Wohldefinierte Überdeckungsmetriken für den Glass-Box-Test

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Rainer Schmidberger
aus Heilbronn

Hauptberichter:	Prof. Dr. Jochen Ludewig
Mitberichter:	Prof. Dr. Martin Glinz
Tag der mündlichen Prüfung:	14. Juli 2014

Institut für Softwaretechnologie der Universität Stuttgart
2014

Dank

Zum Gelingen dieser Arbeit haben viele Personen beigetragen, denen ich an dieser Stelle für ihre Unterstützung danken möchte. An erster Stelle gilt mein ganz besonderer Dank meinem Doktorvater Prof. Jochen Ludewig, der mit seinen vielen wertvollen Anregungen und seiner konstruktiven Kritik ganz wesentlich zum Gelingen dieser Arbeit beigetragen hat. Herrn Professor Martin Glinz danke ich herzlich für die freundliche Übernahme des Zweitgutachtens und die wertvollen Hinweise.

Ein ganz herzlicher Dank geht auch an die Kolleginnen und Kollegen der Abteilung Software Engineering. Die Zusammenarbeit war mir stets eine große Freude. Ivan Bogicevic, Angela Georgescu, Markus Knaus, Daniel Kulesz, Tilmann Hampp, Stefan Opferkuch, Holger Röder, Adolf Veith-Willer und Matthias Wetzel – vielen Dank! Ganz besonders möchte ich mich bei Frau Kornelia Kuhle für die gewissenhafte Korrektur meiner Texte bedanken.

Ebenso möchte ich mich bei den Studenten ganz herzlich bedanken, die am Werkzeug CodeCover mitgearbeitet haben. Stefan Franke, Steffen Hanikel, Robert Hanussek, Benjamin Keil, Steffen Kieß, Johannes Langauf, Christoph Marian Kreidler, Igor Podolskiy, Tilmann Scheller, Michael Starzmann und Markus Wittlinger haben im Rahmen des CodeCover-Studienprojekts das wesentliche Grundgerüst erstellt. Sebastian Schumm, Ralf Ebert und Steffen Hanikel haben durch Weiterentwicklungen von CodeCover im Rahmen Ihrer Diplom- oder Bachelor-Arbeiten wichtige Beiträge geleistet.

Stuttgart, im Juli 2014

Rainer Schmidberger

Zusammenfassung

Der Glass-Box-Test (GBT), der auch als White-Box- oder Strukturtest bezeichnet wird, zeigt den im Test ausgeführten – und viel wichtiger: den nicht ausgeführten – Programmcode an. Dieser Grad der Ausführung wird als Überdeckung bezeichnet. Empirische Untersuchungen zeigen eindeutig, dass eine hohe GBT-Überdeckung mit einer geringeren Restfehlerrate korreliert. Viele brauchbare Werkzeuge sind für den GBT für praktisch alle Programmiersprachen verfügbar, und wichtige Industriestandards zur Zertifizierung sicherheitskritischer Software verlangen den Nachweis einer vollständigen (oder sehr hohen) Überdeckung. Auf den ersten Blick erscheint uns damit der GBT als eine ausgereifte und etablierte Testtechnik, die auf standardisierten Metriken basiert.

Doch bei genauer Betrachtung der zugrundeliegenden Modelle und Metriken zeigen sich erhebliche Mängel, die zu unpräzisen und inkonsistenten Resultaten der verschiedenen GBT-Werkzeuge führen. Eine wesentliche Ursache hierfür bildet der Kontrollflussgraph (CFG), auf dessen Grundlage überwiegend die Definitionen der GBT-Metriken vorgenommen werden. Der CFG hat hierfür gravierende Nachteile: Die Transformation der realen Programme in den CFG ist nicht eindeutig definiert, und es fehlt im CFG eine Repräsentation für die Ausnahmebehandlung sowie die GBT-relevanten Ausdrücke, wie z. B. bedingte Ausdrücke oder die Kurzschlusssemantik boolescher Operationen. Als logische Konsequenz folgen die Werkzeuge des GBT keinem gemeinsamen Standard und zeigen für die gleiche Programmausführung deutlich verschiedene Überdeckungswerte an.

In dieser Arbeit wird ein präzises Modell für den GBT präsentiert. Dieses Modell entsteht in zwei Schritten: Zunächst wird eine GBT-Modellsprache definiert (die Reduced Program Representation, RPR), die die GBT-relevanten Aspekte der realen Sprachen abbildet, die irrelevanten dagegen präteriert. Aus der RPR-Definition entstehen sogenannte Ausführungselemente, die entweder primitiv sind (wie z. B. primitive Anweisungen), oder die sogenannten Verbund-Ausführungselemente (wie z. B. if-Anweisungen), die als Teil der eigenen Struktur andere Ausführungselemente enthalten. RPR bildet dabei sowohl den Kontrollfluss als auch die GBT-relevanten Ausdrücke wie z. B. den bedingten Ausdruck oder zusammengesetzte boolesche Ausdrücke ab. Auch ist RPR so angelegt, dass Ausnahmen berücksichtigt werden und reale Programme systematisch und eindeutig nach RPR transformiert werden können.

Im zweiten Schritt wird die Ausführungssemantik der Ausführungselemente durch Petri-Netze, sogenannte Modellnetze, definiert. Primitive Ausführungselemente lassen sich direkt als Modellnetz beschreiben, die Verbund-Ausführungselemente wie z. B. die Entscheidungsanweisung oder der zusammengesetzte boolesche Ausdruck werden durch Komposition der Teilnetze beschrieben. Die Modellsprache RPR liefert hierzu die Kompositionsregeln. Ein Vorteil des formalen Modells ist, dass die Evaluation der Netze werkzeugunterstützt über die Erreichbarkeitsanalyse erfolgen kann. Ein besonderer Vorteil der

Modellnetze ist auch, dass sich die zur Metrikenbestimmung wichtigen Ausführungszähler präzise im Modell platzieren lassen und so die Spezifikation für eine Werkzeugimplementierung liefern.

Auf Grundlage der Ausführungselemente und der Ausführungszähler der Modellnetze erfolgt eine präzise Definition der populären sowie weiterer GBT-Metriken. Zur formalen Prüfung der Plausibilität, Differenziertheit und Vergleichbarkeit dieser Metriken wird ein Regelsystem aufgestellt, anhand dessen die GBT-Metriken geprüft werden.

Da für den GBT ein Werkzeugeinsatz Voraussetzung ist, wird das Werkzeug CodeCover präsentiert, das eine Referenzimplementierung der definierten Metriken liefert. Als eine Folge der programmiersprachenunabhängigen RPR-Darstellung unterstützt CodeCover mehrere Programmiersprachen: Java, C und COBOL.

CodeCover bietet auch eine neue Funktion, die den Tester durch sogenannte Testfall-Hinweise beim Entwurf von GBT-basierten Testfällen systematisch unterstützt. Diese Testfälle führen einerseits zu einer Erhöhung der Überdeckung. Durch eine gezielte Priorisierung der Testfall-Hinweise wird aber auch eine hohe Fehlersensitivität der neu entwickelten Testfälle angestrebt.

Abstract

The subject of this work is the systematisation of the Glass Box Test. The Glass Box Test (GBT), also known as White Box Test or Structural Test, shows which parts of the program under test have, or have not, been executed. This degree of execution is called coverage. Empirical studies clearly indicate that higher GBT coverage correlates with lower post-release defect density. Many GBT tools are available for almost any programming language, and industry standards for safety-critical-software require a very high or even complete coverage. At first glance, the GBT seems to be a well-established and mature testing technique that is based on standardized metrics.

But upon a closer look at the underlying models and metrics, they show severe shortcomings that lead to imprecise and inconsistent coverage results of the various GBT tools. One main reason for this is the control flow graph (CFG) that is mostly used to define the popular GBT metrics. But the CFG has some severe disadvantages: The transformation of real programs in the CFG is not clearly defined, and the CFG has no representation for exception handling and the GBT-relevant expressions, such as the conditional expressions or the short-circuit semantics of Boolean operations. As a consequence, the various GBT-tools do not follow a common standard and show for the same program execution significantly different coverage results.

In this work a new and precise model for the Glass Box Test is presented. This model is developed in two steps: First a GBT model language is defined (the Reduced Program Representation, RPR), which is reduced to the GBT-relevant aspects of the real programming languages. Using the RPR-definitions, so-called execution items are defined that are either primitive (e. g. primitive statement) or complex (so-called compound items) that contain other GBT items as part of their own structure. RPR models both the control flow and GBT-relevant expressions and real programs can be transformed systematically and precisely into RPR.

In the second step, the execution semantics of the GBT items is defined using Petri nets called GBT model nets. Primitive GBT items were directly described as model nets, the compound items such as the if-statement or the compound Boolean expression are described as a composition of model (sub) nets. For this, RPR provides the composition rules. An advantage of the formal model net is that the evaluation can be done using a tool based reachability analysis. Another important advantage of the model net is the precise placement of the execution counters that have significant influence in the GBT metrics determination. Using this, the model builds a precise specification for GBT tool implementations.

On the basis of the GBT items and the execution counters of the model nets, a precise definition of the popular GBT metrics is presented. Additionally plausibility rules were defined, which were applied to check the GBT metrics' plausibility, differentiation, and comparability.

Because tools are required for the GBT, the tool CodeCover is presented. CodeCover is an implementation that strictly follows the defined metrics. As a result of the programming languages independent RPR-representation CodeCover supports several programming languages: Java, C and COBOL.

Additionally CodeCover provides a new functionality which systematically supports the tester in the development of new test cases using so-called test case recommendations. First, these test cases lead to an increase in the GBT coverage. And second, using systematic prioritization of the test case recommendations, also a high error sensitivity of the newly developed test cases is intended.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1
Einleitung und Überblick

Dieses Kapitel liefert einen Überblick über den Inhalt dieser Arbeit. Zunächst wird die Motivation vorgestellt, die der Arbeit zugrunde liegt und es werden die Zielsetzung sowie der Lösungsansatz der Arbeit beschrieben. Anschließend folgt die Inhaltsübersicht der einzelnen Kapitel.

1.1 Motivation

Der Programmtest hat sich in den letzten Jahren von einer „Randerscheinung“ zu einem zentralen und als erfolgskritisch wahrgenommenen Projektbestandteil entwickelt. Entsprechend verwenden die Unternehmen einen beträchtlichen Anteil der Projektbudgets für den Test, der in vielen Fällen dennoch nicht die angestrebte Güte erreicht – (zu)viele, auch schwerwiegende Fehler bleiben im Prüfling unentdeckt. Gleichzeitig spielt die Qualität der Software eine immer größere Rolle und die Zahl an „Post Release Defects“ entwickelt sich für die Unternehmen zu einem entscheidenden Wettbewerbsfaktor. Dieser gestiegenen Wahrnehmung und wirtschaftlichen Bedeutung des Tests folgend besteht in der Industrie großes Interesse daran, die vormals „intuitiv“ und „hemdsärmelig“ durchgeführten Testtechniken zu systematisieren.

Einen Beitrag zur Systematisierung des Tests kann der Glass-Box-Test (GBT) liefern, der den im Test ausgeführten (und damit auch den nicht ausgeführten) Programmcode anzeigt. Zwar erscheint uns heute der GBT als eine ausgereifte und etablierte Testtechnik, es zeigen sich jedoch bei genauer Betrachtung der hierzu genutzten Modelle und Metriken erhebliche Mängel: Wichtige GBT-relevante Programmmerkmale werden in den Modellen nicht berücksichtigt und für die Übertragung der Programme in das Modell gibt es keinen Standard.

Der Glass-Box-Test ist dabei keineswegs neu: Erste Arbeiten von 1963 gehen auf Miller und Maloney zurück [MM63], und bereits 1975 beschreibt Huang [Hu75] den Glass-Box-Test prinzipiell in der Form, wie er auch heute in den Lehrbüchern (wie z. B. [Li02, SL04]) behandelt wird[1]. Huang definiert Anweisungs- und Zweigüberdeckung auf Grundlage des Kontrollflussgraphen und beschreibt, wie Zähler in ein Programm eingefügt werden

[1] Soweit man es heute zurückverfolgen kann, wurde der GBT unabhängig von mehreren Autoren in der Zeit ab 1963 bis 1975 publiziert. Der Artikel von Huang hebt sich aber dadurch ab, dass er den GBT weitgehend in der heute bekannten Form beschreibt.

und wie die Auswertung des GBT erfolgt. Der Kontrollflussgraph ist seitdem im Wesentlichen unverändert das vorherrschende GBT-Modell. Dabei wird durch den Wegfall von (oder Verzicht auf) Goto-Anweisungen in den aktuellen Programmiersprachen der Kontrollfluss einfacher, Ausnahmen (Exceptions) wiederum machen den möglichen Kontrollfluss deutlich aufwändiger. Beides wird von der aktuellen Literatur zum GBT kaum berücksichtigt. Der CFG erlaubt zwar präzise, auf der Graphentheorie basierende Definitionen. Programme gängiger Programmiersprachen können aber nicht angemessen in den CFG abgebildet werden. Auch Ausnahmebehandlung oder die GBT-relevanten Ausdrücke lassen sich im CFG nicht zufriedenstellend abbilden.

Für das Verständnis einer im GBT ermittelten Überdeckung und der daraus resultierenden Schlussfolgerungen ist das zugrunde liegende Modell von großer Bedeutung. Wenn z. B. von einem Abnahmetest eine 80-prozentige Anweisungsüberdeckung gefordert wird, ist es bedeutsam, ob if-Anweisungen oder Schleifen selbst als Anweisung gewertet werden und damit zur Anweisungsüberdeckung beitragen oder nicht. Ein standardisiertes Referenzmodell des GBT, das diese Details umfassend definiert und auch strukturierte Programmiersprachen unterstützt, gibt es heute nicht. Dadurch sind viele, selbst grundlegende Überdeckungsmetriken bis heute für gängige Programmiersprachen nicht einheitlich definiert. Als logische Konsequenz folgen die Werkzeuge des GBT keinem gemeinsamen Standard und zeigen für die gleiche Programmausführung deutlich verschiedene Überdeckungswerte an. Zwar lassen sich diese Unterschiede durch die z. T. verschiedenen Techniken der GBT-Werkzeuge erklären, aber es gibt auch keine „Referenz“, an der sich die Werkzeughersteller orientieren könnten.

1.2 Zielsetzung und Lösungsansatz

Es ist das Ziel dieser Arbeit ein solches Referenzmodell zu liefern. Dieses Ziel lässt sich wie folgt formulieren:

> Ziel ist die Entwicklung eines programmiersprachenneutralen Modells, das eine einfache und präzise Abbildung der gängigen Programmiersprachen ermöglicht. Das Modell soll die weitgehend standardisierten Kontrollstrukturen wie z. B. Entscheidung oder Schleife enthalten. Ebenso soll die Ausnahmebehandlung berücksichtigt werden, sowie die Ausdrücke, die Auswirkung auf den GBT haben. Sowohl die populären kontrollflussbasierten Metriken, als auch Überdeckungsmetriken für logische und bedingte Ausdrücke sollen auf Grundlage des GBT-Modells definiert werden können.

Dieses Modell wird in zwei Schritten entwickelt: Erstens durch den Entwurf einer primitiven Sprache RPR (Reduced Program Representation), die die GBT-relevanten Aspekte der realen Programmiersprachen abstrahiert. Und zweitens der Definition der Ausführungssemantik mit Petri-Netzen, den sogenannten Modellnetzen. Die Modellnetze sollen auch für GBT-Werkzeugimplementierungen eine präzise Spezifikation liefern, die beschreibt, wie die zur Metrikenberechnung wichtigen Ausführungszähler in das Original-

Programm eingewoben werden sollen. Auf der Grundlage der Modellnetze erfolgt dann die präzise Definition der populären GBT-Metriken sowie weiterer neuer Metriken.

1.3 Übersicht und Gliederung

In Kapitel 2 werden zunächst die wesentlichen Grundbegriffe auf dem Gebiet des Tests definiert. Ausgehend von den Grundbegriffen wird der Test im Kontext des typischen Projektverlaufs für die verschiedenen Teststufen und der schrittweise steigenden Komplexität des Prüflings beschrieben. Hierbei wird insbesondere der Aspekt des GBT betrachtet, der in den verschiedenen Teststufen unterschiedlichen Rahmenbedingungen unterliegt und auch unterschiedliche Ziele verfolgt. Da sich die vorliegende Arbeit im Kern mit einem neuen Modell des GBT beschäftigt, folgt eine Einführung in die Modelltheorie, und es werden die GBT-Überdeckungsmetriken aus der Literatur zusammengefasst. Da die Relevanz der GBT-Metriken die Wirksamkeit des GBT voraussetzt, wird eine Reihe von Untersuchungen hierzu vorgestellt und zusammengefasst.

Kapitel 3 befasst sich mit der Literatur zu GBT-Modellen, den GBT-Werkzeugen und dem GBT-Einsatz in der Praxis. Zunächst wird eine Zusammenfassung der GBT-Modelle der Literatur geliefert, und es werden die Schwachstellen dieser Modelle herausgearbeitet. Es folgt eine Übersicht über die aktuelle GBT-Werkzeuglandschaft. Dies findet zum einen durch eine Zusammenfassung der aktuellen Literatur hierzu statt, zum anderen werden gängige Werkzeuge anhand eines Referenzprogramms und einer Referenzausführung untersucht. In beiden Fällen zeigt sich, dass die Werkzeuge keinem einheitlichen Standard folgen und durchweg verschiedene Ergebnisse liefern. Ergänzend zu diesen technischen Aspekten folgt eine Zusammenfassung der Literatur zum praktischen Projekteinsatz von GBT-Werkzeugen. Abschließend wird eine Übersicht über die Techniken zur GBT-Protokollierung geliefert, die eine zentrale Funktion innerhalb eines GBT-Werkzeugs bildet und über Möglichkeiten des GBT-Werkzeugs zur Metrikenerhebung entscheidet.

In den Kapiteln 4 und 5 wird ein neues GBT-Referenzmodell entwickelt, das deutliche Vorteile gegenüber den in Kapitel 3 beschriebenen GBT-Modellen der Literatur hat. Zunächst werden die Anforderungen an ein solches Modell zusammengefasst. Hierzu werden im Wesentlichen zwei Quellen genutzt: Die Überdeckungsmetriken, die auf Grundlage des Modells definiert werden sollen, sowie Richtlinien zur Zertifizierung sicherheitskritischer Software. Das GBT-Modell dieser Arbeit wird dann aus zwei Perspektiven entwickelt: Zum einen wird in Kapitel 4 eine Modellsprache, die Reduced Program Representation (RPR), definiert, die entsprechend den aufgestellten Anforderungen die GBT-relevanten Aspekte der realen Sprachen abbildet, die irrelevanten dagegen präteriert. Als zweite Perspektive wird in Kapitel 5 ein Ablaufmodell definiert, das präzise beschreibt, wie die genaue Ausführungssemantik eines GBT-Modellprogramms zu verstehen ist. Die Beschreibung einer Übertragung von Programmen der Programmiersprache Java in das GBT-Modell schließt dieses Kapitel ab.

Auf Grundlage des GBT-Referenzmodells werden dann in Kapitel 6 die populären GBT-Überdeckungsmetriken sowie weitere neu entwickelte Überdeckungsmetriken präzise definiert. Hierzu wird zunächst eine allgemeine Definition für Überdeckungsmetrik

und Überdeckung geliefert. Danach wird ein Regelwerk aufgestellt, mit dem Überdeckungsmetriken auf Plausibilität, Differenziertheit und Vergleichbarkeit geprüft werden können. Anschließend werden die verschiedenen Überdeckungsmetriken definiert und anhand des Regelwerks geprüft.

In Kapitel 7 werden prinzipielle Anforderungen an ein GBT-Werkzeug, die sich aus den Kapiteln 4, 5 und 6 ergeben, zusammengefasst. Anschließend wird das GBT-Werkzeug CodeCover vorgestellt, das in weiten Teilen eine Implementierung des Referenzmodells dieser Arbeit bildet. Es erfolgt auch ein Abgleich, in wie weit CodeCover die aufgestellten Anforderungen an ein GBT-Werkzeug erfüllt.

Ein neues, werkzeuggestütztes Verfahren zum Testfallentwurf wird in Kapitel 8 vorgestellt. Das Verfahren basiert auf dem in Kapitel 4 und 5 vorgestellten Modell und nutzt Erkenntnisse zur Fehlerprognose aus der Literatur. Als Resultat des Verfahrens werden dem Tester sogenannte Testfall-Hinweise offeriert, aus denen er systematisch neue und fehlersensitive Testfälle entwickeln kann. Einen wichtigen Aspekt dieses Verfahrens bilden auch Priorisierungen dieser vorgeschlagenen Testfall-Hinweise. Abschließend wird die Umsetzung der werkzeuggestützten Teile des Verfahrens im Werkzeug CodeCover beschrieben.

Abschließend werden in Kapitel 9 die Ergebnisse dieser Arbeit zusammengefasst, und es wird ein Ausblick auf weitere aufbauende Arbeiten geliefert.

2
Grundlagen und Begriffe

In diesem Kapitel werden die für diese Arbeit wichtigen Begriffe definiert. Beginnend mit den Grundbegriffen zur Qualitätssicherung und zum Test werden die Teststufen mit ihrem Bezug zum Glass-Box-Test beschrieben. Es folgen Grundlagen zu Modellen und den Überdeckungsmetriken

2.1 Grundbegriffe

2.1.1 Software-Qualitätssicherung

Nach [Li02] stellt die Software-Qualitätssicherung Techniken zur Erreichung gewünschter Ausprägungsgrade der Qualitätsmerkmale von Software-Systemen zur Verfügung, wobei die Qualitätsmerkmale sich nach [ISO 9126] in Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit sowie Übertragbarkeit untergliedern lassen. Nach [LL10] hat die Software-Qualitätssicherung (QS) die Aufgabe, alle qualitätsrelevanten Aktivitäten und Prozesse zu gestalten, zu organisieren, abzustimmen und zu überwachen.

Def. **quality assurance**. *(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured. Contrast with: quality control (1). [IEEE610]*

In [LL10] werden neben den Prüfungen auch organisatorische und konstruktive Maßnahmen zur Software-Qualitätssicherung gerechnet. Diese organisatorischen und konstruktiven Maßnahmen finden im Gegensatz zu den Prüfungen präventiv statt und zielen auf allgemeine Prozessverbesserung und nicht auf die Prüfung eines einzelnen Artefakts.

Organisatorische Maßnahmen zur Qualitätssicherung sind beispielweise die Regelung von Zuständigkeiten, die Festlegung von Richtlinien, die Einführung von Standards, die Bereitstellung von Checklisten sowie die Qualifizierung der Mitarbeiter. Konstruktive Maßnahmen zielen nach [LL10] darauf ab, Probleme zu vermeiden. Der Einsatz geeigneter Methoden, Sprachen und Werkzeuge sowie die Schulung von Mitarbeitern und die Verwendung bestimmter Prozessmodelle werden als konstruktive Maßnahmen genannt. Die Einführung von Prozessreifegradmodellen wie z. B. [CMM, CMMI, ISO15504] ist in

aller Regel von organisatorischen und konstruktiven Maßnahmen zur QS geprägt. Bei der analytischen QS wird der Prüfling (Teil-, Zwischen- oder Endprodukt) auf Fehler hin untersucht, die Qualität des Prüfgegenstands wird bewertet oder es wird geprüft, ob ein Prüfgegenstand bestimmte vorgegebene Qualitätskriterien erfüllt. Die Prüfungen gliedern sich in die nicht automatisierbaren Prüfungen wie beispielweise Inspektionen oder Reviews und in automatisierbare Prüfungen am Rechner. Dabei haben die nichtmechanischen Prüfungen, die durch Menschen vorgenommen werden, trotz der Aufwände erhebliche Vorteile. So können diese Prüfungen auch für nicht formale Dokumente wie z. B. Anforderungsspezifikationen erfolgen. Hampp liefert in [Ham10] einen umfangreichen Überblick über diese nichtautomatisierbaren Prüfungen.

Abbildung 1: Übersicht über die Prüfverfahren nach [LL10]

Statische Prüfungen am Rechner werden in der Regel nur zur Prüfung formaler Dokumente eingesetzt. Bei Programmcode sind Architekturanalysen sowie Konformitätsanalysen hinsichtlich vorgegebener Programmierrichtlinien möglich. Ein in der Industrie verbreiteter Vertreter dieser Programmierrichtlinien ist der sogenannte MISRA-Standard [MISRA], der bei der Softwareentwicklung im Automobilbereich eine zentrale Rolle spielt. Die statische Prüfung kann Verstöße gegen diese Richtlinie anzeigen. Zudem kann ein Programm auf sogenannte Anomalien (wie z. B. Datenflussanomalien, [SL04]) hin untersucht werden. Mit der statischen Analyse können auch Code-Metriken erhoben und mit vorgegebenen Grenzwerten verglichen werden. Alle diese statischen Prüfungen haben gemeinsam, dass der Prüfgegenstand gelesen, aber nicht ausgeführt wird. Die Prüfungen, in denen der Prüfling ausgeführt wird, werden als dynamische Prüfung oder als Test bezeichnet.

2.1.2 Testen

Nach [My79] und [LL10] ist Testen die – auch mehrfache – Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden. Ergänzend ist nach [LL10] ein systematischer Test ein Test, bei dem die Randbedingungen definiert und präzise erfasst sind, die Eingabedaten systematisch ausgewählt werden und die Ergebnisse dokumentiert und nach Kriterien beurteilt werden, die vor dem Test festgelegt wurden.

Def. **test**. *An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component, [IEEE610].*

Grundsätzlich ist Testen immer eine Stichprobenprüfung, da der vollständige verifizierende Test bei Programmen der Industrie völlig ausgeschlossen ist. Bereits bei kleinen Programmen mit wenigen Integer-Variablen wären für einen vollständigen Test so viele Wertekombinationen zu testen, dass er eine astronomische Anzahl an Testfällen und damit eine nicht praktikable Testdauer erfordern würde. Damit müssen in der Praxis einige wenige Testfälle – ein verschwindend geringer Teil der theoretisch möglichen Testfälle – geschickt gewählt werden, um dennoch möglichst viele Fehler des Programms aufzudecken. Dijkstra hat dies in den berühmten Satz zusammengefasst:

Program testing can be used to show the presence of bugs, but never show their absence!
E.W. Dijkstra (1970)

So wird „to show the presence of bugs“ das zentrale Thema beim Testen, das in dem Satz von Myers zusammengefasst wird:

Testing is the process of executing a program or system with the intent of finding errors.
G. Myers (1979)

Misslingt nun dieses „Fehler entdecken“ trotz großer Anstrengungen, begründet sich nach Hetzel das Vertrauen darin, dass der Prüfling das leistet, was er leisten soll. So definiert Hetzel Testen als

Testing is the process of establishing confidence that a program or system does what it is supposed to.
B. Hetzel (1973)

Und Grimm stellt in [Gr95] dazu fest, dass Testen in der Praxis das einzige Verfahren ist, mit dem die realen Einsatzbedingungen eines Software-Systems angemessen berücksichtigt und die dynamischen Eigenschaften geprüft werden können. So liegt Testen in einem Spannungsfeld, weil einerseits durch den Test kein Nachweis der Korrektheit des Prüflings möglich ist, andererseits aber keine andere Prüftechnik in der Lage ist, das Testen auch nur annähernd zu ersetzen.

2.1.3 Black-Box-Test

Beim sogenannten Black-Box-Test (BBT) betrachtet man nach [LL10] das Programm als Monolithen, über dessen innere Beschaffenheit man nichts weiß und nichts wissen muss. Man prüft, ob das Programm das tut, was die Spezifikation verlangt. In der Literatur wird übereinstimmend festgestellt, dass der Black-Box-Test die wichtigste Form des Tests ist. Auch sind für den Black-Box-Test in der Literatur viele Techniken zum Testfallentwurf und zur Testfalldokumentation beschrieben. Eine Zusammenfassung hierzu folgt in Kapitel 2.6.1. Neben dem Begriff *Black-Box-Test* werden in der Literatur auch die Begriffe *Funktionstest* (functional testing) [IEEE610] oder *Spezifikationsbasierter Test* [ISTQB] verwendet.

Def. **functional testing**. *(1) Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. Syn: black box testing. Contrast with: structural testing. (2) Testing conducted to evaluate the compliance of a system or component with specified functional requirements. See also: performance testing. [IEEE610]*

2.1.4 Glass-Box-Test

Nach [LL10] lässt das Wort „Glass Box“ erkennen, dass der Programmcode bei dieser Art des Tests sichtbar ist. Als Glass-Box-Test (GBT) wird demnach ein Test bezeichnet, wenn beim Testen zusätzlich beobachtet wird, wie weit das Programm ausgeführt wird. Diesen Anteil des ausgeführten Programms, bezogen auf das gesamte Programm, bezeichnet man dabei als GBT-Überdeckung. Nach [LL10] geht es in jedem Glass-Box-Test darum, eine bestimmte auf den Programmcode bezogene Überdeckung zu erreichen. Das Kalkül des GBT ist einfach: Ein Defekt, der sich in einer Anweisung oder in einem Ausdruck befindet, kann nur gefunden werden, wenn die Anweisung oder der Ausdruck ausgeführt wird. Die einzige Chance, einen Defekt beim Testen zu finden, besteht somit darin, Eingabedaten zu wählen, die dazu führen, dass die Anweisung oder der Ausdruck ausgeführt wird. Damit führt eine höhere Überdeckung statistisch zur Entdeckung von mehr Defekten, und, wenn diese behoben sind, zu besserer Produktqualität. Andererseits gibt es natürlich (von sehr wenigen Defekttypen abgesehen) keinerlei Sicherheit, dass mit Ausführen einer Anweisung oder eines Ausdrucks der Fehler tatsächlich angezeigt, d. h. eine Abweichung zum Soll-Resultat erkannt wird. Ein Beispiel hierfür ist ein Ausdruck, der bei bestimmten Eingaben zu einer Division durch null (d. h. zu einem Fehlverhalten) führt. Solange genau diese Eingaben nicht gewählt werden, wird der Prüfling die Fehlersymptome des Defekts nicht zeigen, und es wird kein Fehlverhalten vorliegen. Ntafos fasst dies in [Na88] wie folgt zusammen:

Another problem is that most structural strategies do not provide any guidelines for selecting test data from within a path domain, and many errors along a path can be detected only if the path is executed with values from a small subset of its domain.

Gleichwohl ist sich die Literatur einig, dass der GBT eine wirksame und wirtschaftliche Ergänzung zum gründlichen Funktionstest liefert.

Def. **structural testing.** *Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing. Syn: glassbox testing; white-box testing. Contrast with: functional testing. [IEEE610]*

Für den Glass-Box-Test ist in jedem Fall ein Werkzeug erforderlich, das während der Testausführung die Ausführung der einzelnen Programmelemente protokolliert und daraus die Überdeckung berechnet. In der Regel werden in den Prüfling zusätzliche Anweisungen als Messpunkte eingewoben, die die Ausführungen einzelner Elemente des Programms zählen. In Abschnitt 3.3 wird dies ausführlich behandelt. Die Darstellung der GBT-Resultate kann für den Tester schließlich in anschaulicher Form erfolgen: So werden beispielsweise die Teile des Programms, die nicht ausgeführt sind, in einer speziellen Farbe hervorgehoben.

