallerstellung Verwendung. Ein Test hat nicht zum Ziel, die Übereinstimmung mit einer Spezifikation zu überprüfen. Vielmehr sollen die internen Strukturen nach einem festzulegenden Verfahren angemessen getestet werden. Die Testdaten entstehen aus dem Softwaresystem selbst, so dass dieses nur gegen sich selbst getestet wird, nicht aber die Erfüllung externer Anforderungen an eine Spezifikation. Ziel ist, die Struktur des Testlings zu überprüfen.

Gängige Verfahren des strukturorientierten Softwaretests zielen darauf ab, den Quellcode so zu durchlaufen, dass ein bestimmtes Überdeckungskriterium, d.h. eine

bestimmte Intensität, mit der Testfälle Programmabschnitte durchlaufen, erfüllt wird. Die formale Definition dieser Überdeckungsmaße erfolgt in der Regel auf Kontrollflussgraphen.

Andere Verfahren arbeiten auf Datenflüssen und definieren auf diesen angemessene Kriterien. Im Rahmen dieser Arbeit stehen vor allem kontrollflussbasierte Verfahren im Vordergrund.

So fordert etwa Anweisungsüberdeckung, dass jeder Knoten der Kontrollflussgraphen wenigstens einmal durchlaufen wird, was auf Quellcode-Niveau bedeutet, dass jede Anweisung mindestens einmal ausgeführt wird. So kann sichergestellt werden, dass kein nicht erreichbarer (toter) Code existiert.

Weitere Methoden beobachten etwa Schleifendurchläufe oder die jeweiligen Belegungen von Bedingungen an Verzweigungen des Kontrollflusses, um weitere Aussagen zu gewinnen. Auf die verschiedenen Überdeckungskriterien und ihre Bewertung im Bezug auf Angemessenheit wird im weiteren Verlauf dieser Arbeit eingegangen.

2.7 Kombination beider Ansätze

In der Regel wird man sich beim Test von Softwaresystemen nicht auf einen einzelnen Test-Ansatz beschränken können.

Der Black Box-Test basiert auf Spezifikationen, die in der Regel informal und unpräzise und unter Umständen auch unvollständig sind, da sie in einer früheren Phase der Entwicklung erstellt wurden. Viele der Eigenschaften des tatsächlichen Programms ergeben sich erst, wenn in weiteren Phasen der Entwicklung die abstrakten Anforderungen mit denen einer konkreten Umgebung, sei es eine programmiersprachliche oder eine Hardwareumgebung, kombiniert werden. Testfälle, die auf einer Spezifikation basieren, decken daher zumeist nicht den kompletten Umfang an tatsächlich vorhandener Funktionalität ab, ohne dass dieser Fehler offensichtlich feststellbar ist.

Andererseits ist auch der reine White Box-Test nicht frei von Schwächen. Getestet wird möglicherweise die Korrektheit der konkreten implementierten Programmstruktur, ohne dass bemerkt wird, dass diese zwar möglicherweise fehlerfrei ist, aber eine nicht erwünschte Semantik besitzt.

Auf der Ebene von Kontrollflussgraphen sagt auch eine Überdeckung aller Kanten nicht aus, dass das Programm fehlerfrei arbeitet. Der Fall, dass Kanten fehlen (etwa nicht behandelte Sonderfälle der Eingabedaten) wird nicht erkannt.

Eine sinnvolle Testreihe sollte daher beide Ansätze kombinieren. Testfälle, die der Black Box-Test liefert, lassen sich mit dem White Box-Test auf ihre Vollständigkeit hin überprüfen und bewerten. Ausgehend von dieser Bewertung kann dann die Zahl und der Umfang der Testfälle entsprechend ergänzt werden, um Anforderungen an Überdeckungsmaße gerecht zu werden.

3 Kontrollflussgraphen

3.1 Motivation

3.2 Kontrollflussgraphen

Kontrollflussgraphen sind Graphen, in denen Knoten Anweisungen einer Funktion und Kanten mögliche Übergänge zwischen verschiedenen Anweisungen darstellen. Sie lassen sich verwenden, um die Ablauf der Ausführung von Konstrukten imperativer Programmiersprachen nachzuvollziehen.

Kontrollflussgraphen werden auf einzelnen Modulen betrachtet. Ein Modul entspricht dabei einer Funktion bzw. einer Methode in einer imperativen Programmiersprache.

Eine Reihe von Metriken sind auf Kontrollflussgraphen definiert. Sie geben Aufschluss über die logische Komplexität des betrachteten Programmabschnitts. Ferner dienen die Graphen als Basis für die Bewertung von Testfällen im Zusammenhang mit Überdeckungsmaßen.

3.3 Begriffsbestimmungen

Zum Verständnis von Kontrollflussgraphen und zur Festlegung einiger Eigenschaften, auf deren Basis anschließend Komplexitätsmaßzahlen bestimmt werden, werden zunächst einige Definitionen in Anlehnung an [Zus90] vorgestellt:

Definition 3.3.1 (Graph) *Ein **Graph** ist ein Tupel G=(V,E), wobei V eine endliche Menge von Knoten und E eine endliche Menge von Kanten ist. E ist eine Teilmenge der 2-elementigen Teilmengen von V:* $E \subseteq \binom{V}{2}$.

Definition 3.3.2 (gerichteter Graph) *Ein **gerichteter Graph (Digraph)** ist ein Tupel D=(V,E) mit $E \subseteq V \times V$. Die Elemente aus E heißen gerichtete Kanten.*

Definition 3.3.3 (Eingangsgrad, Ausgangsgrad) *Sei G=(V,E) ein gerichteter Graph und $x \in V$ ein Knoten von G. Dann ist $N^+(x) = \{y \in V | (x,y) \in E\}$ die Menge der Ausgangsknoten und $N^-(x) = \{y \in V | (y,x) \in E\}$ die Menge der Eingangsknoten von x. Weiterhin sei $d^+(x) := |N^+(x)|$ der **Ausgangsgrad** und $d^-(x) := |N^-(x)|$ der **Eingangsgrad** von x.*

Definition 3.3.4 (Weg, Pfad) *Ein **Weg** von x_0 nach x_l in einem gerichteten Graphen G=(V,E) ist eine Folge von Knoten $x_0, \ldots, x_l \in V$ mit $l \geq 1$ und $(x_i, x_{i+1}) \in E$ für $i = 0, \ldots, l-1$. Die Knoten x_0 und x_l werden als Endknoten des Weges bezeichnet.*

*Ein **Pfad** von x_0 nach x_l ist ein Weg von x_0 nach x_l, in dem kein Knoten doppelt vorkommt.*

Definition 3.3.5 (Flussgraph) *Ein **Flussgraph** G=(V,E,s,t) ist ein gerichteter Graph wobei:*

1. $V \neq \emptyset$
2. $E \neq \emptyset$
3. $d^-(s) = 0$
4. $d^+(t) = 0$
5. *jeder Knoten $x \in V$ liegt auf einen Pfad in G von s nach t*

Dabei wird $s \in V$ der Startknoten und $t \in V$ der Endknoten genannt.

Stellt ein Flussgraph die Struktur des Kontrollflusses eines Softwaremoduls dar, so nennt man ihn Kontrollflussgraph. Dabei entsprechen die Knoten des Flussgraphen Anweisungen im betrachteten Programmabschnitt, Kanten zwischen Knoten bilden Kontrollflüsse zwischen je zwei Anweisungen und die Richtung des Kontrollflusses ab. Somit beschreiben Kontrollflussgraphen die logische Struktur eines Moduls. Der Kontrollfluss beginnt immer mit dem Startknoten und endet mit dem Endknoten. Auf diese Weise wird die Semantik eines Funktions- oder Methodenaufrufs abgebildet.

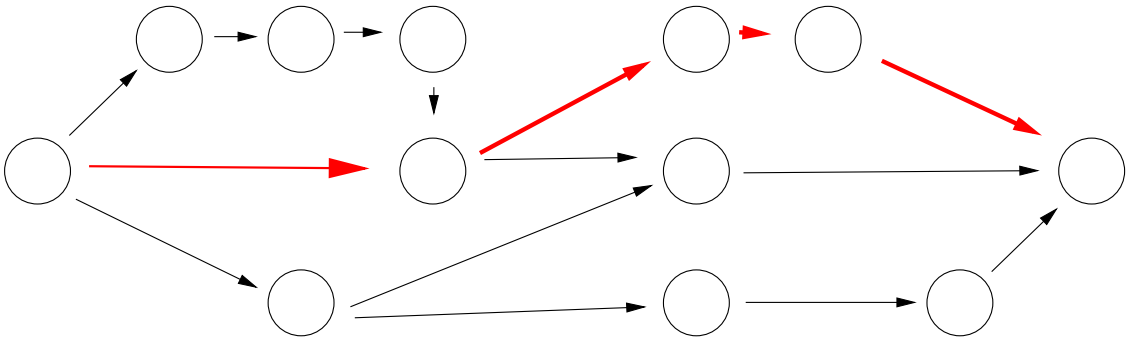


Abbildung 3.1: Beispiel eines Flussgraphen

Der Graph in Abbildung 3.1 zeigt einen Flussgraphen, in dem einer der möglichen Pfade hervorgehoben ist.

Definition 3.3.6 (Subgraph) *Ein Graph G'=(V',E') heißt* ***Subgraph*** *des Graphen G=(V,E), wenn gilt:*

1. $V' \subseteq V$
2. $E' \subseteq E$

Definition 3.3.7 (Eingang in einen Graphen) *Ein Knoten $x \in V_G$ heißt* ***Eingang in einen Graphen*** *$G = (V_G, E_G)$, wenn jeder mögliche Pfad mit Knoten aus V_G auch x als ersten Knoten aus V_G enthält.*

Definition 3.3.8 (Subflussgraph, 1-Eingangs-Subflussgraph) *Ein* ***Subflussgraph*** *G'=(V',E',s',t') eines Flussgraphen G=(V,E,s,t) ist ein Flussgraph, für den gilt:*

1. *G' ist Subgraph von G*
2. *für alle $v' \in V'$, $v \in V$: $v = v' \rightarrow N^+(v') = N^+(v)$*

Wenn der Graph G' genau einen Eingang besitzt, so nennt man diesen Subflussgraphen einen ***1-Eingangs-Subflussgraphen****.*

Definition 3.3.9 (Prime) *Ein Flussgraph ist ein* ***Prime****, wenn er keinen echten 1-Eingangs-Subflussgraphen enthält.*

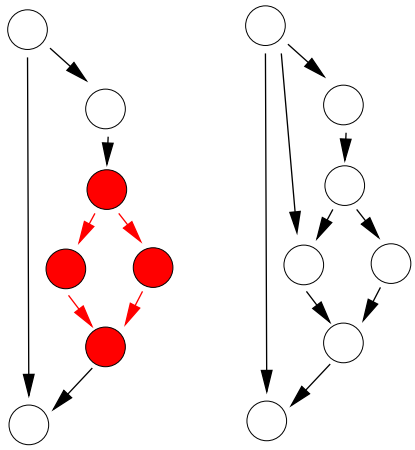


Abbildung 3.2: Flussgraph mit Prime (rot) und Graph ohne Prime

Ein Prime ist somit ein Subflussgraph, der nicht durch Verschachtelung anderer Flussgraphen gebildet werden kann [Zus90]. In Abbildung 3.2 ist rechts ein Prime dargestellt. Wird dem Graphen lediglich eine Kante hinzugefügt (rechter Teil der Abbildung), enthält der Graph keinen Prime als Subgraphen mehr.

Anschaulich sind Primes grundlegende Kontrollstrukturen imperativer Programmiersprachen wie etwa C++ oder Java: `while`, `do-while`, `if` und `if-else` sowie `switch`-Konstrukte.

3.4 Komplexitätsmaße auf der Basis von Kontrollflussgraphen

Kontrollflussgraphen können für verschiedene Einsatzzwecke betrachtet werden. Im Rahmen dieser Arbeit werden zwei Aspekte betrachtet: zum einen die Verwendung im strukturorientierten Softwaretest, zum anderen zur Bestimmung von Komplexitätsmaßzahlen in allen Phasen der Softwareentwicklung.

Eine verbreitete Komplexitätsmaßzahl ist die zyklomatische Komplexität. Sie gibt die logische Komplexität eines Moduls an [WM96].

Definition 3.4.1 (zyklomatische Komplexität) *Sei $G_M = (V, E, s, t)$ der Kontrollflussgraph, der der Kontrollstruktur eines Moduls M entspricht. Dann wird $v(G_M) = |E| - |V| + 2$ die **zyklomatische Komplexität** dieses Moduls genannt.*

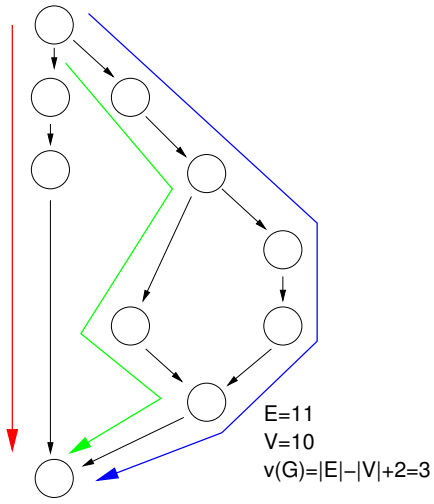


Abbildung 3.3: Kontrollflussgraph mit 3 lin. unabh. Pfaden

Die Definition der zyklomatische Komplexität rührt aus der Betrachtung von stark zusammenhängenden Graphen her.

Definition 3.4.2 (stark zusammenhängender Graph) *Sei G=(V,E) ein gerichteter Graph. Man nennt G* ***stark zusammenhängend****, wenn es für alle* $x, y \in V$ *einen Pfad mit x als Startknoten und y als Endknoten gibt.*

Definition 3.4.3 (linear unabhängiger Pfad) *Ein Pfad p aus einer Menge von Pfaden P heißt ein* ***linear unabhängiger Pfad****, wenn er wenigstens einen Knoten enthält, der in keinem Pfad* $p' \in P$ *enthalten ist.*

Definition 3.4.4 (zyklomatische Zahl) *Sei G=(V,E) ein stark zusammenhängender Graph. Dann wird* $n(G) = |E| - |V| + 1$ *die* ***zyklomatische Zahl*** *von G genannt. Die zyklomatische Zahl gibt die Anzahl linear unabhängiger Pfade durch G an.*

Ein Kontrollflussgraph ist kein stark zusammenhängender Graph. Fügt man allerdings eine Kante (t,s) ein, erfüllt er die Definition. Wegen des Fehlens dieser Kante wird bei der Berechnung der zyklomatischen Komplexität eine 1 addiert. Das Hinzufügen dieser zusätzlichen Kante ist nicht rein willkürlich, man kann sich vorstellen, dass es das Verhalten des Programms beschreibt, aus dem ein Modul aufgerufen wurde.

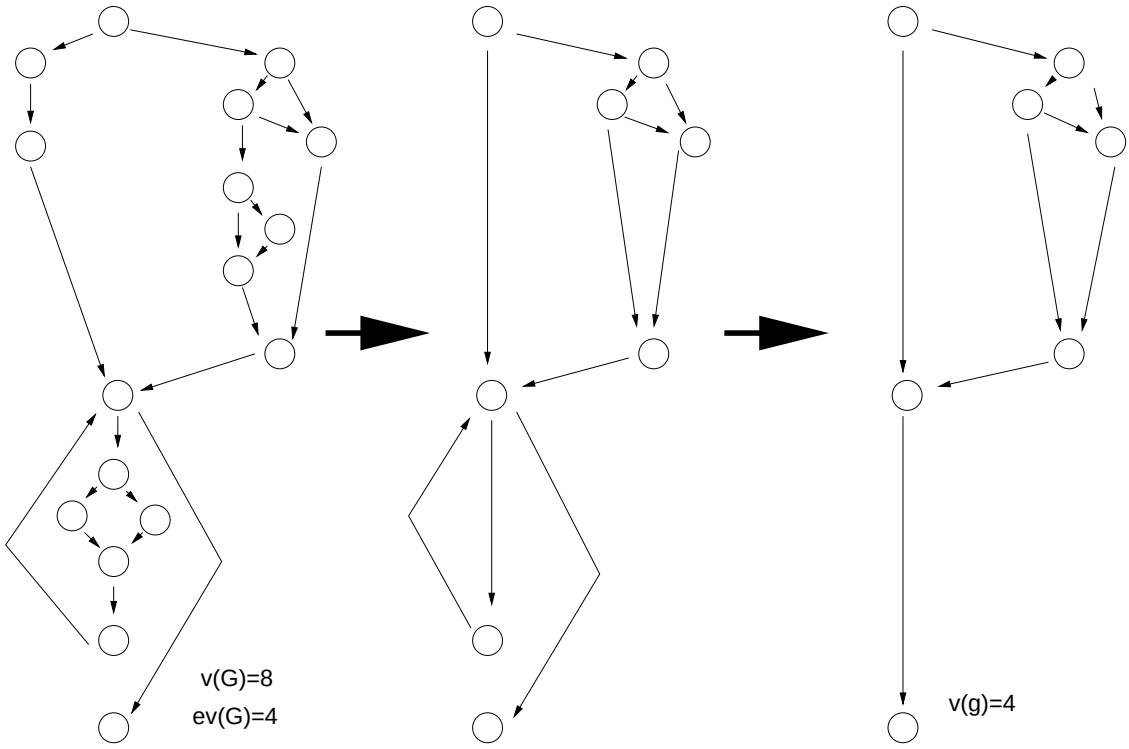


Abbildung 3.4: Reduktion eines Graphen zur Berechnung der ess. Komplexität

Die zyklomatische Komplexität gibt somit die Menge möglicher, linear unabhängiger Kontrollflüsse durch ein Modul an [WM96].

Essentielle Komplexität [WM96] ist ein Maß, dass die Unstrukturiertheit von Software-Modulen misst. Eine hohe zyklomatische Komplexität weist zwar auf eine logisch komplexe Struktur hin, diese ist aber nicht notwendigerweise unübersichtlich und damit schwer zu warten. Diese Abwägung versucht die Messung der essentiellen Komplexität zu berücksichtigen.

Definition 3.4.5 (essentielle Komplexität) *Sei $G_M = (V, E, s, t)$ der Kontrollflussgraph, der der Kontrollstruktur eines Moduls M entspricht. Sei $G_{M'}(V', E', s, t)$ der reduzierte Graph, der entsteht, wenn sukzessiv alle Primes entfernt werden, bis keine weitere Reduktion mehr möglich ist. $ev(G_M) := v(G_{M'})$. Dabei wird $ev(G_M)$ auch die* ***essentielle Komplexität*** *von G_M genannt.*

Der Vorgang der Reduktion eines Graphen ist in Abbildung 3.4 an einem Beispiel dargestellt.

3.5 Bewertung von zyklomatischer und essentieller Komplexität

Die Maße von McCabe wurden von ihn zuerst 1976 [McC76] definiert und sind somit im Vergleich zu den meisten anderen Entwicklungen auf diesem Gebiet vergleichsweise alt und gut untersucht. Darüber hinaus gehören sie zu den Maßen, die in Testsystemen am häufigsten implementiert sind und daher weite Verwendung finden.

[Li87] etwa liefert eine umfangreiche Bewertung dieser Komplexitätsmaße. Die meisten Punkte der Bewertung rühren direkt aus der Art und Weise her, wie die Komplexitätsmaße von McCabe bestimmt werden und welche Faktoren dabei in die Bewertung einfließen und welche nicht.

Stärken

- Einfache Berechnung von zyklomatischer Komplexität aus dem Programmcode bzw. dem Kontrollflussgraphen.
- Da sich auch in früheren Phasen des Software-Entwurfsprozesses Kontrollflussgraphen erstellen lassen, ist eine Verwendung in einem Top Down-Entwicklungsprozess gut möglich.
- Ableitung direkter Entwicklungsrichtlinien, wie etwa die zyklomatische Komplexität nicht über 10 wachsen zu lassen.
- Einfaches Auffinden von komplexen Programmmodulen, die intensiv getestet werden sollten.
- Daher möglicher Wegweiser für Testgestaltung, da fehleranfällige Codeteile rasch aufgespürt werden können.

Schwächen

- Keine Bewertung der Komplexität von Rechenoperationen (drei verschachtelte Schleifen sind für McCabe das Gleiche wie drei aufeinander folgende).
- Statische Zuweisung der gleichen Komplexität ohne Berücksichtigung der betrachteten Operation.

- Länge sequentieller Programmabschnitte wird nicht gewertet.
- Keine Bewertung von Entscheidungen nach Art und Zahl der Variablen.
- Keine Bewertung der Abhängigkeiten von Entscheidungen im Kontrollfluss.
- Modulaufrufe und deren Abhängigkeiten und Abfolgen werden nicht gemessen.
- Aspekte objektorientierter Programmierung werden aufgrund des Alters des Maßes nicht berücksichtigt.

4 Überdeckungsmaße

Die Prüfung eines Softwaresystems auf die Erfüllung von Überdeckungsmaßen basiert auf dem Test des tatsächlichen Systems. Dabei wird das Programm kontrolliert durchlaufen und sein Verhalten gemessen und bewertet. Somit gehört dieser Ansatz zu den testenden Verfahren, die dynamisch arbeiten, also das Programm tatsächlich ausführen. Da der Quellcode bzw. der durch ihn beschriebene Kontrollfluss betrachtet wird, sind diese Verfahren unter den White Box-Tests einzuordnen.

Um das Verhalten eines Testlings bei Durchlauf durch einen Testfall betrachten zu können, ist es nötig, eine Instrumentierung vorzunehmen. Der bei der Instrumentierung eingefügte Code zeichnet an den entsprechenden Stellen die Informationen auf, die zur Rekonstruktion von Kontrollflüssen nötig sind. Für die Verfahren, die die Verzweigung des Kontrollflusses aufgrund von Bedingungen betrachten, ist es darüber hinaus notwendig, diese Bedingungen zu aufzuzeichnen.

Die Erfüllung von Überdeckungskriterien kann genutzt werden, um die verwendeten Testfälle auf ihre Vollständigkeit hin zu überprüfen und sie ggf. zu ergänzen.

4.1 Anweisungs- und Zweigüberdeckung

Das am einfachsten zu erfüllende und zu überprüfende Überdeckungsmaß ist die *Anweisungsüberdeckung*. Sie verlangt den Durchlauf durch alle Knoten der Kontrollflussgraphen, was auf der Ebene des Quelltextes der Ausführung aller Anweisungen entspricht. Ist Anweisungsüberdeckung erfüllt, so ist sichergestellt, dass alle Teile des Programms tatsächlich ausführbar sind. Findet sich dagegen keine Menge von Testfällen, die Anweisungsüberdeckung erfüllt, liegt toter Code vor, der nie ausgeführt wird. Wenn dieser Code keine Funktion mehr erfüllt, kann er entfernt werden, andernfalls liegt ein Fehler in der logischen Programmstruktur vor. Aussagen, die über das Aufspüren toten Codes hinaus gehen, lassen sich mit der Anweisungsüberdeckung nicht machen.

```
void example(int a)
{
    if (a>0)
        whatever();
    return;
}
```

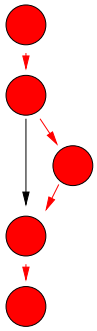


Abbildung 4.1: Anweisungsüberdeckung vs. Zweigüberdeckung

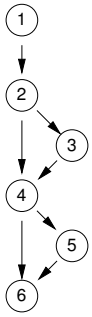
Die Anweisungsüberdeckung lässt es nicht zu, wesentliche Eigenschaften eines Systems zu testen, so etwa den Kontrollfluss und die Bedingungen, die Verzweigungen des Kontrollflusses auslösen. Nichtsdestotrotz ist die Anweisungsüberdeckung ein notwendiges Kriterium bei der Erstellung von Testfällen für umfassendere Überdeckungsmaße.

Zweigüberdeckung ist erfüllt, wenn alle Zweige des Kontrollflussgraphen zumindest einmal durchlaufen wurden. Bei diesem Maß wird auch die Kontrollstruktur in die Betrachtung einbezogen. Erfüllte Zweigüberdeckung stellt sicher, dass alle Verzweigungen des zu testenden Moduls tatsächlich erreichbar sind. Sie wird oft als minimal zu erfüllendes Testkriterium angesetzt. Zweigüberdeckung impliziert die Erfüllung von Anweisungsüberdeckung.

Abbildung 4.1 stellt ein Beispiel dar, in dem zwar Anweisungsüberdeckung erfüllt ist, aber nicht Zweigüberdeckung, wenn die beschriebene Funktion mit einem Wert größer als Null aufgerufen wird. Dies rührt daher, dass der else-Zweig der if-Verzweigung keine Anweisungen enthält. Für die Erfüllung der Zweigüberdeckung müsste zusätzlich ein Testfall, der die Funktion mit einem Wert kleiner-gleich Null aufruft, definiert werden.

Obwohl die Fehlererkennungsrate gegenüber der Anweisungsüberdeckung steigt, lässt die Zweigüberdeckung mehrere Fehlerfelder unberührt. So werden Abhängigkeiten zwischen Zweigen nicht betrachtet, sondern nur jeder Zweig für sich. Pfade, deren Zweige in Abhängigkeit voneinander stehen, werden nicht berücksichtigt. Ebenso werden Schleifen nicht angemessen berücksichtigt. Zweigüberdeckung ermittelt nur, ob diese durchlaufen wurden, nicht aber, wie oft. Pfadüberdeckungstests, die Folgen von durchlaufenen Zweigen betrachten und Wiederholungen von Schleifen bewerten, berücksichtigen diese Faktoren besser.

```
void example(boolean a, boolean b)
{
    if (a)
        do();
    if(b)
        doAlso();
}
```



Pfadüberdeckung:	a	b
1. Testfall: 1,2,4,6	f	f
2. Teilfall: 1,2,3,4,5,6	t	t
3. Testfall: 1,2,3,4,6	t	f
3. Testfall: 1,2,4,5,6	f	t

Abbildung 4.2: Testfälle für Pfadüberdeckung und Pfade

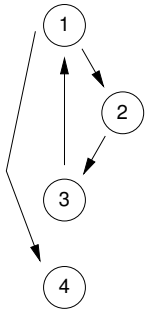
Die Bedingungen, die Verzweigungen im Kontrollfluss auslösen, werden bei der Zweigüberdeckung ebenfalls nicht betrachtet, womit insbesondere bei komplexen Bedingungen, die sich aus vielen einzelnen, verknüpften Bedingungen zusammensetzen, der Test nur unzureichend ist. Bedingungsüberdeckung in ihren verschiedenen Varianten nimmt sich dieses Problems an.

4.2 Pfadüberdeckung

Pfadüberdeckungskriterien berücksichtigen anders als die Zweigüberdeckung nicht nur einzelne Kanten des Kontrollflussgraphen, sondern betrachten Pfade durch den Graphen, also auch die Abhängigkeiten zwischen den einzelnen Zweigen.

Von dieser Voraussetzung ausgehend betrachtet der *Pfadüberdeckungstest* alle Pfade, die in einem Modul existieren. Pfadüberdeckung ist gegeben, wenn alle möglichen Pfade eines Moduls durch die Testfälle durchlaufen werden. Die tatsächliche Überprüfung dieses Kriteriums ist schwierig, da viele Kombinationen, die sich auf dem Kontrollflussgraphen konstruieren lassen, sich aufgrund ihrer Bedingungen gegenseitig ausschließen. Darüber hinaus wächst die Anzahl der nötigen Testfälle bei Modulen, die Schleifen enthalten, ins Unendliche, da deren Zahl von Wiederholungen zumeist nach oben hin nicht abschätzbar ist.

```
void example(int a)
{
while (a<2)
    a++;
}
```



Boundary–Interior–Überdeckung	a
1. Testfall: 1,4	2
2. Testfall: 1,2,3,1,4	1
3. Testfall: 1,2,3,1,2,3,1,4	0

Abbildung 4.3: Boundary-Interior-Überdeckung mit Testfällen

Diese Schwäche bei der Betrachtung von Pfaden versucht der *Boundary Interior-Pfadtest* zu bewältigen. Er teilt Pfade im Bezug auf Schleifen in drei Kategorien ein:

- Pfade, die Schleifenkörper nicht durchlaufen
- Pfade, die den Schleifenkörper genau einmal durchlaufen
- Pfade, die den Schleifenkörper wenigstens einmal wiederholen

Boundary Interior-Überdeckung ist genau dann erfüllt, wenn alle Pfade durch den Kontrollflussgraphen durchlaufen werden. Ausgenommen von dieser Regel, die soweit dem Pfadtest entspricht, sind Pfade durch Schleifen, die diese mehr als einmal wiederholen.

In Abbildung 4.3 ist ein Beispiel einer Schleife dargestellt. Abhängig vom Parameter der Funktion wird die while-Schleife sehr oft durchlaufen, so dass Pfadüberdeckung mit sehr vielen Testfällen verbunden ist. Boundary-Interior benötigt lediglich 3 Testfälle, um die Schleife auf die drei Anforderungen zu testen.

4.3 Bedingungsüberdeckung

Die verschiedenen Maße der *Bedingungsüberdeckung* betrachten nicht mehr nur den Kontrollflussgraphen und Durchläufe durch diesen, sondern beziehen in die Betrachtung die Bedingungen und ihre Belegungen ein, die Verzweigungen des Kontrollflusses auslösen. Dabei werden atomare Bedingungen, die entweder wahr oder falsch sind und die kleinsten Einheiten in Bedingungen bilden, von zusammengesetzten Bedingungen unterschieden, die sich ergeben, wenn atomare Bedingungen durch Operatoren verknüpft werden.

Einfache Bedingungsüberdeckung fordert, dass alle atomaren Bedingungen zumindest einmal mit wahr und einmal mit falsch belegt werden. Einfache Bedingungsüberdeckung umfasst weder Anweisungs- noch Bedingungsüberdeckung und ist daher als alleiniges Testkriterium unzureichend.

Mehrfache Bedingungsüberdeckung fordert, dass alle möglichen Kombinationen von Belegungen der atomaren Bedingungen aufgetreten sind. Da dann auch jede zusammengesetzte Bedingung jedes mögliche Ergebnis annimmt, umfasst mehrfache Bedingungsüberdeckung Zweigüberdeckung. Allerdings benötigt man bei komplexen zusammengesetzten Bedingungen sehr viele Testfälle, um alle Belegungen zu erreichen. Bei n atomaren Bedingungen einer zusammengesetzten Bedingungen sind 2^n Testfälle nötig, was oft nicht zu realisieren ist. Darüber hinaus ist die Interpretation dadurch erschwert, dass nicht alle Kombinationen von Belegungen möglich sind, ohne dass dies einen Fehler bedeutet.