Beim Glass-Box-Test lassen sich zwei prinzipielle Verwendungs-Strategien abgrenzen: Erstens die Verwendung der Überdeckung als Testendekriterim. Nach [LL10] kann beispielsweise das Testziel sein, eine Anweisungsüberdeckung von 80 % zu erreichen. Dann wir getestet, bis 80 % der ausführbaren Anweisungen im Programm mindestens einmal ausgeführt sind. Auch die Vorgaben wichtiger Industriestandards zur Zertifizierung sicherheitskritischer Software sind als Testendekriterium zu verstehen (z. B. [RTCA92, IEC61508]). Sie verlangen in der Regel den Nachweis einer vollständigen (oder sehr hohen) Überdeckung.

Zweitens kann der GBT als Testfallentwurfsstrategie verwendet werden. So wird beispielsweise im Buch von Myers [My79] der GBT auch im Kapitel „Testfallentwurf“ behandelt. Aus Programmverzweigungen und logischen Ausdrücken leitet er Eingabedaten für Testfälle ab, die zu einer vollständigen GBT-Überdeckung führen. Eine ausführliche Beschreibung des GBT als Testfallentwurfsstrategie folgt in Abschnitt 2.6.2 auf Seite 29.

2.1.5 Fehler

Der Begriff „Fehler“ wird im allgemeinen Sprachgebrauch (aber auch in der Softwaretechnik) für prinzipiell verschiedene Sachverhalte genutzt. Die folgende Definition benennt diese verschiedenen Bedeutungen:

Def. **error**. *(1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result. (2) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program. (3) An incorrect result. For example, a computed result of 12 when the correct result is 10. (4) A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator. Note: While all four definitions are commonly used, one distinction assigns definition 1 to the word “error,” definition 2 to the word “fault,” definition 3 to the word “failure,” and definition 4 to the word “mistake.” See also: dynamic error; fatal error; indigenous error; semantic error; syntactic error; static error; transient error. [IEEE610]*

Oft ist aus dem Kontext heraus klar, welche Bedeutung das Wort „Fehler“ jeweils haben soll. Um Missverständnisse zu vermeiden, ist es ratsam, konsistent die eindeutigen Begriffe gemäß der Definition zu verwenden. Die deutschen Begriffe nach [ISTQB] sind in Tabelle 1 angegeben. Die drei Verwendungsmöglichkeiten des Begriffs Fehler lassen sich damit auch übersichtlich auf die drei Bedeutungen zusammenfassen:

- eine Person macht einen Fehler (Definition (4), mistake),
- das Programm enthält einen Fehler (Definition (2), fault) und
- die Programmausführung zeigt einen Fehler (Definition (3), failure)

Die drei Bedeutungen von „Fehler“ nach Tabelle 1 erfordern jeweils völlig unterschiedliche Maßnahmen um den „Fehler“ zu erkennen bzw. zu vermeiden. Organisatorische und konstruktive QS-Maßnahmen zielen auf die Vermeidung von Fehlhandlungen, die nichtmechanischen Prüfungen zielen auf das Erkennen der Defekte, und der Test zielt auf die Erkennung und Vermeidung des Programmfehlverhaltens.

Begriff	Definition
Mistake (**Fehlhandlung**, Versagen, Irrtum) Definition (4) nach [IEEE610]	**Eine Person macht einen Fehler**: Fehlerhafte Aktion einer Person (Irrtum), die zu einer fehlerhaften Programmstelle führt.
Fault, Defect, Bug (**Defekt**, Fehlerursache) Definition (2) nach [IEEE610]	**Der Programmcode enthält einen Fehler:** Fehlerhafte Stelle (z. B. eine Zeile) eines Programms, die eine Fehlwirkung auslösen kann.
Failure (Fehlerwirkung, Mangel) Definition (3) nach [IEEE610]	**Die Programmausführung zeigt einen Fehler:** Fehlverhalten eines Programms gegenüber der Spezifikation, das während seiner Ausführung (tatsächlich) auftritt.

Tabelle 1: Verwendungsbereiche des Worts „Fehler“

2.1.6 Testfall

Testfälle spezifizieren für einen Prüfling Eingabedaten mit den zugehörigen erwarteten Ausgaben – den Soll-Resultaten. Die Eingabedaten steuern die Programmausführung, und ein Fehler (oder eine Fehlerwirkung) wird dann angezeigt, wenn die tatsächlichen Ausgaben – die Ist-Resultate – von den spezifizierten Soll-Resultaten abweichen. Nach [IEEE610] umfasst ein Testfall die Eingabedaten, die für die Ausführung notwendigen Vorbedingungen, die Soll-Resultate sowie weitere erwartete Nachbedingungen. Während zur Auswahl fehlersensitiver Eingabedaten Kreativität und Intuition erforderlich ist, müssen die Soll-Resultate für die ausgewählten Eingabedaten sorgfältig aus der Anforderungsspezifikation abgeleitet werden.

Def. **test case.** *A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE610]*

Def. ***test suite.*** *A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one. [ISTQB]*

Def. Die **Testausführung** *ist die Ausführung aller Testfälle der Testsuite eines Programms.*

Def. Die **Testfallausführung** *ist die Ausführung eines einzelnen Testfalls.*

Nach [LL10] ist ein Testfall gut, wenn er mit hoher Wahrscheinlichkeit einen noch nicht entdeckten Fehler anzeigt. Ein idealer Testfall ist demnach

- repräsentativ, d. h., er steht stellvertretend für viele mögliche Testfälle,
- fehlersensitiv, d. h., er hat nach der Fehlertheorie eine hohe Wahrscheinlichkeit, einen Fehler anzuzeigen,
- redundanzarm, d. h., er prüft nicht, was auch andere Testfälle schon prüfen.

Ausführliche Empfehlungen zur Struktur, in der die Testfälle beschrieben werden, sind in [IEEE829] enthalten. Dort wird auch die Zusammenfassung der Testfälle in eine Testsuite empfohlen.

2.2 Testprozess

2.2.1 Das V-Modell

Viele einschlägige Industriestandards (wie z. B. [ISO15504, IEC61508, VMXT04]) beschreiben den Test im Rahmen eines V-förmig angelegten Entwicklungsmodells, das auf Arbeiten von Boehm [Boe79] zurückgeht. Obwohl sich die Modelle alle geringfügig unterscheiden, sind sie in der Grundstruktur gleich: Die linke Flanke des „V“ enthält die verfeinernden Aktivitäten wie Anforderungsanalyse, Entwurf und Implementierung.

Die rechte Flanke des „V“ enthält die integrierenden Aktivitäten mit den sogenannten Teststufen. Je Teststufe werden die im korrespondierenden verfeinernden Prozessabschnitt erstellten Artefakte gegen die dem Artefakt zugrundeliegende Spezifikation validiert. Damit wird z. B. im Komponententest die implementierte Komponente gegen die der Komponente zugrundeliegende Spezifikation geprüft. Entsprechendes gilt für den Integrationstest, bis schließlich beim System- und Abnahmetest das System gegen die Anforderungsspezifikation des Systems geprüft wird. Neben den genannten V-Modellen sind zahlreiche Varianten in der Literatur und auch als Unternehmensstandards zu finden. In der Test-Literatur (z. B. [SL04]) spielt das V-Modell nach [ISTQB] eine große Rolle. Dieses Modell ist in Abbildung 2 dargestellt. Die gestrichelt dargestellten Pfeile entsprechen Prüfungen, ob die Ergebnisse eines Entwicklungsschritts die Vorgaben der Eingangsdokumente erfüllen. So wird z. B. nach Fertigstellung der Anforderungsdefinition diese daraufhin geprüft, ob sie die vorab festgelegten Vorgaben erfüllt. Diese Prüfungen

werden auch als Verifikation bezeichnet. Die Verifikation kann bei formalen Dokumenten mit formalen Eingangsdokumenten als formale Verifikation und damit als Korrektheitsnachweis stattfinden. Testen validiert für spezielle Datenpunkte den Prüfling gegen die Spezifikation, kann aber praktisch nie einen Korrektheitsnachweis erbringen.

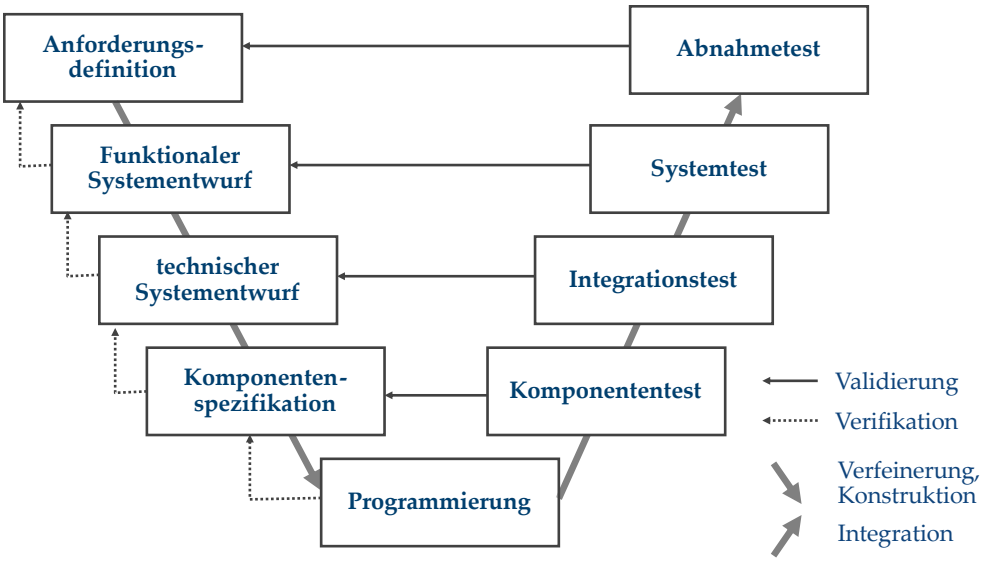


Abbildung 2: V-Modell nach ISTQB [ISTQB, SL04]

Def. **verification**: *(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness. („Are we doing the thing right?") [IEEE610]*

Def. **validation**: *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. („Are we doing the right thing?") [IEEE610]*

2.2.2 Testaktivitäten

Der Testprozess ist nicht eine große gekapselte Aktivität, sondern besteht aus vielen Einzelaktivitäten. Die Aktivitäten im Programmtest bestehen nach [LL10, FLS07, IEEE829, Gri95, SL04] im Wesentlichen aus den nacheinander stattfindenden Teilaktivitäten Testvorbereitung (Testspezifikation), Testdurchführung und Testauswertung. Abbildung 3 zeigt diese Aktivitäten und ihr Zusammenwirken.

Def. **test process**. *The fundamental test process comprises planning, specification, execution, recording, checking for completion and test closure activities. [ISTQB]*

Begleitet werden die Testaktivitäten von einem Testmanagement (oder nach [ISTQB] der Testplanung), das Ziele, Budget, Zeitrahmen und Organisationsstruktur festlegt. Detail-

liert sind diese Management-Aktivitäten in [IEE829] sowie [SL04] beschrieben. Der Testprozess endet mit der Dokumentation der festgestellten Fehler; die Fehleranalyse und Fehlerbehebung zählen nicht mehr zum Testprozess hinzu. In der Praxis ist es auch üblich, Fehler in einem sogenannten Fehlerverfolgungssystem (Bug-Tracking-System) zu erfassen. Neben der genauen Beschreibung der Fehlersymptome werden die genaue Testkonfiguration sowie der Testfall erfasst, der den Fehler angezeigt hat. [IEEE829] und [SL04] enthalten eine ausführliche Übersicht über die Angaben einer solchen Fehlererfassung.

In [LL10] wird der Testfallentwurf als die eigentliche intellektuelle Leistung des Testens bezeichnet. Als wichtigster und umfangreichster Teil der Vorbereitung wird die Auswahl und Spezifikation der Testfälle genannt. Ohne Frage ist für diese Tätigkeit ein besonderes Maß an Kenntnis der Spezifikation und der Anwendungsdomäne erforderlich. Unterstützend liefert die Literatur eine Vielzahl an Verfahren, um den Testfallentwurf zu systematisieren. In Kapitel 2.6 werden diese Verfahren ausführlich beschrieben.

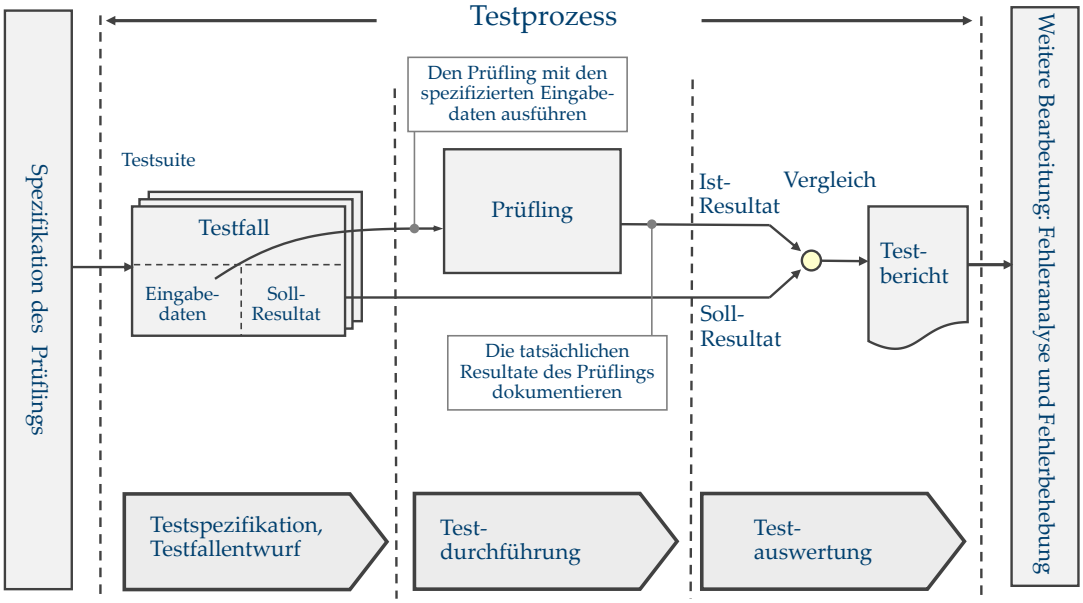


Abbildung 3: Testprozess

2.2.3 Wirtschaftlichkeitsaspekte

Die Wirtschaftlichkeit des Testens wird durch zwei Faktoren bestimmt: Die Kosten, um einen Fehler im Test zu finden (die Testkosten), und die Fehlerkosten, die dadurch entstehen, dass der Fehler unentdeckt bleibt und so lange Schaden anrichtet, bis er in einer späteren Phase, im Betrieb oder gar nicht entdeckt wird. In [LL10] wird daher empfohlen, eine Kostengrenze zu definieren, die das Testende steuert. Überschreiten die durchschnittlichen Testkosten für die Entdeckung eines Fehlers eine vorab festgelegte Kostengrenze, wird der Test beendet. Die Kostengrenze wird dabei nach [SL04] abhängig von der Kritikalität des Projekts und weiteren Faktoren (wie z. B. festgesetzte Vertragsstrafen oder spezielle Fehlerfolgekosten) festgelegt, die die Fehlerkosten beeinflussen.

Amland behandelt in [Am00] ausführlich solche Risikofaktoren und empfiehlt, den Test an diesen Risiken auszurichten. In der Praxis zeigt sich, dass die Testkosten pro entdecktem Fehler über längere Zeit nahezu konstant bleiben und danach deutlich anwachsen. Dementsprechend werden zu Beginn des Tests deutlich mehr Fehler gefunden als in einem späteren Testabschnitt.

Einen ausführlichen Bericht zur Wirtschaftlichkeitsbetrachtung des GBT befindet sich in [EBI06]. Die Autoren geben für ein untersuchtes Projekt auch die Testkosten für Testfälle des logikbasierten GBT mit 8,4 Aufwandstagen pro Fehler an. Die Kostengrenze des Projekts liegt nach Aussage der Autoren deutlich über diesem Wert; das getestete Programm stammt aus dem Bereich der Raumfahrt.

2.3 Klassifikation nach Teststufen

Die Aktivitäten des Tests lassen sich nach ganz unterschiedlichen Merkmalen klassifizieren. [LL10] nennt die folgenden Klassifikationen:

- Klassifikation nach den Grundlagen zur Testfallerstellung: Im Wesentlichen erfolgt hier eine Trennung in den Black-Box- und in den Glass-Box-Test (vgl. Kapitel 2.6). Diese Klassifikation spielt auch in dieser Arbeit eine große Rolle.
- Klassifikation nach dem Aufwand für Vorbereitung und Archivierung: Es erfolgt hier eine Untergliederung in Laufversuch, Wegwerftest und systematischen Test. Da in dieser Arbeit ausschließlich auf den systematischen Test Bezug genommen wird, wird diese Klassifikation nicht weiter behandelt.
- Klassifikation nach der Komplexität des Prüflings. Hier handelt es sich um die am häufigsten in der Literatur verwendete Klassifikation [ISTQB, SL04]. Der Prüfling wird schrittweise integriert und nimmt so eine immer höhere Komplexität an. [LL10] nennt Einzeltest, Modultest, Integrationstest und Systemtest. Diese Klassifikation wird nach [ISTQB] auch als Klassifikation nach Teststufen bezeichnet und wird im Folgenden ausführlich behandelt.
- Klassifikation nach den getesteten Eigenschaften. Im Einzelnen wird Funktionstest, Installationstest, Wiederinbetriebnahmetest, Verfügbarkeitstest, Last- und Stresstest, Regressionstest genannt. Damit wird dem Rechnung getragen, dass jede testbare Anforderung – und damit nicht nur die funktionalen Anforderungen – durch mindestens einen Testfall abgedeckt werden.
- Klassifikation nach beteiligten Rollen: Alpha- und Beta-Test und Abnahmetest.

In der Literatur (z. B. [Ost07]) findet man häufiger noch die Klassifikation nach Entwicklungsphasen. Faktisch wird diese Klassifikation mit der Klassifikation nach Teststufe oder Komplexität des Prüflings in weiten Teilen übereinstimmen. Die Klassifikation nach Teststufen (oder Komplexität des Prüflings) hat, wie im Folgenden noch dargestellt wird, wegen der praktikableren Abgrenzung aber Vorteile.

Durchläuft man den Entwicklungsprozess nach Abbildung 2 in chronologischer Reihenfolge, beginnt der Test nach der Implementierung mit dem Modultest, es folgen der Integrationstest und der Test des Gesamtsystems. Natürlich ist die Testdurchführung immer erst möglich, wenn der Prüfling zur Verfügung steht. So kann beispielsweise der

Test eines Moduls erst dann durchgeführt werden, wenn das Modul von der Entwicklung abgeschlossen und zum Test freigegeben ist. Dagegen können viele andere Arbeiten des Tests wie z. B. die Testplanung, der Testfallentwurf oder die Bereitstellung des Testgeschirrs bereits früher begonnen werden. Der Testfallentwurf kann beispielsweise unmittelbar mit Abschluss der Spezifikation des Prüflings beginnen. In der Literatur wird auch übereinstimmend empfohlen, diese vorbereitenden Arbeiten so früh wie möglich zu beginnen. So bildet der Testfallentwurf auch eine exzellente Prüfung der Spezifikation – sowohl der Anforderungsspezifikation wie auch der Modulspezifikation. Diese vorbereitenden oder nachbereitenden Arbeiten, die zeitlich nicht in der jeweiligen Testphase des Entwicklungsprozess liegen, werden in der sogenannten Teststufe mit den Aktivitäten der Testphase zusammengefasst. Da eine vollständige Darstellung der Teststufen den Rahmen dieser Arbeit bei Weitem sprengen würde, werden im Wesentlichen nur die für den GBT relevanten Eigenschaften der Teststufen im Folgenden behandelt. Eine ausführliche Behandlung der Teststufen befindet sich in [SL04, LL10].

2.3.1 Modultest

Die Begriffe Modul und Modultest werden in der Literatur nicht einheitlich definiert. Liggesmeyer versteht in [Li02] unter einem Modul die kleinste sinnvoll unabhängig testbare Einheit eines Programms. In [LL10] bildet ein Modul einen Verbund mehrerer zusammenhängender Einheiten, die gemeinsam eine Funktion bilden. Ausdrücklich wird der Modultest vom sogenannten Unit-Test abgegrenzt, wobei die Unit im technischen Sinne die kleinste testbare Einheit bildet. Eine Unit ist demnach eine Funktion, ein Unterprogramm oder auch eine Klasse. Ein Modul umfasst nach [LL10] mehrere dieser Units. Der Modultest ist in jedem Fall stark vom Begriff des Unit-Tests beeinflusst, und beide Begriffe werden oft auch als Synonym benutzt. Im IEEE Standard [IEEE610] werden Modul- und Unit-Test zwar nicht als Synonyme genannt, haben aber genau die gleiche Definition. Für diese Arbeit ist der genaue Zuschnitt des Moduls nicht wichtig, wichtig ist lediglich ein für den Test vom Gesamtsystem isolierbarer Betrieb des Moduls. Der Begriff Unit wird in dieser Arbeit im Folgenden als Synonym für Modul genutzt.

Def. **Modul**. *Abgrenzbare Einheit des Gesamtsystems, die isoliert betrieben und getestet werden kann. Synonyme: Unit, Subsystem, Komponente.*

Def. **Modultest**. *Test eines vom restlichen Gesamtsystem isolierten Moduls unter Verwendung von Testgeschirr, das die Eingabedaten liefert und die Ausgabedaten prüft [EBI06]. Synonyme: Unit-Test, Komponententest*

Faktisch ist der Begriff Modul- oder Unit-Test heute mit dem Einsatz der Unit-Test-Werkzeuge wie z. B. JUnit [Be03, JUnit] gleichzusetzen. Unit-Test-Werkzeuge unterstützen den Test einzelner unabhängiger Methoden einer Klasse, die selbst möglichst keine oder sehr wenige Abhängigkeiten von anderen Teilen des Gesamtsystems besitzen. Liggesmeyer bezeichnet den Modultest nach seinem Verständnis auch als „Testen im Kleinen“. Damit impliziert er einige prägnante Merkmale des Modul- oder Unit-Tests: Der

Modultest ist weniger aufwändig als die anderen Teststufen, von geringerer Aussagekraft, was die Qualität des Gesamtprodukts betrifft, und es wird keine spezielle Testorganisation verlangt, weil alle Testaktivitäten von den Entwicklern geleistet werden können. Dennoch erfüllt der Modultest eine wichtige Aufgabe, nämlich das Modul vor der weiteren (aufwändigen) Integration einer möglichst gründlichen Prüfung zu unterziehen. Viele Autoren sehen den Haupteinsatz des GBT beim Modultest (oder sogar die Einsatzmöglichkeiten des GBT auf den Modultest begrenzt [Li02, Ost07]). Die folgenden Gründe werden hierfür genannt:

- Der Modultest ist oft bei Entwicklern angesiedelt, die selbst in der Lage sind, ein GBT-Werkzeug in den Testprozess zu integrieren.
- Bei (kleinen) Modulen ist es vergleichsweise einfach neue Testfälle zu finden, die zu einer Erhöhung der Überdeckung führen.
- Im Modultest ist es einfacher, eine hohe Überdeckung zu erzielen, als in den anderen Teststufen.

Ähnlich argumentiert Tai in [Ta89]. Er nennt die Zweigüberdeckung für den Modultest als vergleichsweise einfach zu erreichen, während eine 90 %-Überdeckung im Systemtest nur sehr schwer zu erreichen ist. Um die Wirksamkeit des Modultests zu verbessern, empfiehlt der Autor daher, im Modultest ergänzend zum Zweigüberdeckungstest weitere schärfere Überdeckungskriterien anzuwenden. Konkret werden Bedingungsüberdeckungen (siehe Kapitel 2.5.3) empfohlen. Auch einschlägige Richtlinien zur Zertifizierung sicherheitskritischer Software (z. B. [IEC 61508]) empfehlen den GBT im Modultest.

2.3.2 Integrationstest

Der Integrationstest hat geradezu ein „Schattendasein" zwischen dem Modul- und dem Systemtest, denn es gibt viele Werkzeuge, viel Literatur und viele Methoden für den Modul- und den Systemtest, während der Integrationstest wenig behandelt wird. Prinzipiell wird mit dem Integrationstest die Modulinteraktion geprüft. Jedes Modul ist dabei im Vorfeld bereits im Modultest getestet worden.

Def. **integration**. *The process of combining software components, hardware components, or both into an overall system. [IEEE610]*

Def. **integration testing**. *Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. See also: component testing; interface testing; system testing; unit testing. [IEEE610]*

Für den Integrationstest sollen solche Testfälle gewählt werden, die gezielt kritische Interaktion zwischen den Modulen testen [SL04]. Die Testfälle sollen dazu speziell fehlersensitiv hinsichtlich der Modulinteraktion sein, es ist nicht das Ziel, Modul-interne Funktionen zu prüfen. Damit ersetzt der Integrationstest den Modultest nicht. Der GBT führt damit beim Integrationstest bei den klassischen Überdeckungsmetriken wie z. B. der Anweisungsüberdeckung in der Regel zu einer geringen Überdeckung. Der Grund liegt im ge-

nannten Testziel des Integrationstests. Es geht primär um das Auffinden von Schnittstellenfehlern und weniger um ein vollständiges Testen der Modulfunktionen. D. h. die klassischen Überdeckungsmetriken sind für den Integrationstest wenig aussagekräftig. Spillner empfiehlt in [Sp95] daher ergänzende Überdeckungskriterien für den Integrationstest:

- Jede exportierte Operation wird mindestens einmal ausgeführt
- Jede exportierte Operation wird mindestens einmal von jedem importierenden Modul aufgerufen
- Jede Aufruf-Anweisung einer importierten Operation wird mindestens einmal ausgeführt
- Jede Aufrufabfolge importierter Operationen wird mindestens einmal ausgeführt

Bei den meisten gängigen Programmiersprachen werden allerdings keine Operationen explizit importiert oder exportiert, sondern der Import erfolgt über die komplette Schnittstelle einer Klasse, und der Export wird über die Sichtbarkeit einer Methode deklariert. Der Tester kann aber in einem separaten Integrationsdokument die geforderten Import- und Export-Interaktionen definieren und zur Vollständigkeitskontrolle die Resultate des GBT gegenüberstellen. GBT-Werkzeuge, die speziell diesen Integrationstest unterstützen, sind aber nicht am Markt erhältlich.

2.3.3 Systemtest

Der Systemtest bildet die letzte Stufe des Integrationstests und testet damit das Zusammenspiel aller integrierten Module. In der Regel erfolgt der Systemtest in der vorgesehenen Betriebsumgebung des Systems.

Def. **system**. *A collection of components organized to accomplish a specific function or set of functions. [IEEE610]*

Def. **system testing**. *Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. See also: component testing; integration testing; interface testing; unit testing. [IEEE610]*

Der Einsatz des GBT im Systemtest wird von den wenigen Publikationen, die es dazu gibt, übereinstimmend als sehr nützlich bezeichnet. In [POC93, Ki03] wird ausführlich zum Einsatz des GBT im Systemtest berichtet. Eine Auffälligkeit beider Untersuchungen ist, dass Tester sich beim Einschätzen der GBT-Überdeckung, die mit den Systemtest-Testfällen erzielt wird, deutlich täuschen. Piwowarski, Ohba und Caruso beschreiben diesen Effekt in [POC93]: *„When test coverage had not previously been measured, testers tended to overestimate coverage of their test cases. The first time testers measured coverage during function test, they found that the coverage was in the range of 50% to 60%. The testers were surprised at the low percentage of coverage they were getting. They expected a much higher percentage of code coverage. Some testers estimated that their coverage was 90% or higher. "*

Während also beim Modultest deutlich höhere Überdeckungswerte erreicht werden, liegt die GBT-Überdeckung beim (gründlichen) Systemtest bei den von [POC93] genann-

ten 50% - 60%. Auch neuere Untersuchungen [Schm11] kommen zu sehr ähnlichen Ergebnissen. Das bedeutet, dass beim klassischen Systemtest nahezu die Hälfte des Programmcodes nicht ausgeführt wird, und es ist kaum anzunehmen, dass dieser Programmcode ausschließlich durch defensive Programmierung begründet ist. Für den GBT sind allerdings die Bedingungen, um beim Systemtest neue Testfälle zu entwickeln, ungleich schwieriger als beim Modultest: Die Menge von Programmcode ist viel größer als beim Modultest, und die Tester kennen den Programmcode nicht oder nur sehr wenig. Dementsprechend gibt es bislang kaum Ansätze, die Tester unter diesen besonderen Bedingungen des Systemtests bei der Entwicklung neuer Testfälle zu unterstützen. Liggesmeyer schreibt in [Li02] zum Anwendungsbereich des GBT: *Der wichtigste Anwendungsbereich der kontrollflussorientierten Testtechniken ist der Modultest. Kontrollflussorientierte Testtechniken können im Integrationstest noch eine gewisse Bedeutung besitzen, während sie im Systemtest nicht eingesetzt werden.* Eine zentrale These dieser Arbeit ist, dass bei geeigneter Werkzeugunterstützung der GBT im Systemtest sinnvoll eingesetzt werden kann (und, wie Piwowarski et al. in [POC93] berichten, bereits eingesetzt wurde). Das in Kapitel 8 vorgestellte GBT-basierte Verfahren zum Testfallentwurf zielt sogar speziell auf den Systemtest.

2.3.4 Abnahmetest

Der Abnahmetest ist in der Regel dem Systemtest sehr ähnlich, hat aber noch zusätzlich eine juristische Bedeutung und prüft auf vertragliche Konformität. Der GBT kann im Abnahmetest zur Überwachung einer Mindestüberdeckung verwendet werden. Auf diese Weise kann die Testsuite, die als Abnahmekriterium verwendet wird, auf eine Mindestgüte geprüft werden. So könnte ein Auftraggeber für den Abnahmetest eine Anweisungsüberdeckung von beispielsweise 70 % verlangen. Gerade wenn der Abnahmetest nicht vom Auftraggeber der Software selbst, sondern vom Lieferanten oder einem Drittdienstleister durchgeführt wird, ist eine solche Forderung sinnvoll. Auch Ramsey und Basili empfehlen in [RB85] den GBT beim Abnahmetest als Kontrollinstrument: *„Structural coverage metrics can be used to aid in the management of the acceptance test process"*.

Mit Blick auf den formalen Aspekt einer solchen Forderung sollte aber zudem das GBT-Werkzeug, mit dem die Messung erfolgt, vorab bestimmt werden. Wie in Kapitel 3.2 gezeigt wird, unterscheiden sich die Überdeckungswerte der verschiedenen Werkzeuge z. T. deutlich.

2.4 Modellbildung beim Glass-Box-Test

In dieser Arbeit spielt das GBT-Modell des Programms und der Programmausführung eine zentrale Rolle. Praktisch alle Definitionen und Schlussfolgerungen werden auf Grundlage des GBT-Modells vorgenommen; der vollständige (reale) Programmcode, aus dem das GBT-Modell gewonnen wird, und die reale Programmausführung werden dagegen selten unmittelbar betrachtet. Dies hat zwar den Nachteil, dass die Definitionen und Schlussfolgerungen, die auf Grundlage des Modells gewonnen werden, zur praktischen Verwendung erst auf die realen Programmiersprachen übertragen werden müssen. Ande-

rerseits wird das Modell so gehalten, dass diese Übertragung in die gängigen Programmiersprachen mit geringem Aufwand und geringem Verlust an Genauigkeit möglich ist. Der Vorteil des programmiersprachenneutralen GBT-Modells ist, dass die Resultate auf viele Programmiersprachen übertragen werden können und die Komplexität (man könnte auch sagen der „Schmutz“) der realen Programmiersprachen im Modell ausgeblendet werden kann.