Betrachtet man neben den atomaren Bedingungen zusätzlich das Ergebnis der zusammengesetzten Bedingungen sowie der Zwischenauswertungen und fordert, dass die atomaren Bedingungen, die zusammengesetzten Bedingungen und die Zwischenergebnisse beide möglichem Wahrheitswerte annehmen, so spricht man von *minimal-mehrfacher Bedingungsüberdeckung*. Sie umfasst die Zweigüberdeckung, ist somit mächtiger als die einfache Bedingungsüberdeckung. Da die Zahl der nötigen Testfälle nicht wie bei der mehrfachen Bedingungsüberdeckung exponentiell in der Zahl der atomaren Bedingungen wächst, ist die Realisierung dieses Testkriteriums auch bei komplexen Softwaresystemen realistisch.

4.4 MC/DC-Überdeckung

MC/DC-Überdeckung (Modified Condition/Decision Coverage) ist ebenfalls eine Form der Bedingungsüberdeckung. Sie wurde mit dem Ziel entwickelt einen

$F = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$

Kombination	A	B	C	F	C2	MC/DC	C3
1	1	1	1	1			+
2	1	1	0	1		+	+
3	1	0	1	1		+	+
4	0	1	1	1	+		+
5	1	0	0	0	+	+	+
6	0	1	0	0		+	+
7	0	0	1	0			+
8	0	0	0	0			+

Abbildung 4.4: Beispiel für drei Maße der Bedingungsüberdeckung

möglichst vollständigen Test von Bedingungen durchzuführen, ohne vollständige Bedingungsüberdeckung erreichen zu müssen [KJHR01].

MC/DC-Überdeckung fordert die Erfüllung von vier Bedingungen [do192]:

1. Jeder Eingang und Ausgang des Moduls wird wenigstens einmal gewählt
2. Jede atomare Bedingung hat jede mögliche Belegung angenommen
3. Jede zusammengesetzte Bedingung hat jede mögliche Belegung angenommen
4. Jede atomare Bedingung hat wenigstens einmal das Ergebnis der zugehörigen zusammengesetzten Bedingung bestimmt

Die erste Bedingung wird von allen hier aufgeführten Überdeckungsmaßen mit Ausnahme der Anweisungsüberdeckung erfüllt. Minimal mehrfache Bedingungsüberdeckung erfüllt das zweite und das dritte Kriterium. Die vierte Bedingung macht daher die MC/DC-Überdeckung aus.

Die Abbildung 4.4 stellt die drei Maße einfache Bedingungsüberdeckung(auch *C2-Test* genannte), MC/DC-Überdeckung und mehrfache Bedingungsüberdeckung (*C3*) an einem Beispiel vor. Untersucht wird die zusammengesetzte Bedingung $F = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$, für die es insgesamt $2^3 = 8$ verschiedene Belegungen gibt. Für die Erfüllung der mehrfachen Bedingungsüberdeckung sind alle acht Kombinationen notwendig, was die große Zahl von notwendigen Testfällen schon bei diesem kleinen Beispiel verdeutlicht. Wesentlich weniger Testfälle erfordert die einfache Bedingungsüberdeckung, nämlich nur zwei.

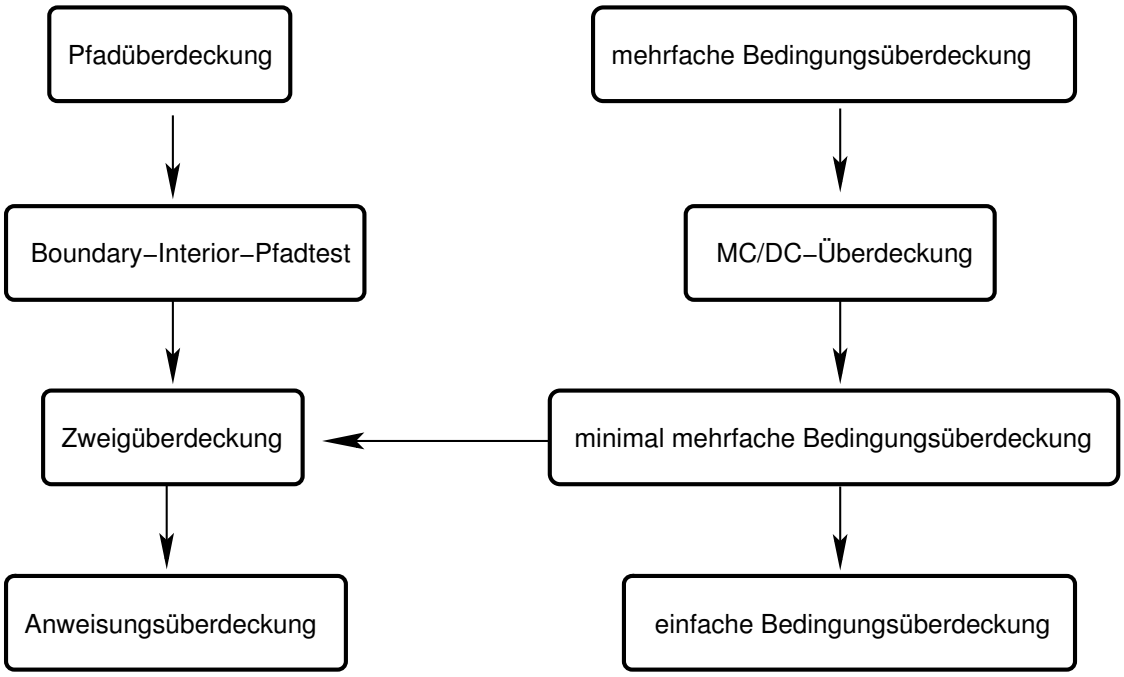


Abbildung 4.5: Beziehungen der verschiedenen Überdeckungsmaße

Die MC/DC-Überdeckung erfordert für dieses Beispiel 4 Testfälle. Dabei bilden für jedes Atom je zwei Belegungen *MC/DC-Paare*, also Paare von Belegungen, die sich nur in der Belegung eines Atoms und der Gesamtauswertung F unterscheiden. Für das Atom A sind dies die Kombinationen 2 und 6, für das Atom B die Kombinationen 2 und 5 und schließlich für das Atom C die Kombinationen 3 und 5.

4.5 Vergleich der verschiedenen Überdeckungsmaße

Bei der Bewertung von Methoden zur Ermittlung von Fehlern in Programmen sollte man sich einige grundsätzliche Eigenschaften solcher Ansätze im Allgemeinen klar machen. Es ist aufgrund der Unentscheidbarkeit des Halteproblems nicht möglich, sicherzustellen, dass ein Programm grundsätzlich fehlerfrei arbeitet. Es kann höchstens die Anwesenheit von Fehlern aufgezeigt werden.

Bei der Bewertung bietet sich daher ein Verfahren an, bei dem in ein bekanntes Programmmodul nachträglich Fehler eingebracht werden (*Errorseeding*). [Wal90] diskutiert kurz das dazu verwendete Verfahren.

[Lig02] stellt die hier aufgeführtem Maße ausführlich vor und bewertet sie nach dem oben angesprochenen Ansatz des Errorseedings.

Anweisungsüberdeckung wird im allgemeinen nicht als hinreichendes Kriterium für die Vollständigkeit eines Tests betrachtet, ihre Auswertung ergibt sich als Nebenprodukt der Auswertung umfassenderer Überdeckungskriterien. Empirische Untersuchungen zeigen eine Fehlererkennungsrate von 15% bis unter 20% für reine Anweisungsüberdeckung. Sie liefert allerdings in allen Untersuchungen höhere Prozentsätze als die statische Analyse des Quelltextes.

Die Kantenüberdeckung gilt als Standard bei kontrollflussbasierten Überdeckungsmaßen und wird oft als Kriterium an Tests gestellt. Sie wird als leistungsfähiger Test vor allem zum Auffinden von logischen Fehlern betrachtet. Untersuchungen bezüglich der Leistungsfähigkeit streuen in einem weiten Bereich von 20% bis 70% gefundener Fehler. Zumeist liegen die Prozentsätze im unteren Drittel dieses Intervalls. Studien, die ebenfalls Anweisungsüberdeckung betrachten, zeigen für Zweigüberdeckung eine Fehlererkennungsrate, die um 50% bis 100% über der der Anweisungsüberdeckung liegt.

Bedingungsüberdeckung findet vor allem bei Systemen mit komplexer Verarbeitungslogik Verwendung, die in der Regel kompliziert aufgebaute Bedingungen verwende. Als Kompromiss zwischen Testaufwand und erzielbarem Ergebnis finden vor allem minimal mehrfache Bedingungsüberdeckung und MC/DC-Überdeckung Verwendung.

5 Anforderungsdefinition

5.1 Ausgangspunkt

Der Lehrstuhl Softwaretechnik unter Prof. Bothe betreut mit dem System XCTL im Rahmen des Softwaresanierungs-Projektes ein umfangreiches Softwaresystem, an dem im Zuge von Reengineering-Maßnahmen und Erweiterungen schon zahlreiche Änderungen und Ergänzungen vorgenommen wurden.

Tests finden vor allem mit dem eigenentwickelten Regressionstest-Tool ATOS statt, das einen Black Box-Test realisiert. Mit diesen Tests kann die Erfüllung der geforderten Funktionalität gut überwacht und einfach geprüft werden.

Über die *Gründlichkeit*, mit der die Testfälle die Struktur des Quellcodes untersuchen, der zu einem Teil noch aus dem Alt-System vor Beginn der Restrukturierung stammt, können keine Aussagen abgeleitet werden. Solche Aussagen lassen sich nur mit einem White Box-Testsystem ermitteln.

5.2 Problemdarstellung und Anforderungsdefinition

Am Beginn der Arbeit stand eine E-Mail, die Ronny Treyße und ich von Herrn Prof. Bothe im Anschluss an ein Einführungsseminar in das Softwaresanierungs-Projekt erhielten, nachdem wir unser Interesse an einer weiteren Mitarbeit in diesem Bereich bekundet hatten.

Diese Mail fasst alle aus damaliger Sicht relevanten Punkte für die weitere Ausarbeitung zusammen:

```
From: Klaus Bothe <bothe@informatik.hu-berlin.de>
Date: Wed, 05 Mar 2003 14:54:48 +0100
Organization: Humboldt-Universitaet zu Berlin

Arbeitsthema:

(Portables ) Überdeckungstesttool für C++ / Java:
    C0, C1 ... - Überdeckungsmessung für den
    strukturorientierten Test

  Als Ergänzung zum funktionsorientierten Test mittels
  CTE soll hiermit eine Bewertung der Testfälle in
  Bezug auf die einbezogenen Programmeinheiten
  (Anweisungen, Zweige, Zyklen, Ausdrücke) vorgenommen
  werden.

  C0: Anweisungsüberdeckung
  C1: Zweigüberdeckung
  C2: einfache Bedingungsüberdeckung
  minimale mehrfache Bedingungsüberdeckung
  C3: Mehrfache Bedingungsüberdeckung
  MC/DC-Überdeckung
  Boundary-Interior-Pfadtest

  Ausgangspunkt: Syntaxanalyse und Preprocessing der Programmquellen

Anwendungen:
a) Fallstudie XCTL:
  Entwicklung eines vollständigen oberflächenorientierten
    Regressionstests mit Hilfe von (attributierten)
    Klassifikationsbäumen und Überdeckungsmessung
  -> zunächst: ein Use Case (z. B. manuelle Justage)
b) im Halbkurs Software Engineering

Arbeitsschritte:
- Dokumente erarbeiten:
   Pflichtenheft
     nach Balzert:
       - jedoch angepasst an Aufgabe:
         z. B. mit Beispielen untersetzt
       - Wunschkriterium: zyklomatische / essentielle Komplexität
       - Wunschkriterium: Architekturinformation
           (welche Komponenten durch welche Testfallmengen berührt)
       - Musskriterium: MC/DC-Überdeckung
           (amerikanische Luftfahrtindustrie)
   Design: Nachnutzung und Erweiterung von Parsern möglich?
   Implementation
   Testfälle für das Tool
```

5.3 Gliederung der Aufgabenstellung

Die Anforderungsdefinition fordert ein Programm, das in der Lage ist, Überdeckungsmaße für ein beliebiges Softwaresystem in einer der geforderten Sprachen zu bestimmen.

Um Überdeckungsmaße zu ermitteln, ist eine Messung der Programmverhaltens zur Laufzeit notwendig. Diese Messung erfordert das Einfügen zusätzlicher Befehle in das Programm, die die durchlaufenen Pfade und die Belegung der Wahrheitswerte, die den Kontrollfluss bestimmen, aufzeichnen und abspeichern, um sie auswerten zu können. Diesen Vorgang, ein Programm um zusätzliche Befehle zu ergänzen, nennt man *Instrumentierung*.

Die Aufgabenstellung wurde von uns in zwei wesentliche Teilaufgaben unterteilt:

- **Instrumentierung**
  Zur Instrumentierung ist ein Einlesen (Parsen) des originalen Quellcodes und anschließendes Analysieren notwendig. Diese Aufgabe ist naturgemäß stark von der Zielsprache abhängig. Es muss eine Schnittstelle bereitgestellt werden, die das Auswerten und Einlesen des geparsten Programms ermöglicht. Auch muss eine Schnittstelle zur Verfügung gestellt werden, über die die Aufzeichnungen der Instrumentierung ausgewertet werden können.

- **Auswertung**
  Unabhängig vom Zeitpunkt und Ort der Ausführung des zu untersuchenden System ist die Auswertung der anfallenden Daten (Quellcode und Aufzeichnungen der Instrumentierung). Diese Auswertung, die im Gegensatz zum Instrumentierer Kenntnis über die auszuwertenden Überdeckungsmaße besitzt, interpretiert diese Daten und stellt sie anschaulich dar.

Ausgehend von dieser Aufteilung wurde die Gesamt-Aufgabenstellung in zwei, aus organisatorischen Gründen getrennte, Diplomarbeiten aufgeteilt, die aber parallel entstanden sind. Die Instrumentierung wurde von Ronny Treyße ausgearbeitet, die Auswertung von mir, Hendrik Seffler.

Im weiteren Rahmen dieser Arbeit wird die auswertende Komponente ausführlich vorgestellt und auf den Parser und Instrumentierer nur im Bezug auf seine nach außen hin zur Verfügung gestellten Schnittstellen Bezug genommen.

5.4 Pflichtenheft

5.4.1 Zielbestimmung

Musskriterien

Ziel ist die Entwicklung eines Programms, welches die Daten eines Programms zum funktionsorientierten Test ergänzt, indem es den zu testenden Programmcode instrumentiert und aus den Testdurchläufen Überdeckungsmaße ermittelt (strukturorientierter Test). Zusätzlich sollen die Ermittlung von Komplexitätsmaßzahlen und die Gewinnung von Architekturinformationen möglich sein.

- Ermittlung von Überdeckungsmaßen aus den Log-Aufzeichnungen
  - C0: Anweisungsüberdeckung
  - C1: Zweigüberdeckung
  - C2: einfache Bedingungsüberdeckung
  - C3: mehrfache Bedingungsüberdeckung
  - minimale mehrfache Bedingungsüberdeckung
  - MC/DC-Überdeckung
  - Boundary-Interior-Pfadtest
- Die Auswertung dieser Messungen ist für Quellcode in Java und C++ zu leisten
- Portabilität des Hauptprogramms ist anzustreben

Wunschkriterien

- Bestimmung von Komplexitätsmaßzahlen
  - zyklomatische Komplexität
  - essentielle Komplexität
- Umfang der Instrumentierung soll auf Klassen- und Methodenebene konfigurierbar sein

- Ableitung von Architekturinformationen
  Darstellung des Zusammenhangs zwischen Mengen von Testfällen und dem von ihnen berührten Codeabschnitten sowie des jeweiligen Beitrags zur Sicherstellung von Überdeckungsmaßen
- grafische Darstellung des Kontrollflusses von Methoden des zu testenden Programms
- Darstellung der folgenden Elemente
  - Testfälle
  - Klassenstruktur
  - Sourcecode
  - Kontrollflussgraph

Abgrenzungskriterien

- Die Erstellung und Verwaltung der Testfälle erfolgt nicht innerhalb des Hauptprogramms, sondern außerhalb. Dabei ist insbesondere das funktionsorientierte ATOS-System zu berücksichtigen.
- Die instrumentierende Komponente ist ein Quellcode-zu-Quellcode-Compiler. Die eigentliche Übersetzung erfolgt unabhängig durch einen Compiler für die benutzte Programmiersprache.

5.4.2 Produkteinsatz

Das Programm unterstützt den Benutzer bei der Erstellung und Auswertung anhand von Kriterien des strukturorientierten Softwaretests. Es ist vorgesehen zum Einsatz im Zusammenhang mit und als Ergänzung zur funktionalen Programmanalyse.

Anwendungsbereich

Die Auswertung findet nach einer gewissen Anzahl von Durchläufen des Programms statt, welches auszuwertende Daten erzeugen. Dazu wird das zu testende Programm instrumentiert, um Informationen über die Programmstruktur und über das Laufverhalten gewinnen zu können.

Zielgruppen

Es sind zwei verschiedene Zielgruppen zu unterscheiden

- Entwickler und Tester eines Programms
  Der Einsatz erfolgt in dieser Zielgruppe vor allem mit dem Zweck, vorhandene funktionsorientierte Testfälle so zu überprüfen und zu ergänzen, dass Überdeckungsmaße erfüllt werden.

- Studenten (Einsatz in der Lehre)
  Während im vorhergehenden Fall Kenntnis von Überdeckungsmaßen angekommen werden kann, ist beim Einsatz in der Lehre zu berücksichtigen, dass dies nicht der Fall ist und das Programm erst zur Gewinnung von Wissen über diese Maße genutzt wird.

Betriebsbedingungen

- Umgebung der Programmentwicklung und -wartung

- Lehrbetrieb

5.4.3 Produktumgebung

Das Hauptprogramm soll auf Endbenutzer-Systemen eingesetzt werden. Es soll nicht an eine bestimmte Plattform gebunden sondern portabel sein.

Software

Java 2-Plattform

Hardware

Keine Hardwareanforderungen, die über die der Java 2-Plattform hinausgehen

Produktschnittstellen

Da der Quellcode des zu untersuchenden Programms und die Ergebnisse der Tests und der Instrumentierung unabhängig vom Hauptprogramm erzeugt werden, müssen Schnittstellen definiert werden, die den Zugriff auf diese Daten erlauben.

Zu unterscheiden sind dabei insbesondere:

- dynamische Daten eines Projektes
  Ergebnisse von Testdurchläufen werden vom instrumentierten Programm in Log-Dateien gespeichert. Ebenso werden aus diesen Log-Dateien die Namen der Testfälle abgeleitet. Das Datenformat muss formal definiert sein, um Interoperabilität sicherzustellen.
- statische Daten eines Projektes
  Zur anschaulichen Darstellung der Auswertung ist Zugriff auf den Quellcode des Programms und die dazugehörigen Kontrollflussgraphen nötig. Aus dem Quellcode werden u.a. die Namen der Klassen eines Programms gewonnen. Da sich der Quellcode eines Projektes während des Auswertungsprozesses nicht ändert, sind Quellcode und Kontrollflussgraphen bzw. die Daten, aus denen diese Graphen gewonnen werden, statisch.

5.4.4 Use Case-Diagramm

Abbildung 5.1 stellt ein Use Case-Diagramm des System vor.

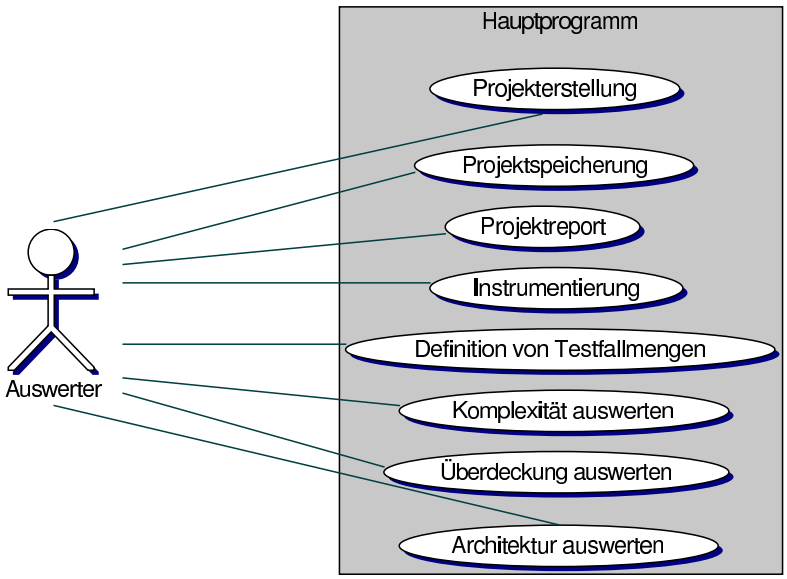


Abbildung 5.1: Use-Case-Diagramm

5.4.5 Produktfunktionen

Projekterstellung

Name	Projekterstellung
Zusammenfassung	Das Erstellen von Projekten umfasst im Sinne dieses Use-Cases das tatsächliche Erstellen von neuen Projekten inklusive der Eingabe der dazu benötigten Daten sowie das Laden eines Projektes und das Wiederherstellen einer konsistenten Projektkonfiguration.
Ergebnis	Ein Projekt wurde erstellt oder geladen.
Möglichkeiten der Interaktion	• /F10/ Erstellen eines neuen Projektes, Abfragen der benötigten Daten vermittels eines Assistenten (1. Variante) • /F20/ Erstellen eines neuen Projektes, Manuelle Eingabe der Daten (2. Variante) • /F30/ Die Manuelle Eingabe von Daten ist auch nach der erstmaligen Projekterstellung zur Änderung von Einstellungen möglich • /F40/ Laden von Projekten ist durch Auswahl von Projektdaten oder durch Auswahl aus einer Historie zuvor genutzter Projekte möglich

Ereignisse	• /F50/ Zur Festlegung der Instrumentierungslevel auf Klassen- und Methodenebene wird der Quellcode analysiert, so dass Klassen- und Methodennamen für die weitere Verarbeitung zur Verfügung stehen. • /F60/ Beim Laden eines Projektes werden die gespeicherten Daten mit der tatsächlichen Situation verglichen und der Nutzer bei Diskrepanzen aufgefordert, entsprechende Korrekturen an den Projekteinstellungen vorzunehmen
Annahmen	Vor der Abarbeitung dieses Use-Cases ist kein Projekt geladen. Sollte dies dennoch der Fall sein, wird dieses Projekt geschlossen entsprechend dem Use-Case *Projektspeicherung*.
Systemzustand vor Ausführung	Kein Projekt ist geladen. Dementsprechend findet auch keine Darstellung von Auswertungsergebnissen statt.
Systemzustand nach Ausführung	Ein Projekt ist geladen und befindet sich in einem konsistenten Zustand, so dass im Weiteren mit ihn gearbeitet werden kann.

Projektspeicherung

Name	Projektspeicherung
Zusammenfassung	Das Speichern von Projekten legt die Daten, die für einen Wiederherstellung des Projektes und seiner Einstellungen nötig sind, an einem vom Nutzer zu bestimmenden Platz ab
Ergebnis	Die Daten, die zur Wiederherstellung des Projektzustandes vor dem Speichern vorlagen, wurden gespeichert

Möglichkeiten der Interaktion	• /F70/ Speichern ein Projektes nach Auswahl der entsprechenden Programmfunktion
Ereignisse	• /F80/ Ein bereits gespeichertes Projekt wird unter dem selben Namen wieder gespeichert, es sei denn, dies wird vom Nutzer explizit anders gewählt • /F90/ Wurde ein Projekt noch nicht gespeichert, wird der Nutzer aufgefordert, einen Namen zu wählen.
Annahmen	Ein Projekt ist geladen
Systemzustand vor Ausführung	Keine Änderung des Systemzustandes
Systemzustand nach Ausführung	Keine Änderung des Systemzustandes

Projektreport

Name	Projektreport
Zusammenfassung	Erstellung eines Berichtes über das Projekt, der wichtige Informationen zusammenfasst. Ablage entweder im HTML-Format oder zur Druckausgabe
Ergebnis	Ein Bericht wurde erstellt
Möglichkeiten der Interaktion	• /F100/ Auswahl der Reportfunktion und Eingabe der nötigen Daten über Typ und Umfang des Berichtes

Ereignisse	• /F110/ Ausgabe nach Auswahl im HTML-Format (Ablage als Datei) • /F120/ Ausgabe zum Drucker
Annahmen	Ein Projekt ist geladen
Systemzustand vor Ausführung	Keine Änderung des Systemzustandes
Systemzustand nach Ausführung	Keine Änderung des Systemzustandes

Definition von Testfallmengen

Name	Definition von Testfallmengen
Zusammenfassung	Möglichkeit, eine Verknüpfung zwischen Klassen und Testfällen herzustellen, um etwa die zu einem Subsystem gehörigen Klassen und Testfälle unter einem Bezeichner zu vereinen
Ergebnis	Eine Änderung an den Definitionen von Testfallmengen wurde durchgeführt (Änderung, Löschung bzw. Erstellung einer Menge)
Möglichkeiten der Interaktion	• /F130/ Anzeige der Klassen eines Projektes mit Auswahlmöglichkeit • /F140/ Anzeigen von Testfällen mit Auswahlmöglichkeit [1] • /F150/ Auswahl von Mengen von Klassen und Testfällen und Speicherung unter einem Namen • /F160/ Änderung von bereits definierten Testfall/Klassen-Mengen
Ereignisse	Keine
Annahmen	Ein Projekt ist geladen

[1] In der Regel (d.h. beim gemeinsamen Einsatz mit dem ATOS-System), sind diese Testfälle diejenigen von ATOS. Sie werden beim Laden eines Projektes eingelesen.

Systemzustand vor Ausführung	-
Systemzustand nach Ausführung	Die Testfallmengen-Definitionen wurden geändert (erstellt, ergänzt, gelöscht)

Auswertung im Bezug am Komplexität

Name	Auswertung im Bezug am Komplexität
Zusammenfassung	Nach Öffnen eines Fensters für die Komplexitätsbetrachtung werden in diesem Bereiche für die Darstellung von Klassen und Methoden, für den Quellcode und den Kontrollflussgraphen angezeigt. Eine Auswahl einer Methode führt zur Anzeige des entsprechenden Quelltextes und des Kontrollflussgraphen, sowie der zyklomatischen und essentiellen Komplexität.
Ergebnis	Eine Auswertung im Bezug auf Programmkomplexität wurde vorgenommen.
Möglichkeiten der Interaktion	• /F170/ Nach Auswahl einer Methode wird der entsprechende Kontrollflussgraph angezeigt • /F180/ Methoden können nach verschiedenen Kriterien (etwa alphabetisch oder nach Komplexität) sortiert werden.

Ereignisse	• /F190/ Aufbau eines Fensters für Darstellung der Komplexitätsmaßzahlen nach Auswahl in oberer Symbolleiste oder im Menü • /F200/ Anzeige einer Klassen- & Methodenliste • /F210/ Essentielle & zyklomatische Komplexität werden gemeinsam mit den Methodennamen angezeigt
Annahmen	Ein Projekt ist geladen.
Systemzustand vor Ausführung	Keine Änderung des Systemzustandes.
Systemzustand nach Ausführung	Keine Änderung des Systemzustandes.

Auswertung im Bezug auf Überdeckung

Name	Auswertung im Bezug auf Überdeckung
Zusammenfassung	Ein Fenster zur Auswertung der Überdeckungsmaße wird geöffnet, in dem Klassen, Kontrollflussgraphen, Quellcode und Testfälle dargestellt werden. Nach Auswahl der entsprechenden Elemente werden werden überdeckter Quellcode, Kontrollflussgraphen und die Erfüllung von Überdeckungskriterien angezeigt.
Ergebnis	Eine Auswertung im Bezug auf Überdeckung hat stattgefunden.

Möglichkeiten der Interaktion	• /F220/ Der Nutzer kann optional alle angezeigten Elemente ein- und ausblenden • /F230/ Nach Selektion von Testfällen werden überdeckte Codeteile farbig hervorgehoben • /F240/ Nach Selektion von Testfällen werden berührte Klassen farbig hervorgehoben • /F250/ Nach Selektion von Testfällen werden berührte Teile des betreffenden Kontrollflussgraphen farbig hervorgehoben • /F260/ Bei allen Hervorhebungen ist jeweils eine Umkehrung des Kriteriums möglich • /F270/ Für die jeweils aktive Auswahl von Klassen- und Testfall-Selektion wird die Erfüllung von Überdeckungskriterien angegeben • /F280/ Belegungen von Konditionalen werden wegen ihrer Komplexität separat dargestellt
Ereignisse	• /F290/ Nach Anwahl aus Menü oder oberer Symbolleiste öffnet sich ein Darstellungsfenster • /F300/ Anzeige und Auswahlmöglichkeit von Klassen, Kontrollflussgraphen, Quellcode und Testfällen

Annahmen	Ein Projekt ist geladen und ein Durchlauf des instrumentierten Quellcode hat stattgefunden (d.h. es liegen Daten über Testfälle und Log-Dateien der Instrumentierung vor).
Systemzustand vor Ausführung	Keine Änderung des Systemzustandes.
Systemzustand nach Ausführung	Keine Änderung des Systemzustandes.

Auswertung im Bezug auf die Programmarchitektur

Name	Auswertung im Bezug auf die Programmarchitektur
Zusammenfassung	Ein Fenster zur Auswertung der Architektur des aktuellen Projektes wird geöffnet. In diesem Fenster werden Klassen und Testfälle dargestellt. Eine Auswahl von Klassen stellt berührte Testfälle dar. Ebenso wird nach Auswahl von Testfällen die berührte Klassenmenge hervorgehoben.
Ergebnis	Eine Architekturauswertung hat stattgefunden.
Möglichkeiten der Interaktion	• /F300/ Nach Auswahl einer Menge von Klassen kann der Nutzer sich die berührten Testfälle anzeigen lassen • /F310/ Nach Auswahl einer Menge von Testfällen ist eine Anzeige der von diesen berührten Klassen möglich
Ereignisse	• /F320/ Aufbau eines Fensters zur Gewinnung von Architekturinformationen nach Auswahl aus Menü oder oberer Symbolleiste • /F330/ Anzeige von Klassen und Testfällen
Annahmen	Ein Projekt ist geladen.

Systemzustand vor Ausführung	Keine Änderung des Systemzustandes.
Systemzustand nach Ausführung	Keine Änderung des Systemzustandes.

Instrumentierung des Quelltextes

Der Use-Case, der die Instrumentierung des Quelltextes des originales Programms beschreibt, ist zweifellos der komplexeste aller hier aufgeführten. Da die dort zu beschreibenden Operationen in einer anderen Umgebung und zu einem anderen Zeitpunkt als die des Hauptprogramms stattfinden können, wurde die Beschreibung in ein separates Pflichtenheft *Instrumentierer* ausgelagert, um diesen Gesichtspunkten gerecht zu werden.

5.4.6 Produktdaten

- /D10/ Projektbezogene Daten, die bei der Projekterstellung ermittelt werden
  - Verzeichnisangaben (Quellcode, Logdateien)
  - zu betrachtende Klassen
  - Angaben zum Ausmaß der Instrumentierung für einzelne Klassen bzw. Methoden
  - Sitzungspezifische Daten
- /D20/ Die Teilmenge der für den Instrumentierer wichtigen Daten werden in einer separaten Konfigurationsdatei gespeichert, die dieser einliest.

5.4.7 Benutzerschnittstelle

- /B10/ Bedienung ist menü- und mausorientiert
- /B20/ Grundregeln zur Gestaltung grafischer Oberflächen sind zu beachten (insbesondere die *Java Look and Feel Design Guidelines* unter `http://java.sun.com/products/jlf/ed1/dg/index.htm`)
- /B30/ Die Benutzeroberfläche ist in 4 Hauptbereiche gegliedert
  - Menüleiste
    Zugriff auf alle Programmfunktionen

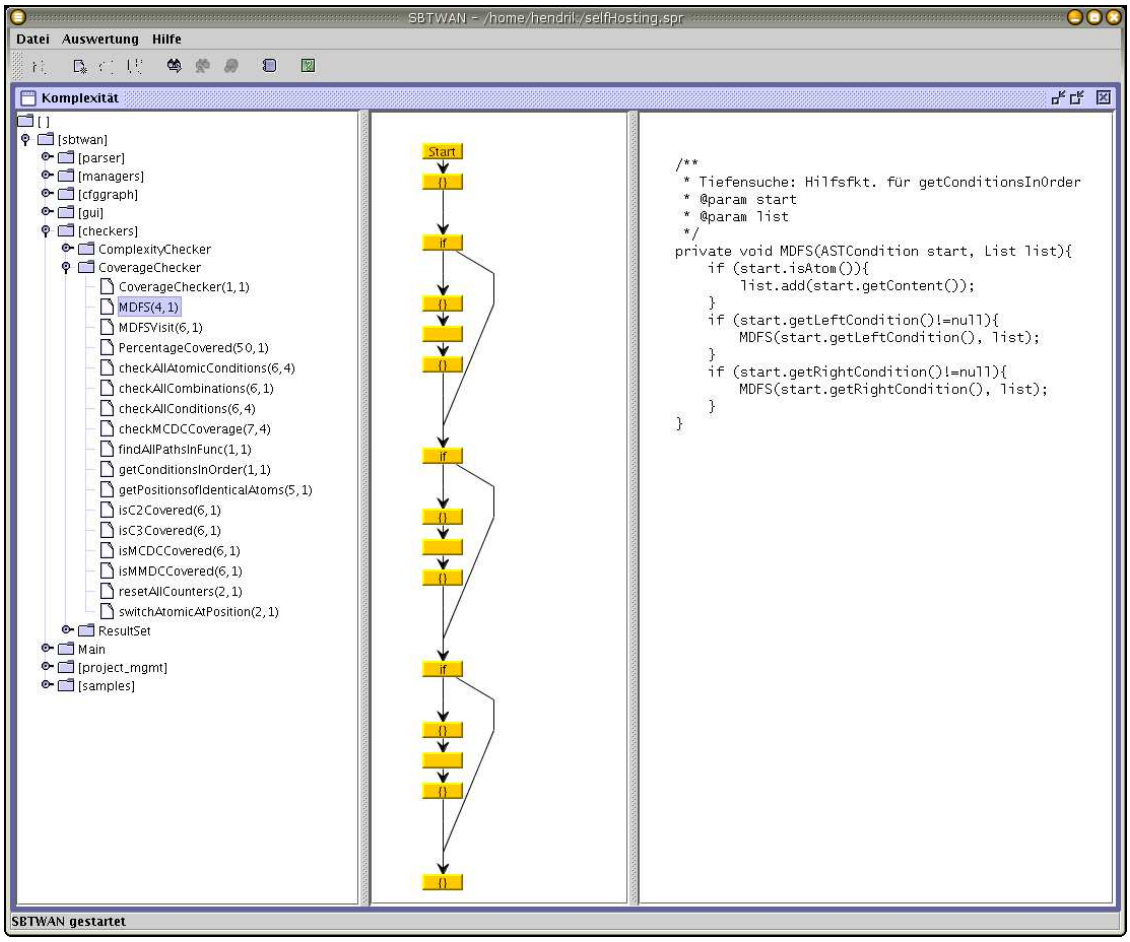


Abbildung 5.2: Aufbau der GUI

- obere Symbolleiste
  Schnellzugriff auf wichtige Funktionen, insbesondere Umschaltung zwischen Ansichtsmodi
- Darstellungsbereich
  Darstellung der ausgewählten Daten

5.4.8 Testfälle

Während beim Use-Case „Instrumentieren“ das besondere Augenmerk darauf liegt, dass das zu instrumentierende Programm in seiner Ausführung möglichst wenig verändert wird (da die gewonnen Daten dann kaum mehr interpretierbar wären),

muss bei der Gestaltung der übrigen Elemente unter anderem auf Robustheit gegenüber Fehleingaben durch Nutzer und auf Konsistenz in der Benutzerführung geachtet werden. Natürlich ist daneben die korrekte Auswertung des durch die Instrumentierung gewonnenen Datenmaterials sicherzustellen.

Grundlegende Bereiche, die getestet werden müssen (und dabei in einzelne, detailliertere Testfälle aufgegliedert werden) sind:

- /T10/ Projekterstellung
  Reaktion auf Fehleingaben, Verifikation der Korrektheit von Eingaben
- /T20/ Laden ein Projektes
  Wiederherstellung von alten Einstellungen, Wiederherstellung von Sitzungsdaten, Abgleich der gespeicherten Projekt-Daten mit der tatsächlichen Situation (geänderte Pfade etc.)
- /T30/ Integration mit ATOS
  Übernahme von Testfällen, Abgleich zwischen gespeicherten Informationen über Testfälle mit tatsächlichem Zustand
- /T40/ Korrektheit der Graphen zur Komplexitätsermittlung
  Vergleich der ermittelten Graphen mit Referenz-Graphen
- /T50/ Verifikation der korrekten Ermittlung der einzelnen Überdeckungsmaße
  Vergleich der ermittelten Überdeckungsmaße mit Referenzen
- /T60/ Architekturanzeige
  Vergleich mit Referenz-Daten

5.4.9 Entwicklungsumgebung

Betriebssystem

Die Wahl der Betriebssystems unterliegt keinen besonderen Einschränkungen, das Hauptprogramm sollte aber insbesondere auf denselben Plattformen wie ATOS (Win32-Umgebung) und für den Einsatz in der Lehre auf den dort gängigen Plattformen (Windows, Solaris, Linux) einsetzbar sein. Die Java 2-Plattform erleichtert diesen Vorgang erheblich, allerdings müssen auf den verschiedenen Plattformen weiterhin Spezifika etwa bezüglich des Aufbaus von Dateipfaden oder der Konfigurationsspeicherung beachtet werden.

Entwicklungsumgebung

Da die Java 2-Plattform genutzt werden wird, die keine besonderen Anforderungen an die Nutzung einer speziellen Entwicklungsumgebung stellt, ist die Festlegung auf eine Solche nicht zwingend notwendig.

Werkzeuge, die aber dennoch Berücksichtigung finden sollten:

- Versionskontrollsystem (CVS)
- GUI-Builder zur Gestaltung der grafischen Oberflächen
- Quellcode-Dokumentation (JavaDoc)

5.4.10 Qualitätsbestimmung

Wegen des Einsatzes in der Lehre, wo Einarbeitungszeiten kürzer sind, als bei einem regelmäßigen Einsatz im Rahmen eines oder mehrerer Software-Entwicklungsprojekte, sollte ein Schwerpunkt auf guter Benutzbarkeit (Verständlichkeit, Erlernbarkeit, Bedienbarkeit) liegen.

Um die Anpassung an neue Programmiersprachen und so zukünftige Einsetzbarkeit zu erleichtern, ist auf einfache Änderbarkeit (Analysierbarkeit, Modifizierbarkeit) zu achten bzw. die Implementierung des Hauptprogramm gänzlich unabhängig von speziellen Sprachkonstrukten zu halten.

5.4.11 Anhang des Pflichtenheftes

Darstellung von Quellcode und Kontrollflussgraph

Die Instrumentierung von Quellcode ist ein Vorgang, der den originalen Quellcode vom Umfang und damit in den meisten Fällen auch im Bezug auf die zur Ausführung notwendigen Ressourcen erheblich vergrößert. Dabei erfordern komplexere Überdeckungsmaße (insbesondere die Auswertung von Bedingungen) aufwendigere Instrumentierung.

Es wird daher angestrebt, die Instrumentierung abgestuft nach den Wünschen und Bedürfnissen des Nutzers individuell konfigurierbar zu gestalten.

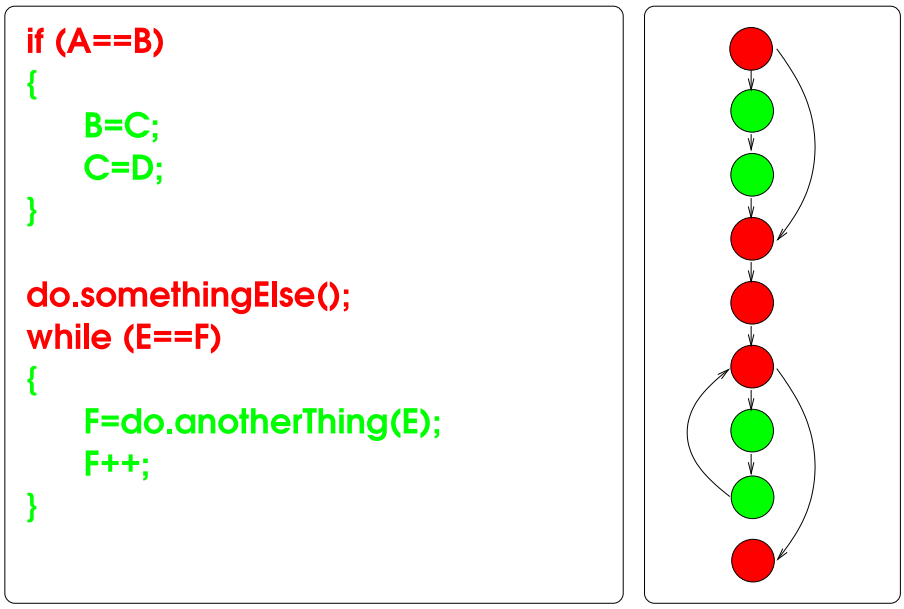


Abbildung 5.3: Mögliche Anzeige auf Basis einfacher Instrumentierung

Mögliche Stufen sind dabei

- Instrumentierung für Anweisungsüberdeckung
- Instrumentierung für Zweigüberdeckung
- Ermöglichung von Bedingungsauswertung

Die Abbildung 5.3 stellt eine beispielhafte Anzeige bei einfacher Instrumentierung für Anweisungsüberdeckung dar.

In dieser einfachsten Form der Überdeckungsbetrachtung, werden nur Knoten betrachtet. Nicht überdeckte Knoten sind grün dargestellt, überdeckte Knoten sind rot dargestellt. Entsprechendes gilt für die ausgeführten bzw. nicht ausgeführten Quellcodezeilen.

Im Beispiel wurden die if-Verzweigung und die while-Schleife nicht durchlaufen.

Abbildung 5.4 stellt den selben Programmablauf wie 5.3 dar. Allerdings wurden in diesem Fall auch die nötigen Informationen durch die Instrumentierung gesammelt ist, die nötig sind, um Zweigüberdeckung auszuwerten. Pfade im Kontrollflussgraphen, die durchlaufen wurden, sind rot vermerkt, nicht benutzte grün.

Abbildung 5.4: Mögliche Anzeige auf Basis von Instrumentierung zur Auswertung der Zweigüberdeckung

6 Design

6.1 Struktur

Die Gliederung der Aufgabenstellung, wie sie im Pflichtenheft vorgenommen wurde, legt eine Trennung der Struktur in Komponenten nahe, die jeweils verschiedene Aufgaben übernehmen und in verschiedenen Kontexten genutzt werden.

Weiterhin ist aus Gründen der Übersichtlichkeit, Wiederverwendbarkeit und nicht zuletzt Wartbarkeit eine Trennung zwischen den Komponenten, die die grafische Oberfläche zur Verfügung stellen und denen, die auf den Eingabedaten operieren und diese auswerten, wünschenswert. Die nicht-GUI Komponenten sind weiter untergliedert in die Teilsysteme, die die Daten aus den verschiedenen Quellen (Quellcode, Testfälle, Logdateien des instrumentierten Programms) lesen, überwachen und in einen konsistenten Zustand bringen und denen, die auf diesen Daten operieren und diese auswerten (Komplexitätsmaße, Überdeckungsauswertung, Betrachtung von Testfällen, Darstellung von Kontrollflussgraphen).

6.1.1 GUI

Für die GUI hat sich aus diesen Beweggründen eine Einteilung in drei Grundmodi, die drei verschiedene Ansichten auf die zur Verfügung stehenden Daten zur Verfügung stellen, ergeben:

- Betrachtung der Komplexität
- Auswertung von Überdeckungsmaßen
- Architekturauswertung

Diese Modi teilen sich untereinander eine Reihe von Darstellungselementen, die jeweils in einer Komponente gekapselt sind. Im Weiteren werden diese Komponenten

als *Einzelsichten* bezeichnet und die Realisierung der drei Grundauswertungen als *zusammengesetzte Sichten*.

Zusammengesetzt sind die zusammengesetzten Sichten aus folgenden Einzelsichten:

- Strukturansicht des geladenen Softwaresystems
- Kontrollflüsse
- Quelltextansicht
- Testfallansicht
- Überdeckungsmaßerfüllung

Diese Elemente der grafischen Oberfläche sind in einem Hauptfenster zusammengefasst, von dem aus die verschiedenen zusammengesetzten Sichten aufgerufen werden.

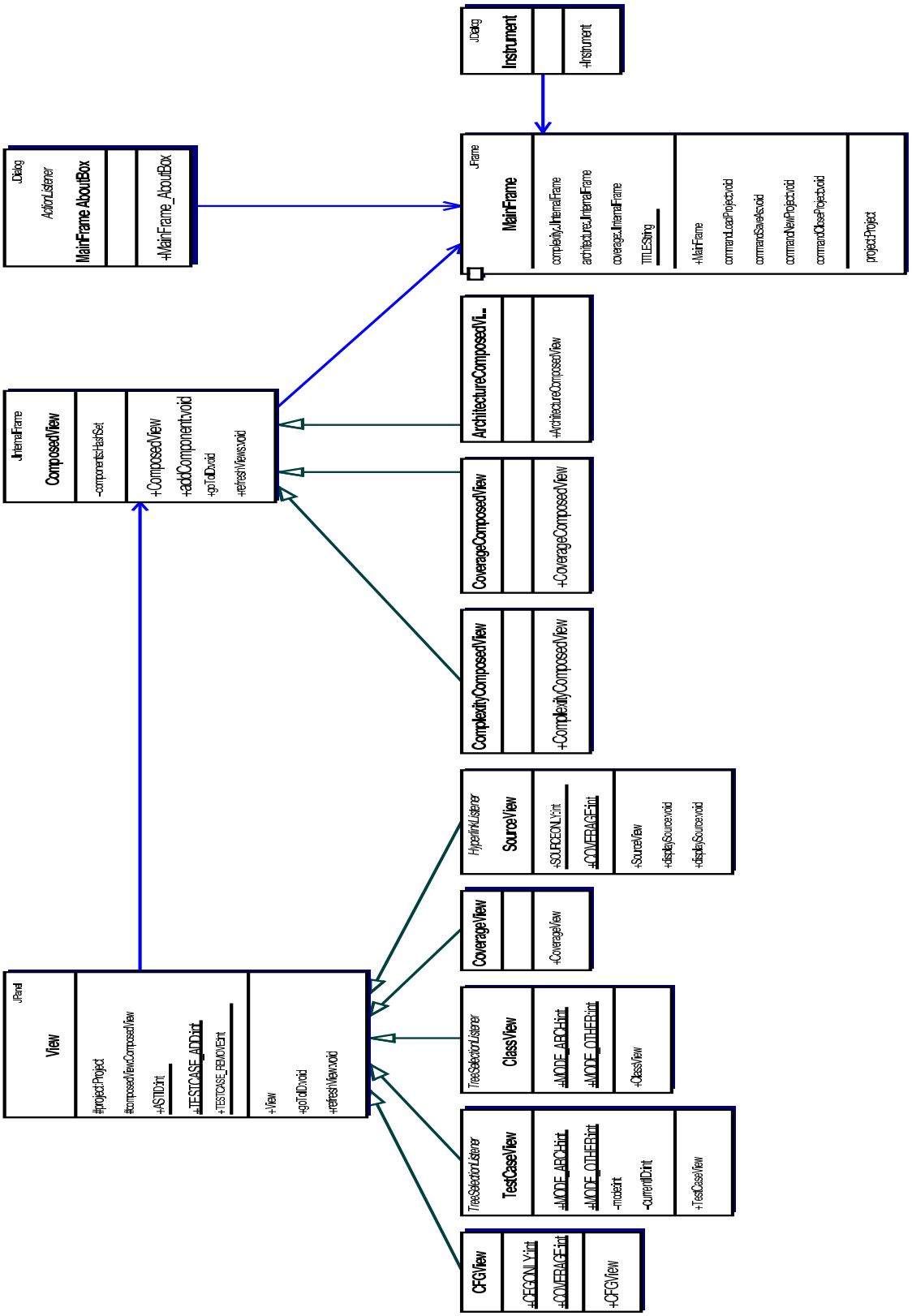


Abbildung 6.1: UML-Diagramm des Pakets GUI

6.1.2 Kommunikation der Sichten untereinander

Die verschiedenen Sichten auf die Daten des Projektes, die zu einer kombinierten Sicht zusammengefasst werden, müssen untereinander kommunizieren. Die Strukturansicht erlaubt das Auswählen von Klassen und Methoden und in der Testfallansicht können Testfälle an- und abgewählt werden.

Sichten verfügen daher über Operationen, um sich untereinander über solche Ereignisse zu unterrichten. Diese Methoden sind in der Klasse *View* (siehe Abbildung 6.1) definiert, ebenso die Typen von Ereignissen, die eintreten können:

- Auswahl einer Klasse oder Methode
- Abwahl einer Klasse oder Methode
- Anwählen eines Testfalls
- Abwählen eines Testfalls

Tritt eines dieser Ereignisse ein, informiert die Sicht, in der die Änderung vorgenommen wurde, die übergeordnete zusammengesetzte Sicht, welche wiederum alle eingebetteten Einzelsichten darüber informiert.

Wenn alle entsprechenden Ereignisse weitergeleitet sind, werden die Einzelsichten aufgefordert, ihre Darstellung dem neuen Zustand anzupassen.

6.1.3 Betrachtung der Komplexität

Die zusammengesetzte Ansicht, die der Auswertung der Komplexität dient, wird aufgebaut aus drei der Grundkomponenten:

- logische Struktur des Programms. D.h. ein Baum, der Klassen und Methoden darstellt. Dieser ermöglicht eine Auswahl von Klassen und Methoden.
- Klassen und Methoden stellt die Quelltextansicht dar.
- Für Methoden wird zusätzlich der entsprechende Kontrollflussgraph konstruiert und angezeigt.

Zusätzlich wird zusammen mit dem Methoden- bzw. Funktionsnamen jeweils die essentielle und zyklomatische Komplexität angezeigt.

6.1.4 Auswertung von Überdeckungsmaßen

Auch die Auswertung von Überdeckungsmaßen nutzt die selben Komponenten wie die Komplexitätsauswertung des vorherigen Abschnitts.

Zusätzlich bestehen folgende Möglichkeiten:

- Testfälle anzeigen und anwählen
- Ergebnisse der Überdeckungsauswertung betrachten

Diese Funktionalität wird von zwei getrennten Komponenten bereitgestellt.

Die Sichten für Quelltextanzeige und Kontrollflussdarstellung werden darüber hinaus in einem erweiterten Modus genutzt, der durch Einsatz von Farbe entsprechende Überdeckungsmaße darstellt. So kann die Kontrollflussansicht Knoten und Kanten farbig unterlegen, um Anweisungs- und Kantenüberdeckung zu verdeutlichen.

Die Quelltextansicht stellt Anweisungsüberdeckung farbig dar und bietet darüber hinaus die Möglichkeit, die Belegungen von zusammengesetzten und atomaren Bedingungen anzuzeigen, nachdem diese durch den Nutzer angewählt worden sind.

Ist ein Testfall oder eine Menge von Testfällen in der Testfallansicht angewählt, so ist es Aufgabe der Sichten für Kontrollflüsse, Quelltext und Überdeckungsmaße, das Ergebnis dieser Auswahl wie beschrieben darzustellen. Die Überdeckungsansicht zeigt darüber hinaus die Erfüllung von Überdeckungskriterien an, wobei sowohl angegeben wird, ob diese erfüllt sind, als auch, zu welchem Anteil sie erfüllt sind.

6.1.5 Architekturauswertung

Die statische Betrachtung des Quellcodes ohne weitere Eingriffe in diesen gestattet die Ermittlung von Komplexitätsmaßzahlen und die Erstellung von Kontrollflussgraphen, die Instrumentierung von Quellcode ermöglicht im Zusammenspiel mit einem funktionsorientierten Testsystem, das das Softwaresystem oder seine Komponenten kontrolliert ausführt, die Ermittlung von verschiedenen Überdeckungsmaßen.

Instrumentierung ermöglicht aber noch andere Einblicke in das zu untersuchende Programm. Überdeckungsmessung ermittelt, auf welche Art und Weise Code durchlaufen wird. Interessant kann aber auch die Frage sein, welcher Code überhaupt

durchlaufen wurde und welche Kombination von Eingabedaten dabei den Ausschlag gab.

Diese Frage mag bei kleineren Systemen trivial erscheinen, aber bei wachsender Komplexität des zu testenden Systems wird sie umso wichtiger. Gesteigert ist das Interesse an dieser Frage insbesondere bei Reverse-Engineering-Prozessen, wie sie im Rahmen der Arbeit am XCTL-System von besonderer Bedeutung sind. Da die ursprünglichen Design-Erwägungen des zu testenden Systems nicht notwendigerweise vollständig bekannt sind, lassen sich durch die Untersuchung, welche Eingabedaten die Ausführung welchen Codes bedingen, wichtige Rückschlüsse ziehen.

Die Auswertung der Projekt-Daten im Bezug auf die Architektur kennt zwei Blickwinkel:

- *Ausgehend von der Programmstruktur*
  Betrachtet man die Klassen und Methoden des Programms, interessiert die Frage, welche Testfälle ihre Ausführung ausgelöst haben.
- *Ausgehend von den Testfällen*
  Ein Testfall ist in der Regel darauf ausgelegt, eine bestimmte Teilfunktionalität zu testen. Von Interesse ist, welcher Code ausgeführt wird, um die getestete Funktionalität bereitzustellen.

Die Architektur-Auswertung muss also eine Übersicht über die logische Programmstruktur bereitstellen und diese mit einer Übersicht über die Testfälle ergänzen und eine Auswertung in beiden zuvor betrachteten Richtungen erlauben. Dazu sind bei Auswahl von Testfällen die Programmkomponenten hervorzuheben, die betroffen sind, und bei Auswahl von Programmkomponenten die Testfälle, die deren Ausführung beinhalten.

6.1.6 Kontrollflussgraphen und Komplexität

Die Daten, die von der GUI dargestellt werden, werden von sog. *Managern* eingelesen und verwaltet. Es gibt Manager für die folgenden Datenquellen:

- Quellcode
- Testfälle
- Log-Dateien

Die Daten, die von Managern verwaltet werden, sind jeweils externe Daten, die außerhalb des Programms erstellt werden und sich ändern können, so dass die Konsistenz der Daten untereinander verloren gehen kann. Aufgabe der Manager ist es daher, die Konsistenz der Daten mit den anderen Daten, die zusammen ein Projekt bilden, sicherzustellen.

Auf Basis dieser dann konsistenten Daten ist eine Auswertung nach verschiedenen Kriterien möglich. Dazu existieren Komponenten, die Kontrollflussgraphen als zentrale Abstraktion des Quellcodes erstellen, und Module, die auf diesen Graphen operieren:

- Kontrollflussgraph-Ersteller
- Ermittlung von Komplexitätsmaßzahlen
- Überprüfung von Überdeckungsmaßen

6.1.7 Erstellung von Kontrollflussgraphen

Kontrollflussgraphen als abstrakte Darstellung des Kontrollflusses einer Methode bzw. Funktion sind eine der zentralen Strukturen dieser Arbeit, da sie bei der Auswertung der meisten hier betrachteten Metriken Verwendung finden. Das bedeutet auch, dass für die Darstellung eine Form gewählt werden muss, die einheitlich für verschiedene Anwendungen ist, aber flexibel genug bleibt, um den jeweiligen Ansprüchen gerecht zu werden.

Kontrollflussgraphen werden aus der Darstellung der Struktur einer Methode in sprachunabhängiger Form konstruiert. Wichtig bei der korrekten Erstellung der Graphen sind naheliegenderweise die Elemente des betrachteten Moduls, die Verzweigungen des Kontrollflusses abbilden.

Der Algorithmus lässt sich folgendermaßen darstellen:

1. Parameter: Quellcode, geparst und in abstrakte Darstellung gebracht
2. Bestimmung von Anfang und Ende des aktuell zu bearbeitenden Bereiches, in der Regel sind dies die Grenzen einer Methode
3. Linearer Durchlauf durch die Methode
4. Bei kontrollflussrelevanten Elementen jeweils rekursiver Aufruf der Kontrollflusskonstruktion

5. Aus der Kontrollflusskonstruktion Aufruf von entsprechenden Methoden, die Knoten und Kanten in den Graphen einfügen

6. Rückgabe des Graphen

Das UML-Diagramm in Abbildung 6.2 veranschaulicht die Struktur der Graph-Erstellung. Die zentrale Klasse dieses Paketes ist `GraphMaker`, die die Routinen zum Analysieren des vom Parser erzeugten Syntaxbaumes und die Routinen zum Layouten des Graphen enthält. Der eigentliche Layoutvorgang findet in der Routine `layout` statt, die rekursiv aufgerufen wird und dabei jeweils Unterroutinen aufruft, die einzelne Konstrukte im zugrunde liegenden Programm entsprechend darstellen:

- `tryCatchLayout`
  Auswertung von try-catch-Ausdrücken
- `switchLayout`
  Auswertung von switch-case-Ausdrücken
- `compoundLayout`
  Auswertung von Blöcken in geschweiften Klammern
- `doWhileLayout`
  Auswertung von do-while-Konstrukten
- `iterationLayout`
  Auswertung von while- und for-Schleifen
- `ifLayout`
  Auswertung von if-elseif-else-Verzweigungen

Kontrollflussgraphen finden für zwei Anwendungen Verwendung:

- Grafische Darstellung des Kontrollflusses
- Interne Darstellung für Komplexitätsermittlung

Um dies zu modellieren, werden von `GraphMaker` zwei Unterklassen abgeleitet, nämlich `JGraphMaker` und `AbstractGraphMaker`.

`AbstractGraphMaker` erzeugt Graphen für die interne Darstellung, auf deren Basis etwa Komplexitätsberechnungen durchgeführt werden. Diese Graphen werden

mit der Datenstruktur `AbstractGraph` dargestellt, die alle relevanten Informationen zu den Graphen und ihren Knoten, die als `GraphVertice` abgelegt werden, kapselt.

Die Klasse, die Graphen für die graphische Darstellung erstellt ist `JGraphMaker`. Die Datenstruktur für diese Art der Darstellung wird in der JGraph-Bibliothek (siehe [jgr]) definiert und wurde somit nicht im Rahmen dieser Arbeit erstellt.

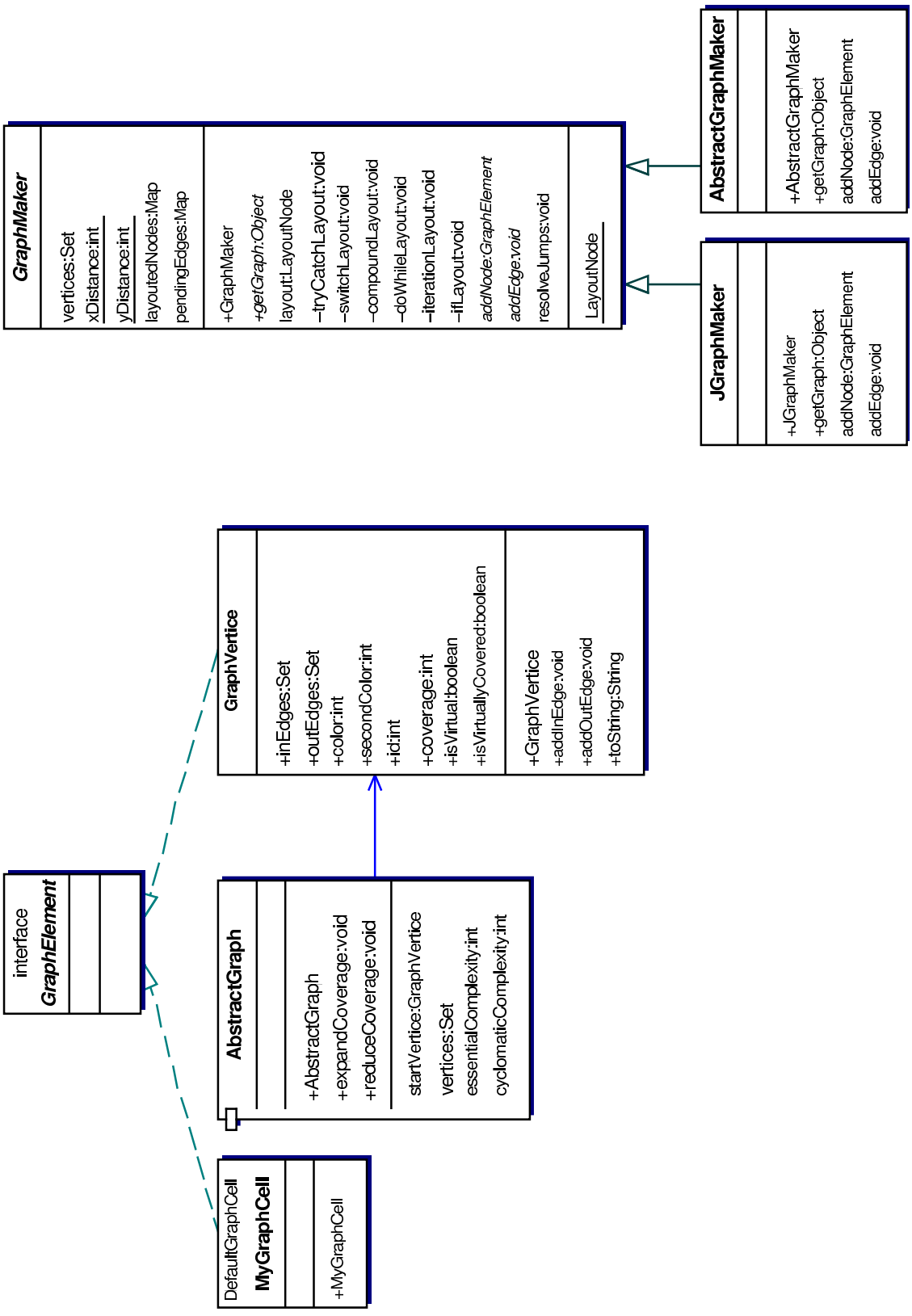


Abbildung 6.2: UML-Diagramm des Pakets Kontrollflussgraph

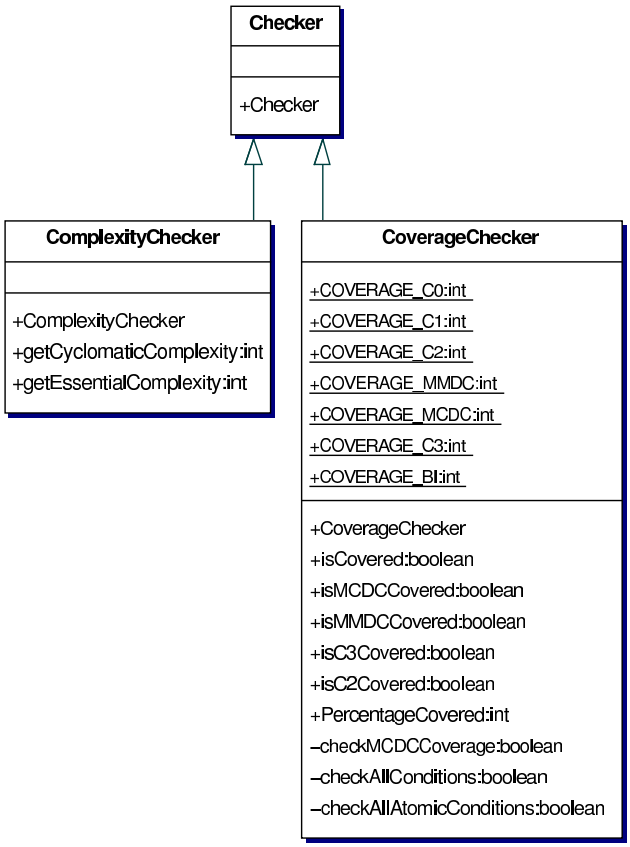


Abbildung 6.3: UML-Diagramm des Pakets Checkers

6.1.8 Ermittlung der Daten-Eigenschaften

Zentraler Teil des Programms ist die Auswertung von geladenen Daten im Bezug auf Komplexitäts- und Überdeckungsmaße. Da diese beiden Aufgaben jeweils sehr umfangreich sind, sind sie je in einer Klasse zusammengefasst (siehe Abbildung 6.3).