2.4.1 Modelltheorie

Nach [LL10] sind Modelle entweder Abbilder von etwas oder Vorbilder für etwas – sogenannte deskriptive (beschreibende) oder präskriptive (vorschreibende) Modelle. Nach Stachowiak [Sta73, LL10] besitzt ein Modell drei Modellmerkmale: Das Abbildungsmerkmal, das Verkürzungsmerkmal und das pragmatische Merkmal

Def. **Abbildungsmerkmal**. *Zum Modell gibt es ein Original.*

Def. **Verkürzungsmerkmal**. *Ein Modell erfasst nicht alle Attribute des Originals, sondern nur einen Ausschnitt.*

Def. **pragmatische Merkmal**. *Modelle können unter bestimmten Bedingungen und bezüglich bestimmter Fragestellungen das Original ersetzen.*

Mit dem Ziel, das Modell so knapp wie möglich zu halten, werden auch möglichst nur die Eigenschaften aus dem Original in das Modell übertragen, die für den Modellzweck unbedingt erforderlich sind. Alle anderen Eigenschaften des Originals werden bei der Modellbildung weggelassen (verkürzt). Das Modell weist dagegen zusätzlich zum Original die sogenannten abundanten Attribute auf, die nichts mit dem Original zu tun haben. Einer Modellbildung liegt dabei immer eine Absicht zugrunde: Das Modell soll an Stelle des Originals genutzt werden, weil durch die Verkürzungen die Handhabung oder das Verständnis einfacher oder erst möglich wird. Trotz dieser Verkürzungen werden aber aus dem Modell Erkenntnisse gewonnen, die auf das Original zurück übertragen werden können.

2.4.2 Abbildungsmerkmale des GBT-Modells

Die angestrebten GBT-Auswertungen – und damit in aller Regel die Überdeckungsmetriken – bilden den Zweck der GBT-Modellbildung. Sollen beispielsweise die Überdeckungsmetriken für logische Ausdrücke erhoben werden, müssen diese Ausdrücke in das Modell mit aufgenommen werden. Ist nur die Anweisungs- oder Zweigüberdeckung angestrebt, können diese Ausdrücke auch verkürzt werden. Es ist dann für den Modellzweck ausreichend, nur die kontrollfluss-bestimmenden Merkmale des Programms in das GBT-Modell abzubilden und alle anderen Merkmale zu verkürzen. Wobei auch hier entschieden werden muss, ob z. B. die Ausnahmebehandlung des Programms und der Programmausführung zur angestrebten Auswertung des Modells gehört. Generell verkürzt

werden Entwurfsstrukturen, die nicht ausführbar sind, wie Datentypen, abstrakte Klassen, Interfaces oder Sichtbarkeiten.

Die Verkürzungen des GBT-Modells gegenüber dem realen Programmcode haben in dieser Arbeit aber nicht nur den bereits genannten Zweck der vereinfachten Handhabung, sondern führen auch zu einer Generalisierung. Hierin besteht der besondere Vorteil, dass die Modellbildung nicht nur für eine Programmiersprache, sondern für eine ganze Klasse von Programmiersprachen (die Klasse der strukturierten Programmiersprachen) genutzt werden kann.

2.4.3 Referenzmodell

Nach Schneider [Schn98] ist in der Wirtschaftsinformatik ein Referenzmodell der Versuch, Ergebnisse bei der Modellierung zu verallgemeinern, um das gewonnene Erfahrungswissen als Grundlage weiterer Modellierungsaktivitäten zu dokumentieren. Nach Fettke und Loos [FL04] beschreibt ein Referenzmodell in deskriptiver Sicht die Gemeinsamkeiten einer Klasse von Modellen, und in präskriptiver Sicht liefert ein Referenzmodell einen Vorschlag, wie eine Klasse von Modellen ausgestaltet sein kann.

Das GBT-Referenzmodell dieser Arbeit beschreibt, wie die (Original-)Programme in ein GBT-Modell übertragen werden. D. h. das GBT-Referenzmodell modelliert die Modellbildung und definiert die verwendbaren Modellelemente und deren Kompositionsregeln. Damit ist das GBT-Referenzmodell das Metamodell der GBT-Modelle.

Def. **GBT-Referenzmodell.** *Ein Modell, das die Modellbildung der Programme in das GBT-Modell modelliert und so das Metamodell des GBT-Modells bildet. Das GBT-Referenzmodell liefert die strukturelle und begriffliche Grundlage zur Definition von GBT-Überdeckungsmetriken.*

Der Begriff Referenzmodell zielt in dieser Arbeit auf die aktuelle Werkzeuglandschaft im GBT. Faktisch gibt es heute keine Referenz, an der sich Werkzeughersteller orientieren könnten. Das GBT-Modell dieser Arbeit ist als Referenz geeignet.

2.5 Glass-Box-Test-Überdeckungsmetriken

Eine Überdeckungsmetrik (oft auch nur als „Überdeckung“ bezeichnet) liefert ein Maß für die Vollständigkeit eines Tests, bezogen auf ein bestimmtes Testverfahren oder Testziel. Für die GBT-Überdeckungsmetriken wird dieser Grad auf die Ausführung einzelner Elemente des Programmcodes bezogen.

Def. **metric**. *A quantitative measure of the degree to which a system, component, or process possesses a given attribute. See also: quality metric. [IEEE610]*

Def. **quality metric**. *(1) A quantitative measure of the degree to which an item possesses a given quality attribute. (2) A function whose inputs are software data and whose output is a single*

numerical value that can be interpreted as the degree to which the software possesses a given quality attribute. [IEEE610]

Def. **test coverage.** *The degree to which a given test or set of tests addresses all specified requirements for a given system or component. [IEEE610]*

Def. **Überdeckungsmetrik.** *Maß der Vollständigkeit eines Tests, bezogen auf ein bestimmtes Testverfahren oder Testziel.*

Neben dem Begriff der Überdeckungsmetrik wird in der Literatur auch oft der Begriff Überdeckungskriterium verwendet.

Def. **Überdeckungskriterium.** *Prädikat, das dann für einen Test erfüllt ist, wenn der Test, bezogen auf das Kriterium, vollständig ist, d. h. die Metrik den Wert 1 annimmt. Ansonsten ist das Prädikat nicht erfüllt.*

Die Überdeckungsmetrik gibt den Anteil oder den Grad der Überdeckung an, und das Überdeckungskriterium ist dann erfüllt, wenn die Metrik den Wert 1 (oder 100 %) annimmt. Man sagt auch, eine Testsuite *T* erfüllt für ein Programm *P* ein Überdeckungskriterium C_K. Oder in Prädikatschreibweise nach [Ost07]: es gilt $C_K(P, T)$.

Fenton und Pfleeger definieren in [FP97] die Überdeckungsmetrik wie folgt: Sei T ein Testverfahren, das die Überdeckung von Programmelementen eines bestimmten Typs fordert (die T-Programmelemente). Für ein Programm und eine Testsuite ergibt sich die Überdeckung *covT* (nach [FP97] das Test-Wirksamkeits-Verhältnis) zu

$$covT = \frac{Anzahl\ der\ \ mindestens\ einmal\ ausgeführten\ T-Programmelemente}{Gesamtanzahl\ der\ T-Programmelemente} \in [0;1]$$

Überdeckungsmetriken liegen auf einer Rationalskala und im Bereich von null bis eins (oder von null bis hundert Prozent). Sie bilden auch ein Maß für eine Qualitätseigenschaft der Testsuite und sind so eine sogenannte Pseudometrik[2] für die Testgüte. Eine ausführliche Darstellung des Zusammenhangs von GBT-Überdeckung und Produktqualität folgt in Kapitel 2.7. In der Literatur ist eine Reihe von GBT-Überdeckungsmetriken beschrieben, die sich im Wesentlichen in die drei Klassen einteilen lassen: kontrollflussbasierte, datenflussbasierte und logikbasierte Überdeckungsmetriken.

2.5.1 Kontrollflussbasierte Überdeckungsmetriken

Die kontrollflussbasierten Überdeckungsmetriken sind die ältesten GBT-Überdeckungsmetriken [MM63, Hu75] und intuitiv plausibel. Beispiele sind Anweisungs- oder Zweigüberdeckung [ZHM97, FP97, Li02, AO08]. Die Grundlage bildet hier der Kontrollflussgraph: Nach [ZHM97] entspricht die Anweisungsüberdeckung der Abdeckung der Kno-

[2] Nach [LL10] werden Metriken dann als Pseudometriken klassifiziert, wenn sie aus gemessenen oder geschätzten Werten berechnet werden, weil man sie nicht direkt messen kann.

ten des Kontrollflussgraphen, die Zweigüberdeckung der Abdeckung der Kanten. Die aktuellen GBT-Werkzeuge für Java (Tabelle 5 von Seite 46) nutzen zudem noch Begriffe wie Blocküberdeckung, Zeilenüberdeckung und Instruction-Überdeckung. Dabei ist keinesfalls sicher, dass die Werkzeuge, die von Blocküberdeckung sprechen, auch die gleiche Definition hierfür heranziehen. Wie Zeilen- oder Instruction-Überdeckung zu verstehen ist, ergründet sich auch bei genauerer Analyse der verfügbaren Dokumentation nicht.

In älterer Literatur wird auch im Stile einer Ordinalskala von C0, C1 oder C2-Überdeckung gesprochen, wobei diese Begriffe im Folgenden nicht weiter genutzt und auch von keinem der aktuellen Werkzeuge aufgegriffen werden. Zu den kontrollflussbasierten Überdeckungsmetriken zählt in der Literatur [Rie97, Li02, SL04] auch die Pfadüberdeckung, die dann erfüllt ist, wenn sämtliche Pfade des Kontrollflussgraphen mindestens einmal durchlaufen werden. Die Literatur stimmt darin überein, dass die Pfadüberdeckung nur bei Programmen ohne Schleifen sinnvoll gefordert werden kann. Damit kann die Pfadüberdeckung in der industriellen Praxis in aller Regel nicht genutzt werden. Es gibt allerdings auch vereinzelt Berichte über Programme, die tatsächlich keine Schleifen enthalten. So wird z. B. in [WH11] über einen sinnvollen Einsatz der Pfadüberdeckung berichtet. Dabei ist eine Überdeckungsmetrik, die dann erfüllt ist, wenn bestimmte oder alle Ausführungsabfolgen eines (kleinen und schleifenfreien) Programms ausgeführt werden, durchaus plausibel. Auch hierfür entwickelte Testfälle können eine ausreichende Fehlersensitivität haben [Rie97]. Riedemann weist in [Rie97] darauf hin, dass es bei k aufeinanderfolgenden Verzweigungen schon 2^k verschiedene Pfade durch ein Programm gibt und dass damit die Erhebung einer Pfadüberdeckung auch bei schleifenfreien Programmen aus praktischen Gründen auf Programme mit kleinem k begrenzt bleibt. Andererseits stellt er fest, dass mit Erreichen der Zweigüberdeckung nicht sichergestellt wird, dass alle Ausführungskombinationen von aufeinander folgenden Verzweigungen auch ausgeführt werden. Er definiert daraus die sogenannte Segmentpaarüberdeckung, die dann erfüllt ist, wenn für jedes Paar von Verzweigungen, die im Kontrollflussgraph direkt aufeinander folgen, alle vier Ausführungskombinationen durchlaufen sind. Allerdings hat diese Überdeckungsmetrik keinerlei praktische Bedeutung, und es gibt auch keine Werkzeuge, um diese Metrik zu erheben. Eine praktikable Alternative zur Pfadüberdeckung, die auch Schleifen mit einbeziehen kann, wird in [BL96] als *path profiling* vorgestellt. Die von [Rie97] angegebene Begrenzung auf kleinere Programme gilt aber für das *path profiling* in gleicher Weise.

2.5.2 Datenflussbasierte Überdeckungsmetriken

Die datenflussbasierten Überdeckungsmetriken werden ursprünglich von Rapps und Weyuker in [RW85] beschrieben und von Oster [Ost07] ausführlich in aktualisierter Form dargestellt. Zur Motivation der datenflussbasierten Überdeckungsmetriken argumentieren Rapps und Weyuker in [RW85]: *"Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed."*

Die Autoren klassifizieren dazu die Variablenzugriffe des Programms in Definition (def), Verwendung in einer Berechnung (c-use) und Verwendung in einer Verzweigungsbedingung (p-use). Tabelle 2 fasst diese Klassifikationen zusammen.

Klassifikation	Beschreibung	Beispiel
def	Definition der Variablen, Variablenzuweisung.	a = 7; b = f(x);
c-use	Die Variable wird in einer Berechnung verwendet.	a = b * b; (Die Variable a wird definiert, die Variable b als „c-use“ verwendet)
p-use	Die Variable wird im Bedingungsausdruck einer Verzweigung verwendet.	if(a > 42) { … }

Tabelle 2: Klassifizierung der Variablenzugriffe

Die Knoten des CFG werden dann mit der jeweiligen Klasse des Variablenzugriffs attributiert. Unter einem vollständigen Pfad verstehen die Autoren einen Pfad innerhalb des CFG, der mit dem Startknoten beginnt und mit dem Endknoten endet. Als Pfad bezeichnen sie einen Pfad im CFG zwischen zwei attributierten Knoten. Ein Pfad heißt für einen Variable x definitionsfrei, wenn x auf diesem Pfad nicht definiert wird. P wird als Menge von vollständigen Pfaden definiert, die durch die Eingabedaten der Testsuite durchlaufen werden. Auf dieser Grundlage definieren die Autoren von [RW85] eine ganze Reihe an datenflussbasierten Überdeckungskriterien. Die wichtigsten zeigt Tabelle 3.

all-defs	Das Kriterium ist dann für ein *P* erfüllt, wenn zu jeder Definition einer Variablen *x* in *P* ein vollständiger Pfad enthalten ist, der einen für *x* definitionsfreien Pfad zu mindestens einem c-use- oder p-use-Knoten enthält, bei dem auf *x* zugegriffen wird.
all-c-uses	Das Kriterium ist dann für ein *P* erfüllt, wenn alle für *x* definitionsfreien Pfade von der Definition zu allen erreichbaren c-use-Knoten, bei denen auf *x* zugegriffen wird, in einem vollständigen Pfad von *P* enthalten sind.
all-p-uses	Wie all-c-uses, nur dass die Pfade von der Definition zu einem p-use betrachtet werden.
all-uses	Das Kriterium ist dann für ein *P* erfüllt, wenn alle für *x* definitionsfreien Pfade von jeder Definition von *x* zu jedem erreichbaren Zugriff auf *x* in einem vollständigen Pfad von *P* enthalten sind.

Tabelle 3: Die wichtigsten datenflussbasierten Überdeckungsmetriken

Während die datenflussbasierten Überdeckungsmetriken in der Literatur sehr ausführlich behandelt werden, muss man davon ausgehen, dass sie in der industriellen Praxis kaum eine Rolle spielen. Es gibt zwar vereinzelt Forschungswerkzeuge, die die genannten Datenflussmetriken erheben können, aktuelle und in der Praxis einsetzbare Werkzeuge gibt es aber nicht. In der Untersuchung von Yang et al. in [YLW09] befindet sich unter den 17 untersuchten GBT-Werkzeugen keines, das datenflussbasierte Überdeckungsmetriken erheben kann. Auch hat die Datenflussanalyse ein prinzipielles Problem mit Referenzdatentypen. Während in [RW85] nur Variablenwerte und keine sogenannten Zeiger oder Referenzen betrachtet werden, sind Zeiger und Referenzen ein wichtiger Teil moderner Programmiersprachen. Es gibt zwar Ansätze, das Problem des sogenannten *pointer-aliasing* zu behandeln [Ost07], der Aufwand hierfür ist aber groß, und die ursprüngliche Präzision des Ansatzes geht verloren.

2.5.3 Logikbasierte Überdeckungsmetriken

Bereits Myers stellt in [My79] fest, dass neben den kontrollflussbasierten Überdeckungsmetriken wie z. B. der Anweisungs- oder Zweigüberdeckung auch Überdeckungsmetriken für Bedingungsausdrücke sinnvoll sind. Auf diese Weise können zusammengesetzte Bedingungsausdrücke hinsichtlich der vollständigen Ausführung der Teilterme bewertet werden – es soll so ein gründlicher Test dieser zusammengesetzten Bedingungen unterstützt werden. Begriffe wie Logik-Testen [AOH03], Bedingungsüberdeckungstest [Li02], Prädikat-Testen [Ta96] oder Termüberdeckung [LL10] sind für diesen Test gebräuchlich. Im Folgenden wird hier überwiegend der Begriff Bedingungsüberdeckung verwendet. Die Bedingungsüberdeckung wird von verschiedenen Zertifizierungsrichtlinien gefordert: Z. B. wird in der Richtlinie DO-178B [RTCA92, FAA01] zur Zertifizierung von Software im sicherheitskritischen Bereich der Luftfahrt eine spezielle Ausprägung, die sogenannte Modified Condition/Decision (MC/DC)-Überdeckung gefordert. Zudem werden sie in verschiedenen Sicherheitsstandards wie z. B. [ISO61508] oder im Automotive-Bereich [IEC26262] genutzt. Dementsprechend sind auch aktuelle Werkzeuge für alle wichtigen Programmiersprachen am Markt verfügbar. Die Bedeutung der Bedingungsüberdeckungen hat nach [AOH03] nicht nur damit zu tun, dass die genannten Sicherheitsstandards die Bedingungsüberdeckung fordern, sondern auch die Relevanz von logischen Ausdrücken außerhalb des Programmcodes, z. B. in Zustandsgraphen oder anderen formalen Spezifikationen, trägt dazu bei.

Mit dem Einsatz praktisch jeder Bedingungsüberdeckung wird das Ziel verfolgt, dass jeder (atomare) Einzelterm eines logischen Ausdrucks das Gesamtresultat mindestens einmal bestimmen soll; in Details des konkreten Nachweises unterscheiden sie sich aber z. T. deutlich. Einzelterme, die das Gesamtresultat bestimmen, werden in der Literatur als aktiv [AOH03] oder wirksam [LL10] bezeichnet. Bei [AOH03] wird in der sogenannten „Active Clause Coverage“ von Haupt- und Nebenklauseln (major und minor clause) gesprochen. Die Hauptklausel ist der Einzelterm, dessen Wirksamkeit betrachtet wird, die Nebenklauseln sind die übrigen Einzelterme.

Def. ***Bedingung****. Logischer Ausdruck, der selbst nicht durch eine logische Operation in einen anderen booleschen Ausdruck eingebettet ist. Synonym: Prädikat, Gesamtausdruck*

Def. **Einzelterm***. Ein Einzelterm ist ein Teilausdruck einer Bedingung, der keinen booleschen Operator mehr enthält [FAA01]. Synonym: atomarer logischer Ausdruck, Klausel*

Def. Eine **Termbelegung** *ist die Wertekombination der Einzelterme für eine Ausführung eines Gesamtausdrucks.*

Nach Liggesmeyer [Li02] existieren unterschiedliche Ausprägungen von Bedingungsüberdeckungen, von denen die schwächste – der einfache Bedingungsüberdeckungstest (simple condition coverage test, *Clause Coverage* nach [AOH03]) – bereits dann erreicht ist, wenn jeder atomare Term einmal true und einmal false wird. Wie man leicht sieht, kann dieses Überdeckungskriterium bereits erfüllt werden, ohne dass sich das Gesamtresultat des Ausdrucks mindestens einmal zu true und einmal zu false ergibt. Daraus leiten [AOH03] die kombinatorische Überdeckung des logischen Ausdrucks ab, die dann erfüllt ist, wenn alle möglichen Wertekombinationen der Einzelterme eines Bedingungsausdrucks ausgeführt werden. Bei Ausdrücken mit n Einzeltermen sind damit 2^n Wertekombinationen zu testen – für Ausdrücke mit vielen Einzeltermen eine zu aufwändige Vorgabe. Als einen praktikablen Mittelweg werden in der Literatur MC/DC und die Termüberdeckung vorgeschlagen [Li02, AOH03, LL10]. Diese werden im Folgenden ausführlich beschrieben.

MC/DC [CM94]

Die sogenannte Modified Condition/Decision (MC/DC)-Überdeckung wird in der Zertifizierungsrichtlinie DO-178B [RTCA92, FAA01, FAA07] zur Softwareentwicklung im sicherheitskritischen Bereich der Luftfahrt gefordert. Die Definition von MC/DC nach RTCA, DO-178B, lautet in [RTCA92] wie folgt: *„… and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that decision while holding fixed all other possible conditions."*

MC/DC verlangt damit für die booleschen Ausdrücke, dass jeder einzelne Term das Gesamtresultat mindestens einmal bestimmt. Der Nachweis hierzu hat nach MC/DC für jeden Term so zu erfolgen, dass der boolesche Wert eines Terms (der sogenannte Hauptterm [AOH03]) variiert wird und sich dabei der Wert des Gesamtausdrucks ändert, während alle anderen Terme (die sogenannten Nebenterme) unverändert bleiben. Der Begriff „variieren" suggeriert hierbei, dass der Tester das Programm stoppen, den Hauptterm verändern und den Gesamtausdruck erneut ausführen könnte. Das ist natürlich nicht möglich, sondern der Tester muss geeignete Eingabedaten wählen, um entsprechende Termbelegungen zu erhalten. Die so tatsächlich angewendeten Termbelegungen werden jeweils „beobachtet" und abgespeichert. Der Nachweis der Wirksamkeit eines Hauptterms wird nach [CM94] durch eine Wertetabelle geliefert, welche die im Test aufgetretenen Werte der Einzelterme und den Wert des Gesamtresultats enthält. Werden durch paarweisen Vergleich zwei Wertebelegungen gefunden, die sich im Hauptterm und dem Gesamtresultat unterscheiden, während die Nebenterme unverändert bleiben, ist für die-

sen Hauptterm die Wirksamkeit nachgewiesen. MC/DC fordert diesen Wirksamkeitsnachweis für alle Einzelterme des Bedingungsausdrucks. Damit kann MC/DC erst nach der Testdurchführung geprüft werden, wenn alle tatsächlich angewendeten booleschen Wertekombinationen der Terme vorliegen und paarweise miteinander verglichen werden können.

Termüberdeckung [LL10]

Die Termüberdeckung nach Ludewig [LL10] ist wie MC/DC eine Ausprägung der Bedingungsüberdeckung und ist in der Zielsetzung MC/DC ähnlich. Der besondere Vorteil der Termüberdeckung gegenüber der MC/DC ist, dass auch Abstufungen des Erfüllungsgrads definiert sind; MC/DC definiert dagegen nur ein vollständiges Erfüllen ohne Abstufungen. So kann für die einzelnen booleschen Operationen über eine Wertetabelle (siehe Tabelle 4, Seite 26) bestimmt werden, bei welcher booleschen Operation und Wertebelegung ein Einzelterm das Gesamtresultat bestimmt – also wirksam wird – oder aber unwirksam bleibt. Anders als bei MC/DC ist es so bei der Termüberdeckung möglich, für eine einzelne Wertebelegung des Gesamtausdrucks die true- und false-Wirksamkeit der Terme zu bestimmen. Dies lässt sich an einem einfachen Beispiel leicht veranschaulichen. Gegeben sei der Ausdruck in der Sprache Java A && B. Bei einer Wertebelegung von *A = true* und *B = false* bestimmt nur *B* den Wert des Gesamtausdrucks, denn eine Wertänderung von *A* ist ohne Effekt auf das Gesamtresultat, ein Wechsel auf *B = true* dagegen verändert das Gesamtresultat. In diesem Sinne ist Tabelle 4 zu verstehen: Getrennt für die and- und die or-Operation kann mit den Werten der Operanden deren Wirksamkeit abgelesen werden. Im genannten Beispiel (*A = true* und *B = false*) greift Regel 3 der and-Tabelle von Tabelle 4. Damit ist *B* wirksam und, da *B* den Wert *false* hat, ist *B* false-wirksam. Wie bei MC/DC tragen bei der Termüberdeckung nur Einzelterme, also atomare Ausdrücke, zur Überdeckungsmetrik bei und nicht die zusammengesetzten Ausdrücke. Die Definition zur Bestimmung der Wirksamkeit lautet nach [LL10] wie folgt:

- Die Wurzel des Baumes (d. h. der logische Ausdruck insgesamt) ist wirksam.
- Ein Unterknoten eines Und-Knotens ist genau dann unwirksam, wenn der Und-Knoten unwirksam oder dessen anderer Unterknoten false ist.
- Ein Unterknoten eines Oder-Knotens ist genau dann unwirksam, wenn der Oder-Knoten unwirksam oder dessen anderer Unterknoten true ist.
- Eine Negation (not) kehrt nur den Wert um, hat aber im Übrigen für die nachfolgende Betrachtung keine Bedeutung.

Damit ergeben sich für einen Ausdruck *X and Y* sowie *X or Y* zusammenfassend die Wirksamkeiten für die Werte von *X* und *Y* nach Tabelle 4.

Regel	X and Y		Wirksam
1	F	F*	
2	F	T*	X
3	T	F	Y
4	T	T	X, Y

Regel	X or Y		Wirksam
1	F	F	X, Y
2	F	T	Y
3	T	F*	X
4	T	T*	

Tabelle 4: Wertetabellen für die Wirksamkeit der Operatoren X und Y

Die mit * gekennzeichneten Werte für Y bilden bei Operationen mit Kurzschlusssemantik ein „don't care", da der Wert von Y dann nicht ermittelt wird. Bei der *and*-Operation entfällt damit bei Kurzschlusssemantik die Regel 1, bei der *or*-Operation die Regel 4. Eine elegante Implementierung der Wertetabelle nach Tabelle 4 schlägt Ludewig [Lud11] vor. Wenn eine boolesche Variable *operation* für den Fall der *and*-Operation den Wert true und für den Fall der *or*-Operation den Wert false annimmt, dann kann die Wirksamkeit für X und Y statt über die Tabellenauswertung auch über einen einfachen Ausdruck festgestellt werden:

```
Wirksam(X) = (Y == operation);
Wirksam(Y) = (X == operation);
```

Die Gültigkeit dieser Ausdrücke lässt sich durch Vergleich mit den Regeln von Tabelle 4 nachprüfen.

2.5.4 Schleifenüberdeckungsmetrik

Die Überlegungen zur Schleifenüberdeckung [Bei90] basieren auf dem Boundary-Interior-Test nach Howden [Ho75]. Vollständige Schleifenüberdeckung ist dann erreicht, wenn der Schleifenkörper bei Ausführung der Schleife mindestens einmal übersprungen wird (d. h. die Wiederholbedingung ist bei der ersten Auswertung bereits false; Howden spricht hier von einem Boundary-Test der Schleife), genau einmal ausgeführt wird und mehrfach ausgeführt wird (diese Ausführungen werden von Howden als Interior-Test bezeichnet). Die Schleifenüberdeckung zielt nach Beizer [Bei90] darauf ab, dass auch Fehler gefunden werden, die im Zusammenhang mit der Anzahl der Schleifendurchläufe stehen. Die null-fache oder mehrfache Ausführung des Schleifenkörpers wird z. B. durch die Zweigüberdeckung nicht verlangt.

2.5.5 Weitere Überdeckungsmetriken

Einen umfangreichen Katalog vieler weiterer Überdeckungsmetriken und –kriterien liefert Kaner in [Kan96]. Neben den hier beschriebenen Überdeckungsmetriken enthält der Katalog viele weitere Überdeckungskriterien, die z. T. auch als Checkliste bei der Testfallerstellung zu verstehen sind. Teilweise ist eine vollständige Erreichung der geforderten Kriterien auch unrealistisch wie z. B. *„Every type of data sent to every object"* oder *„Handling of every potential data conflict"*, aber für besonders kritische Module ein nützlicher Anhaltspunkt beim Entwurf der Testfälle.

2.5.6 Subsumtionsrelation

Viele Autoren (z. B. [Na88, FW93]) beschäftigen sich mit der Frage, ob eine Teststrategie eine andere Teststrategie vollständig ersetzt (subsumiert). Ntafos weist in [Na88] darauf hin, dass verschiedene Teststrategien auch verschiedene Testaufwände verursachen können. So war eine ursprüngliche Intention der Forschungen zur Subsumtionsrelation der Überdeckungskriterien, dass eine Teststrategie gesucht wurde, die eine andere subsum-

miert, dabei aber keinen höheren Testaufwand verursacht. Oster liefert in [Ost07] eine aktualisierte Übersicht über die Subsumtionsrelationen verschiedener, überwiegend datenflussbasierter Überdeckungskriterien. Allerdings ist die praktische Relevanz solcher Überlegungen eher gering.

Def. **Subsumtionsrelation**. *Seien C_K und C_L zwei Überdeckungskriterien. Dann sagt man C_K* ***subsumiert*** *C_L, wenn jedes Programm-Testsuite-Paar (P, T), welches das Überdeckungskriterium C_K erfüllt, auch das Kriterium C_L erfüllt. C_K* ***subsumiert*** *C_L $\Leftrightarrow \forall P, T : C_K(P, T) \Rightarrow C_L(P, T)$ [Ost07]*

Bei Zweig- und Anweisungsüberdeckung liegt diese Subsumtion in aller Regel vor, d. h. führt eine Testsuite zu vollständiger Zweigüberdeckung (erfüllt also das Zweigüberdeckungskriterium), ist auch die Anweisungsüberdeckung vollständig erreicht. Angenommen wird dabei allerdings, dass es nur Anweisungen gibt, die auch über einen Zweig erreichbar sind. Der Nachweis kann leicht anhand der Knoten- und Kantenüberdeckung am CFG erfolgen. Der Subsumptionszusammenhang gilt aber nicht für Abstufungen: Bei 80 % Zweigüberdeckung besteht keine Garantie, dass (mindestens) ebenso 80 % Anweisungsüberdeckung erreicht werden. Da für Programme der Industrie praktisch nie 100 % Überdeckung erzielt werden, sind die Überlegungen zur Subsumtion in der Praxis letztlich bedeutungslos. Dagegen sind in der Praxis die Korrelationen zwischen den Überdeckungsmetriken interessant, weil damit im Wesentlichen nur eine Überdeckungsmetrik überwacht werden muss und die korrelierten Überdeckungsmetriken abgeschätzt werden können.

2.6 Techniken zum Testfallentwurf

Eine ausführliche Darstellung vieler Techniken zum Testfallentwurf befindet sich in [Bei90, Bin99, Li02, SL04, LL10], und eine vollständige Wiedergabe würde den Rahmen dieser Arbeit bei Weitem sprengen. Die wichtigsten Vertreter werden aber im Folgenden vorgestellt.

2.6.1 Funktionstest (Black-Box-Test)

Beim Funktionstest (oder Black-Box-Test, spezifikationsbasierter Test) bildet die Anforderungsspezifikation die Grundlage des Testfallentwurfs. Wie aus dem Namen „Black-Box-Test“ ersichtlich ist, werden beim Testfallentwurf nur die Anforderungen an den Prüfling herangezogen, nicht aber die innere Struktur des Prüflings wie beispielsweise der Programmcode. Man schaut also nicht in die „Box“ hinein, sondern nutzt die Dokumente, welche die von außen wahrnehmbare Funktion des Prüflings beschreiben.