6.1.9 Komplexitätsermittlung

Ist der Kontrollflussgraph konstruiert, wird es möglich, die statischen Metriken zur Abschätzung der Komplexität zu bestimmen.

Die zyklomatische Komplexität lässt sich direkt aus der Darstellung des Kontrollflussgraphen ableiten. Für die Bestimmung ist lediglich ein Zählen von Kanten und Knoten des Kontrollflussgraphen nötig.

1 **REDUCE_GRAPH(Kontrollflussgraph G)**
2 while (G was reduced) durchlaufe $v_{Start}|n^-(v_{Start}) \geq 2$
3 DFS (starte bei v_{Start}, stop bei $v_{Stop} \in V_G|n^-(v_{Stop} \geq 2)$, merke v_{Stop})
4 if (size($\{v_{Stop}\}$) $\geq 1 \rightarrow$ *continue*;
5 //Potenzieller Prime
6 start = v_{Start}
7 stop = v_{Stop}
8 verschiebe (stop)
9 * Existiert zweiter Eingang? dann $\rightarrow$ continue;
10 $G = G \setminus Prime(start, stop)$
11 return G;

Abbildung 6.4: Algorithmus zur iterativen Entfernung von Primes

Die Bestimmung der essentiellen Komplexität gestaltet sich deutlich komplexer. Wie in Kapitel drei erläutert, ist es das Ziel, die Unstrukturiertheit eines Moduls zu messen. Als strukturiert werden Konstrukte betrachtet, die im Kontrollfluss nur durch einen Knoten betreten werden und nur durch einen Knoten verlassen werden sowie selbst keinen derartigen Subgraphen enthalten.

Im folgenden wird ein algorithmischer Ansatz vorgestellt, um diese *Primes* genannten Subgraphen zu finden und iterativ zu entfernen.

Algorithmus zur Bestimmung der essentiellen Komplexität

Gemäß der Definition der essentiellen Komplexität entfernt der Algorithmus Primes aus gegebenen Kontrollflussgraphen in einem iterativen Vorgang solange, bis kein weiterer Prime mehr gefunden und entfernt werden kann. Der verbleibende Restgraph wird dann im Bezug auf seine zyklomatische Komplexität untersucht, die der essentiellen Komplexität des originalen Graphen entspricht.

Der Algorithmus sucht in Zeile 2 beginnend beim Startknoten des Graphen Knoten mit einem Ausgangsgrad größer als 2, d.h. Knoten, bei denen sich der

Kontrollfluss verzweigt. Ausgehend von einem solchen Knoten wird mit einer modifizierten Tiefensuche nach Nachfolgern dieses Knoten gesucht, die mehr als nur einen Nachfolger haben. Diese Knoten sind insofern interessant, als sie potenziell wieder den Beginn eines Primes darstellen können, die in einem Prime nicht verschachtelt vorkommen dürfen.

Wird nur ein solcher Endknoten gefunden, so wurde ein Konstrukt entdeckt, das einen Eingang und einen Ausgang besitzt und keinen Prime als Subgraph enthält. Anschließend untersucht der Algorithmus, ob tatsächlich ein Prime gefunden wurde. Dazu wird zunächst der bisher gefundene Endknoten verschoben, da die Tiefensuche unter Umständen zu tief in den Graphen hinab gestiegen ist. Anschließend wird geprüft, ob im untersuchten Modul nicht von anderer Stelle in den gefundenen Subgraphen gesprungen wird, da dies für Primes nicht erlaubt ist. In diesem Fall gäbe es mehr als einen Eingang in den Subgraphen.

Sind alle diese Bedingungen erfüllt, wird der gefundene Subgraph zwischen Start- und Endknoten aus dem Graphen entfernt und der Algorithmus versucht iterativ, einen weiteren Prime zu finden. Gelingt dies nicht, ist die Entfernung von Primes und damit die Reduktion des Graphen abgeschlossen.

Die Operationen, die im in Abbildung 6.4 dargestellten Algorithmus Verwendung finden, sind:

`verschiebe (stop)`	Zu Beginn der Suche von Prime-Kandidaten durchsucht der Algorithmus mit Hilfe der Tiefensuche den Graphen und markiert alle Knoten, bis er auf einen stößt, an dem sich der Kontrollfluss aufteilt, der also mehr als einen Nachfolger hat. Dabei läuft er unter Umständen zu weit. `verschiebe (stop)` setzt den Endknoten des Prime-Kandidaten auf eine frühere Position im Graphen. Solange der aktuelle Endknoten nur einen markierten Vorgänger hat und dieser Vorgänger keinen markierten Nachfolger außer dem aktuellen Endknoten hat, so wird dieser Vorgänger neuer Endknoten. Dieser Vorgang wird iteriert, solange sich ein neuer Endknoten finden lässt.
$G = G \setminus Prime(start, stop)$	Entfernt ein Prime aus dem Graphen. Dies geschieht, indem zunächst alle markierten Nachfolger- und Vorgänger des Startknotens gelöscht werden, ebenso alle markierten Vorgänger des Endknotens. Anschließend wird zwischen Start- und Endknoten eine neue Kante eingefügt.

6.1.10 Überdeckungsmaße

Die Überdeckungsmaße, die von SBTWAN aufgezeichnet werden, fallen in drei Kategorien:

- Anweisungs- und Zweigüberdeckung
- verschiedene Maße der Bedingungsüberdeckung
- Boundary-Interior-Pfadtest

Jede dieser Kategorien erfordert andere Herangehensweisen bei der Auswertung der Erfüllung der Überdeckungsmaße und stellt somit auch andere Anforderungen an die Daten, die die Instrumentierung eines Programms aufzeichnen muss.

Dabei ist die Anweisungsüberdeckung das am einfachsten auszuwertende Überdeckungsmaß. Jeder eine Anweisung darstellende Knoten des abstrakten Syntaxbaumes des Parsers verfügt über ein Feld, in dem festgehalten wird, wie oft die jeweilige Anweisung ausgeführt worden ist. Anweisungsüberdeckung wird auf Methodenebene betrachtet. Zur Auswertung genügt es daher, die Knoten, die eine Methode darstellen zu durchlaufen und ihre Zähler zu betrachten. Aus dem Verhältnis zwischen Knoten, die einen Zählerwert von wenigstens Eins und einem von Null aufweisen, lässt sich das Maß der Erfüllung des Kriteriums ersehen.

Bei der Auswertung der Zweigüberdeckung werden die selben Daten der Durchlaufhäufigkeit durch die Knoten einer Methode betrachtet. Knoten werden allerdings nicht isoliert betrachtet, sondern jeweils in Paaren, da nicht die Ausführung von Anweisungen interessiert, sondern der Durchlauf durch die Kanten zwischen den Anweisungen. Eine Kante ist dann durchlaufen worden, wenn ihre Start- und Ziel-Knoten wenigstens einmal ausgeführt wurden. Auf diese Weise werden alle Kanten eines Softwaremoduls betrachtet.

Für die Auswertung der Bedingungsüberdeckung ist eine gänzlich andere Herangehensweise notwendig. Zum Einen ist es erforderlich, die Knoten, die Konditionale enthalten, zu kennzeichnen, zum Anderen müssen mit diesen Knoten weitere Daten verknüpft werden.

Welche Daten dies sein müssen, hängt von den verschiedenen Überdeckungsmetriken ab. Für einfache Bedingungsüberdeckung und die mehrfache Bedingungsüberdeckung genügt es, wenn die Belegungen der einzelnen Atome zur Verfügung stehen. Die MC/DC-Überdeckung untersucht zusätzlich noch die Belegung der Gesamtbedingung. Die umfangreichsten Daten erfordert die Auswertung der minimal mehrfachen Bedingungsüberdeckung, da diese zusätzlich auch Zwischenschritte der Auswertung der Gesamtbedingung betrachtet.

Für die Ermittlung dieser verschiedenen Maße muss zuerst eine Liste aller Konditionale des zu untersuchenden Softwaremoduls erstellt werden. Für die einfache Bedingungsüberdeckung muss dann untersucht werden, ob jeweils jedes Atom einmal mit *wahr* und *falsch* belegt worden ist. Für die mehrfache Bedingungsüberdeckung muss die Gesamtzahl aller möglichen Belegungen einer Bedingung ermittelt werden (dies sind 2^n Belegungen, wobei n der Zahl der Atome entspricht). Diese Gesamtzahl wird dann in Verhältnis mit den tatsächlich aufgezeichneten Belegungen gesetzt.

Die minimale mehrfache Bedingungsüberdeckung kann ähnlich wie die einfache Bedingungsüberdeckung ausgewertet werden. Neben den Werten der Atome werden auch die Werte der einzelnen Teilbedingungen und der Wert der Gesamtbedingung überprüft. Sie müssen jeweils beide Wahrheitswerte annehmen.

Für die Ermittlung der MC/DC-Überdeckung ist das Finden von MC/DC-Paaren entscheidend. Dies sind Kombinationen von Belegungen, die sich jeweils nur in der Belegung eines einzelnen Atoms und der Belegung der Gesamtauswertung unterscheiden. Wird für jedes Atom einer Bedingung ein solches Paar gefunden, ist MC/DC-Überdeckung für diese Bedingung erfüllt.

Die Pfadüberdeckung betrachtet immer ein Modul in Gänze, nicht nur wie die anderen Maße einzelne Knoten oder Abschnitte. Betrachtet werden komplette Durchläufe durch ein Modul. Die Instrumentierung muss daher in der Lage sein, diese aufzuzeichnen. Im Rahmen dieser Arbeit werden diese Aufzeichnungen auf die Erfüllung des Boundary-Interior-Kriteriums überprüft. Dazu müssen alle möglichen Pfade durch das betreffende Modul ermittelt werden und diese mit den tatsächlichen Pfade nach dem Boundary-Interior-Kriterium verglichen werden.

6.2 Umgebung

Natürlich arbeitet das Programm, das im Rahmen dieser Diplomarbeit entstand, nicht abgeschlossen von seiner Umwelt.

Neben den eigentlichen Programmen, die untersucht werden, liefern noch zwei weitere Komponenten Eingabedaten:

- Das funktionsorientierte Testsystem ATOS
- Die Parser- und Instrumentierer-Komponente

6.2.1 ATOS

ATOS ist ein funktionsorientiertes Testsystem, das Programme im Black-Box-Test auf Fehler prüft. Kenntnis von den Programminterna hat es bei der Prüfung nicht.

Zur Durchführung von Tests bietet ATOS die Möglichkeit, Programme auf der Windows-Plattform mittels einer eigenen Skriptsprache zu automatisieren, so dass ATOS das Programm bei Testdurchführung fernsteuert. Anschließend vergleicht es bei der Testdefinition abgelegte Referenzwerte mit den tatsächlichen Ergebnissen der Programmdurchführung und stellt auf diese Weise Abweichungen vom gewünschten Verhalten fest.

Ein einzelnes Skript, das einen Programmablauf darstellt, bildet einen *Testfall*. Testfälle wiederum können zu *Testfallmengen* zusammengefasst werden. Dies ermöglicht es etwa, Tests, die ein abgeschlossenes Subsystem beschreiben, unter einem Bezeichner zusammenzufassen. Die Testfälle und Testfallmengen zusammen mit den diversen Einstellungen bilden ein ATOS-Projekt.

Für die Überdeckungsmessung im Rahmen dieser Arbeit wird die Ausführung des zu testenden Programmsystems betrachtet, insbesondere die Ausführung des Programms durch ATOS. Einzelne Programmdurchläufe und deren Log-Files sind also zumeist einem ATOS-Testfall zugeordnet.

ATOS legt seine Testfälle in einer Projekt-Datei ab [1]. Diese Datei enthält die einzelnen Testfälle und Testfallmengen.

6.2.2 Parser und Instrumentierer

Das im Rahmen dieser Arbeit erstellte System kennt zwar Grundkonstrukte imperativer und objektorientierter Programmiersprachen, nicht aber konkrete Sprachen. Quellcode von Softwaresystemen muss auf eine abstrakte Struktur abgebildet werden, die die nötigen Informationen enthält, um Kontrollflüsse, Überdeckungsmaße und Komplexitätsmaßzahlen ermitteln zu können. Mit den Details einer konkreten Sprache soll sich der Auswerter nicht befassen, nicht befassen müssen.

Diese komplexe Aufgabe, eine Schnittstelle zwischen tatsächlichem Programm und dessen Auswertung herzustellen, fällt der Parser-Komponente [Tre] zu. Diese parst den Quellcode und stellt ihn in geeigneter Form dar. Darüber hinaus leistet sie die Instrumentierung des ursprünglichen Programms, so dass der Auflauf dieses Programms, etwa im Rahmen des funktionalen Softwaretests, nachvollziehbar und auswertbar wird.

Die Vererbungsstruktur der Elemente, aus denen sich der Syntaxbaum, den der Parser aufbaut, zusammengesetzt, ist in der Abbildung 6.5 dargestellt.

Da der Parser und die Auswertung logisch voneinander getrennt entwickelt werden, ist die Definition einer Schnittstelle zwischen dem Parser und dem auswertenden Programm besonders wichtig.

Dabei sind verschiedene Situationen zu unterscheiden, in denen Parser und Auswerter miteinander kommunizieren:

[1] Eine Datei mit der Endung *.apf*, ein ATOS Project File

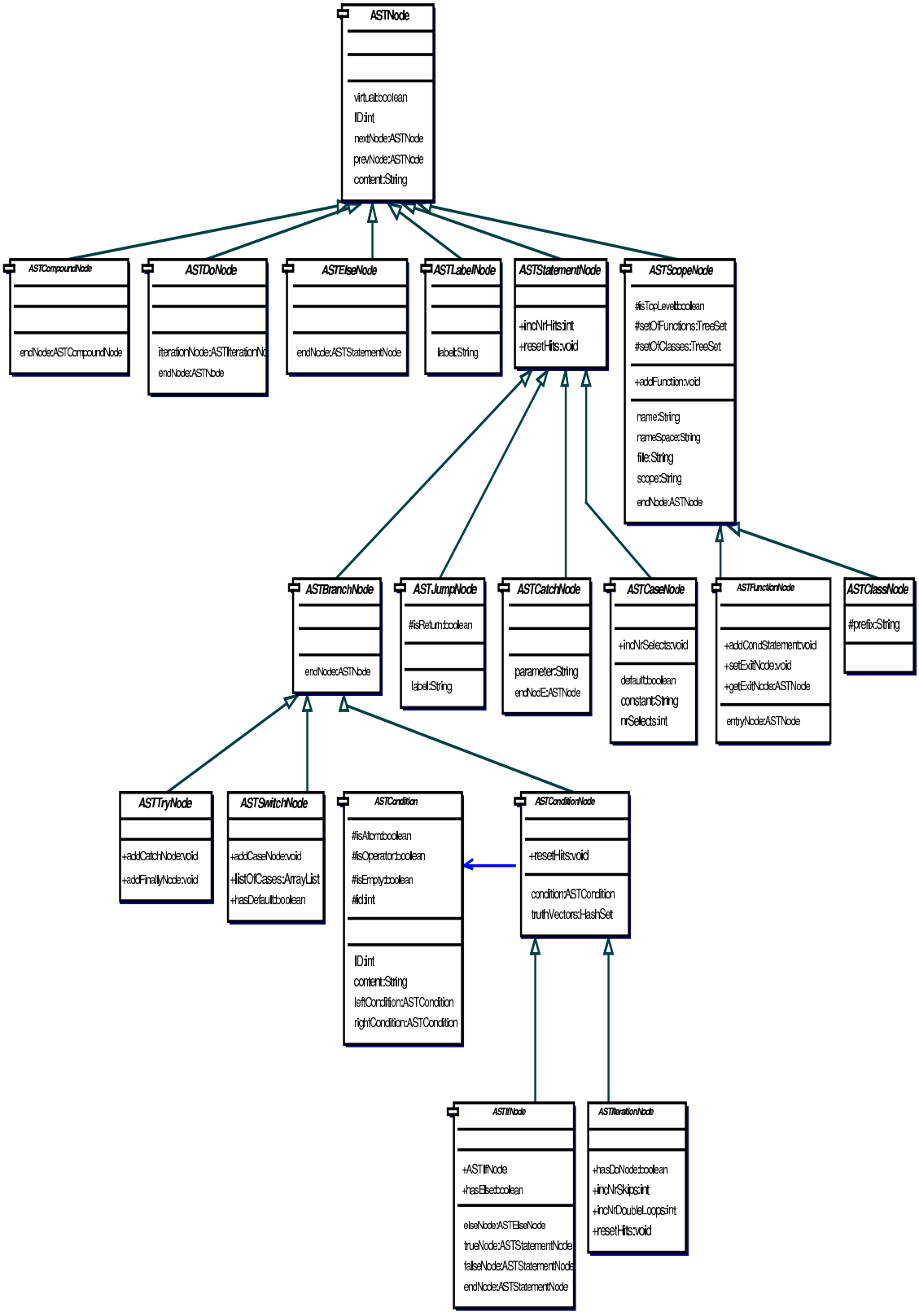


Abbildung 6.5: UML-Diagramm des abstrakten Syntaxbaums

- Initialisieren des Parsers
- Eigentliches Parsen
- Instrumentieren von Quellcode und Konfiguration der Instrumentierung
- Einlesen von Log-Dateien in die geparsten Daten
- Arbeit auf der Datenstruktur der geparsten Dateien

Initialisieren des Parsers

Zentrale Klasse, über die der Parser seine Funktionen nach außen zur Verfügung stellt, ist die Klasse `ASTManager`, über die auch die initiale Konfiguration des Parsers stattfindet. Dazu muss dem Parser eine Konfiguration übermittelt werden.

Diese Konfiguration umfasst:

- Sprache, die zu Parsen ist
- Liste der zu parsenden Dateien

Ist diese Konfiguration erstellt, ist der Parser initialisiert.

Eigentliches Parsen

Der `ASTManager` verfügt nach der Initialisierung über die notwendigen Informationen, um den festgelegten Quellcode zu parsen.

Dazu bietet er eine Methode `parseProject` an, mit der dieser Vorgang gestartet werden kann. Dar Parser startet daraufhin den eigentlichen Vorgang des Parsens, der erst nach der Abarbeitung der `parseProject`-Methode abgeschlossen ist.

Wenn der Quellcode geparst worden ist, steht er im Speicher zur weiteren Bearbeitung zur Verfügung und ist in einer abstrakten Datenstruktur (siehe Abbildung 6.5) abgebildet. Ebenfalls ist auf dieser Basis die Instrumentierung des Quellcodes möglich.

Instrumentieren von Quellcode und Konfiguration der Instrumentierung

Der Vorgang der Instrumentierung erfordert es, weitere Informationen in die Konfiguration des `ASTManager`s auszunehmen:

- Dateiendung der Quellcodedateien nach der Instrumentierung
- Endung, mit der die originalen Dateien versehen werden
- Kürzel, mit dem alle eingefügten Variablen im Quellcode versehen werden, um Namenskonflikte zu vermeiden
- Festlegung des Grades der Instrumentierung (Standard)
- Besondere Instrumentierung für einzelne Klassen und Methoden abweichend vom Standard

Einlesen von Log-Dateien in die geparsten Daten

Wenn ein instrumentiertes Programm abläuft, so legt es Aufzeichnungen über den Programmablauf in einer Log-Datei ab. Der `ASTManager` verfügt über eine Methode `readLog`, der als Parameter der Dateiname der Logdatei übergeben werden kann, um diese einzulesen.

Nach dem Einlesen stehen folgende Informationen zur Verfügung, die den Programmablauf, der der Log-Datei zugrunde liegt, wiedergeben:

- *Zahl der Ausführungen einzelner Kommandos*
  Zugriff über Methode `getNrHits` der Klassen `ASTStatementNode` und `ASTScopeNode`
- *Belegung von Konditionalen*
  Zugriff über Methode `getListOfConditions` der Klasse `ASTConditionNode`
- *Durchlaufene Pfade durch Methoden*
  Die Klasse `ASTFunctionNode` stellt eine Methode `getASTPaths` bereit

Arbeit auf der Datenstruktur der geparsten Dateien

Die Datenstruktur, in der der Parser die Struktur des geparsten Softwaresystems ablegt, ist vergleichsweise komplex, da sie den gesamten Sprachumfang von Java und C++ darstellen muss. Enthalten sind alle Daten, die nötig sind, um Kontrollflussgraphen zu erstellen, den originalen Quellcode zu gewinnen und nach Einlesen von Log-Dateien Überdeckungsmaße auszuwerten. Die Datenstruktur, die dies ermöglicht, ist in UML-Diagramm 6.5 dargestellt.

Der Parser in der für diese Arbeit verwendeten Version unterstützt als Sprachen C++ und Java. Für diese Sprachen sind Grammatiken implementiert. Zur Generierung der sprachspezifischen Komponenten greift der Parser auf den Parsergenerator *JavaCC* [SM] zurück, der Grammatiken für diverse Programmiersprachen bereitstellt. Ziel bei der Entwicklung des Parsers ist, Unterstützung für neue Sprache einfach integrierbar zu machen. Die Menge der bereits verfügbaren Grammatiken trägt ihren Teil dazu bei, solche Erweiterungen für die nahe Zukunft realistisch zu machen.

Bei der Instrumentierung werden in den ursprünglichen Quellcode zusätzliche Kommandos eingefügt. Wichtig ist dabei natürlich vor allem, dass sich an Kontrollfluss und Verhalten des Programms nichts ändert. Allenfalls hinnehmbar ist höherer Ressourcenverbrauch und längere Abarbeitungszeit. Je nach Natur des instrumentierten Programms sind solchen Verhaltensänderungen allerdings ggf. enge Grenzen gesetzt. Daher ist es möglich, verschiedene Abschnitte des Programms weniger stark zu instrumentieren (etwa Verzicht auf Bedingungsauswertung) oder auf Instrumentierung ganz zu verzichten.

6.3 JGraph - Grafische Darstellung des Kontrollflusses

Teil der Auswertung der angesammelten Daten eines Projektes ist die Darstellung des Kontrollflussgraphen in grafischer Form. Die Darstellung von Graphen ist eine sehr komplexe Materie und nicht Thema dieser Arbeit. Daher fiel die Entscheidung, für die Darstellung eine externe Bibliothek zu verwenden, die die benötigten Funktionen bereitstellt.

Voraussetzungen für die Wahl einer solchen Bibliothek sind:

- Integration in Java

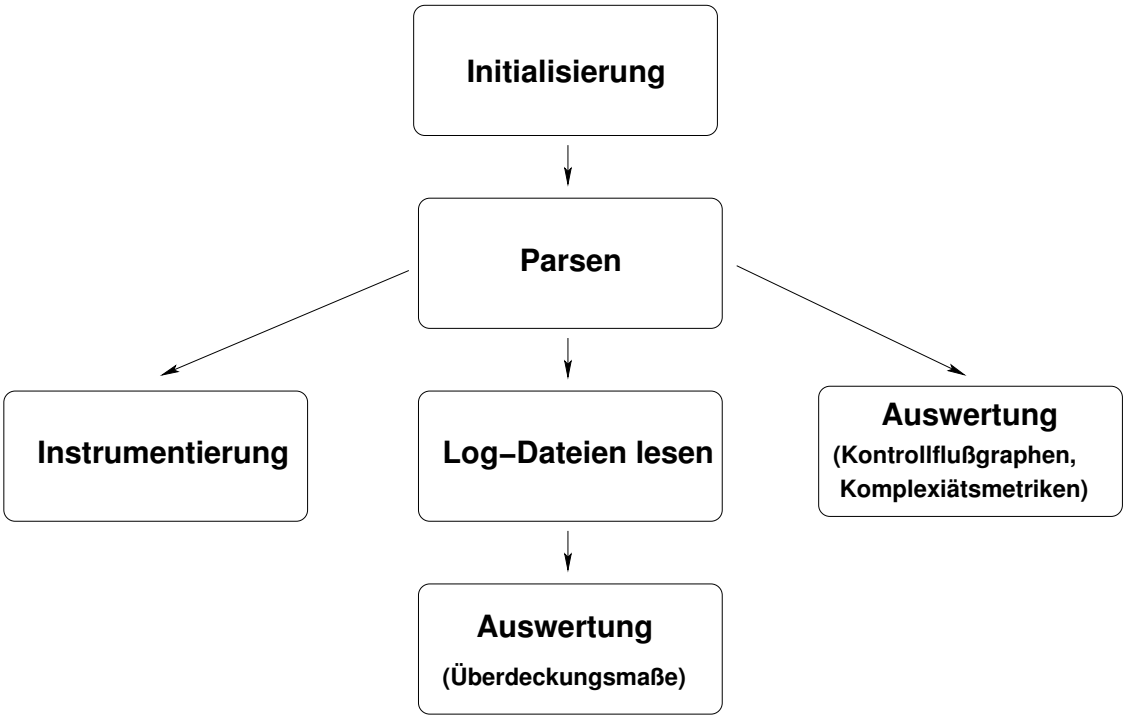


Abbildung 6.6: Ablauf der Parser-Nutzung

- Freie Verfügbarkeit
- Verfügbarkeit von Dokumentation

Unter diesen Kriterien standen folgende Bibliotheken zur Auswahl

- JGraph [jgr]
- Grappa (Java Graph Package) [Lab]

Die wesentlich bessere Dokumentation und weitere Verbreitung gaben letztendlich den Ausschlag für die Nutzung von JGraph. Die Programmierschnittstelle ist ausführlich erläutert und mit Beispielen illustriert. Sie ist für einfache Abläufe, wie sie benötigt werden, kompakt und übersichtlich und erlaubt auch komplexere Operationen, sollten diese im Rahmen späterer Erweiterungen einmal nötig sein.

6.4 Überdeckungsmaße: Zweideutigkeiten

Die Definition der MC/DC-Überdeckung stammt aus [do192], dort sind informal folgende vier Kriterien aufgestellt, die gegeben sein müssen, damit MC/DC-Überdeckung als erfüllt gelten kann:

1. Jeder Eingang und Ausgang des Moduls wird wenigstens einmal gewählt
2. Jede atomare Bedingung hat jede mögliche Belegung angenommen
3. Jede zusammengesetzte Bedingung hat jede mögliche Belegung angenommen
4. Jede atomare Bedingung hat wenigstens einmal das Ergebnis der zugehörigen zusammengesetzten Bedingung bestimmt

Diese Definition weist einige Lücken auf [VB01]:

1. **Mehrfaches Vorkommen derselben atomaren Bedingung**
   Abbildung 6.7 stellt ein solches Beispiel dar. Der Ausdruck `(E&F) | (E&G)` enthält ein zweimaliges Vorkommen der atomaren Bedingung `E`. Ein naiver Ansatz würde beide Vorkommen von `E` getrennt betrachten. In diesem Falle ist MC/DC-Überdeckung nie zu erreichen, da sich das erste Vorkommen nie unabhängig vom zweiten verändern wird.
2. **Unmöglichkeit, eine atomare Bedingung unabhängig zu verändern**
   Es kann Fälle in verschachtelten Bedingungen geben, bei denen es nicht möglich ist, eine atomare Bedingung unabhängig zu verändern. Beispiel 6.7 zeigt diesen Fall am Beispiel der Bedingung A, diese Bedingung wird nach Entscheidung d_1 immer wahr sein, so dass in d_2 keine MC/DC-Überdeckung erreichen kann, wenn man A nicht gesondert berücksichtigt.
3. **Wie sind gekoppelte Bedingungen zu behandeln?**
   Zu unterscheiden sind stark gekoppelte Bedingungen, die immer voneinander abhängen, wenn also das verändern der einen immer die andere verändert.
   Schwach gekoppelte Bedingungen verändern sich manchmal, aber nicht immer in Abhängigkeit.

Folgende Herangehensweisen sind jeweils sinnvoll:

```
1  if (A) // d1
2  {
3    if ((A&B)|(C&D)) // d2
4      sometingElse();
5    else
6      somethingCompletelyDifferent();
7  }
8  else if ((E&F)|(E&G)) // d3
9    something();
```

Abbildung 6.7: Beispiel: Abhängigkeiten von Bedingungen

1. **Mehrfaches Vorkommen derselben atomaren Bedingung**
   Natürlicher ist in diesem Fall der Ansatz beide Vorkommen von `E` als eine atomare Bedingung zu betrachten, was auch einer mathematischen Auffassung der atomaren Bedingungen als Menge näher kommt.

2. **Unmöglichkeit, eine atomare Bedingung unabhängig zu verändern**
   Aus der Analyse der syntaktischen Struktur lassen sich die Informationen, die zur Erkennung derartiger Abhängigkeiten notwendig sind, nicht herauslesen. Im allgemeinen Fall ist bei einem komplexen logischen Programmaufbau nicht möglich, solche Abhängigkeiten zu erkennen.
   Die Entwicklung einer Heuristik, die einen Großteil der praktisch auftretenden Fälle erkennt, geht über den Umfang dieser Arbeit hinaus, ist aber im Rahmen späterer Erweiterungen möglich.

3. **Wie sind gekoppelte Bedingungen zu behandeln?**
   Leider gilt das zuvor Erläuterte auch für diesen Fall. Insbesondere ist die automatisierte Behandlung von schwach gekoppelten Bedingungen, die über komplexe Funktionen miteinander verknüpft sind, nahezu unmöglich.

7 Implementierung

In den Phasen der Anforderungsdefinition und des Designs ist eine umfangreiche Spezifikation entstanden, die eine vielseitige Softwarearchitektur beschreibt. Eingebettet in diese Architektur sind eine Reihe von Systemkomponenten, für die unterschiedlich detailliert ausgearbeitete Spezifikationen entstanden sind, die je nach Komplexität von der Beschreibung eines Konzeptes bis zur Ausformulierung eines Algorithmus in Pseudocode reichen.

Die Implementierung umfasst daher eine Reihe von Vorgängen, die die Feinkonzeption der Klassenstruktur, den Feinentwurf von Algorithmen und angemessenen Datenstrukturen sowie weitere Strukturierung, Verfeinerung und Umsetzung in die Zielsprache umfassen.

Nicht zuletzt ist es möglich, dass sich einzelne Entwürfe früherer Phasen unter den Einschränkungen einer konkreten Implementierungsumgebung als nicht tragfähig erweisen. In diesen Fällen ist es erforderlich, den Designprozess zu iterieren.

7.1 Wahl der Entwicklungsumgebung

In der Entwurfsphase zeigte sich schnell die Komplexität, die mit der Umsetzung der Aufgabenstellung einher geht. Die Nutzung einer integrierten Entwicklungsumgebung ist *ein* Hilfsmittel, diese Komplexität zu beherrschen, da sie Routine-Aufgaben erleichtert oder komplett übernimmt und eine Konzentration mehr auf die technischen und inhaltlichen Aspekte der Implementierung ermöglicht.

7.1.1 JBuilder

Da im Rahmen dieser Arbeit kein bestehendes Softwaresystem erweitert wurde, war die Wahl einer Entwicklungsumgebung nicht von vorne herein festgelegt. Die Implementierungssprache, die gewählt wurde, ist zusätzlich, anders etwa als C++, soweit standardisiert, dass die Wahl einer Umgebung nicht langfristig an diese bindet.

Aufgrund der persönlichen Erfahrungen des Autors mit *JBuilder* der Firma Borland, wurde dieses System zunächst für die ersten Schritte benutzt und bewährte sich im weiteren Verlauf. Die herstellende Firma Borland stellt eine kostenfreie Version bereit, die alle benötigen Features enthält.

Zu den wesentlichen Vorteilen von JBuilder zählen der problemlose Einsatz auf verschiedenen Plattformen, wobei Windows und Linux zum Einsatz kamen, und die nahtlose Integration des Versionsverwaltungssystems *CVS*. Auf diese Weise wird Arbeit von verschiedenen Orten und in verschiedenen Umgebungen sehr erleichtert.

Als besondere Stärken von JBuilder sind zu nennen:

- in Java implementiert und daher auf vielen Plattformen verfügbar
- automatische Anzeige von Klassenmethoden
- Unterstützung von Refactoring-Vorgängen
- integrierte Versionsverwaltung
- manche Funktionen ungewöhnlich langsam
- einfache Navigation durch große Projekte

Schwächen:

- hoher Speicherbedarf
- zeitaufwendiger Start
- unübersichtliche Programmdokumentation
- aufdringliche Codevervollständigung

7.1.2 Versionsverwaltung und Quellcode-Dokumentation

CVS

Die Arbeit an einem umfangreichen System mit komplexen Abhängigkeiten erfordert die Möglichkeit, Änderungen rückverfolgen zu können und frühere Versionsstände einfach wieder herstellen zu können.

Letztendlich war bei der Wahl eines konkreten Versionsverwaltungssystems die Wahl nicht durch spezielle Alleinstellungsmerkmale eines speziellen Systems beeinflusst. Die Anforderungen, die gestellt wurden, waren:

- Zugriff auf frühere Versionen
- Kommentierung von Änderungen
- Änderungsverfolgung
- Verteilter Zugriff
- Plattformunabhängigkeit
- Festschreiben bestimmter Versionsstände

Auch in diesem Fall hat der Autor schon intensive Erfahrungen mit einer Versionsverwaltung gemacht. Es handelte sich dabei um das System *CVS*, das schließlich zum Einsatz kam.

JavaDoc

Ein guter Programmierstil umfasst nicht nur eine klare und übersichtliche Gliederung des Quellcodes und eindeutige Benennung von Bezeichnern. Darüber hinaus sind insbesondere nicht-trivialen Abschnitte und Algorithmen mit Kommentaren zu versehen.

Ein Tool, das darüber hinaus im Rahmen der Erstellung von SBTWAN zum Einsatz kam, ist *JavaDoc*, das von Sun als integraler Bestandteil von Java ausgeliefert wird. JavaDoc ermöglicht die Einbettung von Kommentaren direkt in den Quellcode auf eine Art und Weise, die es erlaubt, automatisch eine übersichtliche Dokumentation von Klassen und Methoden zu erzeugen, die diese Elemente beschreibt und Aufrufparameter erläutert.

7.2 Anmerkungen zur Implementierung

7.2.1 Allgemeine Herangehensweise

Wesentlich bestimmt war die Methodik der Implementierung von der Tatsache, dass Parser und Auswerter praktisch parallel alle Phasen von der Anforderungsdefinition

über den Entwurf bis hin zur Implementierung gleichzeitig durchlaufen haben und dabei von zwei verschiedenen Personen, von Ronny Treyße und mir, dem Autor, betreut wurden.

Damit fiel fast automatisch ein großes Augenmerk einerseits auf die Schnittstellen, die der Kommunikation dienen, und andererseits entstand der Wunsch, möglichst schnell funktionierende Module zum Test der Interoperabilität zu haben.

Dies bedeutete ein inkrementelles Herangehen an den Entwicklungsprozess. Zunächst wurde nach dem Design aus der Entwurfsphase ein funktionsfähiges Grundgerüst erstellt und diesem dann schrittweise weitere Funktionalität hinzugefügt.

Dabei konnte das ursprüngliche Design zu großen Teilen beibehalten werden, was für eine ausreichend gründliche Design-Phase spricht.

7.2.2 Skalierbarkeit der Datenstrukturen

Der grundsätzliche Ansatz des Parsers ist, das komplette Projekt geparst im Speicher in einer verketteten Liste von syntaktischen Elementen zu halten, d.h. die ständig im Speicher gehaltenen Datenmengen wachsen linear mit der Größe des bearbeiteten Softwaresystems.

Im ursprünglichen Entwurf sollten alle Änderungen an der Auswahl der zu betrachtenden Klassen, Methoden und Testfälle unmittelbar nach Auswahl angezeigt und die Erfüllung der diversen zu testenden Kriterien sofort ermittelt werden. Das bedingt natürlich umfangreiche Berechnungen, die nach jeder Operation in der GUI durchgeführt werden müssen. Während der Zeit dieser Berechnungen reagiert das Programm nicht auf Nutzerinteraktion.

Die Bearbeitung von Auswahlereignissen war folgendermaßen vorgesehen:

`void goToID(String id, int type)`	Jede Veränderung der Auswahl führt zum Aufruf dieser Funktion für alle Sichten, wobei • `id` Entweder eine ID im abstrakten Syntaxbaum oder ein Testfallname • `type` Angabe des Typs der `id`: ID im Syntaxbaum, Testfallan- oder abwahl Nach dem Ausruf von `goToID` aktualisieren sich die Sichten sofort, um der neuen Situation Rechnung zu tragen

Nach der Implementierung des ursprünglichen Designs in dem Sinne wie im vorherigen Absatz beschrieben, schienen sich daraus zunächst keine Probleme zu ergeben, da erste Tests immer mit kleinen, überschaubaren Testprogrammen durchgeführt wurden, um das prinzipielle Funktionieren der implementierten Algorithmen und Mechanismen testen zu können. Nachdem diese Phase abgeschlossen war, wurde der Test auf größere Softwaresysteme ausgeweitet, etwa auf einen Test des Systems gegen sich selbst.

Dabei stellte sich heraus, dass auch auf vergleichsweise leistungsfähiger Hardware erhebliche Verzögerungen im Arbeitsfluss auftraten, die ein flüssiges Arbeiten erschwerten. Um diesem Problem zu begegnen, war es notwendig, die Struktur des Mechanismus zur Verteilung von Aktualisierungen zu verändern. Grundsätzlich ist sofortige Aktualisierung für kleinere Softwaresysteme ohne Probleme realisierbar, erst bei größeren treten die beschriebenen Probleme auf.

Daher wird das ursprüngliche Verhalten erhalten, kann aber wahlweise abgeschaltet werden, so dass der Nutzer, nachdem er eine Auswahl der zu betrachtenden Elemente getroffen hat, eine Aktualisierung explizit auf seinen Wunsch hin auslöst.

Dies führte zu einer Trennung von Auswahlereignissen und der Aktualisierung der Ansichten im Bezug auf diese Auswahl. Dies wird folgendermaßen implementiert:

`void goToID(String id, int type)`	Jede Veränderung der Auswahl führt zum Aufruf dieser Funktion für alle Sichten, wobei • `id` Entweder eine ID im abstrakten Syntaxbaum oder ein Testfallname • `type` Angabe des Typs der `id`: ID im Syntaxbaum, Testfallan- oder abwahl Veränderungen an den Selektionen werden registriert, aber nicht unmittelbar in eine Aktualisierung der Sicht umgesetzt
`void refreshView()`	Diese Funktion, über die jede Sicht verfügt, löst eine Anpassung der Sicht an die aktuelle Auswahlsituation, wie sie über `goToID` vermittelt wurde, aus.

Auf diese Weise sind Auswahlereignisse und die Umsetzung der sich daraus ergebenden Änderungen logisch getrennt. Um das alte Verhalten zu erhalten, wird `refreshView` immer nach `goToID` ausgerufen. Ist die sofortige Aktualisierung nicht aktiv, erhält der Nutzer die Möglichkeit, diese explizit anzufordern, woraufhin `refreshView` für die jeweiligen Sichten ausgerufen wird.

Diese Veränderung im Verhalten stellt in der Regel keine Einschränkung in der Nutzbarkeit des Systems dar, da ohnehin bei einer Auswahländerung nicht nur einzelne Elemente an- oder abgewählt werden, sondern mehrere Veränderungen nacheinander vorgenommen werden. Die Ergebnisse einer sofortigen Aktualisierung werden in diesem Fall gar nicht betrachtet oder ausgewertet, sondern sind nur Zwischenschritte, auf deren Berechnung komplett verzichtet werden kann, ohne dass Einbußen in der Funktionalität entstehen.

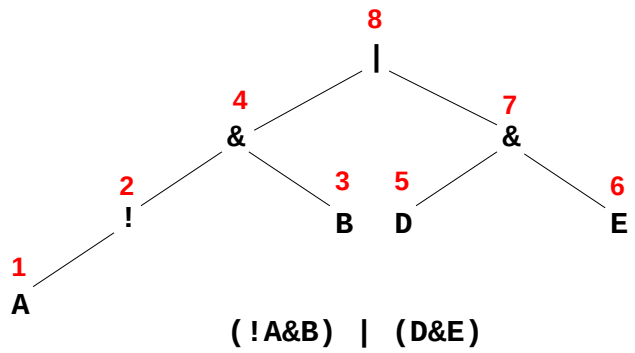


Abbildung 7.1: Aufbau der Darstellung von Bedingungen

7.2.3 Auswertung der durch den Parser bereitgestellten Wahrheitsvektoren

Die Auswertung der Maße der Bedingungsüberdeckung erfordert verschiedene Sichten auf die Struktur der Konditionale und ihrer Zusammensetzung. *Eine* Darstellung, aus der direkt alle notwendigen Informationen ablesbar wären, gibt es nicht.

Der Aufbau von zusammengesetzten Bedingungen und die Art, in der sie vom Parser eingelesen werden ergibt eine Speicherung in einem Baum. Knoten sind dabei Objekte vom Typ `ASTCondition`, die über Attribute `leftCondition` und `rightCondition` verfügen, wobei die rechte Bedingung nicht immer existiert.

Für eine einfache zusammengesetzte Bedingung ist ein solcher Baum in Abbildung 7.1 dargestellt.

Dass die Bedingungen in dieser Struktur abgespeichert werden, ergab sich erst während der Implementierung. In den Wahrheitsvektoren, die die Belegungen dieser Bäume speichern, sind die verschiedenen Belegungen in der Reihenfolge ihres Vorkommens im Baum gespeichert, wobei die Position der einzelnen Atome und zusammengesetzten Auswertungen an den durch die roten Zahlen der Abbildung bezeichneten Position eines Belegungsvektors gespeichert sind.

Für die einfache Bedingungsüberdeckung sind nur die Werte der Atome von Interesse. In der gewählten Implementierung werden nun mit einer modifizierten Tiefensuche die Positionen der Atome, die den Blättern des Baumes entsprechen, bestimmt und extrahiert.

Für die mehrfache und die MC/DC-Überdeckungsmaße ist zusätzlich noch das Ergebnis der Gesamtauswertung des Ausdrucks in Interesse. Dieses kann immer aus

der letzten Stelle des Wahrheitsvektors gewonnen werden, die immer der Wurzel des Bedingungsbaumes entspricht. Für die minimal mehrfache Bedingungsüberdeckung, die auch Zwischenergebnisse der Auswertung in ihre Betrachtung einbezieht, ist der gesamte Wahrheitsvektor von Interesse, d.h. auch die Positionen, die die Auswertung von Zwischenergebnissen im Baum repräsentieren.

7.2.4 Implementierung der Boundary-Interior-Auswertung

Die Problematiken, die sich aus der Auswertung des Boundary-Interior-Kriteriums ergeben, sind erst relativ spät ins Blickfeld gerückt. Dieses Kriterium ist als einziges Überdeckungskriterium aus der Familie der Pfadtests in den Anforderungen und im Design vertreten.

Die anderen Überdeckungsmaße sind insofern verschiedenen, als dass sie immer nur lokal einzelne Anweisungen oder einzelne Bedingungen betrachten. Boundary Interior-Test betrachtet komplette Modul-Durchläufe bzw. ihre Repräsentation als Pfad in einem Kontrollflussgraphen.

Die Kombination aus Parser, Instrumentierer und Auswerter stellt die folgenden Komponenten bzw. Daten bereit, die bei der Pfad-Analyse herangezogen werden können:

- Aufzeichnung der tatsächlich durch ein Modul gewählten Pfade
- In der geparsten Daten-Struktur erfolgt eine Markierung von für Pfaden wichtigen Elementen
- Kontrollflussgraphen

Pfade durch ein Modul können, insbesondere bei Schleifen, sehr lang werden. Daher beschränkt sich der Instrumentierer darauf, nur Knoten, die Verzweigungen oder Vereinigungen des Kontrollflusses abbilden, in die Pfade aufzunehmen. Entsprechend können die Pfade verkürzt dargestellt werden. Elemente, die in Pfaden aufgezeichnet werden können, werden auch in der Datenstruktur, die den geparsten Quellcode repräsentiert, entsprechend markiert.

Bei der Erstellung von Kontrollflussgraphen werden diese Markierungen als Attribute für die Knoten der Graphen übernommen. Die Elemente, die so markiert sind, tragen als Identifizierung eine ID, die für jeden Knoten eindeutig ist. Die Pfadaufzeichnungen setzen sich aus Mengen von ID-Listen zusammen, die über die einzelnen `ASTFunctionNode`s abgefragt werden können.

Diese tatsächlich durchlaufenen Pfade müssen nun einer Menge von Pfaden gegenübergestellt werden, die das Boundary Interior-Kriterium erfüllen. Es ist die Aufgabe von SBTWAN, auf Basis der Kontrollflussgraphen und der Markierungen pfadrelevanter Knoten, diese Pfade zu ermitteln.

Grundsätzlich wird dazu ein Tiefensuche-Algorithmus herangezogen, der allerdings modifiziert werden musste, da klassische Tiefensuche bereits durchlaufene Graphen-Abschnitte nicht nochmal durchsucht, was aber in diesem Fall gewünscht ist. Stattdessen wird, um das Terminieren der Suche sicherzustellen, die Eigenschaft von Kontrollflussgraphen ausgenutzt, einen ausgezeichneten Endknoten zu haben, der von jedem Knoten aus erreicht werden kann.

Die Tiefensuche in der modifizierten Form verzichtet also auf die Markierung bereits besuchter Knoten. Für Graphen ohne Schleifen werden damit bereits alle möglichen Pfade durchlaufen. Da diese mit den tatsächlichen Pfaden verglichen werden, werden sie aufgezeichnet als Folge von Knoten-IDs. Auf diese Weise ist es auch möglich, Schleifen zu erkennen, indem der aktuell in der Tiefensuche betrachtete Knoten mit der Liste der bereits durchsuchten Knoten verglichen wird.

Der Boundary-Interior-Test betrachtet insbesondere diese Pfade durch Schleifen, um den Durchlauf des Schleifeninneren auf besondere Weise zu untersuchen. Daher werden die Pfade, die Schleifendurchläufe enthalten, besonders behandelt, indem der Durchlauf durch das Schleifeninnere aus dem Pfad extrahiert und gesondert vermerkt wird. Im Falle verschachtelter Schleifen werden schrittweise innere Schleifen entfernt, vermerkt und in der umfassende Schleife die entsprechende Position markiert.

Bei der Auswertung werden diese Aufzeichnungen der gesuchten Pfade mit und ohne Schleifen mit den tatsächlich aufgetretenen Pfaden verglichen, indem Pfade ohne Schleifendurchläufe und Schleifenfragmente entsprechend der gesuchten Pfade kombiniert werden.

Da manche Schleifen sehr oft durchlaufen werden und somit die aufgezeichneten Pfade sehr lang werden, werden *reguläre Ausdrücke* eingesetzt, die umfangreiche Möglichkeiten anbieten, um in Zeichenketten nach Mustern zu suchen. Damit können auch Wiederholungen aufgefunden werden. Genutzt wird die Implementierung regulärer Ausdrücke, die in Java 1.4 in der Methode `String.matches(String regexp)` enthalten ist.

7.2.5 Hintergrundaktivitäten und GUI-Aktualisierung

SBTWAN nutzt für die Bereitstellung seiner grafischen Oberfläche das Java-Toolkit *Swing*, das mächtige Hilfsmittel für die Erstellung und Verwaltung von GUI-Komponenten bereitstellt. Swing nimmt alle Operationen aus einem einzelnen Thread vor. Für Applikationen, die ausgelöst durch GUI-Ereignisse umfangreiche Berechnungen durchführen, bedeutet dies, dass während dieser Operationen eine Anzeige von Fortschrittsanzeigern nicht möglich ist, da der einzelne GUI-Thread durch die Berechnung selbst blockiert ist.

Dieses Problem tritt in SBTWAN während des Ladens und Parsens eines Projektes auf, da diese Operation vergleichsweise langwierig ist. Aus Gründen der Nachvollziehbarkeit des Programmablaufes ist es aber wünschenswert, dass während dieses Vorgangs ein Fortschrittsanzeiger angezeigt wird.

Die Lösung, die in diesem Falle gewählt werden muss, ist, für die vorgesehene langwierige Operation einen eigenen zweiten Thread zu starten, der sich mit dem GUI-Thread synchronisiert. Dazu bietet Java ein Hilfsmittel an, dass es erlaubt, Operationen im Kontext des GUI-Threads auszuführen. Die Klasse `SwingUtilities` bietet die Methode `invokeLater` an, der als Parameter eine GUI-Operation übergeben werden kann, die den Fortschrittsanzeiger aktualisiert.

Auf diese Weise ist es möglich, gleichzeitig eine umfangreiche Berechnung durchzuführen und die GUI zu aktualisieren. Eine Voraussetzung dazu ist allerdings, dass diese Berechnung in einem eigenen Thread implementiert wird.

8 Test des Systems

Die einführenden Kapitel dieser Arbeit enthalten einen Überblick über die Grundlagen der systematischen Softwareentwicklung und diese Arbeit beschreibt für das System SBTWAN, wie die Phasen der Anforderungsdefinition, des Designs und der Implementierung durchgeführt wurden. Integraler Bestandteil von Prozessen des Software-Engineering ist nach der erfolgten Implementierung eine Testphase, in der die Eigenschaften der Software überprüft und auf Übereinstimmung mit der Spezifikation geprüft werden. Was im Allgemeinen gilt, gilt natürlich auch für das Softwaresystem SBTWAN, das im Rahmen dieser Arbeit erstellt wurde.

8.1 Erstellung von Kontrollflussgraphen

Die Erstellung von Kontrollflussgraphen auf Basis der Datenstruktur, wie sie der Parser bereitstellt, ist in der Theorie keine schwierige Aufgabe.

In der Praxis sind aber eine Fülle von Details und Möglichkeiten zu berücksichtigen, die in realen Programmen entstehen und die Art beeinflussen, in der Kontrollstrukturen aufgebaut sein können. Dadurch wird die Aufgabe der Erstellung eines entsprechenden Algorithmus vor allem zeitaufwendig.

Im ersten Drittel der Implementierungsphase wurde viel Energie in den Entwurf und die Erstellung eines Konstruktionsalgorithmus verwandt, der den gestellten Anforderungen gerecht wird:

- Korrekte Erstellung von Kontrollflussgraphen
- Überschaubare, wartbare Strukturierung des Algorithmus
- Flexible Verwendbarkeit für verschiedene Anwendungen (interne Graph-Darstellung, Visualisierung am Bildschirm)
- Robustheit in allen Situationen, die in syntaktisch korrekten Programmen entstehen können

- Einfache Erweiterbarkeit sowohl im Bezug auf neue Sprachkonstrukte als auch auf neue Sprachen

Der nach diesen Prinzipien entstandene Algorithmus wurde intensiv getestet. Die wesentlichen Testfälle und Herangehensweisen werden in diesem Abschnitt dargestellt.

Betrachtet werden vier verschiedene, wesentliche Konstrukte der imperativen Programmierung (am Beispiel der Sprache *Java*) und mögliche Variationen. Dazu wird der entsprechende Kontrollflussgraph herangezogen.

`if-else`-Verzweigungen

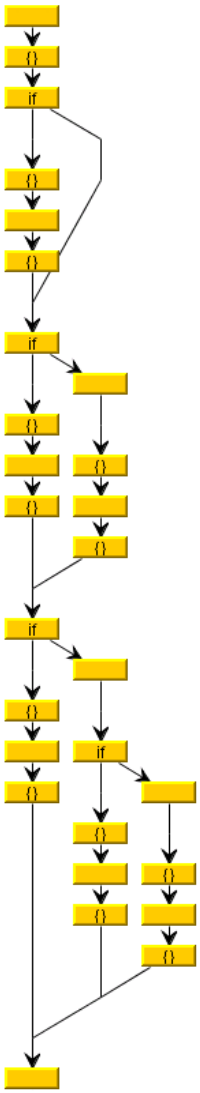
In der Abbildung 8.1 werden `if`-Verzweigungen dargestellt. Dargestellt ist rechts der zugrundeliegende Quellcode und links der von SBTWAN erstellte Kontrollflussgraph. Berücksichtigt werden folgende Situationen:

- Verzweigung im leerem `else`-Zweig
- `if-else` ohne Verschachtelung
- `if-else if-else`, also eine in den `else`-Zweig eingebettete weitere Verzweigung

`switch`-Verzweigungen

In Abbildung 8.2 wird das Verhalten bei der Abbildung von `switch`-Verzweigungen dargestellt. Wieder werden drei Fälle nach dem gleichen Schema betrachtet:

- `switch` mit `default`-Fall und Sprung nach jedem Fall
- `switch` ohne `default`-Fall
- `switch` ohne Sprung nach dem ersten und default-Fall

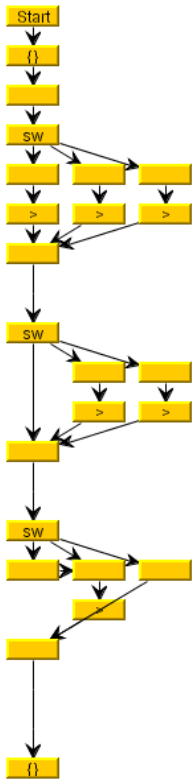



```
boolean a,b;
if (a)
{
   do();
}

if (a)
{
   do();
}
else
{
   doElse();
}

if (a)
{
   do();
}
else if (b)
{
   doElseIf();
}
else
{
   doElse();
}
```

Abbildung 8.1: Kontrollflussgraph für `if-else`-Konstrukte




```
int i;

switch (i)
{
   case 1:
      do();
      break;
   case 2:
      do();
      break;
   default:
      do();
      break;
}

switch (i)
{
   case 1:
      do();
      break;
   case 2:
      do();
      break;
}

switch (i)
{
   case 1:
      do();
   case 2:
      do();
      break;
   default:
      do();
}
```

Abbildung 8.2: Kontrollflussgraph für `switch-case`-Konstrukte

`while`-Schleifen

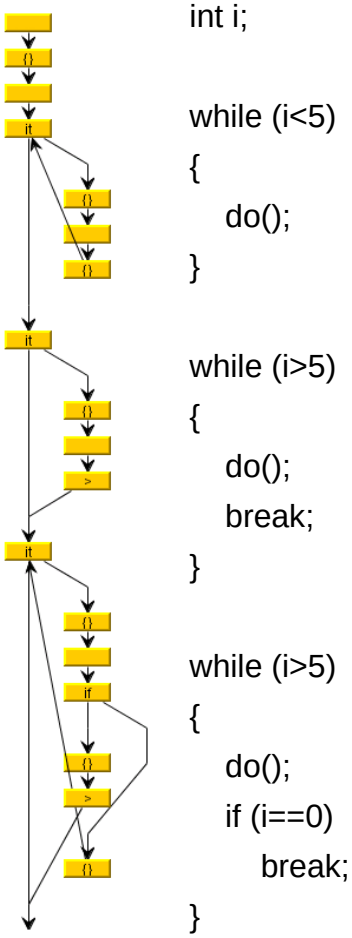
In Abbildung 8.3 sind verschiedenen Varianten einer `while`-Schleife und deren jeweilige Darstellung als Kontrollfluss dargestellt. Es wurden folgende Varianten berücksichtigt:

- `while`-Schleife ohne weitere Besonderheiten

- `while`-Schleife mit Sprung aus der Schleife
- `while` mit bedingtem Sprung aus der Schleife

`do-while`-Schleifen

Abbildung 8.4 zeigt den entsprechenden Kontrollflussgraphen für eine `do-while`-Schleife.



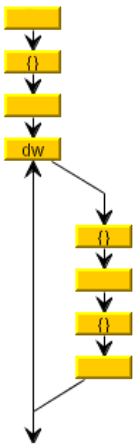

```
int i;

while (i<5)
{
   do();
}

while (i>5)
{
   do();
   break;
}

while (i>5)
{
   do();
   if (i==0)
      break;
}
```

Abbildung 8.3: Kontrollflussgraph für `while`-Konstrukte




```
int i=0;
do
{
   i++;
}
while (i<5);
```

Abbildung 8.4: Kontrollflussgraph für `do-while`-Konstrukte

8.2 Komplexitätsmaßzahlen

SBTWAN ermittelt auf Basis des Quellcodes und der daraus erstellten Kontrollflussgraphen die Komplexitätsmaßzahlen der zyklomatischen und der essentiellen Komplexität, wie sie von McCabe definiert worden sind.

Beide Maße basieren auf der Betrachtung des Kontrollflussgraphen und werten diesen mit unterschiedlicher Methodik aus. Die zyklomatische Komplexität zählt lediglich die Knoten und Kanten des zu betrachtenden Graphen und ist daher sehr einfach zu implementieren, wenn man die korrekte Konstruktion von Kontrollflussgraphen des vorherigen Abschnitts voraussetzt.

Die essentielle Komplexität ist deutlich komplexer zu ermitteln, da dazu aus dem Kontrollflussgraphen iterativ bestimmte Subgraphen zu entfernen sind. Wichtig ist hierbei, die korrekte Reduzierung des Graphen zu überprüfen.

Zur Veranschaulichung eines Beispiels, das nur Konstrukte der strukturierten Programmierung enthält, wird ein Beispiel-Graph mit einer `if-else`-Konstruktion und einer `while`-Schliefe gewählt, der in Abbildung 8.5 dargestellt ist. SBTWAN berechnet die zyklomatische Komplexität korrekt mit 4 und die essentielle mit 1.

Beispiel 8.6 ist strukturell ähnlich aufgebaut, nur, dass hier aus der `while`-Schleife bedingt mit einem `break` nach außen gesprungen wird. Wegen dieses Sprungs sollte eine Reduktion für die `while`-Schleife und das darin eingebettete `if` nicht möglich sein. Erwartungsgemäß wird die zyklomatische Komplexität mit 5 und die essentielle mit 3 ermittelt.

Beispiel	zyklomatische K.	essentielle K.
8.5	4	1
8.6	5	3

8.3 Überdeckungsauswertung

In der Literatur werden eine Reihe von Beispielen besprochen, die zur Illustration von kontrollflussbasierten Verfahren und Überdeckungsmaßen herangezogen werden können. Pagel und Six schlagen in [Pag94] einen Algorithmus zum Parsen von Zahlen vor, für den im weiteren entsprechende Testfälle ausgewählt werden, die die hier vorgestellten Überdeckungsmaße erfüllen. Für die bei Pagel und Six nicht behandelten Überdeckungsmaße wurden entsprechende Testfälle entwickelt.