In der Literatur finden sich zahlreiche Verfahren, wie aus der Anforderungsspezifikation die Testfälle systematisch entwickelt und dokumentiert werden [IEEE829, My79, SL04, LL10]. Da auf diese Weise die Testfälle zu den Anforderungen der Spezifikation geschrieben werden – also in der Regel gezielt die korrekte Umsetzung einzelner Anforderungen getestet wird – haben die Testfälle des Funktionstests einen sehr engen Anfor-

derungs- und damit Domänenbezug. Ein typischer Vertreter dieser funktionalen Testfallentwurfstechnik ist die Äquivalenzklassenbildung. Dabei wird der gesamte Eingaberaum einer Funktion des Prüflings so in Klassen partitioniert, dass alle Eingabedaten einer Klasse sich hinsichtlich ihrer Möglichkeiten zur Fehlerentdeckung gleich verhalten: Die Eingabedaten sind in dieser Eigenschaft äquivalent. Aus diesem Grunde kann ein beliebiger Eingabedatenwert jeder Klasse zusammen mit dem entsprechenden Soll-Resultat – als sogenannter Repräsentant – stellvertretend für die anderen Eingabedaten der Klasse gewählt werden.

Die so gewonnenen Testfälle werden in der Regel in einer Testsuite verwaltet, und nach [IEEE829, My79, SL04, LL10] wird die Dokumentation der Testfälle in einer einheitlichen Struktur empfohlen: Eindeutige Testfall-Nummer oder Testfallbezeichner, Vorbedingung, die zur Ausführung des Tests erfüllt sein muss, ggf. zuvor ausgeführte Testfälle, Eingabedaten oder Benutzeraktion und ein Soll-Resultat, das nach Abschluss der Eingaben ausgegeben werden soll. An dieser Struktur orientieren sich in der Regel auch die Werkzeuge zur Testfallverwaltung [Schm11]. Zudem können bei diesen Werkzeugen Testfälle in einer Ordnerstruktur abgelegt werden.

Als weiteres Attribut eines Testfalls wird von vielen Autoren (z. B. [Am00]) die Priorität empfohlen. Die Priorität eines Testfalls spielt speziell bei der Testwiederholung in der Wartung eine große Rolle. Nach Amland [Am00] ergibt sich die Priorität eines Testfalls aus dessen Chance, einen schwerwiegenden Fehler anzuzeigen. Hierzu bewertet er die Wahrscheinlichkeit, dass der Fehler eintritt, und den dann entstehenden Schaden. Bei begrenztem Testaufwand empfiehlt Amland die risikobasierte Ausführung der Testfälle, also in der Reihenfolge der Priorität, mit dem Ziel, die schwerwiegenden Fehler früh aufzudecken.

2.6.2 Glass-Box-Test

Bereits in den ersten Artikeln zum Glass-Box-Test [Hu75] wird dieser als *An Approach to Program Testing* bezeichnet. Auch Ramsey und Basili empfehlen in [RB85] *„Use structural coverage data to suggest new tests“*. Der GBT wurde damit von Anfang an eher als eine Technik für den Testfallentwurf genutzt und weniger als eine Metrik, um die Testgüte zu quantifizieren. Ein plausibles Beispiel nennt [UZ98]: Wenn die Implementierung eines Sortieralgorithmus zwei verschiedene Algorithmen nutzt, die abhängig von der Größe des zu sortierenden Feldes gewählt werden, und dieses Detail nicht in der Spezifikation enthalten ist, dann kann die Anweisungsüberdeckung zeigen, dass die Eingabedaten nie den Fall der Nutzung des kleinen Feldes enthalten haben. Der GBT kann so als Ergänzung zum BBT verwendet werden. [YCP09] sprechen in diesem Fall von Implementationsinspirierten Testfällen. Eine konkrete Methode, wie der Tester neue Testfälle für den nicht überdeckten Programmcode entwickelt, nennen die Autoren nicht. Eine ausführliche Darstellung eines Verfahrens zum Testfallentwurf im GBT folgt in Kapitel 8.

2.6.3 Modellbasierter Testfallentwurf

Nach Pretschner et al. [Pr05] basiert das modellbasierte Testen (MBT) auf formalen Verhaltensmodellen, aus denen heraus automatisch Testfälle mit Eingabedaten und Soll-

Resultat gewonnen werden. In der Regel wird eine so große Zahl von Testfällen erzeugt, dass eine automatische Testausführung unbedingt erforderlich ist. Die Modelle des MBT sind neben speziellen domänenspezifischen Modellen in vielen Fällen UML-Modelle wie z. B. Klassendiagramme, Sequenzdiagramme, Aktivitätsdiagramme oder Zustandsdiagramme, die um Invarianten ergänzt werden. Diese Invarianten werden in der Regel in der Object Constraint Language, OCL [Schm07b] geschrieben.

Der Testprozess nach Abbildung 3, bei dem direkt aus der Spezifikation die Testfälle „von Hand" entwickelt werden, wird beim MBT so verändert, dass zunächst ein formales Modell der Spezifikation (oder aus Teilen der Spezifikation) entwickelt wird. Die Autoren von [Pr05] argumentieren, dass der Tester sich ohnehin ein sogenanntes mentales Modell bildet, das er als Grundlage zur Erstellung der Testfälle verwendet. Dieses mentale Modell wird beim MBT durch ein explizites formales Modell ersetzt. So entsteht z. B. als explizites Modell ein Zustandsautomat, für den automatisch z. B. sogenannte Round-Trip-Folgen [LL10] als Testfälle generiert werden können. In der Regel sind nach [Pr05] die so gewonnenen Testfälle nicht unmittelbar für den Prüfling ausführbar, sondern es ist noch ein Adapter oder eine Transformation der Testfälle erforderlich.

Die grundsätzlichen Vorteile des MBT gegenüber den bereits beschriebenen Black-Box-Testverfahren sehen die Autoren von [Pr05] in einer früheren und intensiveren Überprüfung der Anforderungsspezifikation und in den anschaulichen Modellen, die beim MBT entstehen. Die Autoren geben an, dass in ihrem Fallbeispiel zwei- bis sechsfach mehr Spezifikationsfehler gefunden wurden als beim konventionellen Testfallentwurf. In ihrer Fallbeschreibung decken die Testfälle des MBT auch 11 % mehr Fehler auf als Testfälle, die von Hand entwickelt wurden.

Allerdings ist der MBT auf Anwendungsbereiche begrenzt, in dem die passenden Modelle verfügbar sind. Neben den bislang verwendeten Programmcode-bezogenen Überdeckungsmetriken kommen beim MBT die Modell-bezogenen Überdeckungsmetriken hinzu, die den Anteil des Modells angeben, der zur Testfallerstellung oder -ausführung genutzt wird. [Pr05] zeigt an einem Beispiel, wie Programmcodeüberdeckung und Modellüberdeckung zusammenhängen. Solche (Testfallentwurfs-)Modell-bezogenen Überdeckungsmetriken werden in dieser Arbeit aber nicht weiter behandelt.

2.6.4 Automatische Testdatengenerierung

Eine Vielzahl von Forschungsarbeiten beschäftigt sich mit der automatischen Generierung von Test-Eingabedaten, die zur Erhöhung der GBT-Überdeckung führen. Einen umfassenden Überblick zu diesen Methoden bieten [McM04] und [Ost07]. Viele dieser Arbeiten gehen auf die Arbeit von Korel zurück, der in [Ko90] das Prinzip der Ziel-orientierten Testdatengeneratoren beschreibt.

Für jede Verzweigung im Programm wird für den Bedingungsausdruck eine Funktion so definiert, dass der gewünschte Ausdruckswert genau dann erreicht wird, wenn diese Funktion ihren minimalen Wert annimmt [Ko90, Ost07]. Im nächsten Schritt wird dann mit sogenannten genetischen Algorithmen versucht, Eingabedaten zu finden, die die Funktion minimieren, was im Falle eines Erfolgs die Auswertung des Ausdrucks zu true bedeutet. Das Testziel ist damit erreicht, und ein neues Testziel wird ausgewählt.

Gupta et al. stellen in [GMS00] ein automatisches Verfahren vor, wie Test-Eingabedaten generiert werden können, die zu einer hohen Zweigüberdeckung führen. Auch sie bilden zunächst den Kontrollflussgraphen des Programms und legen ein Testziel wie beispielsweise einen durchlaufenen Programmzweig fest. Sie starten den Test mit willkürlich gewählten Eingabedaten und beobachten, inwieweit der überdeckte Teil des Kontrollflussgraphen sich dem Testziel nähert. Über die Verzweigungsbedingungen, die auf dem Pfad vom Startknoten bis zum Testziel liegen, bilden sie sogenannte lineare Einschränkungen, die sie ähnlich einem linearen Gleichungssystem lösen.

Wegener et al. beschreiben in [Weg01] ein werkzeuggestütztes Verfahren zur Testsuite-Erweiterung. Sie verwenden den Kontrollflussgraphen als GBT-Modell und wählen die Ausführung einer Anweisung oder einer festgelegten Anweisungsabfolge als Testziel. Ausgehend von den Eingabedaten bestehender, zu Beginn zufällig gewählter Testdaten variieren sie diese Eingabedaten so, dass der daraus resultierende Kontrollfluss das Testziel ausführt oder ihm möglichst nahe kommt. Hierfür definieren sie eine sogenannte Annäherungsstufe, die die minimale Anzahl an Verzweigungen angibt, die zwischen den von einem Testfall überdeckten GBT-Elementen und dem Testziel liegen. Aus dem Prädikat dieser Verzweigungsstellen ermitteln sie den sogenannten lokalen Abstand, der angibt, wie nahe die Eingabedaten an der gewünschten Änderung des booleschen Prädikatwertes der Verzweigungsstelle liegen. Die Annäherungsstufe und die lokale Nähe fassen sie in einer sogenannten Fitness-Funktion eines Testfalls für ein Testziel zusammen, die darüber entscheidet, ob der Testfall weiter variiert oder verworfen wird.

Ähnlich wie bei Wegener et al. generiert Oster [Ost07] Eingabedaten für Testfälle, die vorgegebene Überdeckungskriterien erfüllen sollen. Oster behandelt dabei insbesondere Datenflusskriterien und verfolgt eine Optimierung der Testsuite so, dass möglichst wenige Testfälle die beabsichtige GBT-Überdeckung erzielen. Die Soll-Resultate für die so ermittelten Eingabedaten müssen – wenn kein Orakel für das Prüfobjekt vorliegt – manuell aus der Spezifikation ermittelt werden. Die Anwendbarkeit beider Verfahren sehen die Autoren auf die Teststufe des Modultests begrenzt.

Einen Schwachpunkt von automatisch generierten Testdaten nennen Staats et al. in [Sta12]. Die Resultate ihrer Untersuchungen zeigen, dass automatisch generierte Testdaten, die ausschließlich auf eine Erhöhung der Überdeckung abzielen, nur eine geringe Fehlerentdeckungsquote haben. Die Autoren stellen aber auch fest, dass, sobald zusätzlich zur Erhöhung der GBT-Überdeckung noch weitere Faktoren für die Wahl der Testdaten einfließen, die Fehlerentdeckungsquote deutlich ansteigt.

2.6.5 Kombinierter GBT/BBT

Leider sind in der Literatur sehr wenige Berichte über praktische Erfahrungen beim GBT in Projekten der Industrie außerhalb des Modultests zu finden. Piwowarski et al. berichten in [POC93] über den Testprozess beim Systemtest großer Systeme bei der IBM. In ihren Untersuchungen stellen sie fest, dass auch beim gründlichen Black-Box-Test nur eine Anweisungsüberdeckung von etwa 60 % erreicht wird. Im GBT können sie diese Überdeckung auf ca. 70 % erhöhen. Die Autoren geben an, dass die so im GBT entwickelten Testfälle nur eine geringfügig kleinere Fehlerentdeckungsrate als die Testfälle des Black-Box-

Tests haben. Eine weitere Erhöhung der Anweisungsüberdeckung über 70 % erweist sich in ihren Untersuchungen als nicht wirtschaftlich, weil der Aufwand zur Entwicklung und Ausführung der Testfälle zu groß wird. Piwowarski et al. beschreiben aber kein Verfahren, wie aus den Resultaten des Glass-Box-Tests neue Testfälle hergeleitet werden sollen.

Dupuy und Leveson untersuchen in [DL00] einen kombinierten Black-Box- und Glass-Box-Test anhand einer Satellitensteuerung. Der Programmcode dieser Satellitensteuerung ist allerdings wesentlich kleiner als die von Piwowarski et al. untersuchten Systeme. Als GBT-Überdeckung verwenden sie die für sicherheitskritische Software in der Raumfahrt vorgeschriebene MC/DC-Überdeckung [CM94]. Sie führen dabei zuerst den Black-Box-Test durch. Der überwiegende Teil an Fehlern wird dabei durch die „off-nominal"-Testfälle gefunden, also Testfälle mit Eingabedaten, die im spezifizierten Einsatz des Systems nicht auftreten können. Mit Black-Box-Testfällen entdecken sie aber auch Fehler bei speziellen Parameterkombinationen. Trotz des sehr gründlichen Black-Box-Tests stellen Dupuy und Leveson beim Glass-Box-Test Programmteile fest, die nicht überdeckt wurden. Speziell werden hier Programmteile zur Fehlerbehandlung genannt, in denen schließlich auch Fehler gefunden werden.

Yu, Chan und Poon untersuchen in [YCP09] 56 verschiedene Programme, die von Studierenden für eine vorgegebene Spezifikation einer Kreditkarten-Funktion implementiert werden. Sie verwenden dabei die Klassifikationsbaum-Methode zur Herleitung der Black-Box-Testfälle und als Überdeckungsmetrik die Pfadüberdeckung. Es handelt sich um sehr kleine Programme ohne Schleifen. In ihrer Untersuchung gehen sie der Frage nach, wie hoch die Gesamtüberdeckung der Testsuite für die einzelnen Programme ist und wie sie durch neue Testfälle, die beim Glass-Box-Test entwickelt werden sollen, erhöht werden kann. Sie stellen dabei fest, dass nicht ausgeführte Programmpfade hauptsächlich Implementierungsdetails behandeln. Zur Erweiterung der Testsuite sprechen sie von „Implementations-inspirierten" Testfällen; der Tester lässt sich also vom Programmcode inspirieren und ergänzt aus seiner Domänenkenntnis heraus die Testsuite um die neuen Testfälle.

Alle genannten Arbeiten zeigen den Nutzen eines kombinierten Black-Box-/Glass-Box-Tests, allerdings ohne darauf einzugehen, wie der Tester konkret aus den Resultaten des GBT neue Testfälle entwirft.

2.7 Wirksamkeit des Glass-Box-Tests

In der Literatur sind einige Untersuchungen mit verschiedenen Untersuchungsmethoden zur Wirksamkeitsbewertung des GBT zu finden. Am häufigsten vertreten ist die sogenannte Mutationsanalyse, bei der in der Regel synthetische Fehler in den Prüfling implantiert werden [CL05, An06]. Zudem gibt es einige Untersuchungen von Programmen mit „echten" Fehlern, die in der Regel aus Fehlerdatenbanken gewonnen werden [HLL94, MND09].

Andrews et al. weisen in [An06] darauf hin, dass bei echten Programmen mit echten Fehlern die Datenaufbereitung sehr aufwändig ist und dass oft nicht genügend Fehler enthalten sind, um statistische Signifikanz zu erreichen. Aus diesem Grund werden in vielen Untersuchungen die Fehler in korrekte Programme eingefügt, um so fehlerhafte

Programmversionen mit bekannten Fehlern zu erhalten. Solche Fehler können nach [An06] zum einen von Hand eingefügt werden, indem mit erfahrenen Entwicklern typische Fehlermuster erstellt und Fehler nach diesen Mustern in die Programme eingefügt werden. Zum anderen können Fehler auch automatisch in die Programme durch sogenannte Mutationsoperatoren eingefügt werden. Die so entstandenen fehlerhaften Programmversionen werden auch als Mutanten bezeichnet. Die Autoren von [An06] sehen als besonderen Vorteil der Mutationsanalyse, dass die Mutationsoperatoren präzise beschrieben werden können, womit ein nachvollziehbarer Prozess zur Fehlerindizierung entsteht – Experimente können so leichter nachvollzogen werden. Zudem lassen sich quasi beliebig viele verschiedene Mutanten erzeugen, wodurch eine statistische Signifikanz erreicht werden kann.

Im Folgenden werden mehrere Untersuchungen zur Wirksamkeit des GBT vorgestellt, aus denen übereinstimmend hervorgeht, dass eine hohe GBT-Überdeckung mit einer besseren Fehlerentdeckungsquote einhergeht. Auch ist übereinstimmend zu erkennen, dass sich die Fehlerentdeckungsquote gerade bei Überdeckungen ab etwa 70 - 80 % erkennbar verbessert, wobei aber der Aufwand, um diese hohen Überdeckung zu erreichen, deutlich ansteigt. Es gibt aber auch Autoren, die keinen generellen Schwellwert unterhalb von 100 % erkennen, ab dem ein Weitertesten grundsätzlich unwirtschaftlich wird. Marick schreibt hierzu in [Mar99]: *"85% is a common number. People seem to pick it because that's the number other respectable companies use. I once asked someone from one of those other respectable companies why they used 85%. He said, "When our division started using coverage, we needed a number. Division X has a good reputation, so we thought we'd use the number they use." I didn't follow the trail back through Division X. I have the horrible feeling that, if I traced it all the way back to the Dawn of Time, I'd find someone who pulled 85% out of a hat. I don't know of any particular reasons to prefer one high number over another. Some claim that 85% is the point at which achieving higher coverage gets too hard. Based on my experience, I'd say that whether and where a "knee" in the effort graph appears is highly dependent on the particular application and the approach to test implementation."*

2.7.1 Untersuchung zweier Industrieprojekte [HLL94]

Horgan et al. untersuchen in [HLL94] den Zusammenhang zwischen der GBT-Überdeckung und der Fehlerentdeckungsrate anhand von zwei verschiedenen Industrieprojekten. Sie setzen hierzu ihr Werkzeug ATAC ein [LHL93], das bei vielen Untersuchungen dieser Art (z. B. [An06, CL05]) verwendet wird. Im ersten Projekt (Projekt 1) stehen den Autoren sehr viele Testfälle zur Verfügung. Sie beobachten dabei, dass bereits mit relativ wenigen Testfällen für Blocküberdeckung ein Wert von nahezu 80 % erzielt wird und im weiteren Testverlauf die Zunahme der Überdeckung deutlich geringer ausfällt (Abbildung 4). Insgesamt werden im Verlauf des Tests relativ wenige Fehler gefunden (insgesamt 96).

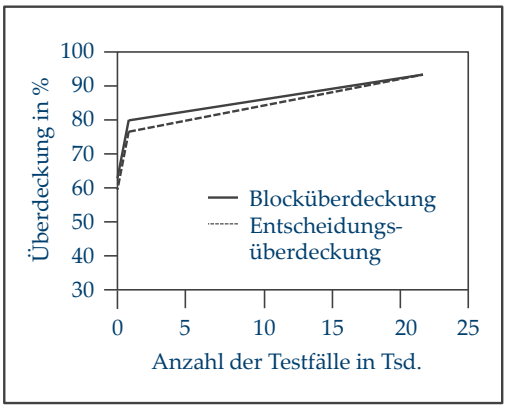


Abbildung 4: GBT-Überdeckung über der Anzahl Testfälle nach [HLL94]

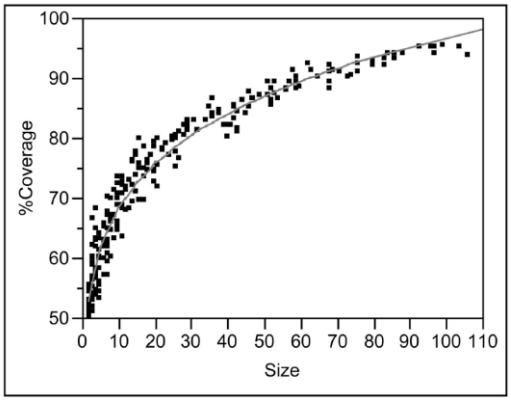


Abbildung 5: Blocküberdeckung über der Testsuitegröße nach [An06]

Die Autoren vermuten eine Korrelation zwischen der Anzahl der gefundenen Fehler und der GBT-Überdeckung, können aber aufgrund der geringen Zahl von Fehlern keine strenge Korrelation nachweisen. In einem zweiten Projekt untersuchen die Autoren den Zusammenhang zwischen der Überdeckung, die im Unit-Test des Projekts erzielt wird, und der Anzahl von sogenannten *modification requests*, MR, die Fehlern entsprechen, die in den folgenden Teststufen erfasst werden. Die Autoren geben an, dass es eine klare Beziehung zwischen hoher Überdeckung im Unit-Test und geringer Fehlerzahl im Systemtest gibt. Als Grenzwert nennen die Autoren 70 % Überdeckung im Unit-Test als Mindestkriterium; unterhalb dieses Wertes bleiben zu viele Fehler in den Modulen unentdeckt, was beim Systemtest zu unnötig vielen Fehlern führt. Das untersuchte Programm hat annähernd 60 kLOC und besteht aus 60 Modulen.

2.7.2 Mutationsanalyse [Hut94, An06]

In [An06] untersuchen Andrews et al. ein Programm mit ca. 6.000 LOC aus der Raumfahrt, für das eine Testsuite mit ca. 10.000 Testfällen vorliegt. Sie erzeugen 736 Mutanten, die jeweils mindestens einen Fehler enthalten, der von mindestens einem der Testfälle auch aufgedeckt wird. Aus der gesamten bereitstehenden Testsuite erzeugen sie zum einen mehrere Teilmengen, die jeweils eine bestimmte Überdeckung erzielen (getrennt nach vier Überdeckungskriterien: Block-, Entscheidungs-, *c-use*- und *p-use*-Überdeckung). Diese Testsuiten werden als *Coverage Suites* (CS) bezeichnet. Zum anderen erzeugen sie zufällig zusammengestellte Teilmengen (*Random Suite, RS*), die die gleiche Anzahl an Testfällen enthalten wie die CS Testsuiten. Für die CS-Testsuiten ermitteln sie den Zusammenhang zwischen der Größe einer Testsuite (die Anzahl der enthaltenen Testfälle) und der Blocküberdeckung, die die Testsuite erzielt (Abbildung 5). Der Verlauf der Kurve entspricht der intuitiven Erwartung: Man sieht einen etwa linearen (und steilen) Anstieg der Überdeckung bis etwa 80 %. Dieser Wert wird mit Testsuiten mit ca. 30 Testfällen erreicht. Danach flacht die Kurve deutlich ab; d. h. es sind für die weitere Erhöhung der Überdeckung von 80 % auf etwa 90 % deutlich größere Testsuiten erforderlich. Der Aufwand, eine Überdeckung über 80 % zu erzielen, steigt damit ganz deutlich an. Dieses Resultat gilt in etwa für alle vier untersuchten Überdeckungsmetriken in gleicher Weise. In einer zweiten

Auswertung stellen die Autoren die Entdeckungsquote eines Mutanten der Größe der eingesetzten Testsuite gegenüber. Ein Mutant gilt für eine Testsuite als entdeckt, wenn mindestens ein Fehler im Mutanten gefunden wird. Die RS-Testsuiten, also die Testsuiten mit zufällig zusammengestellten Testfällen, erreichen dabei bei gleicher Größe wie die CS-Testsuiten nur eine geringere Mutantenentdeckungsquote. Die Autoren leiten daraus ab, dass die GBT-Überdeckung eine gute Unterstützung bei der Wahl der Testfälle liefern kann und dass Testsuiten, die auf Grundlage von GBT-Überdeckung zusammengestellt sind, eine signifikant bessere Fehlerentdeckungsquote haben als zufällig zusammengestellte Testsuiten.

Zu sehr ähnlichen Resultaten wie [An06] kommen in einer früheren Untersuchung Hutchins et al. in ihrem Experiment zur Bewertung der Wirksamkeit von Kontroll- und Datenflussüberdeckungsmetriken [Hut94]. Die fehlerhaften Programme werden ebenso durch künstliches Einfügen von Fehlern (sogenanntes *fault seeding*) erzeugt, allerdings erfolgt das Einfügen der Fehler von Hand. Es werden etwa 130 fehlerhafte Programme erzeugt und mit automatisch zusammengestellten Testsuiten getestet. Die Grundlage zur Herstellung der 5.000 Testsuiten bildet ein Pool von Testfällen, aus dem gezielt Testsuiten mit einer vorgegebenen Anzahl von Testfällen gebildet werden. Für einen Teil der Testsuiten erfolgt dies optimiert für eine bestimmte Überdeckung. Es werden in diese Testsuiten nur Testfälle aufgenommen, die die bereits erreichte Überdeckung erhöhen, ansonsten wird ein nächster Testfall zufällig aus dem Pool gewählt und auf seinen Beitrag zur Überdeckung geprüft. Ein anderer Teil an Testsuiten wird zufällig ohne Betrachtung der Überdeckung zusammengestellt. Was die Wirksamkeit des GBT betrifft, kommen die Autoren zu folgendem Ergebnis: *"Within the limited domain of our experiments, test sets achieving coverage levels over 90 % usually showed significantly better fault detection than randomly chosen test sets of the same size."* Hutchins et al. stellen auch fest, dass eine deutliche Verbesserung der Wirksamkeit der Testsuiten bei einer Überdeckung von über 90 % zu beobachten ist. Dieser Schwellwert liegt höher als bei [An06]. Dort wird dieser Effekt ab 80 % festgestellt. Allerdings sind die untersuchten Programme von [Hut94] auch relativ klein; es handelt sich um mehrere Programme zwischen 150 und 500 LOC.

2.7.3 GBT-Überdeckung und Betriebsfehler [MND09]

Mockus et al. untersuchen in [MND09] zwei verschiedene Industrieprojekte in Bezug auf eine im Unit-Test festgestellte GBT-Überdeckung und die Zahl der sogenannten Post-Verifikationsdefekte, also der Fehler, die nach Abschluss der Prüfungen im Rahmen von Alpha- und Beta-Erprobungen sowie im Betrieb des Systems durch den Kunden entdeckt werden.

Die beiden untersuchten Projekte unterscheiden sich dabei deutlich: Das erste Projekt (Projekt 1) ist ein Informationssystem, das (überwiegend) in der Programmiersprache Java entwickelt wurde und etwa 1 MLOC groß ist. Das zweite untersuchte Projekt (Projekt 2) ist Windows Vista (Microsoft) mit über 40 MLOC, das in den Programmiersprachen C und C++ implementiert ist.

Für beide Systeme wird im Rahmen des Unit-Tests die Überdeckung gemessen. In Projekt 1 wird das Fehlerverfolgungssystem ausgewertet, in dem alle Fehler mit genauer Be-

schreibung und einer eindeutigen Kennung vermerkt sind. Mit dieser eindeutigen Fehlerkennung sind auch die sogenannten Submit-Vorgänge im Versionskontrollsystem gekennzeichnet. Diese Submit-Vorgänge bilden die Änderungen am Programmcode, die die Entwickler zur Fehlerbehebung vornehmen. Auf diese Weise können den Fehlermeldungen die betroffenen Module, in denen die Fehlerbehebung stattfindet, zugeordnet werden. Unter Modul wird eine Java-Klasse verstanden. Die Klassen werden nun in fünf GBT-Überdeckungs-Kategorien eingeteilt (0 % , 0 - 30 %, 30 – 60 %, 60 – 80 % und über 80 %), je nach dem Ergebnis, das die Klassen im Unit-Test erzielt haben.

Nach Aussage der Autoren zeigt sich ein konstantes Abschwächen der Betriebsfehlerrate bei höher überdeckten Klassen. Die Autoren stellen auch fest, dass der stärkste Rückgang der Betriebsfehlerrate bei den höchsten Werten der Überdeckung entsteht, wo sich die Fehlerrate beim Überdeckungsbereich 60 – 80 % zu 80 – 100 % halbiert. Damit wird es schwierig, den kosteneffektivsten Überdeckungsbereich zu benennen, weil einerseits mit Erreichung der hohen Überdeckung eine geringere Fehlerrate entsteht, aber andererseits mit hoher Überdeckung auch hohe Testaufwände verbunden sind.

Auch im zweiten untersuchten Projekt stellen die Autoren fest, dass mit höherer Überdeckung im Modultest weniger Betriebsfehler verbunden sind. Die beiden Untersuchungen zeigen nach Auffassung der Autoren konsistent, dass mit höherer Überdeckung im Modultest eine bessere Qualität einhergeht. Die Daten weisen aber auch darauf hin, dass der Testaufwand exponentiell mit der GBT-Überdeckung wächst, so dass für viele Projekte eine 100-%-Überdeckung nicht das kosteneffektivste Kriterium ist. Diese Resultate entsprechen damit qualitativ und quantitativ den Ergebnissen von Andrews et al. in [An06].

2.8 Nutzungsmöglichkeiten des Glass-Box-Tests

Mit den Mitteln des GBT wird in der Literatur eine Reihe unterschiedlicher Nutzungsmöglichkeiten beschrieben. Die wichtigsten nach [UZ98, Schm08] sind wie folgt:

- **Überdeckungsmetrik**: Der GBT liefert eine objektive Metrik zur Testgütebestimmung. Die GBT-Überdeckung wird durch die Korrelationen mit der Anzahl an gefundenen Fehlern [RB85, Gi06, An06, MND09] als sogenannte Pseudometrik [LL10] für die Testgüte genutzt. Auch die Nutzung der Überdeckung als Testendekriterium bezieht sich auf diese Korrelation. Die Überdeckungsmetriken kann man als „Paradedisziplin“ des GBT bezeichnen. Sie spielen auch in dieser Arbeit eine zentrale Rolle.
- **Testfallentwurf**: Die Auswertungen des GBT bilden den Ausgangspunkt zum Testfallentwurf [Hu75, UZ98, POC93, YCP09]. Dieses Thema wird ausführlich in Kapitel 8 behandelt.
- **Testsuite-Reduktion:** Das Ziel ist die Kostensenkung beim Test, speziell bei wiederholter Testausführung, durch eine Verkleinerung der Testsuite, ohne dabei aber (wesentlich) an Testgüte einzubüßen. Die Grundüberlegung basiert darauf, dass bei zwei Testfällen, die den (nahezu) gleichen Programmcode ausführen eine geringere Chance auf das Auffinden von Fehlern eingeräumt wird als bei zwei Testfällen, die deutlich verschiedenen Programmcode ausführen. Testfälle, die auf diese Weise als re-

dundant erkannt werden, können entweder ganz aus der Testsuite entnommen oder in der Priorität weiter nach hinten versetzt werden. Damit kommen sie abhängig von der angestrebten Testgüte zum Einsatz [HG93, JH03, JG05, Schm09, KGM09]. Eine weitere Nutzung der Testsuite-Reduktionstechnik wird in [Schm09] vorgeschlagen. Ausgangspunkt der Reduktion bilden dabei nicht eine Testsuite, sondern Produktions-Eingabedaten. Es wird ein bereits in Betrieb befindliches System angenommen, zu dem es entsprechend viele Eingabedaten aus der normalen Nutzung gibt. Soll nun eine neue Funktion des Systems mit diesen Daten getestet werden, ist eine Reduktion der äquivalenten Eingabedaten auf einen Repräsentanten aus Wirtschaftlichkeitsgründen unbedingt erforderlich, da, selbst wenn die Ausführung automatisiert stattfindet, die Soll-Resultate für jeden Eingabedatensatz zuvor ermittelt werden müssen. Ein prinzipieller Schwachpunkt dieser Überlegungen zur Testsuite-Reduktion ist die zugrunde liegende Äquivalenzrelation der Ausführungen: Bei zwei Ausführungen mit äquivalenter Programmcode-Ausführung wird unmittelbar auf eine äquivalente Fehlersensitivität geschlossen.