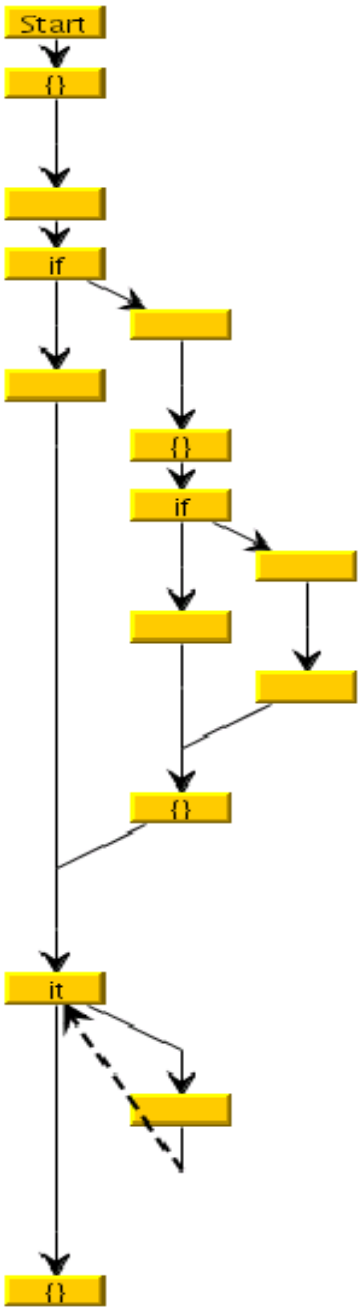


Abbildung 8.5: Beispiel-Graph: zyklomatische Komplexität

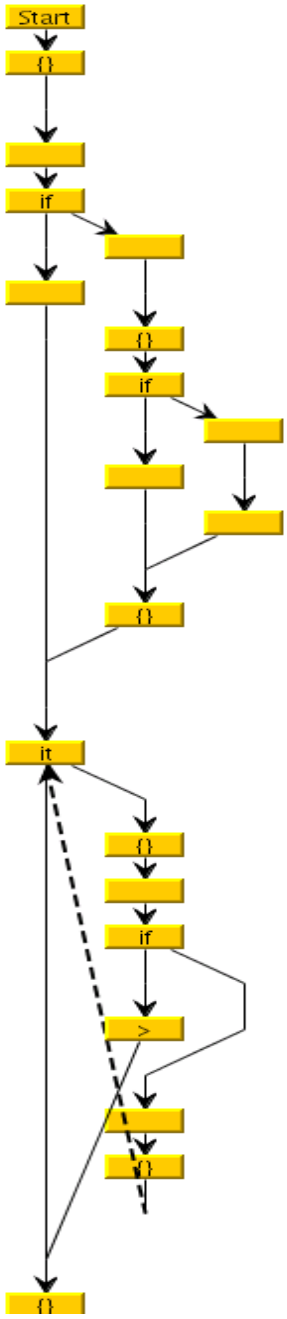


Abbildung 8.6: Beispiel-Graph: essentielle Komplexität

Auf dieser Basis kann das entstandene Programm auf seine korrekte Funktion hin überprüft werden. Im Original liegt der Algorithmus in PASCAL-Notation vor, zum Test des Systems wurde er nach Java übertragen. Diese Fassung ist in Abbildung 8.7 dargestellt.

Der Kontrollflussgraph, auf dem die weiteren Untersuchungen basieren, wird vom entwickelten System Abbildung 8.8 entsprechend dargestellt und entspricht, abgesehen von der anderen Kanten- und Knotenanordnung, der Referenz von Pagel und Six.

8.3.1 Anweisungsüberdeckung

Betrachtet man den Algorithmus, fällt schnell auf, dass man zur Anweisungsüberdeckung unter anderem einen Testfall benötigt, der den Fehlerfall aufruft. Die dann verbliebenen Anweisungen lassen sich durch eine Zahl, die ein Komma enthält, aufrufen.

Pagel und Six schlagen für die Erfüllung von Anweisungsüberdeckung Testfälle vor, die die Eingabedaten `..` und `.9` für den Algorithmus beinhalten.

Wie in Abbildung 8.10 sichtbar, führt die Auswertung dieser Testfälle tatsächlich zur Erfüllung von Anweisungsüberdeckung.

Welchen Prozentsatz der Anweisungsüberdeckung die beiden Testfälle und deren Kombination leistet, ist aus Abbildung 8.9 ersichtlich. Jeder Teilfall für sich überdeckt schon über drei Viertel der Anweisungen, was sich daraus erklärt, dass beiden Testfällen natürlich viele Anweisungen gemeinsam sind.

```
1  public double werteZiffernfolgeAus (String inZiffernString) {
2      double wert = 0.0;
3      double genauigkeit = 1.0;
4      String woBinIch = "VorDemKomma";
5      boolean fehlerfrei = true;
6      int position = 1;
7      while (position <= inZiffernString.length() & fehlerfrei) {
8          String zchn = inZiffernString.substring(position-1, position);
9          if (zchn.matches("[0-9]")) {
10             if (woBinIch.equals("NachDemKomma"))
11                 genauigkeit = genauigkeit / 10.0;
12             wert = 10.0 * wert + Double.parseDouble(zchn);
13         }
14         else if (zchn.equals(".") & woBinIch.equals("VorDemKomma"))
15             woBinIch = "NachDemKomma";
16         else
17             fehlerfrei=false;
18         position++;
19     }
20     if (!fehlerfrei | inZiffernString.length()==0 |
21           ((woBinIch.equals("NachDemKomma") &
22             inZiffernString.length()==1)))
23         return -1.0;
24     else
25         return wert * genauigkeit;
26 }
```

Abbildung 8.7: Algorithmus zur Ermittlung des Zahlenwerts aus einer Zeichenfolge

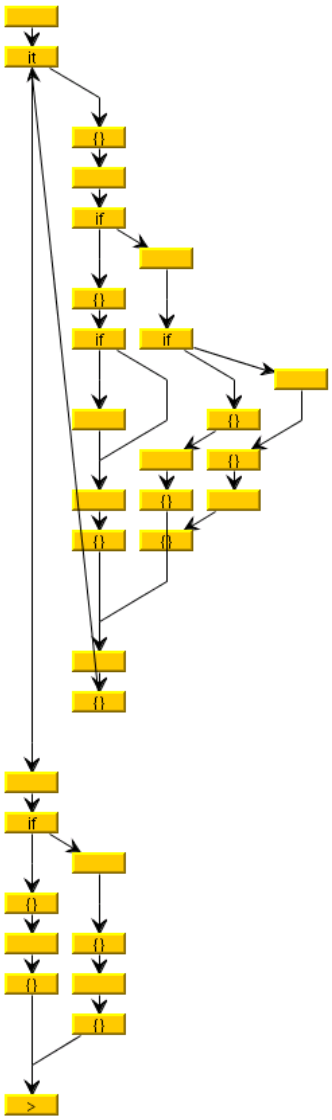


Abbildung 8.8: Kontrollflussgraph des Algorithmus

Eingabe	Prozentsatz der Erfüllung
`..`	75%
`.9`	81%
`..` und `.9`	100%

Abbildung 8.9: Testfälle für Anweisungsüberdeckung

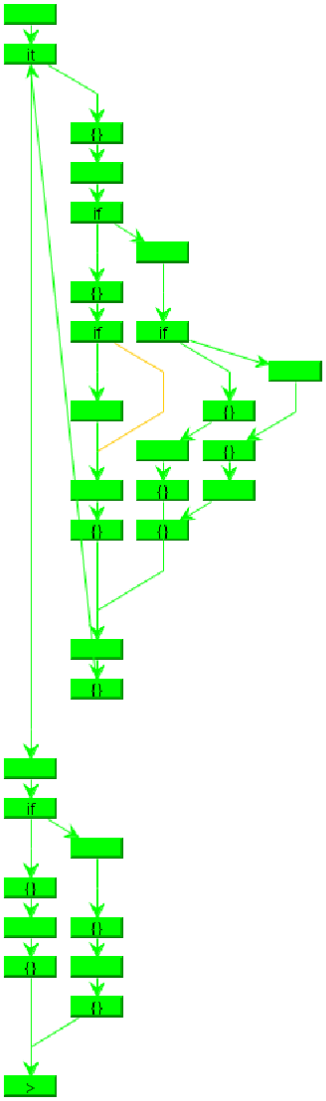


Abbildung 8.10: Anweisungsüberdeckung

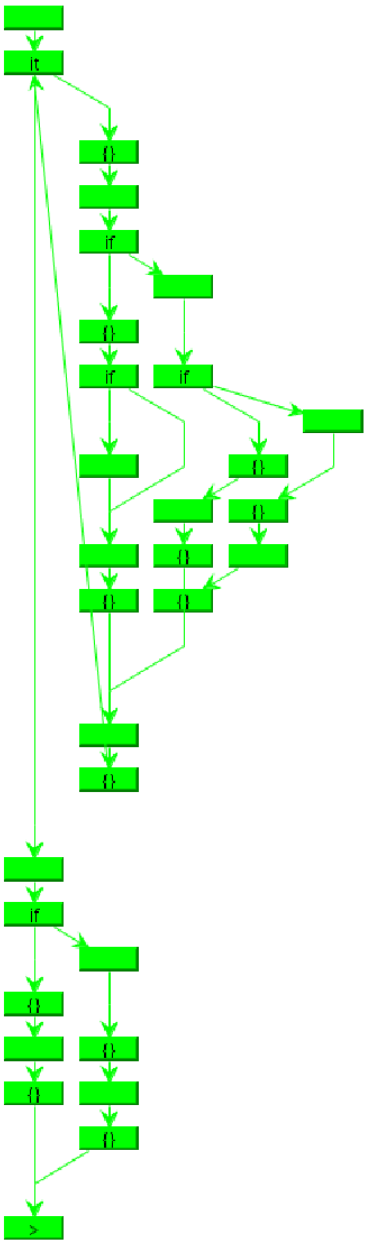


Abbildung 8.11: Zweigüberdeckung

Eingabe	Prozentsatz der Erfüllung
`..`	68%
`.9`	75%
`..` und `.9`	95%
`9`	58%
`..`, `.9` und `9`	100%

Abbildung 8.12: Testfälle für Zweigüberdeckung

8.3.2 Zweigüberdeckung

Betrachtet man die Abbildung 8.10 der Anweisungsüberdeckung, so fällt ins Auge, dass zur Zweigüberdeckung lediglich eine Kante fehlt (diese Kante ist orange dargestellt). Diese Kante stellt den fehlenden `else`-Zweig der Anweisung `if (woBinIch.equals("NachDemKomma"))` dar.

Da dieser fehlende `else`-Zweig keine Anweisung enthielt, war es nicht nötig, ihn für die Erfüllung der Anweisungsüberdeckung mit einem Testfall aufzurufen. Zur Erfüllung der Zweigüberdeckung muss ein Testfall gefunden werden, der diese Kante umfasst. Ein solcher Testfall ist `9`.

Die drei Testfälle `..`, `.9` und `9` erfüllen somit das Zweigüberdeckungskriterium, was korrekt erkannt wird, wie man aus Abbildung 8.11 ersehen kann.

Auch hier (vgl. Abbildung 8.12) fällt das schon bei der Anweisungsüberdeckung beobachtete Verhalten auf, dass einzelne Testfälle prozentual schon einen großen Anteil der Kriteriumserfüllung bedeuten, aber die vollständige Erfüllung noch vergleichsweise viele weitere Testfälle erfordert.

8.3.3 Minimal mehrfache Bedingungsüberdeckung

Um minimal mehrfache Bedingungsüberdeckung erfüllen zu können, sind mehr Testfälle als bei der Zweigüberdeckung nötig, da diese nur bedeutet, dass jede zusammengesetzte Bedingung einmal *wahr* und einmal *falsch* ausgewertet wurde. Die minimal mehrfache Bedingungsüberdeckung fordert zusätzlich, dass auch jede atomare Bedingung und jedes Zwischenergebnis einmal mit *wahr* und einmal mit *falsch* belegt wurden. Auf diese Weise wird der Komplexität zusammengesetzter Bedingungen besser Rechnung getragen.

Eingabe	Prozentsatz der Erfüllung
`..`	66%
`.9`	59%
`9`	57%
`..`, `.9` und `9`	90%
`.`	56%
`z`	60%
leer	50%
`.`, `..`, `.9`, `9`, `z` und *leer*	100%

Abbildung 8.13: Testfälle für minimal mehrfache Bedingungsüberdeckung

position <= inZiffernString.length()	fehlerfrei	Gesamt
0	1	0
1	1	1
0	0	0

Abbildung 8.14: `position<=inZiffernString.length() & fehlerfrei`

Die Testfälle, die für die Zweigüberdeckung genutzt wurden, dienen als Basis der Testfälle für die minimal mehrfache Bedingungsüberdeckung. In Abbildung 8.13 sind diese drei Testfälle dargestellt. Bereits mit diesen drei Testfällen sind die ersten drei Bedingungen der Funktion wie in den Abbildungen 8.14, 8.15 und 8.16 belegt und erfüllen jeweils das Überdeckungskriterium.

Für die beiden letzten Bedingungen ist die minimal mehrfache Überdeckung mit diesen Eingabedaten allerdings noch nicht erfüllt. In der vierten Bedingung ist das Atom `zchn.equals(".")` nie mit *falsch* belegt. Nach der Struktur lässt sich diese Belegung erfüllen, indem eine Eingabe gewählt wird, die keine Ziffer ist und nicht mit einem Komma beginnt. Gewählt wird in diesem Beispiel das Zeichen `z`.

Um die letzte, komplexeste Bedingung zu überdecken, ist zunächst einmal eine leere Eingabe notwendig, um das Atom `inZiffernString.length()==0` einmal mit *wahr* zu belegen. Damit das Prädikat `woBinIch.equals("NachDemKomma") & inZiffernString.length()==1` wahr werden kann, ist ein `.` als Eingabe notwendig.

In diesem Fall treten die in den Abbildungen 8.17 und 8.18 dargestellten Belegungen auf. Die minimale mehrfache Bedingungsüberdeckung ist damit erfüllt und diese Situation wird von SBTWAN erkannt.

zchn.matches("\\d")	Gesamt
0	0
1	1

Abbildung 8.15: `zchn.matches("[0-9]")`

woBinIch.equals("NachDemKomma")	Gesamt
1	1
0	0

Abbildung 8.16: `woBinIch.equals("NachDemKomma")`

zchn.equals(".")	woBinIch.equals("VorDemKomma")	Gesamt
1	1	1
1	0	0
0	1	0

Abbildung 8.17: `zchn.equals(".") & woBinIch.equals("VorDemKomma")`

fehlerfrei	inZiffernString.length()==0	woBinIch.equals("NachDemKom...	inZiffernString.length()==1	Gesamt
1	0	1	0	0
0	0	1	0	1
1	1	0	0	1
0	0	0	1	1
1	0	1	1	1
1	0	0	1	0

Abbildung 8.18:
```
!fehlerfrei | inZiffern.length()==0 |
        ((wo.equals("Nach...") & inZiffern.length()==1))
```

8.3.4 MC/DC-Überdeckung

Ausgehend von den Testfällen, die für die minimal mehrfache Bedingungsüberdeckung zusammengestellt worden sind, können entsprechende Testfälle ergänzt werden, um MC/DC-Überdeckung zu erfüllen.

MC/DC-Überdeckung fordert, dass jedes Atom wenigstens einmal das Ergebnis der zusammengesetzten Bedingung bestimmt. Dazu sucht man MC/DC-Paare, also Paare von Belegungen, in denen jeweils nur das Atom und die Bewertung des Gesamtausdrucks verschieden sind, während die anderen Variablen fixiert bleiben. Das Auffinden eines solchen Paares für jedes Atom bedeutet die Erfüllung des Überdeckungskriteriums.

Betrachtet man die Testfälle der minimal mehrfachen Bedingungsüberdeckung, so stellt man fest, dass lediglich für die Ausdrücke

- `position <= inZiffernString.length() & fehlerfrei`
- `!fehlerfrei|inZiffernString.length()==0| ((woBinIch.equals("NachDemKomma") & inZiffernString.length()==1))`

das MC/DC-Kriterium nicht erfüllen.

Für den ersten Ausdruck ist das zusätzliche hinzufügen eines Testfalls mit der Eingabe ... notwendig. Für den zweiten Ausdruck lässt sich MC/DC-Überdeckung mit einem Testfall `99` herbeiführen.

Diese theoretischen Überlegungen entsprechen dem, was SBTWAN tatsächlich erkennt.

position <= inZiffernString.length()	fehlerfrei	Gesamtausdruck
falsch	wahr	**falsch**
wahr	wahr	**wahr**
wahr	**wahr**	**wahr**
wahr	**falsch**	**falsch**

!fehlerfrei	inZiffernString.length()==0	woBinIch.equals(”NachDemKomma”	inZiffernString.length()==1	Gesamtausdruck
wahr	falsch	falsch	wahr	**falsch**
falsch	falsch	falsch	wahr	**wahr**
wahr	**falsch**	falsch	falsch	**falsch**
wahr	**wahr**	falsch	falsch	**wahr**
wahr	falsch	**falsch**	wahr	**falsch**
wahr	falsch	**wahr**	wahr	**wahr**
wahr	falsch	wahr	**falsch**	**falsch**
wahr	falsch	wahr	**wahr**	**wahr**

8.3.5 Mehrfache Bedingungsüberdeckung

Da die Anzahl der Testfälle in der Anzahl der Atome in zusammengesetzten Bedingungen steigt, werden sehr viele Testfälle benötigt. Aus diesem Grund ist die mehrfache Bedingungsüberdeckung ein Kriterium, das sich im Allgemeinen nicht sinnvoll erfüllen lässt. Die mehrfache Bedingungsüberdeckung fordert, dass in zusammengesetzten Bedingungen *jede* mögliche Kombination von Belegungen der Atome auftritt. Dadurch tritt für dieses Beispiel der Fall ein, der eine automatische Auswertung schwierig macht, da viele der geforderten Kombinationen nicht durch Eingabedaten erreichbar sind. Automatisch lässt sich diese Situation nicht erkennen, vielmehr müssen einzelne Bedingungen, die die Überdeckung nicht erfüllen, manuell untersucht werden.

Die Testfälle, die zur Erfüllung der MC/DC-Überdeckung herangezogen wurden, erfüllen die mehrfache Bedingungsüberdeckung zu 64%. Betrachtet man in der Detailansicht von SBTWAN dieses Ergebnis genauer, sieht man rasch, dass es die letzten beiden Bedingungen sind, die nach der minimal mehrfachen Bedingungsüberdeckung nicht überdeckt sind. Die vorhergehenden Bedingungen bestehen maximal aus zwei Atomen, so dass sich die Zahl der notwendigen Testfälle in Grenzen hält.

Die vorletzte Bedingung

- `zchn.equals(".") & woBinIch.equals("VorDemKomma")`

ist noch vergleichsweise überschaubar. Vier Belegungen werden gefordert, nur eine fehlt.Ergänzt man einen Testfall, der nach dem Komma eine Nicht-Ziffer enthält, kann man die fehlende, vierte Belegung erzeugen. Eine mögliche Eingabe ist etwa `.z`.

Dieser Testfall ändert allerdings nichts daran, dass für die letzte Bedingung nur sieben von sechzehn geforderten Belegungen auftreten. Diese sind in der nachfolgenden Tabelle durch Fettschrift hervorgehoben. Die kursiv hervorgehobene Kombination ist durch den Testfall `9z` erreichbar.

Alle anderen Kombinationen sind durch Eingabedaten nicht erzeugbar, weil sich die Belegungen logisch widersprechen würden. Viele Kombinationen fallen bereits heraus, da sie fordern, dass `inZiffernString.length()==0` und `inZiffernString.length()==1` gleichzeitig wahr sein sollen, die anderen aufgrund komplexerer Widersprüche.

!fehlerfrei	inZiffernString.length()==0	woBinIch.equals(”NachDemKomma”	inZiffernString.length()==1
falsch	*falsch*	*falsch*	*falsch*
falsch	**falsch**	**falsch**	**wahr**
falsch	**falsch**	**wahr**	**falsch**
falsch	falsch	wahr	wahr
falsch	wahr	falsch	falsch
falsch	wahr	falsch	wahr
falsch	wahr	wahr	falsch
falsch	wahr	wahr	wahr
wahr	**falsch**	**falsch**	**falsch**
wahr	**falsch**	**falsch**	**wahr**
wahr	**falsch**	**wahr**	**falsch**
wahr	**falsch**	**wahr**	**wahr**
wahr	**wahr**	**falsch**	**falsch**
wahr	wahr	falsch	wahr
wahr	wahr	wahr	falsch
wahr	wahr	wahr	wahr

Alles in allem sind für die Erfüllung der mehrfachen Bedingungsüberdeckung insgesamt 28 verschiedene Kombinationen aus fünf Bedingungen zu erfüllen. Erfüllbar

sind aber nur 20, die verbleibenden 8 der letzten Bedingung sind nicht erreichbar. Dies entspricht einem maximal erreichbaren Überdeckungsgrad von 71%, was genau der Wert ist, den SBTWAN für die beschriebenen Eingaben ermittelt.

8.3.6 Boundary-Interior-Pfadtest

Wie im Implementierungs-Kapitel beschrieben, ermittelt SBTWAN die für die Boundary-Interior-Überdeckung notwendigen Pfade bei der Erstellung der Kontrollflussgraphen. Dabei werden die für die Auswertung wichtigen Pfade in drei Gruppen unterteilt:

- Pfade, die keine Schleifen durchlaufen
- Pfade, die Schleifen beinhalten und nur einmal durchlaufen
- Pfade, die mehr als einmal Schleifen durchlaufen

Der für den Test herangezogene Algorithmus enthält eine `while`-Schleife. Für den Test betrachten wir zunächst die Pfade, die die Schleife nicht betreten. Theoretisch existieren dabei zwei Pfade, nämlich je einer für die beiden Zweige der `if`-Verzweigung. Tatsächlich wird die Schleife nur dann nicht ausgeführt, wenn die Eingabe leer ist, was genau eine der Bedingungen der `if`-Abfrage ist, so dass der `else`-Zweig nie ausgeführt wird, ohne dass vorher die Schleife betreten wurde. Es existieren also zwei Pfade ohne Schleifendurchläufe, wovon nur einer erreichbar ist.

Innerhalb der Schleife befindet sich keine weiteren Subschleifen, so dass nur die Pfade betrachten werden müssen, die sich durch die geschachtelten `if`-Anweisungen ergeben. Die `if- else if-else`-Konstruktion erzeugt drei weitere Pfade. Im true-Abschnitt des `if`s ist ein weiteres `if` verschachtelt, allerdings nur mit einem true-Teil, wodurch ein weiterer Pfad gebildet wird.

Somit finden sich in der Schleife insgesamt 4 verschiedene Pfade. Die Bedingung `woBinIch.equals("NachDemKomma")` des inneren `if`s kann beim ersten Durchlauf der Schleife nie wahr werden, da `woBinIch` erst in den darauf folgenden Anweisungen anders gesetzt werden könnte. Für die innere Schleife bleiben also im ersten Durchlauf 3 Pfade.

Um zu überprüfen, ob das Schleifeninnere mehr als einmal in allen möglichen Kombinationen durchlaufen wurde, werden die vier Pfade des einfachen Durchlaufs entsprechend kombiniert, so dass sich eine Zahl von insgesamt 16 Pfaden ergibt.

Eingabe	Prozentsatz der Erfüllung
`.`	4%
`.9`	4%
`.z`	4%
`9`	4%
`9.`	4%
`99`	4%
`9z`	4%
leer	4%
`z`	4%
`.`,`.9`,`.z`,`9`, `9.`,`99`,`9z`,`z` und *leer*	40%

Abbildung 8.19: Testfälle für Boundary Interior-Pfadtest

Eine Überprüfung ergibt hier, dass von diesen Pfaden nur fünf tatsächlich erreichbar sind.

Es existieren also insgesamt 22 Pfade, von denen nur 9, also 40%, erreichbar sind. Wie SBTWAN diese Pfade bewertet, ist in Tabelle 8.19 dargestellt. Natürlich erzeugt ein einzelner Testfall dabei immer die gleiche Überdeckung, da er genau einen Pfad beinhaltet.

9 Nutzerdokumentation

9.1 Einführung

SBTWAN ist ein Programm, dass der automatisierten Auswertung von Tests im Bereich des strukturorientierten Softwaretests dient. Es ermöglicht die statische Auswertung von Quellcode von Softwaresystemen, um Informationen über die Komplexität der enthaltenen Klassen, Methoden und Funktionen zu erhalten. Auch ist eine grafische Darstellung des Kontrollflusses einzelner Module möglich.

Die Komplexitätsmaße, die im einzelnen ermittelt werden können, sind:

- zyklomatische Komplexität
- essentielle Komplexität

Im Zusammenspiel mit einem funktionsorientierten Testsystem, das das zu untersuchende Softwaresystem kontrolliert zur Ausführung bringt, oder durch manuelle Ausführung des zu testenden Systems lassen sich zu diesen einzelnen Durchläufen zusätzliche Informationen gewinnen. Dabei wird untersucht, auf welche Weise die verschiedenen Programmmodule durchlaufen wurden und ob dabei bestimmte Kriterien erfüllt wurden. Damit ist die Überprüfung von nicht-funktionalen Testkriterien möglich.

Kriterien, die überprüft werden können, sind dabei:

- Anweisungsüberdeckung
- Zweigüberdeckung
- verschiedene Kriterien der Bedingungsüberdeckung
- Boundary-Interior-Pfadtest

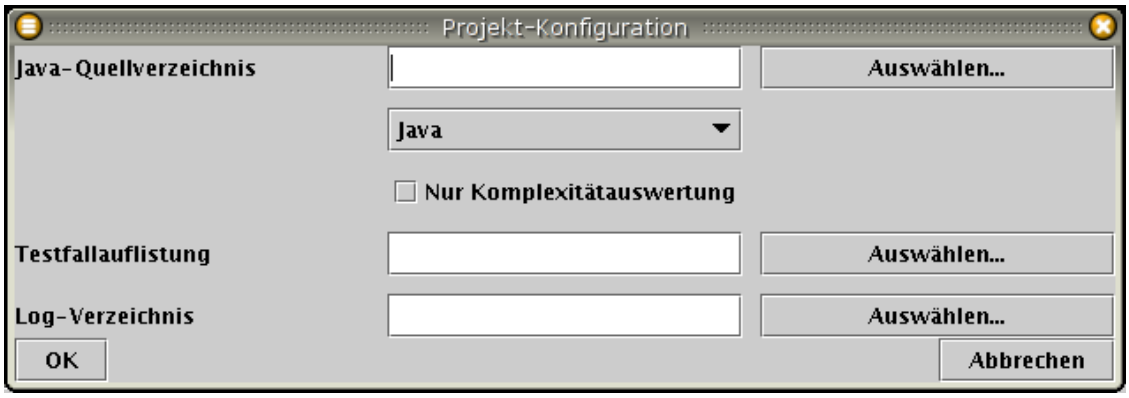


Abbildung 9.1: Dialog zur Projekt-Erstellung

Um Informationen zu erfolgten Programmdurchläufen zu gewinnen, ist es notwendig, in den Quellcode des Softwaresystems zusätzlichen Code einzufügen, der zur Laufzeit Aufzeichnungen über das Programmverhalten vornimmt (*Instrumentierung*). Dieser Vorgang wird dialoggesteuert unterstützt.

9.2 Erstellung eines Projektes

Die Erstellung eines Projektes kann aus der Symbolleiste und aus dem Menü *Datei* heraus gestartet werden.

Die Daten, die SBTWAN für seine Auswertungen benötigt, werden in einem *Projekt* zusammengefasst. Ein solches Projekt enthält die Informationen, die spezifisch zu einem Auswertungsvorgang gehören. Der Konfigurationsdialog (siehe Abbildung 9.1) ermöglicht die Zusammenstellung der benötigten Informationen.

Es gibt zwei Varianten von Projekten. Im ersten Fall, in dem nur Quellcode zur Analyse vorliegt, ist es möglich, ein Projekt zur rein statischen Analyse des Quellcodes zu erstellen. Diese Analyse umfasst dann lediglich eine Bewertung der Komplexität. Weiterführende Auswertungen, die eine Ausführung des Softwaresystems zu Voraussetzung hätten, sind in diesem Fall nicht möglich.

Die zweite Variante der Projekterstellung erlaubt weitergehende Auswertung für den Fall, dass der Quellcode des zu prüfenden Systems instrumentiert und ausgeführt wurde und so Aufzeichnungen vorliegen. Dies ermöglicht die Untersuchung von Überdeckungs- und Architektureigenschaften des Softwaresystems.

9.2.1 Quellcode

Zur Analyse wird der Quellcode des zu untersuchenden Systems benötigt. SBTWAN unterstützt mehrere verschiedene Programmiersprachen:

- Java
- C++
- C

Bei der Projekterstellung ist das Verzeichnis auszuwählen, das den zu untersuchenden Quelltext enthält. Zusätzlich muss die Sprache angegeben werden, in der der Programmcode vorliegt.

9.2.2 Testfalldefinition

SBTWAN benötigt für die Auswertung von dynamischen Überdeckungsmaßen nicht nur Quellcode, sondern auch Logdateien der Instrumentierung und die Namen der Testfälle, auf die sich diese Logdateien beziehen. Darüber hinaus sollte es möglich sein, verschiedene Testfälle unter einem Sammelbegriff zusammenzufassen, etwa Testfälle, die sich auf ein Subsystem beziehen.

Was sich hinter einem Testfallnamen verbirgt, welche Art der Programmausführung er also beschreibt, wird *nicht* in SBTWAN definiert. Dies ist Aufgabe eines funktionalen Testsystems oder des Testers selbst.

Die Daten über Testfälle werden einer Testfalldefinition entnommen, die in einer XML-Datei abgelegt wird.

Deren Format ist dabei folgendermaßen festgelegt:

```
<?xml version="1.0"?>
<testcases>
        <testcase>Testfallname</testcase>
        [... weitere testcase-Elemente ...]

        <testcaseset name="Bezeichnung">
                <case>Testfallname</case>
```

```
[... weitere case-Elemente ...]
        </testcaseset>
[... weitere testcaseset-Elemente ...]
</testcases>
```

Dabei können jeweils beliebig viele `testcase` und `testcaseset` aufgeführt werden. Die `case`-Elemente, die einzelne Testfälle von Testfallmengen festlegen, müssen auch als einzelne `testcase`-Elemente definiert werden (vgl. auch Abbildung 9.8 auf Seite 120).

9.2.3 Log-Dateien

Wenn das instrumentierte Programm ausgeführt wird, erzeugt es Log-Dateien, in denen Statusinformationen über die Programmausführung vermerkt werden. In der Regel ist dabei jedem Testfall eine Logdatei zugeordnet.

Bei der Projekterstellung muss das Verzeichnis angegeben werden, in dem sich diese Log-Dateien befinden. Die Log-Dateien müssen direkt in diesem Verzeichnis liegen, nicht in einem oder mehreren Unterverzeichnissen. Andernfalls werden sie nicht gefunden.

9.2.4 Laden und Speichern eines Projektes

Nach der Erstellung eines Projektes sind die Informationen, die eingegeben wurden, noch nicht permanent gespeichert und würden beim Schließen von SBTWAN verloren gehen. Um die Daten eines Projektes zu speichern und es später wieder zu laden, sind entsprechende Funktionen im Menü *Datei* und der Symbolleiste vorhanden.