- **Grundlage für einen selektiven Test:** Zur Kostenreduktion beim Test in der Wartung sollen anstelle der „rerun-all"-Strategie nur einzelne, ausgewählte Testfälle ausgeführt werden. Die Grundüberlegung ist, dass nach einer (kleinen und abgegrenzten) Programmänderung nur die Testfälle ausgeführt werden, die vor der Programmänderung die betreffenden Programmstellen auch ausgeführt hatten [Sn04, LGJ07]. Von Schumm wird das Thema des selektiven Tests ausführlich in [Schu08] anhand des auch in dieser Arbeit genutzten GBT-Werkzeugs CodeCover behandelt.

Während in dieser Arbeit Überdeckungsmetriken und der GBT-Testfallentwurf ausführlich behandelt werden, werden die Testsuite-Reduktion und der selektive Test nicht weiter behandelt.

3
Glass-Box-Test

In diesem Kapitel wird der aktuelle Stand der Literatur zum GBT zusammengefasst. Beginnend mit der Übersicht über die in der Literatur beschriebenen GBT-Modelle werden die Stärken und Schwächen dieser Modelle herausgearbeitet. Da die Relevanz der GBT-Metriken und das in dieser Arbeit entwickelte Verfahren zum Testfallentwurf die Wirksamkeit des GBT voraussetzen, wird eine Reihe von Untersuchungen hierzu vorgestellt und zusammengefasst. Darauf folgt eine Übersicht über die aktuelle GBT-Werkzeuglandschaft. Weil die Laufzeitprotokollierung für GBT-Werkzeuge eine zentrale Funktion bildet, werden abschließend die in der Literatur beschriebenen Techniken zur Laufzeitprotokollierung zusammengefasst.

3.1 Modelle

Mit den ersten Artikeln zum GBT wurde zur Veranschaulichung der Kontrollflussgraph als GBT-Modell eingeführt [Hu75, My79]. Huang spricht in [Hu75] beim Kontrollflussgraphen allerdings nicht von einem „Modell“ des Programms, sondern vom „Programm“. So wie Huang den CFG beschreibt, liegt die Vermutung nahe, dass der CFG 1975 eher ein Entwurfsinstrument als ein Test-Modell war. Gleichwohl hat sich damals der Kontrollflussgraph auch als Test-Modell zur Definition von Anweisungs- oder Zweigüberdeckung gut geeignet. Heute spielt der Kontrollflussgraph als Entwurfsinstrument längst keine Rolle mehr, beim GBT dagegen ist er noch immer präsent – und das praktisch ohne Veränderung gegenüber dem Stand von 1975.

Neben den bereits von Huang beschriebenen kontrollflussbasierten Überdeckungsmetriken spielen in der Literatur die Überdeckungsmetriken für logische Ausdrücke (oft auch als Bedingungsüberdeckung bezeichnet) die größte Rolle. Während bei den kontrollflussbasierten Überdeckungsmetriken der CFG das dominierende Modell bildet, zeichnet sich bei den Bedingungsüberdeckungen kein vorherrschendes Modell ab. Die GBT-Modelle der Literatur werden im Folgenden ausführlich beschrieben.

3.1.1 Kontrollflussgraph

Der Kontrollflussgraph (CFG) ist das in der Literatur wie z. B. bei [My79, Bi90, ZHM97, Rie97, Li02, PC90] am häufigsten verwendete Modell des GBT. Liggesmeyer beschreibt in [Li02] den GBT anhand eines Modells wie in Abbildung 6.

```
void ZaehleZchn(int& VokalAnzahl,
                    int& Gesamtzahl)
{
  char Zchn;
  cin >> Zchn;                                  // n1
  while(Zchn >= 'A' && Zchn <= 'Z' &&           // n2
        Gesamtzahl < INT_MAX)
  {
    Gesamtzahl = Gesamtzahl + 1;                // n3

    if(Zchn == 'A' || Zchn == 'E' ||            // n4
       Zchn == 'I' || Zchn == 'O' ||
       Zchn == 'U')
    {
      VokalAnzahl =  VokalAnzahl + 1;           // n5
    }
    cin >> Zchn;                                // n6
  }
}
```

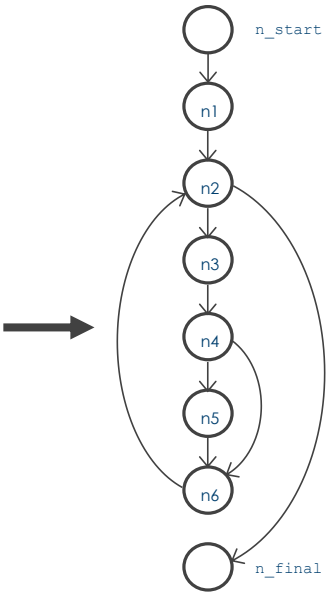
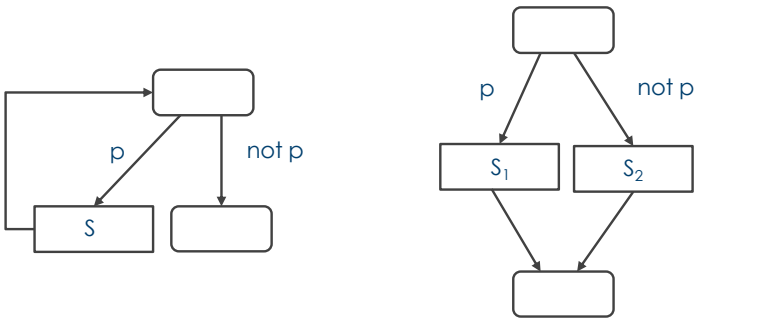


Abbildung 6: Der Kontrollflussgraph als GBT-Modell nach [Li02]

Für die oben genannten Quellen zum CFG ist ein Kontrollflussgraph $G := (N, E, n\_start, n\_final)$ ein gerichteter Graph, bestehend aus einer Menge N von Knoten und einer Menge $E \subseteq N \times N$ von gerichteten Kanten zwischen den Knoten sowie dem Startknoten $n\_start \in N$ und dem Endknoten $n\_final \in N$. Jeder Knoten im Kontrollflussgraphen liegt auf mindestens einem Pfad von *n_start* nach *n_final*. Nach [ZHM97] wird ein CFG aus einem Programm erzeugt, indem die Anweisungen des Programms auf die Knoten, der mögliche Kontrollfluss auf die Kanten abgebildet werden. Sie beschreiben auch nach Abbildung 7, wie die Kontrollstrukturen der prozeduralen Programmiersprachen, z. B. Verzweigung oder Schleifen, in einen Kontrollflussgraphen überführt werden.



(a) while p do S end (b) If p then S_1 else S_2 end

Abbildung 7: Kontrollflussgraph für imperative Strukturelemente aus [ZHM97]

Liggesmeyer empfiehlt, nur Knoten in den CFG aufzunehmen, die auch einer Anweisung im Programm entsprechen. Damit hat der zu einem if-then-else gehörige CFG bei Liggesmeyer einen Knoten für die Verzweigungsstelle, aber keinen Knoten für die Zusammenführung der beiden Zweige. Die Zusammenführung des then- und des else-Zweigs findet nach Liggesmeyer im ersten Statement bzw. Knoten nach dem if-Konstrukt statt. Er ergänzt wie [ZHM97] einen Startknoten und einen Endeknoten; diese beiden Knoten ent-

sprechen aber keiner Anweisung des Programms. Riedemann definiert in [Rie97] den Kontrollflussgraphen eines Programms ebenfalls als einen gerichteten Graphen, dessen Knoten Anweisungen oder Entscheidungsprädikate des Programms zugeordnet sind und dessen Kanten die Verbindungen zwischen Anweisungen und Anweisungsnachfolgern darstellen. Ganze Anweisungsfolgen, die keine bedingten Anweisungen enthalten, fasst er zu einem Knoten zusammen. Die Knoten mit Entscheidungsprädikat bezeichnet er als Entscheidungsknoten, die von Entscheidungsknoten ausgehenden Kanten als Entscheidungskanten. Die Entscheidungskanten sind mit true oder false markiert. Er fügt dem Kontrollflussgraphen ebenso wie [Li02] einen Anfangs- und Endknoten hinzu, denen keine Anweisungen des Programms zugeordnet sind.

3.1.2 Datenflussgraph

Rapps und Weyuker [RW85], Zhu, Hall und May [ZHM97] sowie Oster [Ost07] verwenden für die Datenfluss-Überdeckung (vgl. Kapitel 2.5.2, Seite 22) einen annotierten Kontrollflussgraphen. Nach [RW85] werden die Knoten mit lesenden Variablenzugriffen (sogenannte c-use Variablenzugriffe) sowie mit Wertzuweisungen der Variablen (sogenannte def-Variablenzugriffe) annotiert. Die lesenden Variablenzugriffe sind hier beispielsweise Verwendungen von Werten innerhalb eines Ausdrucks.

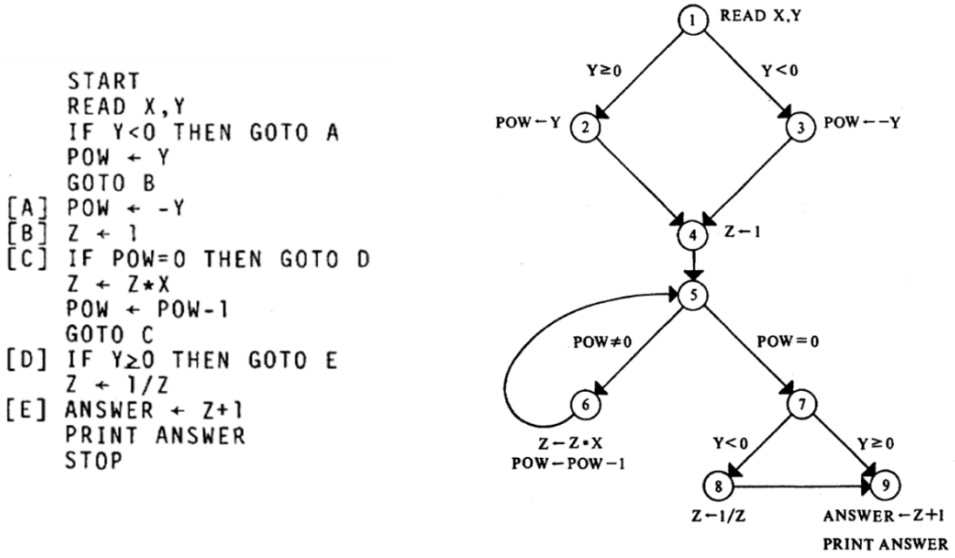


Abbildung 8: Programmbeispiel und annotierter CFG nach [RW85]

Die Bildung von Wahrheitswerten durch Auslesen von Variablenwerten innerhalb von Bedingungen der Verzweigungsstellen (der sogenannte p-use einer Variablen) annotieren sie an den Kanten, die von der Verzweigungsstelle ausgehen. Beispielhaft wird nach [RW85] ein so annotierter CFG in Abbildung 8 wiedergegeben. Rapps und Weyuker definieren in [RW85] die Datenfluss-Überdeckungsmetriken auf Grundlage einer einfachen formalen Sprache, die die Datendefinition, die Zuweisung und die Verwendung innerhalb einer bedingten Verzweigung enthält. Der Kontrollfluss entsteht durch goto-Anweisungen in Form von bedingten und nicht-bedingten goto-Sprüngen. Ein Beispielprogramm dieser Sprache zeigt Abbildung 8, das unmittelbar ein Problem vieler Modelle

zur Definition von GBT-Überdeckungsmetriken aufzeigt: Die Modellsprache erlaubt zwar präzise Definitionen der Überdeckungsmetriken, die Modelle entsprechen aber bei Weitem nicht den Programmen der gängigen Programmiersprachen. Der weitere Schritt, die Modelle auf praxisübliche Programme zu übertragen, wird von den Autoren nicht unternommen.

3.1.3 Modelle für logische Ausdrücke

Zur Veranschaulichung der Überdeckungsmetriken logischer Ausdrücke nutzen Chilenski und Miller in [CM94] eine Baumdarstellung des Ausdrucks sowie Wertetabellen. Die Grundlage einer Theorie bilden diese Modelle aber nicht. Ähnlich einfach definieren Ammann, Offutt und Huang in [AOH03] den logischen Bedingungsausdruck p als Prädikat, das aus einer Menge von Klauseln C_p (*clause*, im Folgenden auch als Einzelterm bezeichnet) als nicht weiter zerlegbaren booleschen Teilausdrücken des Prädikats besteht. Zur Veranschaulichung werden in [Li02] Wahrheitstabellen als GBT-Modell gewählt, die die möglichen logischen Werte der Einzelterme und das Gesamtresultat enthalten. In [LL10] wird die Termüberdeckung ähnlich wie in [CM94] anhand der Baumdarstellung des logischen Ausdrucks veranschaulicht. Die Autoren nutzen eine Baumdarstellung nach Abbildung 9 zur Veranschaulichung des Algorithmus zur Wirksamkeitsbestimmung der Terme. Der Gesamtausdruck bildet die Wurzel des Baumes, die Teilausdrücke die Teilbäume, und die atomaren Terme bilden die Blätter. Der praktische Vorteil dieses Modell ist, dass es für viele Programmiersprachen (wie z. B. Java) auch dem abstrakten Syntaxbaum des Ausdrucks entspricht.

(A AND (B OR (C AND D))) OR E

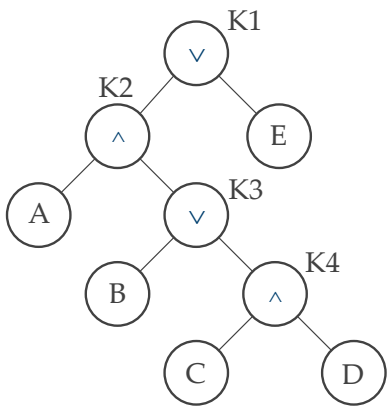


Abbildung 9: Baumdarstellung eines logischen Ausdrucks nach [LL10]

Insgesamt zeigt sich aber kein einheitliches Bild des verwendeten GBT-Modells bei logischen Ausdrücken.

3.1.4 Ablaufgraph für Kontrollfluss und Ausdrücke

In der Literatur ist der Ansatz, dass die Ausführung von Ausdrücken und Anweisungen in einem gemeinsamen Modell behandelt werden und so Ausdrücke auch zur Zweigüberdeckung beitragen, nicht sehr verbreitet. Hutchins et al. fordern in [Hut94], dass Ausdrücke zur Zweigüberdeckung beitragen sollen, und nennen das Werkzeug Tactic

[OW91], das eine solche Implementierung der Zweigüberdeckung enthält. Für den bedingten Operator als Ausdruck in einer Verzweigung geben Ostrand und Weyuker in [OW91] im Beispielprogrammcode

```
if(a?b:c) D;
else E;
```

den entsprechenden CFG nach Abbildung 10 an.

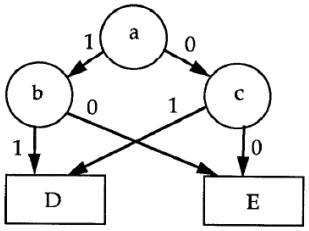


Abbildung 10: CFG für den bedingten Operator nach [OW91]

Die Ziffern (0 oder 1), mit denen die Kanten attributiert sind, sind als Kantenbedingung (false oder true) zu verstehen. Die Variablen a, b und c sind boolesche Variablen. Eine weitere Beschreibung oder eine Definition dieses GBT-Modells liefern die Autoren aber nicht. Auch Binder nutzt in [Bin99] (Seite 362 ff) in einer Beispieldarstellung einen CFG, der sowohl den Kontrollfluss der Strukturelemente als auch den Kontrollfluss zusammengesetzter boolescher Ausdrücke enthält.

Code-Segment-Darstellung

Line		Segment
1	public int displayLastMsg(int nToPrint) {	A
2	np = 0;	
3	if ((msgCounter > 0) && (nToPrint > 0)) {	B
4	for (int j = lastMsg; ((j != 0) && (np < nToPrint)); --j) {	C, D, E
5	System.out.println(messageBuffer[j]);	F
6	++np;	
7	}	
8	if (np < nToPrint) {	G
9	for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {	H, I, J
10	System.out.println(messageBuffer[j]);	K
11	++np;	
12	}	L
13	}	
14	}	
15	return np;	
16	}	

CFG-Darstellung

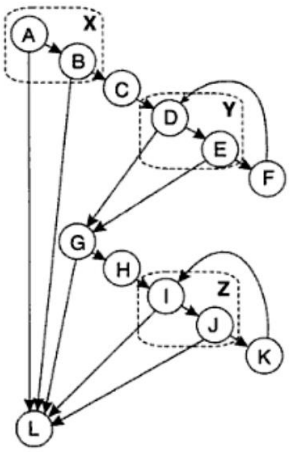


Abbildung 11: CFG eines Beispielprogramms nach [Bin99]

Die zusammengefassten Code-Bereiche der linken Darstellung von Abbildung 11 bezeichnet er als Code-Segmente, die rechte Darstellung bildet den zugehörigen CFG. Auch Binder vergibt für einzelne Code-Segmente Bezeichner, die im Modell zur Beschriftung des CFG genutzt werden. Allerdings definiert Binder keine Abbildungsvorschrift, die präzise und vollständig festlegt, wie der Programmcode in Segmente und aus den Segmenten der CFG entwickelt werden soll.

3.1.5 Eine reale Programmiersprache als GBT-Modell

In [FAA01, FAA07] werden die Vollständigkeitskriterien des GBT durch konkrete Programmbeispiele in den Sprachen Ada und C beschrieben. Für Ada und C werden auch Referenzprogramme geliefert, die den Werkzeugherstellern ganz präzise Vorgaben für den GBT machen.

Oster [Ost07] nutzt als Grundlage zur Generierung von Eingabedaten für Testfälle statt eines generischen Modells die Programmiersprache Java. Diese Vorgehensweise hat Vorteile, weil unmittelbar spezielle Eigenschaften einer Programmiersprache berücksichtigt werden können, ohne dass in einem Modell eine Abstraktion dieser Eigenschaften gebildet werden muss. Ein Nachteil dagegen ist eine möglicherweise schlechte Übertragbarkeit auf weitere Programmiersprachen, insbesondere dann, wenn sehr spezielle Eigenschaften der Programmiersprache in die Metrikendefinition einfließen, die so ansonsten nicht vorliegen. Auch lassen sich Schlussfolgerungen, die so entwickelt werden, nicht oder nur nach einer entsprechenden Abstraktion auf andere Programmiersprachen übertragen. Ein weiterer Nachteil ist die Abhängigkeit von künftigen Veränderungen der Programmiersprache, die dann möglicherweise auch Änderungen an der Definition der GBT-Metriken erforderlich machen.

GBT-Metriken, die auf Grundlage eines Modells definiert werden, haben darüber hinaus weitere Vorteile: Die Metriken lassen sich besser formal definieren, und zudem kann auf Grundlage eines (knappen) Modells deutlich einfacher eine Theorie entwickelt werden, die z. B. den Nachweis von Subsumtionsrelationen erlaubt.

3.1.6 Zusammenfassung und Bewertung

In der Literatur wird praktisch kein anderes GBT-Modell als der CFG genutzt – der CFG ist damit das „De facto"-GBT-Referenzmodell. Der besondere Vorteil des CFG besteht darin, dass er eine anschauliche und einfache Definition für Anweisungs- und Zweigüberdeckung ermöglicht. Zudem ist der CFG flexibel: Zwischen den Knoten können beliebige Kanten gezogen werden. Allerdings kann man hierzu einschränken, dass die gängigen Programmiersprachen „goto-less" sind (was bereits im Jahr 1972 Dahl, Dijkstra und Hoare in [DDH72] fordern). D. h. die Flexibilität wird letztlich nicht gebraucht.

Ein weiterer Vorteil des CFG ist, dass sich die Graphentheorie unmittelbar anwenden lässt. Es steht somit eine „fertige" und sehr mächtige Theorie zur Verfügung. Die Nachteile bestehen zunächst in Bezug auf den zugrundeliegenden Programmcode. Es gibt keine präzise und in der Literatur einheitliche Definition, wie Programme der gängigen Programmiersprachen in einen CFG transformiert werden. So wird z. B. in [ZHM97] (vgl. Abbildung 7) nach einem if-then-else ein Knoten für die Zusammenführung der then- und else-Zweige vorgesehen, in [Li02] wird hierfür kein Knoten vorgesehen mit dem Argument, dass Anweisungen nur mit einem Knoten repräsentiert werden und die if-Anweisung bereits durch den Verzweigungsknoten repräsentiert wird. Wenn aber die Anweisungsüberdeckung durch die Zahl der durchlaufenen Knoten bestimmt wird, ist eine einheitliche Abbildung des Programms in den CFG (und in die Knoten des CFG) erforderlich.

Anders als zur Entstehungszeit des CFG [Hu75] werden Programme heute in der Regel strukturiert programmiert, d. h. unter Verwendung der Kontrollflussstrukturen Entscheidung, Fallunterscheidung und Schleife. Auch wenn es wie in [ZHM97] eine präzise Vorschrift gibt, wie die Kontrollstrukturen imperativer Programme in den Kontrollflussgraphen übertragen werden, stellt sich die Frage, ob ein GBT-Modell, das die „Original"-Strukturelemente des Programms enthält, hinsichtlich der Schlussfolgerungen des GBTs nicht mächtiger ist. Als Beispiel kann man hier die Schleifenüberdeckung nennen, die eine mehrfache Ausführung des Schleifenkörpers fordert. Sind die Schleifen über die genannte Abbildungsvorschrift in Knoten und Kanten im Kontrollflussgraphen aufgegangen, ist es nachträglich schwer, solche Besonderheiten zu berücksichtigen. Ein weiterer Nachteil des Kontrollflussgraphen als GBT-Modell ist, dass nicht alle Überdeckungsmetriken eine ausreichende Repräsentation im Kontrollflussgraphen haben. So können dort beispielsweise die Überdeckungen der logischen oder bedingten Ausdrücke nicht angemessen dargestellt werden.

Und abschließend ist noch der gravierende Nachteil des CFG zu nennen, dass er keine Behandlung von Ausnahmen (Exceptions) kennt. Natürlich könnte man von jedem Knoten aus zum Programmende eine Kante für abruptes Beenden vorsehen, nur entspricht das nicht der Art, wie z. B. Java oder C++ Ausnahmen behandeln. Dort werden die Ausnahmen eines try-Blocks im catch-Block behandelt, und nur, wenn es diesen catch-Block nicht gibt, werden die Ausnahmen weiterpropagiert. Der CFG kennt aber keine Kompositionsstruktur, die diesem kaskadierten Ablauf beim Behandeln der Ausnahmen entspricht. Ein korrektes Nachvollziehen des Kontrollflusses bei Ausnahmen ist im CFG damit ausgeschlossen. [FAA07] fasst dies wie folgt zusammen: "*In Ada and in C, it is possible to abandon the execution of an apparent basic block by raising exceptions or by long jumps. The coverage reported in these cases should reflect only those statements that were actually reached*".

3.2 Werkzeuge

Eine Werkzeugunterstützung ist Voraussetzung des GBT. Der Markt bietet auch viele GBT-Werkzeuge für praktisch alle gängigen Programmiersprachen. Mehrere Untersuchungen zu GBT-Werkzeugen [YLW09, FAA07, Kle09, SI11] geben einen guten Überblick über die am Markt verfügbaren Werkzeuge und über deren jeweiligen Funktionsumfang. Die Produkte sind zum Teil kommerziell, zum Teil Open-Source, und als Resultat des Glass-Box-Tests liefern die Werkzeuge typischerweise Angaben über Anweisungs-, Zweig- oder Blocküberdeckung. Allerdings wird sich im Folgenden zeigen, dass es keinesfalls übereinstimmende Definitionen zu diesen Überdeckungsmetriken gibt. Unterschiede zwischen den Werkzeugen stellen Yang et al. in [YLW09] im Wesentlichen in der Integrationsmöglichkeit in einen bestehenden Build-Prozess sowie bei den Reportfunktionen fest. In ihrer Untersuchung werden von keinem der 17 Werkzeuge Datenflussmetriken unterstützt, was die Autoren auf die Schwierigkeiten bei der Interpretation der Ergebnisse zurückführen. Bei der Anweisungsüberdeckung basieren die Werkzeuge teils auf Programmblöcken, auf Anweisungen im Sinne der Sprachgrammatik oder auf Programmzeilen.

3.2.1 Vergleich von Überdeckungswerten

Tabelle 5 listet einige der aktuell verbreitetsten GBT-Werkzeuge für die Programmiersprache Java auf [PAR09, SI11] und zeigt die Resultate der GBT-Überdeckung für ein Referenz-Java-Programm mit einer fest vorgegebenen Programmausführung (das Programmlistin befindet sich in Abschnitt 6.6 auf Seite 166). Aus Tabelle 5 geht hervor, dass sich für die verschiedenen GBT-Werkzeuge deutliche Unterschiede in der angezeigten Überdeckung ergeben. Zudem herrscht auch bei den Überdeckungsmetriken eine Begriffsvielfalt (oder -verwirrung); so wird teilweise von Anweisungs-, Line- oder Instruction-Überdeckung gesprochen, ohne dass die Begriffe geeignet definiert oder gegeneinander abgegrenzt werden.

	Anweisungs-überdeckung	Zweig-/Block-überdeckung
CodeCover [CC13] Version: 1.0.2.2	62,8 %	Zweig: 50,0 % Block: 52,2 %
Clover [Clover] Version: 3.1.0	58,5 %	
Emma [Emma] Version: v2.1.5320	Line: 62,0 %	Block: 54,0 %
EclEmma [EclEmma] Version: 2.2.1	Instruction: 56,7 % Line: 63,6 %	Block: 50,0 %
eCobertura [eCob] Version: 0.9.8	64,3 %	Zweig: 50,0 %
CodePro [CodePro] Version: 7.1.0	Instruction: 57,7 % Line: 58,5 %	Block: 60,6 %
Rational Application Developer [RAD] Version 9.0.0	Line: 67,0 %	

Tabelle 5: Überdeckungswerte verschiedener GBT-Werkzeuge

Trotz dieser unbefriedigenden Streuung hat es den Anschein, dass sich die Bedeutung des GBT und die Verbreitung der GBT-Werkzeuge in der Praxis in den letzten zehn Jahren verändert haben. So zeigte sich in der Untersuchung von Müller aus dem Jahr 1998 [Mü98], dass Kontrollfluss-orientierte Testverfahren keine hohe Verbreitung haben. Selbst Anweisungsüberdeckung erhoben nur 12 % der befragten Firmen. In der im Jahr 2011 durchgeführten Untersuchung [SVW11] zu den in der industriellen Praxis eingesetzten Testtechniken antwortet fast die Hälfte der Befragten, dass Anweisungsüberdeckung im Test eine Rolle spielt. Das umfangreiche Angebot an z. T. kostenlosen GBT-Werkzeugen spiegelt diese Veränderung wider.

3.2.2 Werkzeugbewertung von [FAA07]

Eine konkrete Anforderungsspezifikation für GBT-Werkzeuge liefern Santhanam et al. in „Software Verification Tools Assessment Study“ [FAA07]. Grundlage der Anforderungen bildet die Richtlinie DO-178B [RTCA] für sicherheitskritische Software im Bereich der Luftfahrt.

Resultats-Kategorie	Beschreibung	Anzahl Resultate für die getesteten Werkzeuge		
		X	Y	Z
Pass	Das Werkzeug hat die Überdeckung korrekt bewertet.	120	75	122
False-Negative[3]	Das GBT-Werkzeug gibt fälschlicherweise an, dass die Überdeckung nicht vollständig ausgefallen ist, obwohl die Überdeckung gem. Referenz als vollständig zu bewerten ist.	11	26	3
False-Positive[3]	Das GBT-Werkzeug gibt fälschlicherweise an, dass die Überdeckung vollständig erzielt ist, obwohl die Überdeckung gem. Referenz als nicht vollständig zu bewerten ist.	13	15	24
Sonstiges	wie z. B. Abbruch, ohne dass ein GBT-Bericht entsteht	14	30	7

Tabelle 6: Testresultate nach [FAA07] für drei GBT-Werkzeuge

Die Autoren definieren getrennt für drei Sicherheitsstufen Referenzprogramme mit Referenzausführungen und deren GBT-Sollresultate. Diese Referenz nennen die Autoren „Prototype Testsuite“ – und verstehen diese als Testsuite zum Test von GBT-Werkzeugen. Anhand dieser Testsuite bewerten sie die Resultate von drei GBT-Werkzeugen (Werkzeug X, Y, Z – die konkreten Werkzeugnamen geben die Autoren leider nicht an) und stellen fest, dass keines der untersuchten Werkzeuge für alle Testfälle der Prototype Testsuite die geforderten GBT-Resultate liefert.

Die Bewertung der drei Werkzeuge X, Y, Z nach Tabelle 6 erfolgt derart, dass für Werkzeug X 158, für Werkzeug Y 146 und für Werkzeug Z 156 Testfälle ausgeführt werden. Ohne Beanstandung führt Werkzeug X 120, Werkzeug Y 75 und Werkzeug Z 122 Testfälle aus. Obwohl in dieser Untersuchung nur Werkzeuge betrachtet werden, die nach Herstellerangaben die Vorgaben der Sicherheitsrichtlinie erfüllen, haben die Werkzeuge

[3] Die Bezeichnungen False-Positiv und False-Negativ werden hier von der Quelle [FAA07] genau andersherum verwendet, als es sonst bei Test-Resultaten üblich ist.

dennoch viele Fehler – selbst die besser bewerteten Werkzeuge X und Z versagen bei etwa 20 % der Testfälle.

3.2.3 Experiment zum GBT nach [CK11]

Mehrere Untersuchungen zu Programmen der industriellen Praxis zeigen, dass ohne den Einsatz eines GBT-Werkzeugs – und damit ohne verfügbare GBT-Überdeckungsmetriken – nur eine geringere GBT-Überdeckung erzielt wird, als wenn entsprechende Überdeckungsmetriken mitsamt der für GBT-Werkzeuge üblichen Visualisierung der Überdeckung im Programmcode den Testern vorliegen (vgl. [POC93, BWK07, MND09]). Allerdings wird in diesen Untersuchungen vor der GBT-Werkzeugeinführung auch keine hohe GBT-Überdeckung als Testziel verfolgt.

Craver und Kraft beschreiben in [CK11] ein Experiment, das sie mit Studenten durchführen, die an einer Software-Engineering-Lehrveranstaltung teilnehmen. Sie untersuchen, ob die Studenten ohne GBT-Werkzeugunterstützung in der Lage sind, eine Testsuite für ein relativ kleines Programm (200 LOC) so zu entwickeln, dass eine möglichst vollständige Überdeckung erzielt wird. Im weiteren Verlauf des Experiments wird untersucht, ob ein GBT-Werkzeug den Studenten dabei hilft, die GBT-Überdeckung zu erhöhen, und ob die Studenten in der Lage sind, hierzu eine möglichst redundanzfreie (bezogen auf die GBT-Überdeckung) Testsuite zu entwickeln. Die Autoren untersuchen die folgenden Hypothesen:

Hypothese 1: Die Studenten sind nicht in der Lage, eine Testsuite für das bereitgestellte kleine Programm zu entwickeln, mit der vollständige GBT-Überdeckung erzielt wird. Ergebnis: Die Hypothese wird bestätigt; die Studenten erreichen deutlich weniger als 100 % Überdeckung. Die Autoren schließen daraus, dass ein GBT-Werkzeug erforderlich ist, wenn eine bestimmte GBT-Überdeckung ein Testziel bildet.
Hypothese 2: Der Einsatz eines GBT-Werkzeugs ermöglicht den Studenten, die GBT-Überdeckung zu erhöhen. Ergebnis: Die Hypothese wird bestätigt; mit Werkzeugunterstützung erreichen die Testsuiten der Studenten eine signifikant höhere Überdeckung als ohne GBT-Werkzeug.
Hypothese 3: Die Studenten sind nicht in der Lage, die bereitgestellte Information des GBT-Werkzeugs so zu nutzen, dass mit möglichst wenigen Testfällen die hohe Überdeckung erzielt wird. Ergebnis: Die Hypothese wird bestätigt; die Studenten ergänzen unsystematisch die Testfälle. Ob ein neuer Testfall tatsächlich die Überdeckung erhöht, wird von den Studenten nicht systematisch verfolgt.

Im Experiment wird das GBT-Werkzeug CodeCover [CC13] eingesetzt, allerdings ohne die in Kapitel 8 beschriebene Funktion zur Generierung von Testfall-Hinweisen. Die Autoren stellen insgesamt fest, dass die Studenten zwar in der Lage sind, durch Hinzufügen weiterer Testfälle die Überdeckung zu erhöhen, aber diese Erhöhung wird nicht systematisch erreicht. Aus dem Experiment schließen die die Autoren, dass trotz der in der Software-Engineering-Lehrveranstaltung vermittelten Lehrinhalte zum Test und zum GBT eine für die Praxis nicht ausreichende Qualifizierung der Studenten entsteht. Diese Feststellung stimmt mit dem Erfahrungsbericht aus [BWK07] überein, in dem für die Junior-Entwickler bei einer GBT-Einführung umfangreiche methodische Hilfestellung empfohlen wird.

3.2.4 Effekte des GBT-Werkzeugeinsatzes nach [La05] und [BWK07]

Lawrance et al. untersuchen in einem Experiment [La05] die Wirkung eines GBT-Werkzeugeinsatzes mit der für GBT-Werkzeuge üblichen Visualisierung der Code-Überdeckung auf das Verhalten der Tester. Sie bilden dazu zwei Gruppen mit jeweils 15 Teilnehmern: Die Teilnehmer der ersten Gruppe – im Folgenden als Kontrollgruppe bezeichnet – testen ein bereitgestelltes und mit Fehlern versehenes Programm ohne ein GBT-Werkzeug. Die Teilnehmer einer zweiten Gruppe – im Folgenden als GBT-Gruppe bezeichnet – testen das gleiche Programm, nutzen aber ein GBT-Werkzeug mit einer für GBT-Werkzeuge üblichen Visualisierung der Überdeckung im Programmcode. Alle Teilnehmer des Experiments sind erfahrene Softwareentwickler. Der Prüfling enthält Implementierungen bekannter Funktionen wie Zeichenkettenfunktionen oder arithmetische Funktionen, d. h. eine spezielle Domänenkenntnis ist für die Tester nicht erforderlich. Den Teilnehmern beider Gruppen liegt eine Spezifikation der Programmfunktion vor, und Teilnehmer beider Gruppen sollen Unit-Tests schreiben, um möglichst viele der in den Prüfling implantierten Fehler zu finden. Die folgenden Forschungsfragen (RQ) werden im Experiment untersucht:

RQ1: Werden Tester durch die Visualisierung der Überdeckung motiviert, mehr und wirksamere Testfälle zu schreiben? Antwort: Nein. Die Visualisierung der Überdeckung hat keinen Einfluss auf die Anzahl der gefundenen Fehler. Die Resultate dieses Experiments stimmen damit mit denen vorausgehender Untersuchungen (z. B. [KL95]) überein.
RQ2: Beeinflusst die Visualisierung der Überdeckung die Anzahl der geschriebenen Testfälle? Antwort: Die Tester der GBT-Gruppe schreiben geringfügig weniger Testfälle als die der Kontrollgruppe. Auffällig ist, dass die Anzahl der Testfälle in der GBT-Gruppe weniger stark variiert als in der Kontrollgruppe. Die Autoren führen dies darauf zurück, dass Tester mit Visualisierung der Überdeckung den Test bis zum Erreichen einer bestimmten Überdeckung fortsetzen, dann aber beenden. Den Teilnehmern der

Kontrollgruppe dagegen fehlt dieses Testendekriterium.
RQ3: Führt die Visualisierung der Überdeckung die Tester dazu, die erzielte Testgüte zu überschätzen? Antwort: Im Experiment haben die Teilnehmer beider Gruppen einen zu optimistischen Eindruck von der Wirksamkeit der geschriebenen Testfälle; es blieben jeweils mehr Fehler unentdeckt, als die Teilnehmer schätzen. Dabei ist die Einschätzung der Teilnehmer der GBT-Gruppe sogar etwas schlechter als die der Kontrollgruppe. Die Autoren führen dies darauf zurück, dass für den Prüfling relativ leicht eine Blocküberdeckung erreicht werden konnte und so den Testern durch die Visualisierung der Überdeckung eine optisch zu positive Rückmeldung geliefert wird. Die Autoren von [BWK07] bestätigen ebenso diesen negativen Effekt des GBT aus Beobachtungen in Industrieprojekten und liefern einen Katalog von Empfehlungen, um diesem Problem vorzubeugen.
Welche Teststrategien verwenden die Tester mit und ohne Visualisierung der Überdeckung? Antwort: In diesem Experiment lassen sich keine Unterschiede zwischen den Teilnehmern beider Gruppen feststellen. Die Autoren stellen aber für die Teilnehmer beider Gruppen fest, dass etwa ein Drittel der Arbeitszeit unsystematisch und in Folge unproduktiv gearbeitet wird. Als Resultat empfehlen die Autoren auch erfahrenen Testern und Entwicklern methodische Hilfestellungen zu geben um unproduktive Vorgehensweisen beim Test zu vermeiden.

Zusammenfassend stellen die Autoren von [La05] fest, dass die im GBT übliche Visualisierung der Überdeckung die Tester weder zu produktiveren Teststrategien führt noch dazu beiträgt, dass die Tester mehr Testfälle schreiben oder mehr Fehler finden.

Dass der GBT für die Tester eine begleitende methodische Unterstützung erfordert, beschreiben auch Berner, Weber und Keller in [BWK07]. Sie berichten über ihre Erfahrungen bei der Einführung eines GBT-Werkzeugs in einem laufenden Projekt. Sie untersuchen getrennt nach Entwicklergruppe (Senior und Junior) die Auswirkung der GBT-Werkzeugeinführung. Sie betrachten dazu ein Industrieprojekt mit acht Entwicklern über einen Zeitraum von einem halben Jahr. In der Mitte des Betrachtungszeitraums wird das GBT-Werkzeug eingeführt. Überdeckungswerte aus dem Zeitraum vor der Werkzeugeinführung werden durch Wiederherstellen der Programmversionen und der Testsuite aus der Versionsverwaltung nachträglich erhoben. Erhoben werden die Anweisungs-, Zweig- und Methodenüberdeckung, wobei alle drei Metriken im gesamten Beobachtungszeitraum sehr stark miteinander korrelieren.

Für die Gruppe der Senior-Entwickler erhöht sich mit der Einführung des GBT-Werkzeugs die Überdeckung schnell, allerdings nur um einen geringen Wert. Die Anwei-

sungsüberdeckung, die im Zeitraum vor der GBT-Werkzeugeinführung weitgehend konstant bei 70 % liegt, erhöht sich in nur zwei Wochen auf fast 80 %, erhöht sich dann aber innerhalb des restlichen Beobachtungszeitraums nicht weiter. Die Autoren begründen dies damit, dass das GBT-Werkzeug mit der Visualisierung der Überdeckung den Senior-Entwicklern geholfen hat, fehlende Testfälle zu identifizieren. In der Regel waren das Testfälle zum Test von Fehlerbehandlung. Nach Ergänzung dieser neuen Testfälle in die Testsuite liefert das GBT-Werkzeug den Senior-Entwicklern kaum weiteren Nutzen.

Bei der Gruppe der Junior-Entwickler zeigt die Einführung des GBT-Werkzeugs dagegen eine deutlich stärkere Wirkung. Die Anweisungsüberdeckung steigt von 50% am Anfang des Beobachtungszeitraums auf nahezu 90% am Ende an. Auch bei der Art der neu entwickelten Testfälle zeigen sich Unterschiede: Während die Senior-Entwickler fast ausschließlich Testfälle für die Fehlerbehandlung ergänzen, entwickelten die Junior-Entwickler zur Erhöhung der Überdeckung auch einen nennenswerten Anteil an Testfällen zum Test der normalen Funktionen. Zudem stellen die Autoren fest, dass Komponenten mit kollektiver Zuständigkeit (die Autoren nennen als Beispiel sogenannte Utility-Komponenten) im Gegensatz zu den Komponenten mit fester Zuordnung der Zuständigkeit eine vor Einführung des GBT-Werkzeugs deutlich geringere Überdeckung haben. Im Beobachtungszeitraum erhöht sich diese Überdeckung auch auf das durchschnittliche Niveau.

Wenn mit dem GBT das Ziel einer Verbesserung der Testgüte verbunden ist, dann lassen sich die beiden Untersuchungen [La05] und [BWK07] so zusammenfassen, dass für die Tester ein bloßes „Bereitstellen“ eines GBT-Werkzeugs nicht automatisch den gewünschten Effekt bewirkt. Vielmehr ist eine gründliche Einführung und methodische Hilfestellung erforderlich. So sollten z. B. nach [BWK07] die Tester immer wieder darauf hingewiesen werden, dass es nicht das ausschließliche Ziel ist, eine hohe Überdeckung zu erzielen, sondern es sollen nicht überdeckte Programmcodebereiche identifiziert werden, die auf neue und besonders fehlersensitive Testfälle hinweisen. Die methodische Unterstützung ist nach [BWK07] besonders für die Junior-Entwickler wichtig, weil diese Gruppe dann den größten Nutzen durch den GBT erzielen kann.

3.3 Ausführungsprotokollierung

Zur GBT-Überdeckungsmessung ist ein sogenanntes Ausführungsprotokoll (engl. *execution trace*, im Folgenden auch GBT-Protokoll genannt) erforderlich. Im Verlauf der Ausführung des Programms werden alle Daten protokolliert, die nötig sind, um eine Aussage bezüglich der Überdeckung vornehmen zu können.

Def. **GBT-Protokoll.** *Protokoll der Programmausführung, aus dem die GBT-Überdeckungen abgeleitet werden können.*

Nach Ball und Larus [BL94] ist ein Ausführungsprotokoll nicht auf einen bestimmten Verwendungszweck begrenzt. Neben der Verwendung beim GBT werden in der Literatur weitere Verwendungszwecke genannt wie z. B. Laufzeituntersuchungen oder Analyse häufig ausgeführter Programmteile (sogenannte „hot-path“-Analysen). Die Techniken der

Ausführungsprotokollierung unterscheiden sich dabei für die verschiedenen Verwendungen nur geringfügig. Da für diese Arbeit nur die GBT-Aspekte relevant sind, wird auf die anderen Verwendungsmöglichkeiten des Ausführungsprotokolls nicht weiter eingegangen.

3.3.1 Nicht-invasive Ausführungsprotokollierung

Bei der nicht-invasiven Ausführungsprotokollierung werden Schnittstellen der Laufzeitumgebung des Programms ausgenutzt. Bei Java-Programmen kann beispielsweise über eine der Schnittstellen der *Java Platform Debugger Architecture (JPDA)* auf die Laufzeitumgebung eingewirkt und diese angewiesen werden, bei bestimmten Aktionen des Prüflings (z. B. der Ausführung einer Anweisung) eine Nachricht an ein überwachendes Programm zu senden. Solche Ereignisse werden dann im GBT-Protokoll festgehalten. Diese nicht-invasive Laufzeitprotokollierung ist allerdings nur bei Programmiersprachen mit ausgeprägter Laufzeitumgebung (in der Regel also bei Interpretern) möglich.

Der Vorteil eines solchen Vorgehens ist, dass der Programmcode des Prüflings nicht verändert werden muss. Allerdings ist die angebotene Funktionalität, die über die Schnittstellen der Laufzeitumgebung genutzt werden kann, in der Regel für einige Überdeckungsmetriken (wie z. B. für die Bedingungsüberdeckungen) nicht ausreichend. Ein weiteres praktisches Problem stellt die hohe Abhängigkeit von dieser Schnittstelle dar. So wurde z. B. die Debug-Schnittstelle [JVMPI] von Java mit der Java-Version 6.0 stark verändert. Einige GBT-Werkzeuge (wie z. B. [PureCoverage]) sind damit mit neueren Java-Versionen in ihrer ursprünglichen Form nicht mehr einsetzbar.

3.3.2 Invasive Ausführungsprotokollierung

Die invasive Laufzeitprotokollierung verändert den Programmcode des Prüflings. Es werden sogenannte Sonden in das Programm eingewoben, die während der Programmausführung die für das GBT-Protokoll erforderlichen Ausführungsinformationen erheben. Diese Veränderung wird als Instrumentierung bezeichnet.

Def. **Instrumentierung.** *Vorgang, der im Prüfling Programmcode (sogenannte Sonden) hinzufügt, um die für den GBT gewünschten Ausführungsinformationen zu gewinnen.*

Die Instrumentierung kann entweder im Quelltext oder auch im sogenannten Byte-Code vorgenommen werden [LGJ07]. Nach [Ost07] liegt der Vorteil einer Byte-Code-Instrumentierung in der Einfachheit der Byte-Code-Sprache, wodurch sich die Instrumentierung relativ leicht gestaltet. Als größter Nachteil der Byte-Code-Instrumentierung wird genannt, dass die protokollierte Information nicht für alle Überdeckungsmetriken (insbesondere nicht für die Bedingungsüberdeckungen) genügend Rückschlüsse auf die tatsächliche Ausführung im Quelltext zulässt.

Yang et al. stellen in [YLW09] fest, dass die Instrumentierung des Programmcodes die am meisten verwendete Technik zur Laufzeitprotokollierung der von ihnen untersuchten GBT-Werkzeuge ist. Alle Autoren sind sich einig, dass die allerwichtigste Voraussetzung für eine erfolgreiche Instrumentierung die Vermeidung jeglicher Seiteneffekte ist. D. h. es

muss sichergestellt werden, dass der Prüfling vor und nach der Instrumentierung (abgesehen vom gewünschten Instrumentierungseffekt) das gleiche Verhalten aufweist. Zudem wird das Programm durch die ergänzten Anweisungen unvermeidbar größer und in der Ausführung langsamer. In bestimmten Anwendungsdomänen wie z. B. der Steuergeräte-Software kann dies dazu führen, dass der GBT nicht (zumindest nicht in der Zielumgebung) durchgeführt werden kann. Für Java-Programme hat sich in der Praxis gezeigt, dass beide Nachteile in der Regel zu keiner nennenswerten Beeinträchtigung führen. Das für den GBT unvermeidliche „application blowup“ und „execution slowdown“ wird im Folgenden auch nicht als Semantikveränderung des Programms gewertet.

3.3.3 Instrumentierung

Die heute verwendeten Techniken zur Instrumentierung für Anweisungs- oder Zweigüberdeckung gehen im Wesentlichen auf Knuth und Stevenson [KS73] zurück. Sie beschreiben, wie im Programm die Sonden platziert werden, damit der Aufwand während der Programmausführung minimiert wird. Ammann und Offutt beschreiben in [AO08] konkrete technische Details der Instrumentierung für Anweisungs- und Zweigüberdeckung, für Datenflussüberdeckung und Bedingungsüberdeckung.

Auch Oster beschreibt in [Ost07] technische Details der Instrumentierung seines Werkzeugs „.gEAr“, das für Java-Programme Datenflussmetriken erheben kann. Ball und Larus behandeln in [BL94] ausführlich das optimale Instrumentieren, also das Einfügen der Sonden mit dem Ziel, bei geringem Aufwand die nötigen Informationen der Programmausführung zu gewinnen.

Für die Anweisungs- und Zweigüberdeckung (nach [BL94] das sogenannte *Vertex Profiling* und *Edge Profiling*) wird nach [BL94] je Anweisung sowie Zweig ein Zähler vorgesehen, der bei Durchlaufen der Anweisung oder des Zweigs inkrementiert wird. In [BL94] wird in diesen Fällen auch von einer Zählerprotokollierung (Counter Profiling) gesprochen. Die Zähler werden vor Programmstart mit null initialisiert, und nach Programmende wird der Wert des Zählers in die Protokolldatei geschrieben. Die Anzahl der Ausführungszähler und damit die Größe des GBT-Protokolls korreliert nach [BL94] im Wesentlichen mit der Programmgröße (in LOC). Beim Edge Profiling nach [BL94] werden nur die Verzweigungskanten mit einem Zähler versehen. In [AO08] werden auch Zweige instrumentiert, die nicht von einem Verzweigungsknoten ausgehen. Man sieht aber leicht (und in [KS73] wird der formale Nachweis geliefert), dass, wie in [BL94] beschrieben, eine Instrumentierung ausschließlich für Verzweigungskanten erforderlich ist, da auch nur diese Kanten zur Zweigüberdeckung beitragen (vgl. [Rie97]). Mit Programmende, d. h. mit Erreichen des Knotens *Exit*, werden die Zählerstände in die Protokolldatei geschrieben. Solche mittels *Counter-Profiling* gewonnenen GBT-Protokolle der Zweig- und Anweisungsüberdeckung haben aber nach [BL94] den Nachteil, dass Ausführungsabfolgen nicht rekonstruiert werden können. So lässt ein solches GBT-Protokoll nicht genügend Rückschlüsse auf die tatsächliche Ausführung zu, um beispielsweise die Pfadüberdeckung [My79] oder die Segmentpaarüberdeckung nach [Rie97] bestimmen zu können.

Um diesen Nachteil auszugleichen, wird in [BL94] erweiternd zur Laufzeitprotokollierung das sogenannte *Program Tracing* beschrieben. Das *Program Tracing* protokolliert jeden

durchlaufenen Knoten: Mit jedem Durchlauf wird ein eindeutiger Wert (ein sogenannter *witness*) in das GBT-Protokoll geschrieben. Es ist auf diese Weise möglich, den Kontrollfluss der Programmausführung vollständig aus dem GBT-Protokoll zu rekonstruieren. Allerdings ist beim *Program Tracing* die Größe des GBT-Protokolls und die Ausführungszeit des Programms proportional. D. h. längere Testausführungen führen zu sehr großen GBT-Protokollen, deren Auswertung zudem sehr aufwändig wird. Als praktikabler Mittelweg wird das *Path Profiling* in [BL96] vorgeschlagen. Auch wenn dieses *Path Profiling* eine praktikable Alternative zum *Program Tracing* bildet und damit das Pfadtesten auch für Programme mit Schleifen umsetzbar wird, gibt es derzeit kein GBT-Werkzeug, das diese Technik nutzt. Eine ausführliche Beschreibung des Algorithmus zum *Path Profiling* befindet sich in [BL96].

3.3.4 Testfallgenaues Ausführungsprotokoll

Die Programmausführung beim Test wird definitionsbedingt durch Testfälle gesteuert. Mit der Eingabe der Testdaten hat die Testfallausführung einen genau definierten Start und mit der Auswertung der Resultate ein genau definiertes Ende. Wenn beim Test die Testfälle nacheinander (d. h. nicht nebenläufig) ausgeführt werden, lässt sich das Ausführungsprotokoll mit einfachen Mitteln auch getrennt für die einzelnen Testfälle bestimmen. Ein solches GBT-Protokoll heißt im Folgenden testfallgenau. Beim „klassischen“ GBT [ZHM97, Li02, AO08] erfolgt die Auswertung der GBT-Überdeckung für die gesamte Testausführung, eine Detaillierung für einzelne Testfälle der Testsuite gibt es nicht. Entsprechend verhalten sich die am Markt verfügbaren GBT-Werkzeuge, die nur für die gesamte Testsuite kumuliert GBT-Überdeckung erheben. Beim testfallgenauen GBT sind neben der Auswertung wie beim „kumulativen“ GBT auch präzise Aussagen zum Überdeckungsverhalten einzelner Testfälle möglich.

Def. Ein GBT-Protokoll heißt **testfallgenau**, *wenn auf die Überdeckung geschlossen werden kann, die sich durch die Ausführung eines einzelnen Testfalls ergibt.*

Testfallgenaue GBT-Protokolle können z. B. dazu genutzt werden, das GBT-Ausführungsverhalten zweier Testfälle zu vergleichen. Praktisch alle Techniken zur Testsuite-Reduktion basieren auf solchen Vergleichen. Auch basieren viele Arbeiten zum selektiven Regressionstest auf einem testfallgenauen GBT-Protokoll. Sneed beschreibt beispielsweise in [Sn04] eine Technik, um ein testfallgenaues GBT-Protokoll beim Test einer großen in C++ geschriebenen Mainframe-Anwendung zu gewinnen. Er nutzt dazu Zeitstempel, die sowohl vom Testtreiber als auch vom instrumentierten Prüfling in je eine getrennte Protokolldatei geschrieben werden. Da durch den Abgleich beider Protokolldateien die Testfälle bekannt sind, die zur Ausführung eines C++-Moduls führen, können daraus Vorschläge für den selektiven Regressionstest gewonnen werden. Gegenüber dem vollständigen Regressionstest gibt Sneed für den auf diese Weise entwickelten selektiven Regressionstest eine Aufwandsreduktion von bis zu 75 % an. Allerdings sind die Werkzeuge, die Sneed bei seinen Arbeiten verwendet, eigenentwickelt und nicht am Markt verfügbar.

Ein weiteres Werkzeug, das testfallgenau das GBT-Protokoll erheben kann, beschreiben Lyu, Horgan und London in [LHL93]. Ihr Werkzeug ATAC, das für viele der Untersuchungen zur Wirksamkeit des GBT in Kapitel 2.7 verwendet wird, erhebt Kontroll- und Datenflussüberdeckung und enthält Funktionen, um die Ausführung einzelner Testfälle zu vergleichen. Jeder Programmlauf – von Programmstart bis Programmende – entspricht der Ausführung für einen Testfall. Eine Einflussnahme durch den Tester ist nicht vorgesehen. So ist es z. B. für Programmstellen, die eine Benutzereingabe erwarten, nicht vorgesehen, dass der Tester die Überdeckung, die durch die weitere Programmausführung entsteht, einem anderen Testfall zurechnen kann. Für den Test einzelner kleiner Funktionen ist diese Einschränkung akzeptabel, für den Test großer Systeme ist dieses Vorgehen aber ungeeignet.

In Kapitel 7.4 werden für das Werkzeug CodeCover und die Programmiersprache Java technische Lösungen vorgestellt, wie das testfallgenaue Laufzeitprotokoll so umgesetzt werden kann, dass der Prüfling zwischen zwei Testfällen nicht beendet werden muss. In dieser Arbeit wird das testfallgenaue GBT-Protokoll in Kapitel 8 zur Unterstützung beim Entwurf neuer Testfälle genutzt.

3.4 Zusammenfassung

Der GBT erscheint heute als eine etablierte und ausgereifte Testtechnik, die auf standardisierten Metriken basiert. Der CFG bildet dabei „De facto“ das GBT-Referenzmodell. Mit den in diesem Kapitel vorgestellten Untersuchungen zeigen sich beim GBT aber erhebliche Mängel:

- Die GBT-Metriken sind keinesfalls einheitlich definiert, und die GBT-Werkzeuge folgen keinem einheitlichen Standard.
- Auf Grundlage des CFG existieren zwar präzise Definitionen der GBT-Metriken, allerdings fehlt eine Vorschrift zur Übertragung der Programme in den CFG.
- Der CFG kann wichtige GBT-relevante Strukturen nicht abbilden, und es gibt heute kein GBT-Modell, das z. B. Ausnahmen oder GBT-relevante Ausdrücke abbilden könnte.
- Prinzipiell korreliert die Erhöhung der GBT-Überdeckung mit der Fehlerentdeckungsquote, als alleinige Technik zum Testdatenentwurf ist der GBT aber ungeeignet.
- Das pure „Bereitstellen“ eines GBT-Werkzeugs für ein Testteam führt keinesfalls zur beabsichtigten Qualitätsverbesserung. Eine Methode, die neben der Erhöhung der GBT-Überdeckung weitere Kriterien der Testdaten berücksichtigt, ist unbedingt erforderlich.
- Das testfallgenaue GBT-Protokoll ist zwar für viele GBT-Auswertungen erforderlich, die am Markt befindlichen Werkzeuge unterstützen diese Funktion allerdings nicht (vom Werkzeug ATAC [LHL93] sowie CodeCover abgesehen).

Die folgenden Kapitel 4 und 5 stellen ein GBT-Modell bereit, das insbesondere bei den genannten Schwächen des CFG deutliche Verbesserungen aufweist. Kapitel 8 behandelt

den methodischen Aspekt des GBT und stellt eine GBT-basierte Testtechnik bereit, die die genannten Resultate der neueren Untersuchungen zum GBT berücksichtigt.

4
Eine Modellsprache für den Glass-Box-Test

In diesem Kapitel wird eine GBT-Modellsprache, die Reduced Program Representation (RPR), entwickelt. Hierzu werden zunächst die Anforderungen an ein GBT-Referenzmodell aufgestellt. Den Ausgangspunkt dieser Anforderungen bilden die Überdeckungsmetriken der Literatur, deren Definition auf Grundlage des GBT-Modells erfolgen soll, sowie Richtlinien zur Zertifizierung sicherheitskritischer Software. RPR wird dann in zwei Abschnitten entwickelt: Zunächst werden die Kontrollstrukturen in RPR abgebildet und anschließend die GBT-relevanten Ausdrücke.

4.1 Einführung

In der Literatur findet man die Definitionen für die Metriken des GBT auf zwei verschiedenen Abstraktionsebenen: Zum einen werden die Metriken auf Grundlage eines Modells des Prüflings definiert. In der Regel werden hierfür der CFG und die Graphentheorie genutzt (vgl. Kapitel 3.1.1). Diese Definitionen sind anschaulich und bezogen auf das Modell präzise. Allerdings werden im CFG viele wichtige Eigenschaften der gängigen Programmiersprachen wie Ausnahmebehandlung oder Ausdrucksauswertung kaum oder gar nicht berücksichtigt, und es fehlt auch eine präzise Vorschrift zur Übertragung der Programme gängiger Programmiersprachen in das Modell. Für die Implementierung eines GBT-Werkzeugs bleiben damit viele Detailvorgaben unpräzise – was sich eindrucksvoll in den verschiedenen Resultaten der GBT-Werkzeuge für gleiche Programmausführung widerspiegelt (vgl. Tabelle 5 auf Seite 46). Der CFG mit den Definitionen der Literatur ist damit als GBT-Referenz nicht geeignet. Auf der anderen Seite werden durch Richtlinien zur Zertifizierung sicherheitskritischer Software sowie Ergänzungen zu diesen Richtlinien die GBT-Definitionen ganz konkret an Beispielen für einzelne Programmiersprachen beschrieben. Auf eine Modellbildung wird dabei weitgehend verzichtet; in [FAA07] wird beispielsweise kein Modell entwickelt, und der CFG wird für Anweisungs- oder Entscheidungsüberdeckung nicht genutzt, während viele spezielle Eigenschaften der Programmiersprachen Ada und C einzeln behandelt werden.

Gegenüber dieser sehr konkreten, auf Einzelbeispiele fixierten GBT-Definition haben GBT-Metriken, die auf Grundlage eines Modells definiert werden, Vorteile: Die Metriken lassen sich besser formal definieren und leichter auf weitere Programmiersprachen übertragen. Zudem kann auf Grundlage eines (knappen) Modells deutlich einfacher eine Theorie entwickelt werden, die z. B. den Nachweis von Subsumtionsrelationen erlaubt oder die Entwicklung einer Strategie zum Testdatenentwurf unterstützt.

Es wird also ein GBT-Modell gesucht, das die Anschaulichkeit und Knappheit des CFG mit der Präzision und Praxistauglichkeit einer Programmcode-basierten Betrachtung verbindet. Damit soll für Programme in strukturierten Programmiersprachen wie COBOL, Pascal, C oder Java im Folgenden die Abbildung in Modelle definiert werden, so dass alle für den GBT irrelevanten Merkmale unterdrückt, die relevanten aber abgebildet werden. Zwei in verschiedenen Sprachen formulierte Fassungen desselben Algorithmus, deren Programmlogik völlig gleich ist, sollen so auf das gleiche Modell abgebildet werden.

Die so entwickelte Modellsprache soll zwei Kriterien erfüllen: Als programmiersprachenunabhängige Darstellung der Programme sollen daran die Definitionen der Überdeckungsmaße festgemacht werden. Und zweitens sollen die für den GBT wichtigen Eigenschaften der strukturierten Programmiersprachen unmittelbar berücksichtigt werden, damit eine Abbildung auf möglichst viele reale Programmiersprachen einfacher und präziser möglich ist, als dies beim CFG der Fall ist.

4.2 Anforderungen an ein Referenzmodell für den Glass-Box-Test

GBT-Modelle der Literatur werden ganz überwiegend in zwei Bereichen genutzt: Zum einen zur Definition von Überdeckungsmetriken, zum anderen zur Formalisierung von Strategien zum Testdatenentwurf. Den Ausgangspunkt der Anforderungsdefinition an ein GBT-Referenzmodell bilden zunächst die Überdeckungsmetriken der Literatur. Im Folgenden werden dann weitere Aspekte genannt und beschrieben, die ebenso wichtige Anforderungen an ein GBT-Referenzmodell bilden.

4.2.1 Überdeckungsmetriken

Bereits in den ersten Artikeln und Büchern zum GBT [Hu75, My79, He84] sowie den einschlägigen Zertifizierungsrichtlinien [RTCA, FAA07, IEC61508, ISO26262] werden für einen vollständigen Test die sogenannte Anweisungs- und die Zweigüberdeckung gefordert. D. h. jede Anweisung soll mindestens einmal ausgeführt werden, und für jede Entscheidungsanweisung soll mindestens einmal jeder Zweig durchlaufen werden.

Auch wenn in der aktuelleren Literatur zum GBT weitere Überdeckungskriterien hinzugekommen sind, bleiben die Anweisungs- und Zweigüberdeckung unverändert die wichtigsten Überdeckungsmetriken im GBT (vgl. Kapitel 2.5). Für beide Metriken stehen zudem empirische Daten zur Wirksamkeit des GBT zur Verfügung. Ein GBT-Modell sollte somit diese beiden Metriken und damit Anweisungen und möglichen Kontrollfluss gut abbilden können. Aber obwohl beide Metriken schon lange in der Literatur behandelt werden und auch „intuitiv“ klar definiert erscheinen, hat sich keine präzise Definition etabliert, die z. B. dazu führt, dass Messungen mit verschiedenen Werkzeugen auch glei-

che Resultate ergeben. Eine solche präzise Definition, die auch für Werkzeughersteller präzise Vorgaben macht, wird in dieser Arbeit entwickelt.