Zu einem Zeitpunkt kann höchstens ein Projekt geladen sein. Der aktuelle Projektname wird dabei in der Titelzeile des Programmfensters angezeigt.

Ein Projekt kann explizit geschlossen werden, indem der Befehl *Projekt schließen* aus dem Menü *Datei* gewählt wird. Erfolgt dies nicht, bevor ein neues Projekt geladen wird, erfolgt das Schließen automatisch. Auch nicht gespeicherte Projekte werden beim Laden eines anderen Projektes automatisch geschlossen.

Im Menü *Datei* wird eine Chronik über die zuletzt geladenen Projekte geführt, um so das einfache Laden eines in einer vorherigen Programmsitzung angelegten und gespeicherten Projektes zu erlauben.

9.3 Auswertung

Wenn ein Projekt geladen ist, können die Daten, die es umfasst, ausgewertet werden. Der Aufruf der Auswertung erfolgt über die Symbolleiste oder das Menü *Auswertung*.

Dabei stehen drei Auswertungsmodi zur Verfügung:

- Komplexität
- Überdeckung
- Architektur

Auswertung nach Komplexität ist bereits möglich, wenn nur Programmcode zur Verfügung steht und ermöglicht Einblicke in die Programmstruktur.

Wenn zusätzlich Testdurchläufe und Aufzeichnungen dieser Testdurchläufe zur Verfügung stehen, können auch die Architektur des Programms und Überdeckungsmaße ausgewertet werden.

9.3.1 Komplexität

Die Komplexitätsauswertung beinhaltet drei Sichten auf das geladene Projekt:

- Struktur
- Kontrollflussgraph-Darstellung
- Quellcodeansicht

Dabei dient die Strukturansicht der Navigation und Steuerung der anderen Sichten. Dargestellt wird die logische Struktur des Programms, d.h. eine Aufgliederung

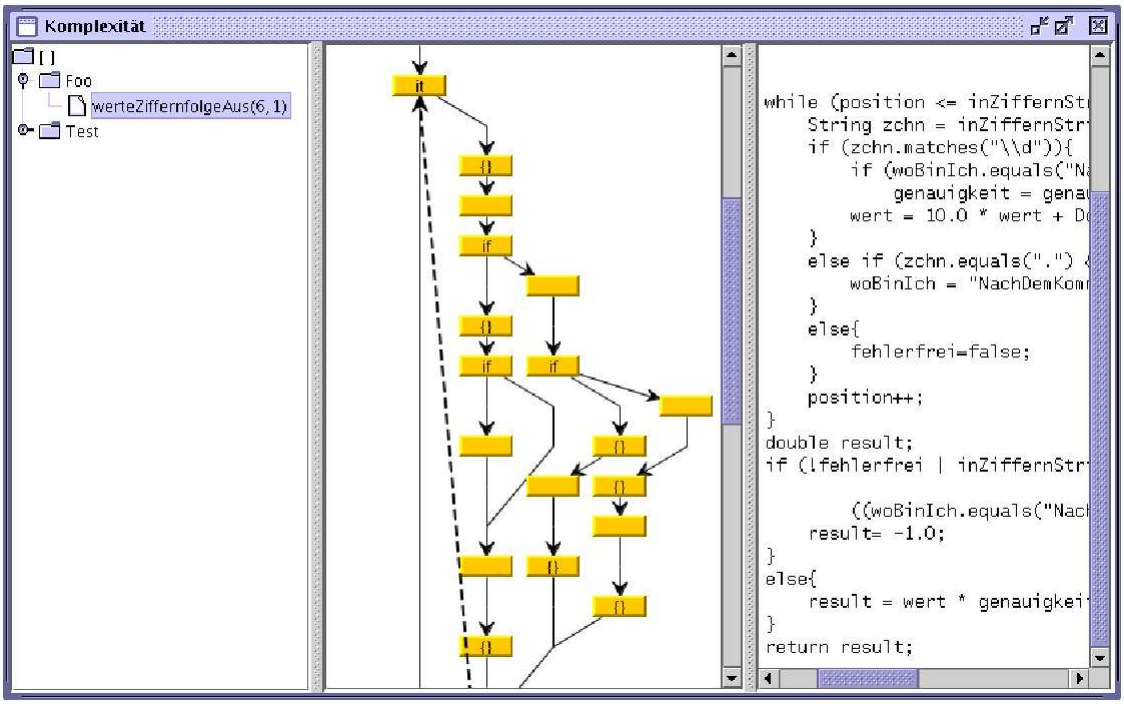


Abbildung 9.2: Komplexitätsauswertung

nach Klassen und Methoden. Hinter dem Namen einer Methode sind dabei in Klammern die zyklomatische und die essentielle Komplexität dieser Methode dargestellt.

Bei Anwahl einer Methode wird der Kontrollflussgraph und der dazugehörige Quellcode dargestellt (siehe Abbildung 9.2).

In der Quellcode- und der Kontrollflussansicht ist es möglich, über das Kontextmenü der jeweiligen Ansicht die Darstellungsgröße zu beeinflussen. Wird im Kontrollflussgraphen ein Element angeklickt, wird es in der Quelltextansicht hervorgehoben.

9.3.2 Überdeckung

Die Überdeckungsauswertung, die der komplexeste Auswertungsmodus ist, bildet den Kern der Funktionalität von SBTWAN (siehe Abbildung9.3).

Die Auswertung setzt sich aus den folgenden Komponenten zusammen:

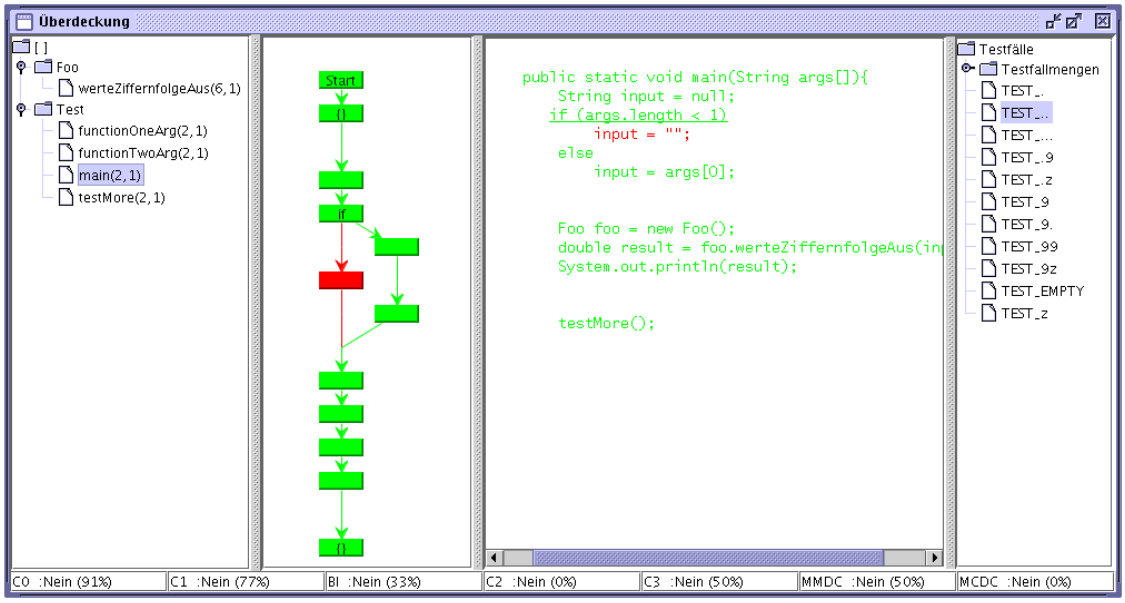


Abbildung 9.3: Überdeckungsauswertung

- Strukturansicht
- Kontrollflussgraph
- Quelltext
- Überdeckungsanzeige
- Testfälle

In dieser Ansicht wird die Erfüllung verschiedener Überdeckungsmaße analysiert:

- Anweisungsüberdeckung (C0-Test)
- Zweigüberdeckung (C1-Test)
- einfache Bedingungsüberdeckung (C2-Test)
- minimale mehrfache Bedingungsüberdeckung (MMDC-Test)
- mehrfache Bedingungsüberdeckung (C3-Test)
- MC/DC-Überdeckung (MC/DC-Test)

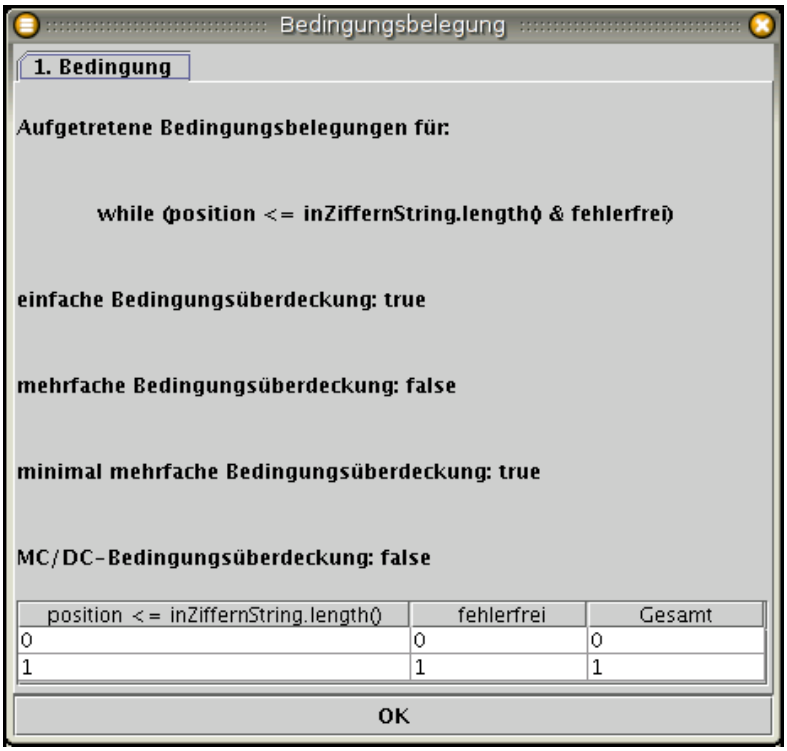


Abbildung 9.4: Darstellung einer Belegung

- Boundary-Interior-Überdeckung (BI-Test)

Die zu analysierenden Daten werden in der Strukturansicht des Programms, in der Klassen und Methoden ausgewählt werden können, und in der Testfallansicht, in welcher Testfälle und Testfallmengen ausgewählt werden können, festgelegt. Ist nur eine Methode ausgewählt, wird zu dieser Methode der entsprechende Kontrollflussgraph und der Quelltext angezeigt. Bei der Auswahl von Testfällen werden die durchlaufenen Elemente im Graphen und im Quelltext farbig hervorgehoben.

Treten im Quelltext Bedingungen auf, werden diese unterstrichen und durch einen Klick können die Wahrheitsbelegungen dieser Bedingung eingesehen werden (siehe Bild 9.4).

Werden mehrere Methoden oder Klassen ausgewählt, wird kein Quelltext oder Kontrollflussgraph dargestellt.

Für die aktuelle Auswahl wird jeweils im unteren Bereich des Fensters die Erfüllung von Überdeckungsmaßen angegeben.

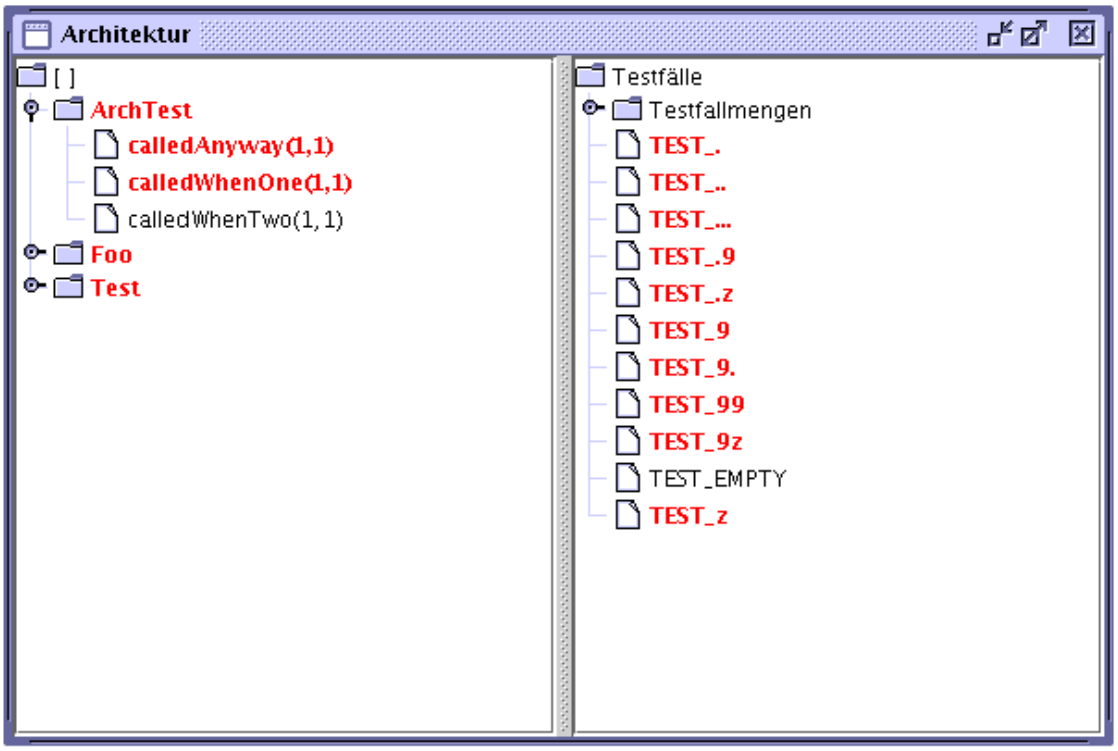


Abbildung 9.5: Architekturauswertung

In der Quellcode- und der Kontrollflussansicht ist es möglich, über das Kontextmenü der jeweiligen Ansicht die Darstellungsgröße zu beeinflussen. Wird im Kontrollflussgraphen ein Element angeklickt, wird es in der Quelltextansicht hervorgehoben.

9.3.3 Architektur

Die Architekturauswertung ermöglicht einen einfachen Einblick in die Struktur eines Programms. Es ist möglich, zu erfahren, welche Programmteile mit welchen Testfällen wie zusammen hängen(siehe Abbildung 9.5).

Die Architekturauswertung besteht aus zwei Komponenten:

- Strukturansicht
- Testfallansicht

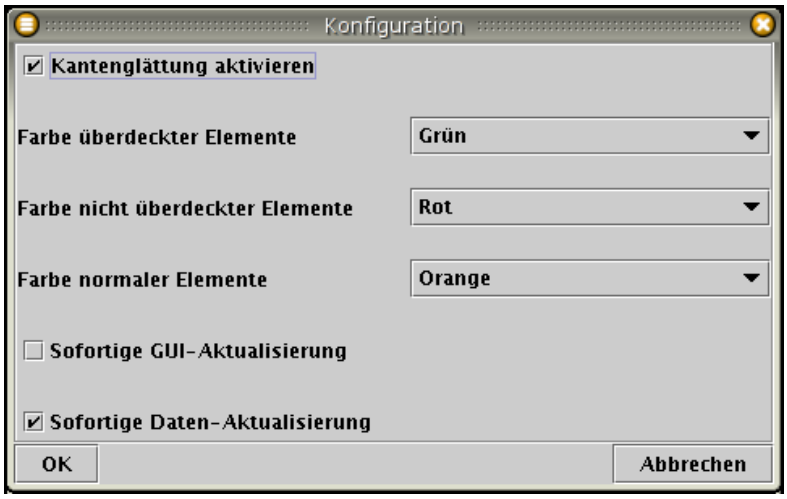


Abbildung 9.6: Konfiguration

In beiden Ansichten können jeweils Elemente, d.h. Testfälle und Klassen/Methoden ausgewählt werden.

Bei Auswahl von Testfällen werden die Klassen und Methoden farbig hervorgehoben, die durch von der Ausführung dieses Testfalls berührt wurden. Umgekehrt hebt die Auswahl einer Klasse oder Methode die Testfälle hervor, die deren Ausführung bedingt haben.

Es ist jeweils die gleichzeitige Auswahl mehrerer Elemente in beiden Sichten möglich.

9.4 Konfiguration

Es gibt eine Reihe Einstellungen von SBTWAN, die vom Nutzer konfiguriert und verändert werden können. Der Konfigurationsdialog kann aus dem *Datei*-Menü ausgewählt werden und ist in Abbildung 9.6 dargestellt.

Es sollte beachtet werden, dass Konfigurationsänderungen immer erst mit dem Laden bzw. Neu-Laden eines Projektes gültig werden. Unter Umständen muss also ein Projekt gespeichert und wieder geöffnet werden, wenn die sofortige Umsetzung von Konfigurationsänderungen gewünscht ist.

Die Konfiguration umfasst folgende Optionen:

- **Kantenglättung aktivieren**
  Beeinflusst die Darstellung von Kontrollflussgraphen und Quelltext. Aktivieren der Kantenglättung verbessert die Qualität der grafischen Darstellung. Aktivierte Kantenglättung erfordert zusätzliche Rechenleistung und verringert die Geschwindigkeit der Graphen- und Text-Darstellung.

- **Sofortige GUI-Aktualisierung**
  Einige Elemente der grafischen Oberfläche, wie etwa das Verschieben von Fenstern oder das Verändern der Trenner zwischen den Auswertungsbereichen, können während einer Veränderungsoperation oder erst nach deren Abschluss aktualisiert werden. Dieses Verhalten lässt sich mit dieser Option umschalten. Diese Option beeinflusst ebenfalls die Geschwindigkeit der Darstellung.

- **Sofortige Daten-Aktualisierung**
  SBTWAN führt nach Auswahl von Testfällen oder Funktionen umfangreiche Berechnungen bezüglich verschiedener Überdeckungsmaße durch. Ist diese Option aktiviert, findet eine Neu-Berechnung nach *jeder* Änderung der Selektionen statt. Andernfalls muss eine Aktualisierung der Darstellung explizit über eine Auswahlschaltfläche *Aktualisieren* in den Darstellungsfenstern ausgelöst werden.

- **Farbe überdeckter Elemente**
  Bestimmt die Farbe der überdeckten Elemente in der Überdeckungsauswertung. Die gewählte Farbe wird sowohl für die Darstellung von Knoten und Kanten im Graphen als auch für den Quelltext verwandt.

- **Farbe nicht überdeckter Elemente**
  Bestimmt die Farbe der nicht überdeckten Elemente in der Überdeckungsauswertung. Die gewählte Farbe wird sowohl für die Darstellung von Knoten und Kanten im Graphen als auch für den Quelltext verwandt.

- **Farbe normaler Elemente**
  Bestimmt die Farbe der Knoten des Kontrollflussgraphen in der Komplexitätsdarstellung. Kanten und Quelltext werden in diesem Modus immer in einer neutralen Farbe (d.h. schwarz) dargestellt.

9.5 Instrumentierung von Quellcode

Unter *Instrumentierung* wird der Vorgang verstanden, mit dem Quellcode in einer der unterstützen Sprachen mit zusätzlichen Anweisungen versehen wird, so dass er während seiner Abarbeitung Aufzeichnungen vornimmt, mit deren Hilfe sich die

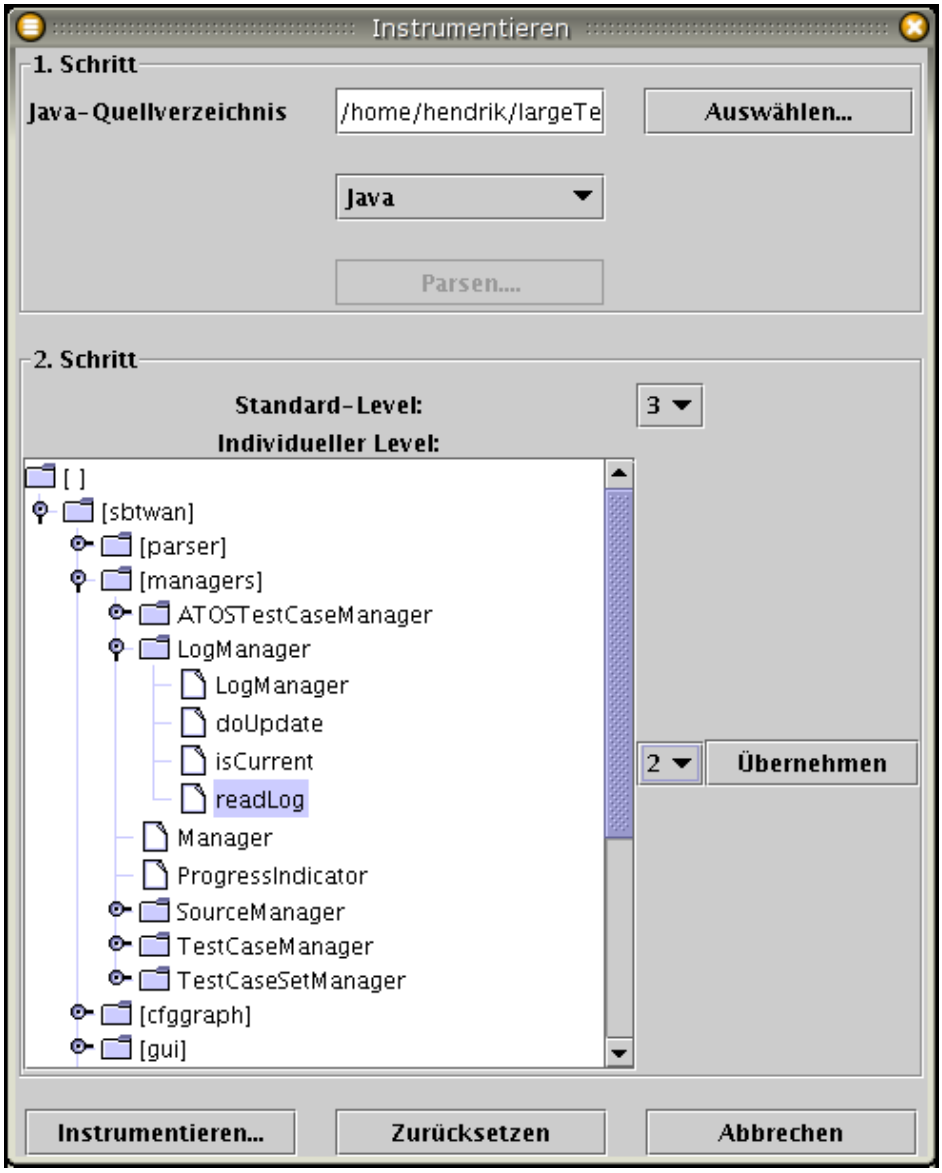


Abbildung 9.7: Instrumentierungsdialog

von SBTWAN unterstützen Überdeckungsmaße auswerten und bestimmen lassen. Die Instrumentierung erfolgt dialoggesteuert(siehe Abbildung 9.7) und ist über das Menü *Datei* oder die Symbolleiste erreichbar.

Der Vorgang der Instrumentierung ist in mehrere Schritte gegliedert.

Im ersten Schritt wird der zu instrumentierende Quellcode ausgewählt, seine Sprache festgelegt und der Einlesevorgang durch den Parser zur Vorbereitung gestartet, indem die Schaltfläche *Parsen* angewählt wird.

Ist der Quellcode geparst, wird seine logische Struktur für den zweiten Schritt angezeigt. Im zweiten Schritt ist es möglich, den globalen Instrumentierungslevel sowie den Level für einzelne Klassen und Methoden festzulegen.

Folgenden Instrumentierungslevel werden dabei unterstützt:

- **Level 0**
  In Level 0 wird keine Instrumentierung vorgenommen.
- **Level 1**
  Dieser Level stellt die Instrumentierung bereit, die für die Ermittlung von Anweisungs- und Zweigüberdeckung notwendig ist. Ebenfalls wird mit diesem Level die Architekturauswertung möglich.
- **Level 2**
  Zusätzlich zu Level 1 ist mit diesem Level die Auswertung von Bedingungs- und Pfadüberdeckung möglich.
- **Level 3**
  Dieser Level aktiviert eine Instrumentierung, die alle Anweisungen umfasst und nicht nur die Knotenpunkte der Verzweigungen des Kontrollflusses.

Für alle Klassen wird global ein Standard-Level festgelegt. Werden keine weiteren Einstellungen vorgenommen, wird dieser für das gesamte Projekt übernommen.

Zusätzlich ist es möglich, für einzelne Klassen und Methoden individuell Ausnahmen vom globalen Level vorzunehmen. Zu diesem Zweck wird eine Baumansicht der logischen Programmstruktur angezeigt. Dort können Klassen und Methoden ausgewählt werden. Rechts neben der Baumansicht ist eine Auswahl des Levels möglich, der mit der Schaltfläche *Übernehmen* gesetzt werden kann.

Abschließend kann mit der Schaltfläche *Instrumentieren* der eigentliche Vorgang der Instrumentierung gestartet werden. Dabei werden die originalen Quelltextdateien nach `<Dateiname>.OLD` umbenannt und an Ihre Stelle die instrumentierte Quellcodedatei gesetzt.

Weiterhin ist es möglich, den Instrumentierungsvorgang in den Ausgangszustand zurückzuversetzen, d.h. alle Dialogfelder wieder zu löschen, und den gesamten Vorgang abzubrechen.

9.6 Anhang der Dokumentation

9.6.1 Installation von SBTWAN

Das Verzeichnis von SBTWAN kann vom Installationsmedium an einen beliebigen Ort auf der Festplatte kopiert werden. Er enthält alle Bibliotheken, die zur Ausführung von SBTWAN notwendig sind, insbesondere den Parser und die JGraph-Bibliothek.

Da SBTWAN in Java entwickelt wurde, ist zusätzlich eine Java-Runtime in der Version 1.4 oder höher zur Ausführung notwendig.

Das SBTWAN-Verzeichnis enthält zwei Start-Dateien: `sbtwan.sh` und `sbtwan.cmd`, wobei die erste für den Start unter Unix/Linux und die zweite für den Start unter Windows benutzt wird.

In diesen Dateien finden sich jeweils zu Beginn einige Variablen, die dem Installationsort und der Systemumgebung angepasst werden müssen.

Die Variablen im einzelnen sind:

- `JAVA`
  Installationsort der Java-Runtime. Liegt diese im Systemsuchpfad, reicht es aus, nur den Dateinamen anzugeben.
- `INST`
  Der Pfad, in dem SBTWAN installiert wurde.
- `VMARGS`
  Wenn gewünscht, können hier weitere Parameter für die JAVA-Umgebung gesetzt werden.

9.6.2 Beispiel eines Auswertungsvorganges

Dieser Abschnitt beschreibt an einem Beispiel die Instrumentierung und anschließende Ausführung eines kompakten Beispielprogramms. Nach der Ausführung wird die Erstellung eines SBTWAN-Projektes und ein Teil des Auswertungsvorganges vorgestellt.

Dieser Vorgang ist folgendermaßen gegliedert:

1. Erstellung der Umgebung
2. Beschreibung der Testfälle
3. Instrumentierung und Übersetzung des Testprogramms
4. Einrichtung eines SBTWAN-Projektes
5. Auswertung

Als Beispiel wird ein einfacher Algorithmus namens `werteZiffernfolgeAus` gewählt, der in Java implementiert ist. Dieser Algorithmus parst positive Fließkommazahlen aus einer Zeichenkette und gibt dieses Ergebnis aus. Im Fehlerfall wird als Ergebnis eine `-1.0` zurückgegeben.

Das Beispiel setzt sich aus zwei Klassen zusammen. Die Klasse `Test` enthält die `main`-Funktion, aus der der Algorithmus, der in der Klasse `Foo` implementiert ist, aufgerufen wird.

Zur Durchführung werden in einem Verzeichnis zwei Unterverzeichnisse angelegt. Eines nimmt den originalen Quellcode auf, ein anderes die Log-Dateien.

Damit SBTWAN die Testfälle kennt, müssen diese in der Syntax beschrieben werden, die in dieser Dokumentation beschrieben ist. Im vorliegenden Fall beschreiben die Testfälle jeweils direkt die Eingaben für den Algorithmus und sind im Format `TEST_<Eingabe>` benannt.

Mehrere logisch zusammengehörige Testfälle sind zu Mengen zusammengefasst. In diesem Fall beschreiben die Mengen, welche Testfälle zusammengenommen bestimmte Überdeckungskriterien erfüllen.

Die Daten werden wie in Abbildung 9.8 in einer Datei `test.stc` zusammengefasst.

Um eine Auswertung von dynamischen Überdeckungsmaßen vornehmen zu können, muss der originale Quelltext instrumentiert werden. Darunter versteht man einen Vorgang, bei dem in den Quelltext zusätzliche Anweisungen eingefügt werden, die zur Laufzeit Aufzeichnungen vornehmen, die zur Auswertung notwendig sind.