Einige GBT-Werkzeuge liefern Überdeckungswerte für eine sogenannte Blocküberdeckung (vgl. Kapitel 6.3.3), ohne dass die Werkzeuge eine Definition für diese Metrik mitliefern. Tabelle 5 von Seite 46 zeigt zudem, dass die GBT-Werkzeuge keine übereinstimmende Definition der Blocküberdeckung verwenden. Im GBT-Modell sollte mit Blick auf eine knappe Definition der Blocküberdeckung eine einheitliche Modellierung der „Blöcke“ wie z. B. der then- oder else-Blöcke, der Schleifenkörper, catch-Blöcke oder der Prozedurkörper gewählt werden. Dagegen wird die Zeilenüberdeckung (line coverage), die einige Werkzeuge angeben, in dieser Arbeit nicht weiter verfolgt. Es ist weder formal noch intuitiv klar, was unter einer „Zeile“ in diesem Zusammenhang zu verstehen ist.

Neben der Anweisungs- und Zweigüberdeckung spielen die Bedingungsüberdeckungen sowohl in der Literatur als auch in den Zertifizierungsrichtlinien eine große Rolle (vgl. Abschnitt 2.5.3). Die Bedingungsüberdeckungen betrachten zusammengesetzte logische Bedingungsausdrücke, die aus einzelnen atomaren logischen Ausdrücken, den sogenannten Klauseln, bestehen. Ein GBT-Modell sollte diese Komposition von atomaren logischen Ausdrücken somit ebenso abbilden können. Auch folgt diese Arbeit der Argumentation von [FAA07], wonach alle booleschen Ausdrücke im Modell abgebildet werden. In der Literatur (z. B. [AOH02]) werden sonst nur solche booleschen Ausdrücke betrachtet, die das Prädikat einer Verzweigungsstelle bilden.

Um die Schleifenüberdeckungsmetrik (vgl. Kapitel 2.5.4) definieren zu können, ist es für das Modell vorteilhaft, wenn Schleifen so abgebildet werden, dass die Schleifenanweisung mit Wiederholbedingung und Schleifenkörper im Modell als solche zu erkennen sind. Eine Abbildung wie beim CFG, bei der eine Schleife im Modell nur aufwändig durch eine bestimmte Kantenanordnung erkennbar wird, ist mit Blick auf die Metrikendefinition zur Schleifenüberdeckung unpraktisch.

In den schon genannten Zertifizierungsrichtlinien wird bei Programmen mittlerer Sicherheitsrelevanz (z. B. DO-178B, Level B [RTCA, FAA07]) für den Test eine Entscheidungsüberdeckung (decision coverage) gefordert. Die Entscheidungsüberdeckung wird dabei von der Zweigüberdeckung (branch coverage) abgegrenzt und bezieht Ausnahmebehandlung, logische Ausdrücke, die die sogenannte Kurzschlusssemantik nutzen, sowie den sogenannten bedingten Ausdruck zum möglichen Kontrollfluss des Prüflings ein. Damit sollte für diese booleschen und bedingten Ausdrücke eine Abbildung gewählt werden, sodass Zweige des Kontrollflusses und „Zweige“ der logischen Ausdrücke in einer möglichst vergleichbaren Form vorliegen.

Dagegen spielen die verschiedenen Datenflussüberdeckungen (vgl. Abschnitt 2.5.2) in den genannten Zertifizierungsrichtlinien keine Rolle. Die bekannten Probleme einer präzisen Datenflussanalyse [Ost07] durch das sogenannte *pointer aliasing* und den interprozeduralen Datenfluss wirken sich hier aus. Das GBT-Modell dieser Arbeit wird zwar so angelegt, dass es prinzipiell für die Definition von Datenflussmetriken erweiterbar ist, die Datenflussmetriken werden aber nicht behandelt. Tabelle 7 fasst die Überdeckungsmetriken zusammen, die Auswirkungen auf das GBT-Modell haben.

Die Überdeckungsmetriken für Bedingungs- und Entscheidungsüberdeckung sollen generell so angelegt werden, dass bei 100 % Überdeckung das in den genannten Zertifizierungsrichtlinien geforderte Kriterium erfüllt ist.

Überdeckungs-metrik	Kurzbeschreibung und Auswirkung auf das GBT-Modell
Anweisungs-überdeckung	Anteil der ausgeführten Anweisungen. Das Modell definiert den Begriff Anweisung und die Umstände, unter denen eine Anweisung für eine Testfall-ausführung als ausgeführt gilt.
Zweigüberdeckung (branch coverage)	Kantenüberdeckung des CFG [ZHM97, Rie97]. Das Modell definiert die Zweige, die von den Kontrollstrukturen ausgehen, und die Umstände, unter denen die Zweige für eine Testfallausführung als durchlaufen gelten.
Blocküberdeckung	Anteil der ausgeführten Anweisungsblöcke. Das Modell definiert den Begriff Anweisungsblock und die Umstände, unter denen ein Anweisungsblock für eine Testfallausführung als ausgeführt gilt.
Schleifen-überdeckung	Anteil der vollständig ausgeführten Schleifen (vgl. Kapitel 2.5.4, Seite 27). Das Modell definiert den Begriff Schleife und Schleifenkörper und die Umstände, unter denen der Schleifenkörper für eine Testfallausführung als ausgeführt gilt.
Entscheidungs-überdeckung (decision coverage)	Anteil der durchlaufenen Verzweigungen des Kontrollflusses und der „Verzweigungen“ innerhalb der Ausdrücke. Der Definition von [FAA07] folgend werden zusätzlich zu den Verzweigungen des Kontrollflusses die Bedingungsausdrücke mit Kurzschlusssemantik, der bedingte Ausdruck sowie die Ausnahmebehandlung im GBT-Modell berücksichtigt.
Bedingungs-überdeckung	Anteil wirksamer Einzelterme zusammengesetzter boolescher Ausdrücke (vgl. Kapitel 2.5.3). Das Modell definiert primitive und zusammengesetzte boolesche Ausdrücke. Zur Bestimmung der Termüberdeckung nach [LL10] wird die Baumstruktur, die

	den zusammengesetzten booleschen Ausdruck repräsentiert, in das GBT-Modell integriert.

Tabelle 7: GBT-Überdeckungsmetriken und deren Auswirkung auf das GBT-Modell

4.2.2 Ausführungselemente

Jedes Original-Programm wird auf ein GBT-Modell-Programm abgebildet, das aus einer Komposition sogenannter Ausführungselemente besteht. Diese sind entweder primitiv oder strukturiert: Primitiv sind beispielsweise solche Ausführungselemente, die eine Wertzuweisung oder einen Prozeduraufruf darstellen.

Die strukturierten Ausführungselemente, sogenannte Verbundausführungselemente, bilden dagegen Schleifen oder Entscheidungen ab und enthalten als Teil der eigenen Struktur weitere Ausführungselemente. Praktisch repräsentieren Ausführungselemente überwiegend ausführbare Anweisungen der Programme.

Da für einige Überdeckungsmetriken auch die Auswertung logischer oder bedingter Ausdrücke zu beachten ist, werden auch diese durch (strukturierte) Ausführungselemente repräsentiert.

Die Typen der Ausführungselemente, die das GBT-Modell bilden, werden im Folgenden durch Schlussfolgerungen aus den genannten Überdeckungsmetriken und durch die Analyse der Einzelbeispiele von [FAA07] identifiziert und im Folgenden über die Grammatik einer GBT-Modellsprache definiert. Diese Typen der Ausführungselemente wie z. B. primitive Anweisung, Entscheidungsanweisung, Schleifenanweisung oder logischer Ausdruck entsprechen dort den einzelnen Produktionen. Der abstrakte Syntaxbaum eines Programms führt so zu einer hierarchischen Struktur der Ausführungselemente, die für die strukturierten Programmiersprachen auch mit der Ausführungsstruktur korrespondiert. Eine detaillierte Ablaufbeschreibung der Ausführungselemente in Form von Petri-Netzen folgt in Kapitel 5.

4.2.3 Kontrollstrukturen

Da beim GBT der Kontrollfluss des Programms von ganz zentralem Interesse ist und sich die gängigen Programmiersprachen in vielen Eigenschaften, die den Kontrollfluss betreffen, sehr ähnlich sind, werden diese Eigenschaften im Folgenden zusammengetragen und schließlich in einer neuen Modellsprache, der Reduced Program Representation (RPR) zusammengefasst. In der Literatur (z. B. [My79, Bei90, FP97, ZHM97, Rie97]) werden für strukturierte Programmiersprachen übereinstimmend Sequenz, Entscheidung, Fallunterscheidung und Schleife genannt, die den Kontrollfluss bestimmen. Auch praktisch alle Beispiele der Lehrbücher, anhand derer GBT-Überdeckungsmetriken beschrieben werden, nutzen ausschließlich diese sogenannten Kontrollstrukturen, die auch in allen gängigen Programmiersprachen vorhanden sind.

Tabelle 8 zeigt Beispiele dieser Kontrollstrukturen für die Programmiersprachen Java und COBOL. Obwohl diese beiden Programmiersprachen sich in vielen weiteren Eigenschaften erheblich unterscheiden, sind die Kontrollstrukturen recht ähnlich. Auch kennen

beide Sprachen (und viele andere der gängigen Programmiersprachen) Abweichungen von der in Tabelle 8 beschriebenen einfachen Struktur. So gibt es in beiden Sprachen die Option, bei einer Entscheidung den else-Block wegzulassen, und bei Schleifen gibt es Möglichkeiten für Abbrüche (wie z. B. mit break oder continue), die in die Metrikendefinition einbezogen werden sollten.

Kontrollstruktur	Beispiel in der Programmiersprache Java	Beispiel in der Programmiersprache COBOL

Sequenz	{
	System.out.print("Hallo");
	System.out.print("Welt");
	}
		DISPLAY "Hallo".
		DISPLAY "Welt".

Entscheidung	if(a == 0) {
	System.out.print("Hallo");
	} else {
	System.out.print("Welt");
	}
		IF a = 0 THEN
		DISPLAY "Hallo".
		ELSE
		DISPLAY "Welt".
		END-IF.

Schleife	while(a > 0) {
	System.out.print("Hallo Welt");
	a--;
	}
		PERFORM UNTIL A <= 0
		DISPLAY "Hallo Welt".
		COMPUTE A = A – 1.
		END-PERFORM.

Fall-unterscheidung	switch(a) {
	case 1:
	System.out.print("Hallo");
	break;
	case 2:
	System.out.print("Welt");
	break;
	default:
	System.out.print("Hallo Welt");
	}
		EVALUATE A
		WHEN 1
		DISPLAY "Hallo".
		WHEN 2
		DISPLAY "Welt".
		WHEN OTHER
		DISPLAY "Hallo Welt".
		END-EVALUATE

Tabelle 8: Kontrollstrukturen für Java und COBOL

Da sich viele der gängigen Programmiersprachen in diesen für den GBT wichtigen Kontrollflussmerkmalen sehr ähnlich sind, ist es vorteilhaft, eine Modellsprache als Grundlage des GBT-Modells zu entwickeln, die die genannten Kontrollstrukturen und die übereinstimmenden Ausführungsmerkmale möglichst vieler strukturierter Programmiersprachen zusammenfasst.

Der Übergang vom CFG zum Modell dieser Arbeit lässt sich gut mit dem Übergang vom Programmablaufplan zum Struktogramm von Nassi und Shneiderman [NS73] ver-

gleichen. Zwar ist das Struktogramm nicht mehr so flexibel wie der Programmablaufplan, um beliebige Kontrollflüsse abzubilden, dafür werden aber Programme strukturierter Programmiersprachen kompakter und originalgetreuer dargestellt. Der Nachteil der geringeren Flexibilität fällt zudem kaum ins Gewicht, weil die strukturierten Programmiersprachen goto-Programmsprünge nicht zulassen. Nassi und Shneiderman sprechen hier vom „goto-less programming“. Während Programmierrichtlinien für C oder C++ (z. B. [MISRA]) das Verwenden der goto-Anweisungen untersagen, enthält beispielsweise Java bereits keine goto-Anweisung mehr. In diesem Punkt schränkt somit die abstraktere Darstellung des Struktogramms gegenüber dem Programmablaufplan nicht ein.

Unter einer strukturierten Programmiersprache wird in dieser Arbeit nach Nassi und Shneiderman [NS73] eine „goto-less“-Sprache verstanden, die zumindest die Kontrollstrukturen Entscheidung, Fallunterscheidung und Schleife bietet.

4.2.4 Ausnahmebehandlung

Eine wichtige Anforderung an das GBT-Modell bildet auch die Ausnahmebehandlung. In [FAA07, Kapitel 4.2.4] wird festgestellt: *„It is not clear how coverage criteria should be applied to exception handling“*; Ausnahmen bilden jedoch in vielen gängigen Programmiersprachen einen zentralen Bestandteil. Für den CFG gibt es beispielsweise keine Ansätze, Ausnahmen (Exceptions) in einer Form zu behandeln, die auf die gängigen Programmiersprachen angemessen übertragbar wäre.

In den gängigen Programmiersprachen werden die Ausnahmen eines try-Blocks im catch-Block behandelt (in Ada begin- und exception-Block) und, wenn es diesen catch-Block nicht gibt, propagiert. Der CFG kennt aber keine Kompositionsstruktur, die diesem kaskadierten Ablauf beim Behandeln der Ausnahmen entspricht. Ein korrektes Nachvollziehen des Kontrollflusses bei Ausnahmen ist im CFG damit praktisch ausgeschlossen. In der Konsequenz berücksichtigen auch die Definitionen der Überdeckungsmetriken die Ausnahmebehandlung nicht.

Nach [FAA07] ist die Entscheidungsüberdeckung bezogen auf die Ausnahmebehandlung dann erreicht, wenn jeder catch-Block mindestens einmal ausgeführt wird. Es spielt aber keine Rolle, von welchen Anweisungen oder Ausdrücken aus die Ausnahme geworfen wird, da nur die Behandlung der Ausnahme im catch-Block (oder in Ada im Exceptionhandler) relevant ist.

4.2.5 Boolesche Ausdrücke

Beim CFG (z. B. nach [Li02, ZHM97]) werden Anweisungen wie z. B. eine Zuweisung oder ein Prozeduraufruf zu einem atomaren Knoten abstrahiert. Mit Blick auf die Entscheidungsüberdeckung, die auch den Kontrollfluss in Ausdrücken mit Kurzschlusssemantik und dem bedingten Ausdruck betrachtet, ist eine Anweisung aber nicht immer atomar. An einem einfachen Beispiel lässt sich leicht zeigen, dass ein Ausdruck, der z. B. in einer Zuweisungsanweisung verwendet wird, selbst wie ein kleines Programm aufgefasst werden kann. Als Beispiel wird zunächst eine „konventionell“ geschriebene Funktion betrachtet, die bestimmt, ob es sich bei einem gegebenen Jahr um ein Schaltjahr han-

delt. In Abbildung 12 ist eine exemplarische Implementierung als Variante 1 in Java mit dem entsprechenden CFG nach [Li02] angegeben.

Eine vollständige Zweigüberdeckung nach [Li02] ist für diesen CFG mit vier Testfällen erreicht. Beispieleingaben dieser Testfälle, die zu einer Zweigüberdeckung führen, sind im CFG von Abbildung 12 auf die entsprechenden Kanten geschrieben.

```
// Variante 1
boolean istSchaltjahr(int jahr) {
 // Annahme: kein Schaltjahr
 boolean istSchaltjahr = false;

 if(jahr % 4 == 0) {
   if(jahr % 100 != 0) {
     istSchaltjahr = true; // s1
   } else {
     if(jahr % 400 == 0) {
       istSchaltjahr = true; // s2
     }
   }
 }
 return istSchaltjahr;
}
```

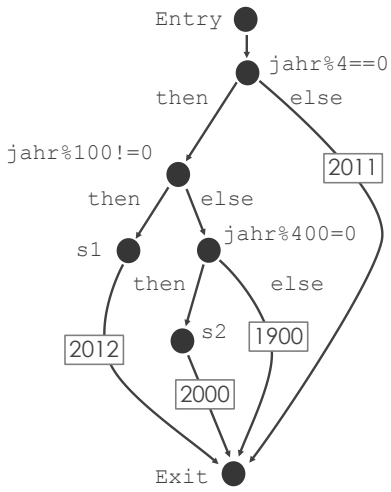


Abbildung 12: Beispiel zur Bestimmung eines Schaltjahrs

Die gleiche Funktion lässt sich aber in vielen gängigen Programmiersprachen (z. B. C, Java oder Ada) auch nach Variante 2 in einem einzigen Ausdruck formulieren:

```
// Variante 2
boolean istSchaltjahr(int jahr) {
  return jahr%4 == 0 && jahr%100 != 0 || jahr%400 == 0;
}
```

Da der &&- und ||-Operator in Java oder C links-assoziativ sind und die Kurzschlusssemantik haben (entsprechend in Ada dem *and then*- und dem *or else*-Operator), entspricht der „Kontrollfluss“ dieses Ausdrucks dem Kontrollfluss des Programmcodes nach Abbildung 12. Bedingt durch die Kurzschlusssemantik geht vom ersten Teilausdruck *jahr%4==0* genauso eine Verzweigung aus wie bei der Entscheidungsanweisung von Variante 1.

Gleiches gilt für den Teilausdruck *jahr%100!=0*. Eine weitere Implementierungsmöglichkeit der Beispielfunktion liefert der bedingte Ausdruck in Variante 3.

```
// Variante 3
boolean istSchaltjahr(int jahr) {
  return jahr%4 == 0 ?
     (jahr%100 != 0 ? true : jahr%400 == 0) : false;
}
```

Die ternären bedingten Ausdrücke stehen in der Programmiersprachen C sowie vielen von C abgeleiteten Programmiersprachen zur Verfügung wie z. B. in Java, C++ oder C#. Auch in diesem Fall entspricht der Kontrollfluss innerhalb des Ausdrucks dem Kontrollfluss des Programmcodes nach Abbildung 12.
Damit wird der CFG nach [ZHM97, Rie97, Li02], der die Ausdrücke nur als atomaren Knoten abbildet, dem tatsächlichen Programmablauf nicht gerecht. Die Autoren von [FAA07] fordern entsprechend folgerichtig das Einbeziehen dieser Ausdrucks-internen Kontrollflüsse in die Entscheidungsüberdeckung. Nach [FAA07, Kapitel 4.2.5] ist ein zusammengesetzter boolescher Ausdruck mit Kurzschlusssemantik dann für die Entscheidungsüberdeckung vollständig ausgeführt, wenn die äquivalente Darstellung nach Tabelle 9 vollständig ausgeführt ist. [FAA07] formuliert hier wie folgt: "*the short-circuit operators … should be considered as splitting a decision in which they occur into separate decisions, each of which must be covered separately.*"

Die Betrachtung eines booleschen Ausdrucks mit Kurzschlusssemantik als Kontrollflussgraph ist in der Literatur zum GBT zwar nicht weit verbreitet, beschreibt aber – wie auch im Beispiel gezeigt – die Ausführungssemantik der realen Programmiersprachen recht treffend. Auch Chilenski und Miller benutzen in [CM94] eine dem Kontrollflussgraphen sehr ähnliche „Object-Code"-Darstellung zur Veranschaulichung der MC/DC und sprechen von einer „Zweigüberdeckung" in diesem Graphen. Der Vorteil dieser Darstellung ist, dass der sonst übliche Modellbruch zwischen Kontrollfluss-Modell und Bedingungsausdrucks-Modell entfällt.

Ausdruck mit Kurzschlusssemantik	Äquivalente Darstellung als Kontrollfluss

if A and then B then …	if A then
	  Temp := B;
	else
	  Temp := False;
	end if;
	if Temp then …

if A or else B then …	if A then
	  Temp := True;
	else
	  Temp := B;
	end if;
	if Temp then …

Tabelle 9: Ausdrücke mit Kurzschlusssemantik nach [FAA07]

Auch Hutchins et al. weisen in [Hut94] darauf hin, dass das Programm

```
if (A && B && C) {
  x=5;
} else {
```

```
  x=10;
}
```

sechs Entscheidungskanten enthält und nicht nur die zwei der Entscheidungsanweisung.

Zusammenfassend ist eine vollständige Entscheidungsüberdeckung dann für einen Ausdruck erreicht, wenn die linken Operanden der Operationen mit Kurzschlusssemantik und die Bedingungen der bedingten Ausdrücke jeweils mit dem Wert true und false ausgeführt werden.

Während bei der Entscheidungsüberdeckung der ausdrucksinterne Kontrollfluss betrachtet wird, zielen die Bedingungsüberdeckungen darauf ab, dass jeder einfache (atomare) boolesche Term mindestens einmal das Resultat des Gesamtausdrucks bestimmt. Für das GBT-Modell bedeutet das, dass die Einzelterme der zusammengesetzten booleschen Ausdrücke abgebildet werden. Für die Termüberdeckung nach [LL10] sind zudem die Auswertesemantik des Ausdrucks und die Werte der Teilausdrücke relevant. Aus diesem Grund wird auch der Baum, der den Ausdruck repräsentiert, in das GBT-Modell mit abgebildet. Die Wurzel dieses Baums repräsentiert den Gesamtausdruck und jeder Teilbaum einen Teilausdruck, die Blätter schließlich die Einzelterme.

4.2.6 Numerische Ausdrücke

Im Gegensatz zu den beschriebenen Kontrollflüssen innerhalb boolescher Ausdrücke spielen für den GBT die Kontrollflüsse in numerischen Ausdrücken keine Rolle – weder in der Literatur noch in den Zertifizierungsrichtlinien. Daher werden in dieser Arbeit numerische Ausdrücke auch nicht weiter behandelt; im GBT-Modell werden sie vollständig unterdrückt. In den realen Sprachen sind allerdings auch Kombination von booleschen und numerischen Ausdrücken möglich. So kann z. B. in Java oder C der bedingte Ausdruck innerhalb eines numerischen Ausdrucks verwendet werden. Das GBT-Modell dieser Arbeit wird dann die numerischen Ausdrucksteile unterdrücken und im Modell nur die relevanten bedingten und booleschen Ausdrücke abbilden.

4.2.7 GBT bei objektorientierten Programmen

Mit Vererbung, Verbergen von Objekt-Interna oder Abstraktion bieten objektorientierte Programmiersprachen vielfältige Mittel zur Entwurfsunterstützung an. Diese haben jedoch keine Auswirkung auf den GBT, weil mit den genannten Mitteln die Entwurfsstruktur des Programms, nicht aber die Programmausführung beeinflusst wird. Daher werden diese Programmkonstrukte im GBT-Modell unterdrückt. Die bei objektorientierten Programmiersprachen übliche dynamische Bindung hat in der Programmausführung aber die Charakteristik einer Fallunterscheidung und kann aus diesem Grund vom GBT berücksichtigt werden.

Binder stellt in [Bin99] hierzu fest: *„Dynamic binding presents a coverage problem“* und fordert, dass für einen gründlichen Test von objektorientierten Programmen alle Überladungen polymorpher Methoden mindestens einmal ausgeführt werden sollen. Beispielhaft seien dazu nach Programmbeispiel 1 die drei Klassen A, B und C in Java-Syntax gegeben. Die Klassen B und C sind sogenannte Kindklassen der abstrakten Klasse A und

liefern jeweils eine Implementierung der Methode *m*. Der Methodenaufruf *a.m()* wird in der Programmausführung so interpretiert, dass dynamisch das durch *a* referenzierte Objekt die aufzurufende Methoden-Implementierung bestimmt. Konkret wird, wenn *a* ein Objekt der Klasse *B* referenziert, die Methoden-Implementierung der Klasse *B* aufgerufen, und entsprechend, wenn *a* ein Objekt der Klasse C referenziert, die Methoden-Implementierung der Klasse *C*.

```
abstract class A {
  abstract void m();
}

class B extends A {
  void m() { ... }
}

class C extends A {
  void m() { ... }
}

// Methodenaufruf:
A a = …
a.m();
```

Programmbeispiel 1: Beispiel eines polymorphen Methodenaufrufs

Solche Methodenaufrufe, die zur Laufzeit dynamisch die Implementierung wählen, werden in der Literatur als polymorphe Methodenaufrufe bezeichnet. Lösch behandelt in [Lö05] ausführlich diese polymorphen Methodenaufrufe, um daraus Fallunterscheidungen zur Generierung von Entscheidungstabellen zu gewinnen. Er ermittelt statisch zu jedem Methodenaufruf den Typ des Empfängerobjektes und mit dieser Klasse als Wurzel den gegebenenfalls ausgehenden Vererbungsbaum. Im Vererbungsbaum werden anschließend die Klassen gesucht, die eine zum betrachteten Aufruf passende Implementierung enthalten. Diese Methoden-Implementierungen werden im Folgenden als Alternativ-Implementierungen bezeichnet, die Klassen, in denen diese Implementierung enthalten ist, als Alternativ-Datentypen. Lösch betrachtet nun jede dieser Alternativ-Implementierungen wie einen Fall einer Fallunterscheidung.
Eine im Sinne des GBT vollständige Ausführung eines polymorphen Methodenaufrufs ist dann erreicht, wenn das Empfängerobjekt eines Methodenaufrufs mindestens einmal jeden der Alternativ-Datentypen annimmt. Lösch zeigt aber auch am Beispiel der Java Standard-Klasse *Iterator*, dass eine solche vollständige Ausführung in vielen Fällen praktisch nicht möglich ist. So gibt es bereits im Java-Standard zu *Iterator* sehr viele Kindklassen (fast 100!) mit entsprechend vielen Alternativ-Implementierungen.

```
Iterator i = ...
while(i.next()) {  ... }
```

Programmbeispiel 2: Polymorpher Aufruf der Klasse Iterator

Damit führt der Aufruf *i.next()* in der typischen Verwendung der Klasse *Iterator* von Programmbeispiel 2 theoretisch zu einer Fallunterscheidung mit fast 100 Fällen, von denen aber praktisch nur einer zur Ausführung kommen kann. Eine GBT-Metrik, die den Anteil der ausgeführten Alternativ-Implementierungen angibt, wäre in diesem Fall nutzlos. In dieser Arbeit werden daher polymorphe Methodenaufrufe nicht weiter behandelt und auch nicht im GBT-Modell berücksichtigt. Gleichwohl ist eine entsprechende spätere Erweiterung des Modells möglich; das skizzierte Problem muss dann aber möglicherweise so behandelt werden, dass zumindest polymorphe Aufrufe, die auf Datentypen aus Bibliotheken basieren, nicht in die Überdeckungsmetrik einfließen.

4.2.8 GBT bei der Behandlung des gegenseitigen Ausschlusses

Zur Behandlung von gegenseitigem Ausschluss (engl. mutual exclusion) bei nebenläufigen Programmen werden in vielen Programmiersprachen sogenannte Mutex-Konstrukte angeboten. In Java steht für diesen Zweck beispielsweise die synchronized-Anweisung zur Verfügung; das Beispielprogramm von Programmbeispiel 3 zeigt eine typische Verwendung dieser synchronized-Anweisung (vgl. [Go05, Kapitel 14.19]).

```
void m() {
  // ...
  Object lock = ...
  synchronized(lock) {
    // ... der kritische Abschnitt ...
  }
}
```

Programmbeispiel 3: synchronized-Anweisung

Der Programmierer bezweckt damit, dass der kritische Abschnitt für ein gewähltes Objekt *lock* gleichzeitig nur von einem Programm-Thread ausgeführt wird. Wenn der kritische Abschnitt für ein *lock*-Objekt von einem Programm-Thread bereits ausgeführt wird, muss ein zweiter Thread mit diesem *lock*-Objekt an der Synchronisationsstelle solange warten, bis der kritische Abschnitt wieder freigegeben wird. Ob aber beim Test eine solche Synchronisation tatsächlich wirksam wird, also an der Synchronisationsstelle Threads tatsächlich auch in einer Warteschlange warten müssen, bis der kritische Abschnitt frei gegeben wird, ist für einen Tester praktisch nicht nachvollziehbar. Auch geben die anderen Überdeckungsmetriken darüber keine Auskunft. Cornett nennt in [Co11] Flaschenhälse und Deadlocks als mögliche Fehlersymptome, die im Zusammenhang mit dem gegenseitigen Ausschluss auftreten können. Dern und Tan weisen in [DT09] darauf hin, dass Tests, auch wenn sie eine vollständige Anweisungsüberdeckung erzielen, auf diese Fehler nicht zwingend prüfen. Die Autoren empfehlen daher eine „Synchronization Coverage“, zu der sie auch eine Technik zur Instrumentierung vorstellen. Die synchronized-Überdeckung gibt damit den Anteil an entsprechend wirksamen synchronized-Anweisungen an, und der Tester kann so gezielt Testfälle entwickeln, die zur nötigen Nebenläufigkeit führen. Da aber die Synchronization-Überdeckung ansonsten in der Literatur kaum behandelt

wird, fließt sie nicht in die Anforderungen an das GBT-Modell dieser Arbeit ein. Eine entsprechende spätere Erweiterung ist aber denkbar.

4.3 Die Reduced Program Representation

Aus den Anforderungen an ein GBT-Modell wird im Folgenden eine strukturierte GBT-Modellsprache, die Reduced Progran Representation (RPR) entwickelt, die für den GBT relevante Merkmale der realen Programmiersprachen abbildet, irrelevante dagegen unterdrückt. Die Syntax der GBT-Modellsprache besteht dazu aus zwei Bereichen: Den Kontrollstrukturen zur Abbildung des „klassischen" Kontrollflusses, die in Abschnitt 4.2.3 bereits behandelt wurden, sowie den Ausdrücken, die für den GBT relevante Merkmale der bedingten und booleschen Ausdrücke der realen Sprachen abbilden. Für diese Strukturelemente der Ausdrücke gibt es allerdings im Gegensatz zu den Kontrollstrukturen kaum Anhaltspunkte aus der Literatur, wie die Modellierung vorgenommen werden soll. Aus diesem Grund wird dieser Teil von RPR schrittweise gebildet und gründlicher hergeleitet als die Strukturelemente des Kontrollflusses, die bereits in der Literatur ausführlich behandelt werden.

4.3.1 Kontrollstrukturen

Im Folgenden werden die für eine GBT-Modellsprache erforderlichen Kontrollstrukturen definiert. Die Grundlage dazu bilden die in Abschnitt 4.2.3 genannten Anforderungen. Von der Ausnahmebehandlung durch die try-Anweisung abgesehen entsprechen die Kontrollstrukturen der GBT-Modellsprache der strukturierten Sprache von Dahl, Dijkstra und Hoare [DDH72]:

- **Programm**: Das Startsymbol eines RPR-Programms bildet *Program*, das vergleichbar einer Prozedur oder Java-Methode aus einem Programmkörper besteht.
- **Anweisung**: Die Anweisungstypen primitive Anweisung, Entscheidung, Fallunterscheidung, Schleife und try-Anweisung werden unterschieden. „Anweisung" bildet dabei die abstrakte Generalisierung dieser konkreten Anweisungstypen.
- **Anweisungsblock**: Der Programmkörper, die then- und else-Blöcke, der Schleifenkörper, die „Fälle" der Fallunterscheidung und der try- sowie die catch-Blöcke sind vom Typ Anweisungsblock, also eine (auch leere) Sequenz von Anweisungen.
- **Entscheidungsanweisung**: Bei der Entscheidungsanweisung steuert ein boolescher Ausdruck die Ausführung des then- und else-Blocks. Dabei ist der else-Block in RPR im Gegensatz zu vielen realen Sprachen nicht optional. Das macht die Definitionen einiger Überdeckungsmetriken einfacher. Wenn im Original-Programm bei einer Entscheidungsanweisung kein else-Block enthalten ist, dann wird im GBT-Modell ein leerer else-Anweisungsblock ergänzt.
- **Schleifenanweisung**: Vergleichbar der Entscheidungsanweisung steuert ein boolescher Ausdruck die Ausführung des Schleifenkörpers. Es wird nur die ablehnende Schleife (direkt) in RPR modelliert. Die annehmende Schleife und die Zählschleife der realen Programme werden auf diese ablehnende Schleife abgebildet. Der Schleifen-

abbruch mit „break“ sowie der Sprung an den Schleifenkopf mit „continue“ werden berücksichtigt.