Zur Instrumentierung wird der Instrumentierungsdialog von SBTWAN aus der Symbolleiste aufgerufen (siehe Abbildung 9.9). Im ersten Schritt wird das Verzeichnis mit dem Quellcode und die Sprache dieses Quellcodes angegeben und mit

```
<?xml version="1.0"?>
<testcases>
        <testcase>TEST_9.</testcase>
        <testcase>TEST_..</testcase>
        <testcase>TEST_.9</testcase>
        <testcase>TEST_9</testcase>
        <testcase>TEST_z</testcase>
        <testcase>TEST_EMPTY</testcase>
        <testcase>TEST_.</testcase>
        <testcase>TEST_...</testcase>
        <testcase>TEST_99</testcase>
        <testcase>TEST_.z</testcase>
        <testcase>TEST_9z</testcase>
        <testcaseset name="Anweisungsueberdeckung">
                <case>TEST_..</case>
                <case>TEST_.9</case>
        </testcaseset>
        <testcaseset name="Zweigueberdeckung">
                <case>TEST_..</case>
                <case>TEST_.9</case>
                <case>TEST_9</case>
        </testcaseset>
        <testcaseset name="minimal mehrfache Bed.-Ueberd.">
                <case>TEST_9</case>
                <case>TEST_z</case>
                <case>TEST_..</case>
                <case>TEST_EMPTY</case>
                <case>TEST_.</case>
                <case>TEST_.9</case>
        </testcaseset>
        <testcaseset name="Boundary-Interior">
                <case>TEST_.z</case>
                <case>TEST_.9</case>
                <case>TEST_9z</case>
                <case>TEST_9.</case>
                <case>TEST_99</case>
                <case>TEST_EMPTY</case>
                <case>TEST_z</case>
                <case>TEST_.</case>
                <case>TEST_9</case>
        </testcaseset>
        <testcaseset name="MC/DC">
                <case>TEST_9</case>
                <case>TEST_z</case>
                <case>TEST_..</case>
                <case>TEST_EMPTY</case>
                <case>TEST_.</case>
                <case>TEST_.9</case>
                <case>TEST_...</case>
                <case>TEST_99</case>
        </testcaseset>
</testcases>
```

Abbildung 9.8: Definition der Testfälle und Testfallmengen

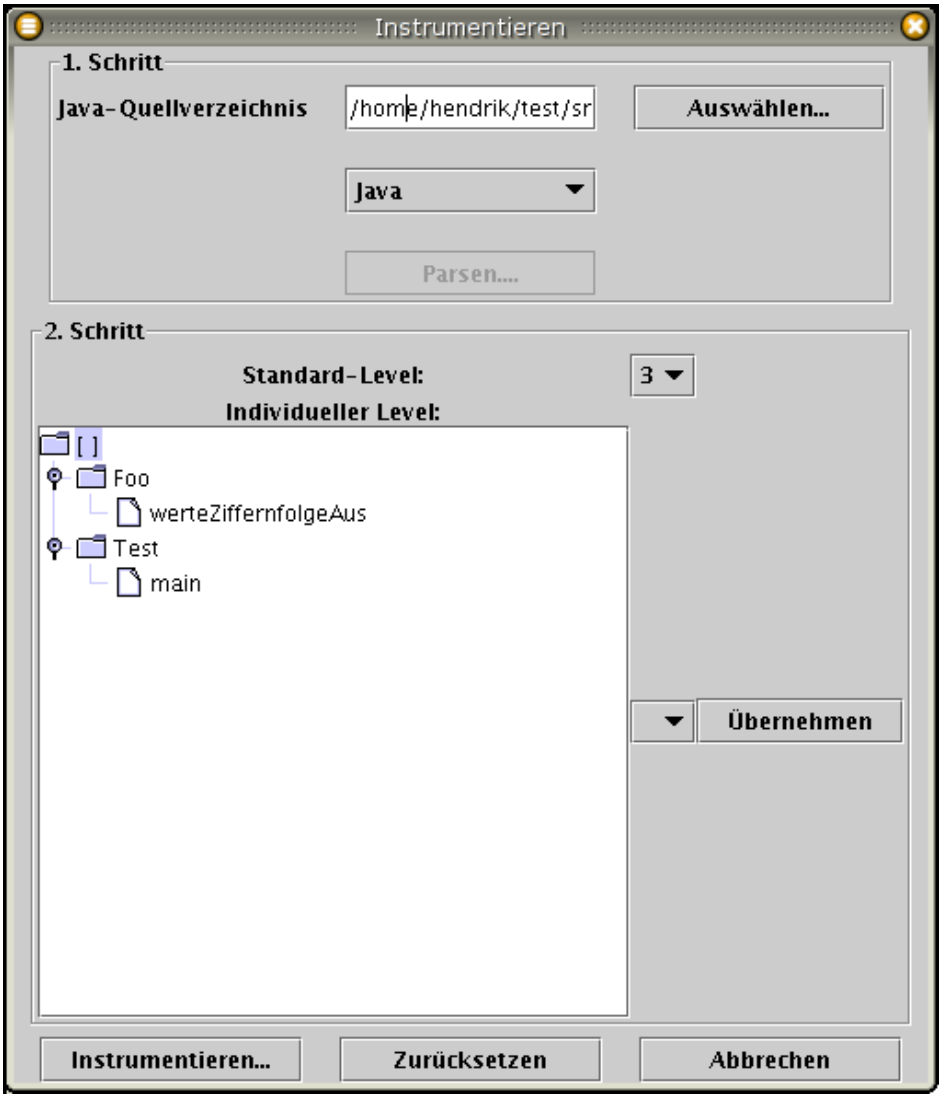


Abbildung 9.9: Instrumentierungsvorgang

Parsen die Struktur eingelesen. Im zweiten Schritt wird nun der globale Instrumentierungslevel auf drei festgesetzt und der Vorgang der Instrumentierung gestartet. Dabei werden die originalen Java-Dateien nach `<Dateiname>.OLD` umbenannt und durch instrumentierte Versionen ersetzt.

Der instrumentierte Quellcode kann jetzt kompiliert werden. Da durch die Instrumentierung zusätzlicher Code eingefügt wurde, muss die Parser-Bibliothek in den Klassenpfad aufgenommen werden. Wird das Programm nun ausgeführt, so wer-

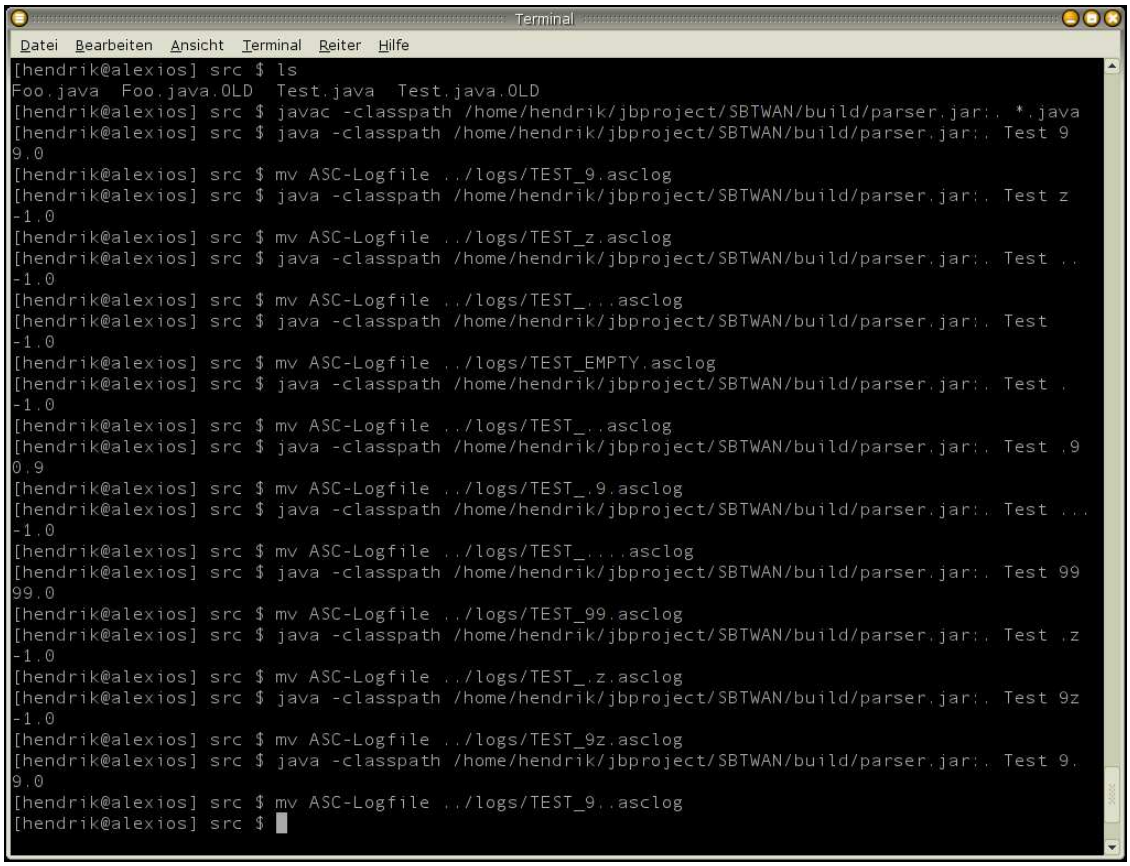


Abbildung 9.10: Kompilierung und Behandlung der Log-Dateien

den Aufzeichnungen über sein Laufverhalten vorgenommen. Diese werden in einer Log-Datei mit dem Namen `ASC-Logfile` abgelegt.

Diese Dateien müssen in das Log-Verzeichnis abgelegt werden, jeweils unter dem Namen, den der entsprechende Testfall trägt, wobei sie die Endung `.asclog` erhalten. Dies ist in Abbildung 9.10 dargestellt.

Nach der Ausführung muss zur weiteren Arbeit die instrumentierte Version des Quellcodes gelöscht und die alte Version aus den bei der Instrumentierung bereitgestellten Kopien wiederhergestellt werden.

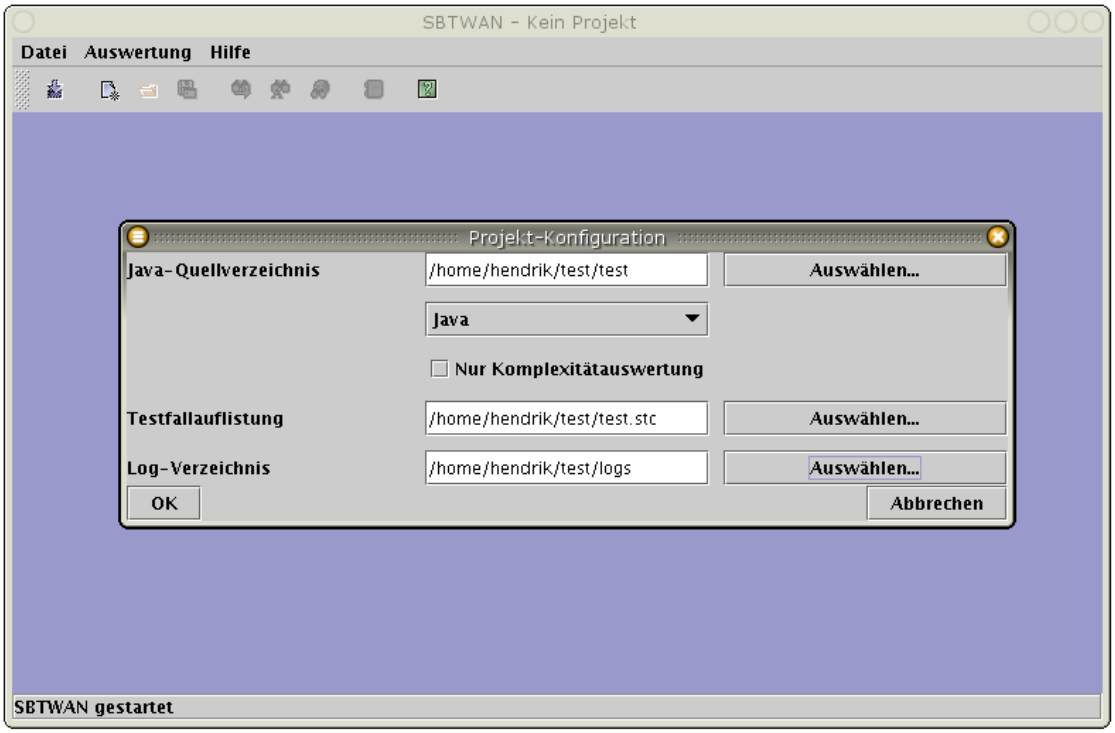


Abbildung 9.11: Projekt-Erstellung

Nun sind alle Daten vollständig, um ein Projekt in SBTWAN zu erstellen. Die Daten werden in der Projekt-Erstellung von SBTWAN angegeben (siehe Abbildung 9.11) und anschließend das Projekt erstellt.

Nach der Projekterstellung können die angefallenen Daten der Testdurchführung einer Auswertung unterzogen werden. In Abbildung 9.12 ist die Analyse der Überdeckung dargestellt. Es sind zwei Testfälle ausgewählt und im Quelltext und im Kontrollflussgraphen sind die durchlaufenen Elemente jeweils grün dargestellt und die nicht durchlaufenen rot. Im unteren Bereich sind die einzelnen Überdeckungsmaße und ihre jeweilige Erfüllung für die aktuelle Auswahl angezeigt.

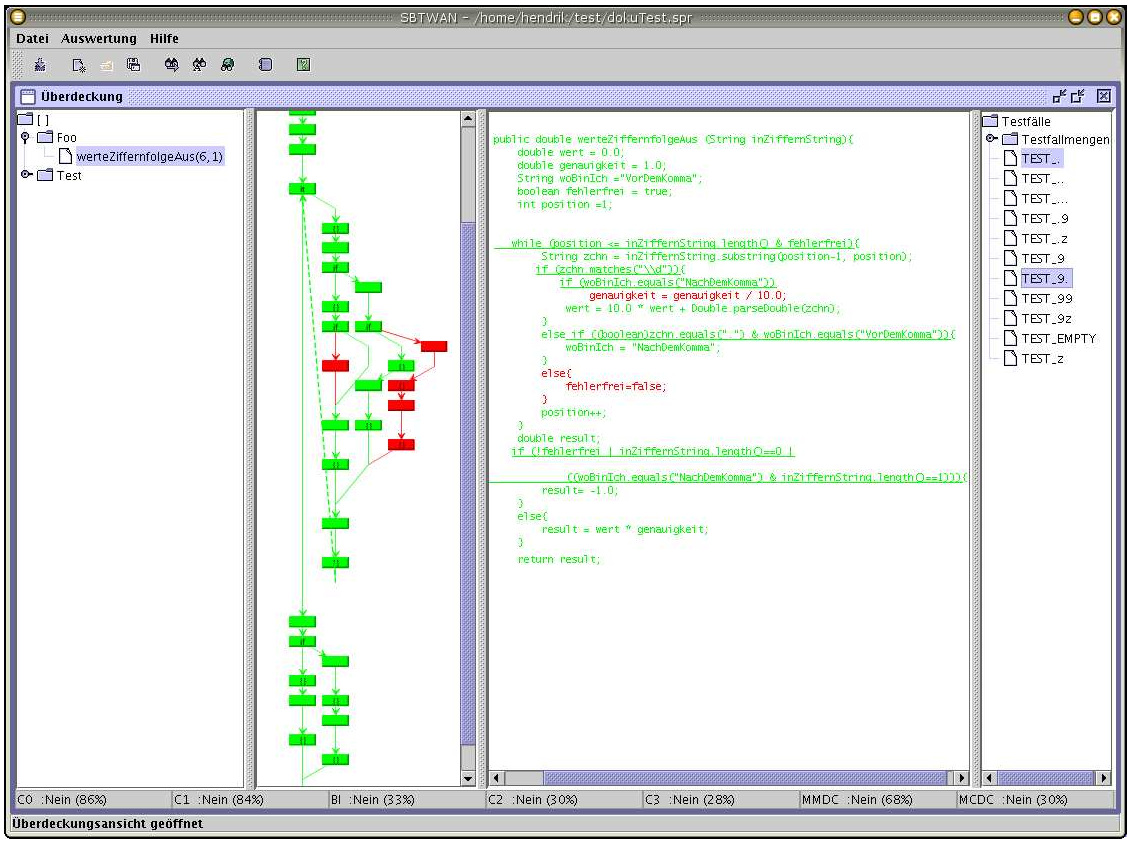


Abbildung 9.12: Auswertungsvorgang

Für die einzelnen Bedingungen können in der Quellcodeansicht, wie in Abbildung 9.13 dargestellt, genaue Daten über die Erfüllung von Bedingungsüberdeckungskriterien und die Wahrheitsbelegungen aufgerufen werden, indem auf die unterstrichenen Bedingungen geklickt wird.

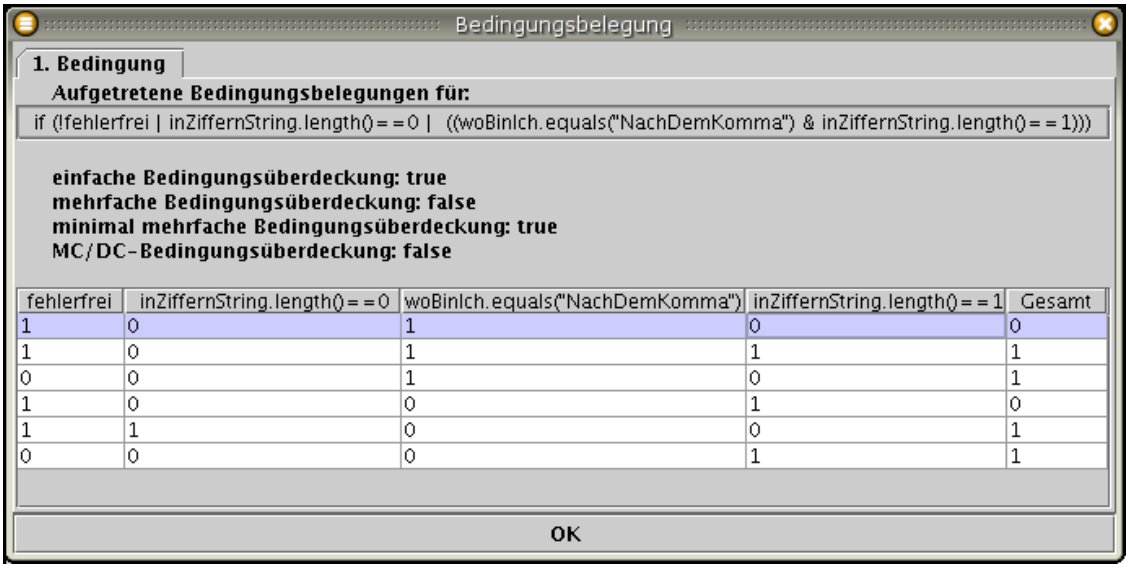


Abbildung 9.13: Detail des Auswertungsvorgangs

10 Ausblick und Bewertung

10.1 Zusammenfassung der Ergebnisse der Arbeit

In dieser Arbeit ist ein Softwaresystem von Grund auf entstanden, ohne dass auf eine bestehende Basis zurückgegriffen werden konnte. Daher lagen dem Ergebnis keine Einschränkungen zugrunde, die von vorne herein vorgegeben waren. Dies hat aber auch bedeutet, dass es notwendig wurde, fast alle notwendigen Komponenten selbst zu erstellen und Ihnen eine sinnvolle Struktur zu geben.

Als besonders wichtig hat sich in diesem Zusammenhang die Existenz eines klaren Designs und Konzeptes für die grundlegenden Strukturen herausgestellt. Insbesondere trifft dies auf die Schnittstellen zwischen dem Parser und Instrumentierer einerseits und dem Auswerter andererseits zu. Da beide parallel entwickelt und iterativ um neue Funktionen erweitert wurden, konnte so auch das Design ständig auf seine Tragfähigkeit untersucht werden.

Mit der Abbildung beliebiger vom Parser unterstützter Sprachen auf Kontrollflussgraphen in einer einheitlichen Struktur, ist eine einfache Erweiterung auf neue Sprachen und Sprachkonstrukte möglich. Dies bedeutet auch, dass die Algorithmen, die auf diesen Graphen operieren, sei es, um Komplexitätsmaße zu ermitteln oder um Überdeckungsmaße zu verifizieren, universell einsetzbar sind und keiner (oder zumindest keiner umfangreichen) Modifikation bedürfen, um neue Sprachen zu unterstützen.

Die ursprüngliche Stellung des Themas und seine Struktur haben für die GUI die realisierte dreigeteilte Struktur in die drei Auswertungsmodi nach Komplexität, Überdeckung und Architektur nahegelegt. Da in allen drei Modi verschiedene Sichten auf die Daten mehrfach genutzt werden, sind diese als getrennte Komponenten realisiert.

Durch eine Kombination der bestehenden Sichten auf die Daten mit neuen zusammengesetzten Ansichten, die andere Kriterien auswerten, ist eine Erweiterung innerhalb der bestehenden Strukturen möglich.

10.2 Ausblick und weiterführende Themen

Das Gesamtsystem aus Parser, Instrumentierer und Auswerter stellen eine umfangreiche Sammlung von Tools zur Verfügung, um Quelltext auf strukturorientierter Ebene statisch und zur Laufzeit auszuwerten. Durch die Abbildung des Quelltextes auf eine syntax- und sprachunabhängige Struktur ist eine Schnittstelle gegeben, um einheitlich mit Programmcode umgehen zu können.

Die auswertende Komponente leistet bisher Auswertung und Darstellung des Kontrollflusses und die Ermittlung der Komplexität des Kontrollflusses. Im Zusammenspiel mit einem funktionsorientierten Testsystem wird die Überprüfung der Einhaltung von Überdeckungsmaßen möglich.

Gegeben die Rahmen-Struktur und die grundsätzliche Instrumentierungs- und Auswertungsinfrastruktur, die zu schaffen einen Großteil der Arbeit umfasst hat, sind Erweiterungen in verschiedene Richtungen mit vergleichsweise geringem Aufwand möglich.

Neben diesen Erweiterungsmöglichkeiten im Bezug auf die Kriterien an Software, die ausgewertet werden können, ist auch im Bereich der grafischen Aufbereitung und Präsentation der Daten für den Nutzer noch ein weites Feld an Verbesserungen und Erweiterungen denkbar. Der Schwerpunkt bei der Erstellung von SBTWAN lag vor allem auf dem Erreichen der zu erzielenden Funktionalität in Hinsicht auf die auszuwertenden Testkriterien. Die GUI stellt Ergebnisse in übersichtlicher und einfach bedienbarer Art und Weise dar. Es sind aber eine Reihe von Erweiterungen denkbar, um die zur Verfügung stehenden Daten in komplexeren Zusammenhängen, einfacher erreichbar und konfigurierbarer darzustellen.

10.2.1 Erweiterungen der GUI

Ergänzung bestehender Elemente

Die grafische Oberfläche ist bisher vom Aufbau recht statisch und für den Nutzer bieten sich nur wenige Möglichkeiten, den Aufbau seinen Wünschen und Bedürfnissen anzupassen. Insbesondere bei der Auswertung ist der Aufbau der Fenster, die dazu zur Verfügung stehen, derzeit statisch. Man kann sich vorstellen, hier einen flexibleren Aufbau zu realisieren. Die Implementierung der einzelnen Sichten würde es problemlos erlauben, diese aus der zusammengesetzten Sicht heraus zu lösen und auf Nutzerwunsch in einem einzelnen Fenster oder an anderer Stelle darzustellen.

Wenn man eine solche beliebige Sortierung erlaubt, sollte zusätzlich eine Speicherung vorgenommener Änderungen realisiert werden.

Ebenfalls Erweiterungsspielraum ist für die Darstellung *in* den einzelnen Sichten denkbar. Bei der Klassen- und Methodendarstellung zeigen etwa gängige Entwicklungsumgebungen wie *JBuilder* oder *Eclipse* eine ähnliche Darstellung, die allerdings weitaus flexibler konfigurierbarer ist. Hier kann man wichtige Funktionen identifizieren und in SBTWAN ergänzen.

Die Darstellung des Kontrollflusses bietet eine Reihe von vorstellbaren Erweiterungsmöglichkeiten. Bisher werden Knoten und Kanten stets gleich dargestellt. Die JGraph-Bibliothek würde aber auch eine Erweiterung im Bezug auf die Darstellung erlauben. Es wäre etwa möglich, verschiedene Formen für Knoten und Kanten anzubieten oder per Tooltip für Knoten den dazugehörigen Quellcode anzuzeigen. In der Graphendarstellung könnte ebenso eine Darstellung von Komplexitätsmaßen aufgenommen werden.

Die Quellcodeansicht könnte bei intensiverer Verzahnung mit dem Parser eine Art von Syntax-Hervorhebung erlauben, etwa in der Weise, dass Schlüsselwörter besonders dargestellt werden.

Obwohl thematisch in dieser Arbeit von zentraler Bedeutung ist die reine Darstellung der Überdeckungsmaße im Moment sehr schlicht realisiert. Es werden rein textuelle Informationen gegeben. Die Prozentsätze könnten aber auch in Form von Balken oder je nach Überdeckungsgrad ausgefüllten Kreisen dargestellt werden.

Hinzufügen neuer Elemente

Die Definition von Testfällen und Testfallmengen erfolgt bisher außerhalb von SBTWAN mit einem beliebigen Editor. In diesen manuellen Vorgang können sich leicht Fehler einschleichen. Es wäre daher denkbar auch diesen Vorgang ähnlich wie die Instrumentierung mit GUI-Unterstützung, möglicherweise als Teil der Projekterstellung, stattfinden zu lassen.

Die Daten, die zu einem Projekt gehören, sind nach der Erstellung und Speicherung derzeit nicht mehr änderbar, etwa im Bezug auf die dazugehörigen Dateien, es sei denn, man editiert manuell die Projektdateien. Auch diesen Vorgang könnte man über die GUI zugänglich machen.

10.2.2 Profiling

Durch die Instrumentierung werden eine Reihe von verschiedenen Informationen aufgezeichnet, u.a. Informationen darüber, wie oft eine einzelne Anweisung durchlaufen worden ist. Im Rahmen der Auswertung, wie sie im Rahmen dieser Arbeit zu leisten war, wurde für die Analyse von Anweisungs- und Zweigüberdeckung hierbei nur die Tatsache, *dass* eine Anweisung ausgeführt wurde, beachtet. Die genaue Zahl der Durchläufe findet bisher nicht Verwendung bei der Betrachtung. Um diese Informationen nicht ungenutzt zu lassen, wäre daher eine Erweiterung in Richtung einer *Profiling*-Anwendung möglich, in deren Rahmen Daten über besonders intensiv genutzte Programmteile gewonnen werden könnten. Diese Daten könnten etwa benutzt werden, um besonders wichtige, d.h. oft durchlaufene, Programmteile aufzufinden, etwa um sie besonders auf ihre Fehlerfreiheit oder Performanz zu untersuchen und zu optimieren.

10.2.3 Konstruktion von Aufrufgraphen

Zur Erstellung der Pfade führt der instrumentierte Quellcode weitere Aufzeichnungen mit sich, die nach der Konstruktion der Pfade und deren Bereitstellung keine weitere Verwendung finden. Es werden Zeitstempel erzeugt, die die Aufrufe wichtiger Elemente in einer chronologischen Reihenfolge darstellbar machen. Auf diese Weise ist die Konstruktion von Aufrufgraphen möglich. Aufrufgraphen beschreiben Abhängigkeiten zwischen Funktionen oder Methoden, die sich aufgrund von Aufrufanweisungen ergeben. Dies würde einen weiteren tiefen Einblick in die Codestruktur erlauben.

10.2.4 Testfall-Reduktion

Bei der Erstellung von Testfällen, insbesondere, wenn komplexe Überdeckungsmaße wie MC/DC erfüllt werden sollten, entsteht schnell eine sehr große und in ihrer Handhabung unübersichtliche Menge von Testfällen. Auf der anderen Seite liegt die Vermutung nahe, dass ein Großteil der erstellten Testfälle sehr ähnliche und in Abschnitten gleiche Programmteile testen. Bei der iterativen Erstellung von Testfällen tritt außerdem ohne Absicht der Fall ein, dass zu einem späterem Zeitpunkt erstellte Testfälle in früher erstellten komplett enthalten sind, die damit eigentlich überflüssig wären.

Diese Zusammenhänge manuell zu überprüfen, ist sicherlich nicht machbar. Es existieren eine Reihe von Herangehensweisen, um die Optimierung von Testfällen zu automatisieren, sieh dazu etwa [RUCH01]. Obwohl MC/DC-Überdeckung, die auch im Rahmen dieser Arbeit implementiert wurde, ein vergleichsweise neues Überdeckungsmaß ist, gibt es auch für dieses Maß Arbeiten zur Optimierung von Testfallmengen, [Dec] etwa beschreibt einen Algorithmus, der dazu Verwendung finden kann.

In SBTWAN stehen alle Informationen bereit, die nötig sind, um die entsprechenden Auswertungen im Bezug auf die Testfallreduktion vornehmen zu können.

Literaturverzeichnis

[Bal98] Balzert, Helmut: *Lehrbuch der Software-Technik: Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung*, Seite 253ff. Heidelberg; Berlin: Spektrum, Akad. Verlag, 1998.

[Dec] Decision, Modified Condition: *Test-Suite Reduction and Prioritization for*.

[do192] *Software Considerations in Airborne Systems and Equipment Certification*. Technischer Bericht RTCA/DO-178B, RTCA, Washington, D.C., U.S., 1992.

[jgr] jgraph.com: *JGraph Swing Component*. http://www.jgraph.com.

[JH02] Jens Hanisch, Johann Letzel: *Automatisierung von Regressionstests eines Programms zur Halbleiter-Strukturanalyse*. Diplomarbeit, November 2002.

[KJHR01] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski und Leanna K. Rierson: *A Practical Tutorial on Modified Condition/ Decision Coverage*. Technischer Bericht NASA/TM-2001-210876, U.S. National Aeronautics and Space Administration, Langley Research Center, Hampton, Virginia, U.S., May 2001.

[Lab] Labs, AT&T: *Grappa - A Java Graph Package*. http://www.research.att.com/ john/Grappa/.

[Lam03] Lamport, Leslie: *Specifying systems : the TLA+ language and tools for hardware and software engineers*. 2003.

[Li87] Li, Eldon: *On the Cyclomatic Metric of Programm Complexity*. Data Processing, Journal of the Quality Assurance Institute, 1987. California Polytechnic State University, School of Business Quality.

[Lig02] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verfizieren von Software*. Nummer ISBN 3-8274-1118-1. Spektrum Verlag, Heidelberg Berlin, 2002,.

[McC76] McCabe, T.: *A Complexity Measure*, Band SE-1 der Reihe *Transactions on Software Engineering*. IEEE, 1976. Seiten 312-327.

[Pag94] Pagel, Bernd-Uwe und Six, Hans-Werner: *Software Engineering : Die Phasen der Softwareentwicklung*, Seite 434ff. Bonn, Paris, Reading (Mass.): Addison-Wesley, 1994.

[RUCH01] Rothermel, Gregg, Roland H. Untch, Chengyun Chu und Mary Jean Harrold: *Prioritizing Test Cases For Regression Testing*. Software Engineering, 27(10):929–948, 2001.

[SF01] Sebastian Freund, Derrick Hepp: *Vom Reverse Engineering zur Programmerweiterung: Automatische Justage für ein Röntgentopogrophie-Steuerprogramm*. Diplomarbeit, Mai 2001.

[SM] Sun Microsystems, Inc: *Java Compiler Compiler*. https://javacc.dev.java.net/. Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator.

[Spi92] Spivey, J.M.: *The Z notation: a reference manual*. Computer Science. Prentice Hall International, 2. Auflage, 1992.

[Tre] Treysse, Ronny: *Erstellung eines Tools zur Überdeckungsmessung, Use-Case Instrumentierer*.

[uKG93] K. Grimm, M. Grochtmann und: *Classification Trees for Partition Testing*. Software Testing, Verification and Reliability, 3(2):63–82, Jun 1993.

[VB01] Vilkomir, Sergiy A. und Jonathan P. Bowen: *Formalization of Control-flow Criteria of Software Testing*. Technischer Bericht SBU-CISM-01-01, South Bank University, SCISM, London, UK, January 2001.

[Wal90] Wallmüller, Ernest: *Software-Qualitätssicherung*. Nummer ISBN 3-446-15486-4. Carl Hanser Verlag, München Wien, 1990,. Seite 183.

[WM96] Watson, A. und T. McCabe: *Structured testing: A testing methodology using the cyclomatic complexity metric*. Technischer Bericht NIST Special Publication, 500–235., U.S. National Institute of Standards and Technology, Washington, D.C., U.S., Sep 1996.

[Zus90] Zuse, H.: *Software Complexity: Measures and Methods*, Seite 67ff. Amsterdam: de Gruyter, 1990.

Abbildungsverzeichnis