- **Fallunterscheidung**: Die RPR-Fallunterscheidung enthält beliebig viele case-Blöcke – die „Fälle“ –, sowie einen nicht optionalen default-Block. Falls im Original-Programm kein default-Block angegeben ist, wird im GBT-Modell ein leerer default-Block ergänzt. Da der steuernde Ausdruck der Fallunterscheidung in den gängigen Programmiersprachen kein boolescher Wert ist, wird er im GBT-Modell präteriert.
- **try-Anweisung**: Die RPR-try-Anweisung enthält genau einen try-Block und beliebig viele catch-Blöcke; das entspricht den realen Sprachen, die auch zur Ausnahmebehandlung mehrere der sogenannten Exception-Handler oder catch-Blöcke enthalten können. Weil die in den Programmiersprachen bei catch-Blöcken üblichen Typ-Angaben der zu behandelnden Ausnahmen keine booleschen Ausdrücke sind, werden sie im Modell präteriert.
- **Terminierungsanweisungen:** „throw“, „return“, „break“ und „continue“ sind primitive Anweisungen, die wegen ihrer „Sprungsemantik“ in das GBT-Modell aufgenommen werden.

Die Syntax der RPR wird im Folgenden in einer an die Backus-Naur-Form (BNF) [ISO14977] angelehnten Notation definiert. In Tabelle 10 sind die Notation und die Darstellung der Symbole beschrieben.

Symbol	Bedeutung
Nichtterminal	Nichtterminalsymbol, fett gedruckt
"Terminal"	Terminalsymbol, in Anführungszeichen
=	Produktionszeichen
… | …	Alternative
(…)	Gruppe: Klammert Elemente im mathematischen Sinn (nach [ISO14977])
empty	Die leere Folge von Terminalen
.	Punkt, das Endezeichen der Produktionsregel

Tabelle 10: Syntax der BNF-Sprache

Die Präzedenzen gelten auch nach [ISO14977]; d. h. die Alternative hat eine geringere Präzedenz als die Sequenz der Symbole. Ebenso gilt die Für die genannten Kontrollstrukturen lässt sich damit die Modellsprache RPR wie folgt beschreiben:

```
Program            = StatementBlock.

StatementBlock     = "{" StatementList "}".

StatementList      = Statement StatementList | empty.

Statement          = PrimitiveStatement
                   | TerminateStatement
                   | WhileStatement
                   | IfStatement
                   | SwitchStatement
                   | TryStatement.


IfStatement        = "if" "(" BoolExpression ")"
                     "then" StatementBlock
                     "else" StatementBlock.

WhileStatement     = "while" "(" BoolExpression ")"
                     StatementBlock.

SwitchStatement    = "switch" CaseHandler.

CaseHandler        = "case" StatementBlock CaseHandler
                   | "default" StatementBlock.

TryStatement       = "try" StatementBlock CatchHandler.

CatchHandler       = "catch" StatementBlock CatchHandler
                   | empty.

PrimitiveStatement = "stmt".

TerminateStatement = "throw" | "return" | "break" | "continue".

BoolExpression     = ...
```

Grammatikausschnitt 1: Die Reduced Program Representation (Teil 1)

Die Definition für *BoolExpression* wird im folgenden Abschnitt noch geliefert, und *Statement* wird um eingebettete Ausdrücke, wie sie z. B. bei Zuweisungen auftreten können, erweitert.

4.3.2 Strukturelemente des Ausdrucks

Die Anforderungen an das GBT-Modell der Ausdrücke ergeben sich im Wesentlichen aus der Bestimmung der Entscheidungs- und der Bedingungsüberdeckung. Im GBT-Modell dieser Arbeit werden hierzu nicht nur die Bedingungsausdrücke der if-Anweisungen, sondern entsprechend den Empfehlungen aus [FAA07] alle logischen und bedingten Ausdrücke des Programms berücksichtigt. Bei der Definition der Überdeckungsmetriken kann die Einschränkung auf bestimmte Ausdrücke später aber vorgenommen werden. Auch können in realen Programmen der bedingte Ausdruck und Ausdrücke mit Kurz-

schlusssemantik in praktisch allen Anweisungen genutzt werden. Eine Einschränkung auf den Bedingungsausdruck der if-Anweisungen ist hier ebenso wenig sinnvoll.

Um zu einem möglichst knappen GBT-Modell der Ausdrücke zu gelangen, das den Anforderungen der Überdeckungsmetriken gerecht wird, werden im Folgenden Programmcode-Beispiele der Sprache Java betrachtet, und aus den daraus abzuleitenden Anforderungen wird das GBT-Modell schrittweise entwickelt. Ausgangspunkt der Überlegungen zum GBT-Modell bildet der Programmcode nach Beispiel 1:

```
if(A && B || C) { … }     // Beispiel 1
```

Hier liegt der "klassische" Ausdruck vor, der in der Literatur (z. B. [FAA07, AOH03]) ausführlich behandelt wird. Die Teilausdrücke *A* und *A*&&*B* steuern jeweils einen Operator mit Kurzschlusssemantik, der Gesamtausdruck steuert die Verzweigung der if-Anweisung, und alle Einzelterme tragen zur Bedingungsüberdeckung bei. *A*, *B* und *C* werden als Einzelterme oder auch *Conditions* bezeichnet; Einzelterme sind nach [FAA01]: "*A Boolean expression containing no Boolean operators*".

Da es für den Einzelterm definitionsbedingt keine weitere boolesche Zerlegung gibt, wird er in der Modellsprache als Terminal „expr" angegeben. In einer späteren Erweiterung der GBT-Modellsprache wird für jeden Term ein eindeutiger GBT-Bezeichner eingeführt. Die binären booleschen Operatoren && und || bilden die sogenannten Verbundausdrücke (CompoundExpression) und haben einen ersten und einen zweiten Operanden, dazwischen das Operationssymbol. Bei den booleschen Operationen wird im GBT-Modell in Anlehnung an die Sprache Ada zwischen Kurzschlusssemantik (andThen und orElse) und ohne Kurzschlusssemantik (and und or) unterschieden. Der unäre Negationsoperator kehrt dagegen einen booleschen Wert nur um, hat aber für die weiteren Betrachtungen keine Bedeutung. Die Negation wird daher im GBT-Modell unterdrückt. Die folgende erste Version der RPR-Grammatik für Ausdrücke zeigt Grammatikausschnitt 2.

```
BoolExpression     = Condition | CompoundExpression.

Condition          = "expr".

CompoundExpression = ( "andThen" | "orElse" | "and" | "or" )
                     "(" BoolExpression "," BoolExpression ")"
```

Grammatikausschnitt 2: RPR (Teil 2), Version 1

Da die binären booleschen Operatoren in Java und vielen anderen Sprachen linksassoziativ sind, wird der Ausdruck von Beispiel 1 in RPR nach Grammatikausschnitt 2 als

```
orElse(andThen(expr, expr), expr)
```

geschrieben. Es wird damit in der GBT-Modellsprache die Baumstruktur eines Ausdrucks sichtbar, die faktisch auch im Java-Ausdruck enthalten ist. Erweiternd zur einfachen Struktur von Beispiel 1 enthält Beispiel 2 einen sogenannten eingebetteten Ausdruck:

```
if(A && f(B || C)) { … }     // Beispiel 2
```

Der Teilausdruck *f(B || C)* wird im Kontext des Gesamtausdrucks als Einzelterm verstanden, d. h. für die Bedingungsüberdeckung des Gesamtausdrucks spielt der (eingebettete) Ausdruck *B || C* keine Rolle. Gleichwohl bildet *B || C* selbst einen logischen Ausdruck, der isoliert vom übergeordneten Ausdruck zur Entscheidungs- und Bedingungsüberdeckung beiträgt. Es liegt also mit *f(…)* ein übergeordneter Ausdruck vor, der mit *B || C* einen untergeordneten Ausdruck, einen sogenannten Subausdruck, enthält.

Subausdrücke werden daher in der Modellsprache bei den Einzeltermen (Condition) vorgesehen, also an Blättern des Ausdrucksbaums; hier sind auch beliebig viele Subausdrücke pro Einzelterm möglich. So könnte z. B. die Funktion *f(…)* von Beispiel 2 durchaus mehrere boolesche Ausdrücke als Parameter enthalten. Im Modell wird aus diesem Grund für Einzelterme eine Sequenz von Subausdrücken vorgesehen. Jeder Subausdruck einer solchen Ausdruckssequenz kann isoliert von seinem übergeordneten Ausdruck betrachtet werden und trägt in gleicher Weise wie der übergeordnete Ausdruck zur Entscheidungs- und Bedingungsüberdeckung bei.

Die RPR-Grammatik von Grammatikausschnitt 2 wird in Grammatikausschnitt 3 nun entsprechend weiter entwickelt: Der Einzelterm (Condition) kann mit *SubExpressions* eine Sequenz von booleschen (Sub-)Ausdrücken enthalten.

```
BoolExpression     = Condition | CompoundExpression.

Condition          = "expr" SubExpressions.

CompoundExpression = ( "andThen" | "orElse" | "and" | "or" )
                     "(" BoolExpression "," BoolExpression ")"

SubExpressions     = "[" ExpressionList "]".

ExpressionList     = BoolExpression ";" ExpressionList |
                     empty.
```

Grammatikausschnitt 3: RPR (Teil 2), Version 2

Der Ausdruck von Beispiel 2 wird damit wie folgt in der GBT-Modellsprache abgebildet:

Java	RPR
A && f(B || C)	andThen(expr [], expr [orElse(expr [], expr []);])

Die Grammatik der GBT-Modellsprache ist damit aber noch nicht abgeschlossen, weil der bedingte Ausdruck noch nicht berücksichtigt wird. Das folgende Beispiel 3 zeigt eine in der Praxis typische Verwendung des bedingten Ausdrucks. Abhängig von einer Bedin-

gung (im Beispiel „B“) werden alternative Teilausdrücke wirksam. Während die Bedingung immer ein boolescher Ausdruck ist, können die jeweiligen Alternativausdrücke einen zwar immer gleichen, aber beliebigen Datentyp wie z. B. Ganzzahl, Boolean oder Referenz annehmen, der auch den Datentyp des gesamten bedingten Ausdrucks bildet.

```
A = B ? f() : g();      // Beispiel 3
```

In der Modellsprache wird die Bedingung des bedingten Ausdrucks als BoolExpression modelliert, und da die beiden Alternativausdrücke auch (Sub-)Ausdrücke enthalten können, werden sie als *SubExpressions* modelliert. Zudem wird in die Modellsprache ein neuer Datentyp *Ausdruck (Expression)* als Generalisierung des booleschen Ausdrucks *(BoolExpression)* und des bedingten Ausdrucks aufgenommen. Die *ExpressionList* wird damit derart verändert, dass sie diese „allgemeinen“ Ausdrücke (statt der booleschen Ausdrücke) enthält. Auf diese Weise können die Subausdrücke sowohl aus booleschen, als auch aus bedingten Ausdrücken bestehen. Die Grammatik nach Grammatikausschnitt 4 zeigt diese Veränderungen.

```
Expression          = BoolExpression | ConditionalExpression.

ConditionalExpression = BoolExpression "?"
                        SubExpressions ":" SubExpressions.

BoolExpression      = Condition | CompoundExpression.

Condition           = "expr" SubExpressions.

CompoundExpression = ( "andThen" | "orElse" | "and" | "or" )
                     "(" BoolExpression "," BoolExpression ")"

SubExpressions      = "[" ExpressionList "]".

ExpressionList      = Expression ";" ExpressionList | empty.
```

Grammatikausschnitt 4: RPR (Teil 2), Version 3

Eine weitere und letzte Veränderung der GBT-Modellsprache ist dadurch bedingt, dass in den gängigen Programmiersprachen auch viele Anweisungen wie z. B. Zuweisungen oder Prozeduraufrufe Subausdrücke im Sinne der Modellsprache enthalten können. Beispiel 3 zeigt eine solche Zuweisungsanweisung, deren Semantik im GBT-Modell vom Grundsatz her unterdrückt wird. Neben den Zuweisungsanweisungen können zudem viele Verbundanweisungen Subausdrücke enthalten, die bislang nicht durch die Modellsprache abgebildet sind. So ist z. B. die Wiederholbedingung der Schleife im Modell abgebildet, die Initialisierungs- und Inkrementausdrücke der Zählschleife aber nicht. Auch sind bedingte oder boolesche Subausdrücke im Argument einer Fallunterscheidung in vielen Sprachen möglich.

```
Statement           = ( PrimitiveStatement
                      | TerminateStatement
```

```
                    | WhileStatement
                    | IfStatement
                    | SwitchStatement
                    | TryStatement )
                   SubExpressions.
```

Grammatikausschnitt 5: RPR, Anweisungen mit Subausdrücken

Um diese eingebetteten bedingten und booleschen Ausdrücke abbilden zu können, werden die Anweisungen vergleichbar dem Einzelterm um Subausdrücke erweitert. Grammatikausschnitt 5 zeigt die entsprechende Erweiterung. Damit können die booleschen oder bedingten Ausdrücke, die eine Anweisung des Original-Programms enthält, im Modell als untergeordneter Teil der Anweisung abgebildet werden. Der Ausdruck von Beispiel 3 wird damit wie folgt in der GBT-Modellsprache RPR abgebildet:

Java	RPR
`A = B ? f() : g();`	`stmt [ expr [] ? [] : []; ]`

4.3.3 Bezeichner der Ausführungselemente

Speziell für die Strategien zum Testfallentwurf ist eine verallgemeinerte Betrachtung von Anweisungen, Anweisungsblöcken und Ausdrücken hilfreich. Alle drei haben gemeinsam, dass sie im Verlauf eines Tests zur vollständigen oder teilweisen Ausführung kommen können und so für jedes einzelne Ausführungselement auch ein Grad an Ausführung für eine Testdurchführung angegeben werden kann. Diese Verallgemeinerung bildet der abstrakte Typ *Ausführungselement*, der Anweisungen, Anweisungsblöcke und Ausdrücke abstrahiert. Für viele Auswertungen, die mit dem GBT-Modell vorgenommen werden, ist es praktisch, wenn die Ausführungselemente eindeutig identifizierbar sind. Aus diesem Grund werden die Ausführungselement-Typen der GBT-Modellsprache um einen für das GBT-Modell eindeutigen Bezeichner erweitert.

```
Program           = Identifier StatementBlock.

Statement         = Identifier
                    ( PrimitiveStatement

                    | TerminateStatement

                    | WhileStatement
                    | IfStatement
                    | SwitchStatement
                    | TryStatement )
                    SubExpressions.

StatementBlock    = Identifier "{" StatementList "}".

BoolExpression    = Identifier
                    ( Condition | CompoundExpression ).
```

```
ConditionalExpression = Identifier BoolExpression "?"
                        SubExpressions ":" SubExpressions.

Identifier          = …
```

Grammatikausschnitt 6: RPR, Ausführungselemente mit Bezeichner

Grammatikausschnitt 6 zeigt diese Erweiterung. Gegenüber dem bislang hergeleiteten GBT-Modell ist der *Identifier*, ein Nichtterminal, ergänzt. In den folgenden Beispielen wird dieser Bezeichner durch eine Ganzzahl gebildet, die durch Traversieren des Ableitungsbaums eines RPR-Programms allen Knoten vom Typ Ausführungselement in aufsteigender Folge vergeben wird. Dieser Bezeichner eines Ausführungselements liegt im Originalprogramm nicht vor; d. h. er ist ein abundantes Attribut des GBT-Modells. In den im Folgenden genutzten Beispielen wird der Bezeichner aus einem Buchstaben (*S* für Anweisung, *B* für Anweisungsblock und *E* für Ausdruck) und einer Zahl, gefolgt von einem Doppelpunkt, gebildet. Das folgende Beispiel zeigt eine primitive Anweisung mit dem Bezeichner S1 sowie eine if-Anweisung S2 mit der primitiven Bedingung E1 und den (leeren) then- und else-Blöcken B1 und B2.

```
S1: stmt []
S2: if(E1: expr []) then B1: { } else B2: { } []
```

Identifier für break und continue

In den realen Sprachen können mit break und continue nicht nur die direkt umschließenden Schleifen gesteuert werden, sondern auch weiter außen liegende, die in den realen Sprachen über einen sogenannten Label angegeben werden. Die Abbildung in RPR erfolgt nun derart, dass bei break oder continue statt des „Labels“ des Originalprogramms der Identifier der Schleifenanweisung verwendet wird. Die Grammatik für break und continue wird damit wie folgt verändert:

```
TerminateStatement = "throw" | "return"
                   | "break" Identifier
                   | "continue" Identifier.
```

4.3.4 Beispielprogramm

Beispielhaft soll nun die folgende primitive Java-Anweisung, die einen Ausdruck enthält, in die RPR abgebildet werden.

```
A = B & f("Hallo", C && D, E ? 7 : 9)   // Java
```

Die Zuweisungsanweisung wird als primitive Anweisung „stmt“ modelliert, die einen Subausdruck B && f(…) enthält.

```
S1: stmt        // die primitive Anweisung
```

```
[                 // mit einem booleschen Subausdruck
 E1:              // der Gesamtausdruck B & f(…)
 and(             // &
  E2: expr [],// B
  E3: expr        // f(…)
                  // und jetzt folgen die SubExpressions
  [               // von f(…)
                  // "Hallo"
   E5: andThen(E6: expr [], E7: expr []); // C && D
   E8: E9: expr [] ? [] : [];                // E ? 7 : 9
 ]
);              // die schließende Klammer des and(...)
]               // und das Ende des eingebetteten Ausdrucks
```

Gemäß Kapitel 4.2.5 werden nur boolesche und bedingte Ausdrücke des realen Programms in das GBT-Modell abgebildet, alle anderen Ausdrücke werden unterdrückt – wie im Beispiel die Zeichenkette oder die Ganzzahlen.

Das folgende Beispiel einer Fakultät-Funktion zeigt, wie ein Java-Programm in die RPR transformiert wird.

Java	RPR
`int fakultaet(int n) {`	`P1: B1: {`

`  if(n < 0 || n > MAX) {`	`  S1: if(`
	`  E1: orElse(E2: expr [],`
	`  E3: expr [])`
	`  ) then B2: {`

`  return -1;` `  }`	`  S2: return []` `  } else B3: { } []`
`  int resultat = 1;`	`S3: stmt []`
`  while (n > 1) {`	`  S4: while(E4: expr [] ) B4: {`

`  resultat *= n;`
`  n--;`
`  }`
	`  S5: stmt []`
	`  S6: stmt []`
	`  } []`

`  return resultat;` `}`	`  S7: return []` `}`

Eine vollständige Beschreibung der Transformationsvorschrift von Java-Programmen in GBT-Modell-Programme liefert Kapitel 5.13.

5
Modellnetze

In diesem Kapitel wird die GBT-relevante Ausführungssemantik der einzelnen Sprachelemente der Modellsprache in Form von Petri-Netzen, sogenannten Modellnetzen beschrieben. Zwei wichtige Ziele werden damit verfolgt: Erstens soll für die Übertragung einer realen Programmiersprache in das GBT-Modell eine Ablaufbeschreibung der GBT-Modellelemente vorliegen, aus der präzise hervorgeht, ob und wie sich die reale Programmiersprache in das GBT-Modell abbilden lässt. Und zweitens beschreibt das Ablaufmodell auf Grundlage der Petri-Netze genau, wie die für die Überdeckungsmetriken wichtigen Ausführungszähler im realen Programm platziert werden sollen.

5.1 Einführung

Während die GBT-Modellsprache RPR beschreibt, welche Merkmale der realen Programmiersprachen in das GBT-Modell aufgenommen werden, beschreibt das im Folgenden vorgestellte Ablaufmodell, wie die Ablaufsemantik der Ausführungselemente im GBT-Modell zu verstehen ist. So definiert beispielsweise die Modellsprache, dass es im GBT-Modell eine Schleifenanweisung gibt, die aus einer Wiederholbedingung und einem Schleifenkörper besteht. Das Ablaufmodell definiert nun für diese Schleifenanweisung, wie die Wiederholbedingung ausgewertet wird und wie z. B. die Ausführungssemantik für normales oder abruptes Beenden des Schleifenkörpers zu verstehen ist. Zudem werden die Ausführungszähler präzise innerhalb der Ablaufstruktur platziert, so dass für die Adaption des Modells auf eine reale Programmiersprache eine Spezifikation zur Anordnung der konkreten Ausführungszähler vorliegt. Da die GBT-Metriken auf Grundlage der Ausführungszähler definiert werden, ist die Präzision dieser Spezifikation wichtig.

Die Modellierung der Ablaufsemantik erfolgt in dieser Arbeit durch ein Petri-Netz. Für die verschiedenen Ausführungselemente werden (Teil-)Netze definiert, und aus deren Komposition wird das Gesamtnetz gebildet. Dieses Gesamtnetz modelliert damit das Programm. Gegenüber dem CFG hat diese Modellbildung Vorteile:

- Das Petri-Netz beschreibt mit der Anfangsmarkierung einen initialen Zustand des Prüflings vor der Testfallausführung, und mit der Endmarkierung lässt sich auch das Ende der Testfallausführung präzise formalisieren.
- Petri-Netze kennen durch Vergröberung und Verfeinerung eine Kompositions- und Dekompositionsmöglichkeit.

- Der Nichtdeterminismus von Konflikten des Petri-Netzes abstrahiert das a priori ebenso unbestimmte Ausführungsverhalten von Anweisungen oder Ausdrücken des realen Programms.
- Stellen bilden als Berandung eines Ausführungselements eine präzise Grenze.
- Das Schalten einer Transition definiert präzise ein Ereignis, das z. B. das Inkrement eines Ausführungszählers auslösen kann.
- Das Modell ist prinzipiell auch für nebenläufigen Kontrollfluss nutzbar.

Es wird jedoch nur ein kleiner Teil der Theorie genutzt, die für Petri-Netze zur Verfügung steht. So spielen beispielsweise Lebendigkeits-, Erreichbarkeits- oder Strukturanalysen in dieser Arbeit kaum eine Rolle. Auch sind nicht beliebige Netze zulässig, sondern nur solche, die induktiv aus den (Teil-)Netzen der in der Modellsprache definierten Ausführungselemente wie z. B. Anweisung, Anweisungsblock oder Ausdruck gebildet werden können. Da die Modellsprache RPR wie die realen Sprachen das Programm durch Komposition einzelner Strukturelemente bildet, entsteht auch das Netz des Gesamtprogramms durch schrittweises Einbetten von Teilnetzen. Als Grundlage dieses Kompositionsprinzips werden für die primitiven Ausführungselemente Teilnetze definiert, die als sogenannte Vergröberung in Teilnetzen der Verbundausführungselemente eingebettet werden. So ist z. B. der then- oder else-Block einer Entscheidung als Einbettung eines Anweisungsblocks in die Entscheidungsanweisung zu verstehen. Petri-Netze, die in dieser Weise induktiv aus den Netzen der Ausführungselemente gebildet werden können, heißen GBT-Modellnetze (oder auch kurz Modellnetze).

5.2 Grundlagen der Modellnetze

5.2.1 Grundlagen zu Petri-Netzen

Für die allgemeinen Grundlagen zu Petri-Netzen wird an dieser Stelle auf die entsprechende Literatur [Re85, Re86, Ba90, DE95] verwiesen. In dieser Arbeit werden die sogenannten Stellen/Transitionsnetze betrachtet und im Folgenden auch kurz als *Netze* bezeichnet.

Def. *Ein* **Netz** *ist ein Tripel* $N = (S, T, F)$ *mit*
S *: die Menge der Stellen,*
T *: die Menge der Transitionen* $T \cap S = \emptyset$,
F *: die Flussrelation* $F \subseteq (S \times T) \cup (T \times S)$ *[Ba90]*

Eine wichtige Rolle bildet in dieser Arbeit die Dekomposition eines (Gesamt-)Netzes in Teilnetze.

Def. *Ein Netz* $N' = (S', T', F')$ *ist* **Teilnetz** *des Netzes* $N = (S, T, F)$, *wenn* $S' \subseteq S$, $T' \subseteq T$ *und* $F' = F \cap ((S' \times T') \cup (T' \times S'))$ *[Ba90]*

Def. Der **Rand** *eines Teilnetzes N' (bezüglich des Gesamtnetzes N) sind diejenigen seiner Knoten, die über Kanten mit dem Restnetz verbunden sind.*
Rand(N', N) = { x ∈ S' ∪ T' |(x● ∪ ●x) \ (S' ∪ T') ≠ Ø } Vor- und Nachbereich sind hierbei bezüglich N zu verstehen. [Ba90, Seite 52]

Baumgarten unterscheidet in [Ba90] bei Teilnetzen in einen relativen und einen absoluten Rand. Der absolute Rand sind dabei die Knoten des Teilnetzes, die entweder einen leeren Vor- oder einen leeren Nachbereich haben, also mit dem Restnetz nicht verbunden sind. Da der absolute Rand in dieser Arbeit nicht betrachtet wird, wird statt „relativem Rand" vereinfacht von „Rand" gesprochen.

Die Teilnetze, die in dieser Arbeit eine Rolle spielen, sind so angelegt, dass der Rand ausschließlich durch Stellen (und nicht durch Transitionen) gebildet wird. Man spricht dann von stellenberandeten Teilnetzen.

Def. Ein Teilnetz N' heiß **stellenberandet**, *wenn Rand(N', N) ⊆ S' [Re85, Ba90]*

Zwei Netztransformationen sind in dieser Arbeit wichtig: Die Vergröberung, die ein stellenberandetes Teilnetz durch eine Stelle ersetzt, und die Verfeinerung, die umgekehrt eine Stelle durch ein stellenberandetes Teilnetz ersetzt. Diese Transformationen sind bei Baumgarten [Ba90 S. 58-63] und Reisig [Re85 S. 67 – 74] ausführlich beschrieben. Reisig behandelt hier insbesondere die sogenannte markentreue Verfeinerung, die auch in dieser Arbeit eine große Rolle spielt. Die Markentreue des Teilnetzes ist nach Reisig die Voraussetzung dafür, dass das grobe und das verfeinerte Netz sich dynamisch gleich verhalten.

Zunächst wird die Vergröberung eines Netzes betrachtet, die nach [Ba90] bezüglich des modellierten realen Systems zu einer lokalen Abstraktion führt. Ein stellenberandetes Teilnetz N' eines Gesamtnetzes N wird durch eine Stelle s ersetzt, wobei alle Flüsse zwischen dem Rand(N', N) und N auf s „zusammengezogen" werden.

Def. Seien N = (S, T, F) ein Netz und N' = (S', T', F') ein stellenberandetes Teilnetz von N und $s_{N'} \notin S$ die „neue" Stelle, die N' in N ersetzt [Ba90]. Wir definieren für ∀(x, y) ∈ F \ F'

$$\varphi(x,y) = \begin{cases} (x,y) & wenn\ x,y \in S \cup T \setminus S' \cup T' \\ (x,s_{N'}) & wenn\ y \in Rand(N',N) \\ (s_{N'},y) & wenn\ x \in Rand(N',N) \end{cases}$$

*Dann ist N'' = (S'', T'', F'') das aus N und N' **vergröberte** Netz, mit*
S'' = S \ S' ∪ { $s_{N'}$ }
T'' = T \ T'
F'' = { φ(x, y) | (x, y) ∈ F \ F' }

φ spiegelt nach [Ba90] die Operation wider, die alle Flüsse zwischen dem Rand von N' und seiner Umgebung auf $s_{N'}$ „zusammenzieht".

Bei der Verfeinerung wird eine Stelle $s_{N'}$ durch ein stellenberandetes Teilnetz N' ersetzt. Nach [Ba90] bedeutet die Verfeinerung eine Detaillierung der Modellierung und bildet so die Umkehrung der Vergröberung.

Def. Die Verfeinerung einer Stelle s in N durch ein Teilnetz N' heißt ***markentreu****, wenn N' (nach dem Schalten interner Transitionen) genauso viele Marken abgeben kann, wie s in N [Re85].*

Es dürfen also durch die Verfeinerung weder Marken im System erzeugt noch vernichtet werden. Innerhalb des Teilnetzes dürfen aber beliebig Marken kreiert oder vernichtet werden, solange beim Ein- und Austreten in die Verfeinerung wieder die ursprüngliche Markenanzahl erreicht wird. Bei einer markentreuen Verfeinerung einer Stelle kann in der vergröberten Sichtweise das Teilnetz als einzelne Stelle betrachtet werden. In allen folgenden Betrachtungen der GBT-Modellnetze ist die Kapazität der Stellen nicht beschränkt; damit sind die GBT-Modellnetze immer kontaktfrei (d. h. das Schalten einer Transition wird nie durch eine belegte Stelle im Nachbereich verhindert). Der Nachweis, dass ein (Teil-)Netz markentreu ist, kann dadurch erfolgen, dass die S-Netz-Eigenschaft nachgewiesen wird.

Def. Ein Netz N = (T, S, F) ist nach [DE95] dann ein **S-Netz***, wenn gilt:*
$\forall t \in T: \; | \bullet t | = 1 = | t \bullet |$

Ein Netz ist somit dann ein S-Netz, wenn jede Transition genau eine Eingangs- und genau eine Ausgangsstelle hat. Da in S-Netzen Transitionen weder Marken erzeugen noch Marken vernichten können, sind S-Netze als Teilnetze immer markentreu.

5.2.2 Allgemeine Eigenschaften der GBT-Modellnetze

Das wesentliche Strukturprinzip der GBT-Modellnetze ist die Komposition von (Teil-)Netzen zu einem übergeordneten (Gesamt-)Netz. Ausgehend von elementar definierten Netzen der primitiven Ausführungselemente wie z. B. primitive Anweisung oder Einzelterm werden die Verbundelemente durch schrittweises Einbetten der Teilnetze gebildet. Das Original-Programm wird schließlich durch das Gesamtnetz modelliert. Viele Eigenschaften der (Gesamt-)Netze können so induktiv aus den Eigenschaften der Teilnetze hergeleitet werden. Auch wenn sich – wie aus der Grammatik der GBT-Modellsprache ersichtlich ist – die inneren Strukturen der einzelnen Ausführungselemente unterscheiden, haben die (Teil-)Netze der Ausführungselemente viele Gemeinsamkeiten: Alle (Teil-) Netze sind stellenberandet und haben, wie in Abbildung 13 dargestellt ist, genau eine Eintrittsstelle und mindestens eine Austrittsstelle. Auch wird für jedes Teilnetz der Ausführungselemente der Nachweis geliefert, dass es markentreu ist und damit im übergeordneten Netz zu einer Stelle vergröbert werden kann.

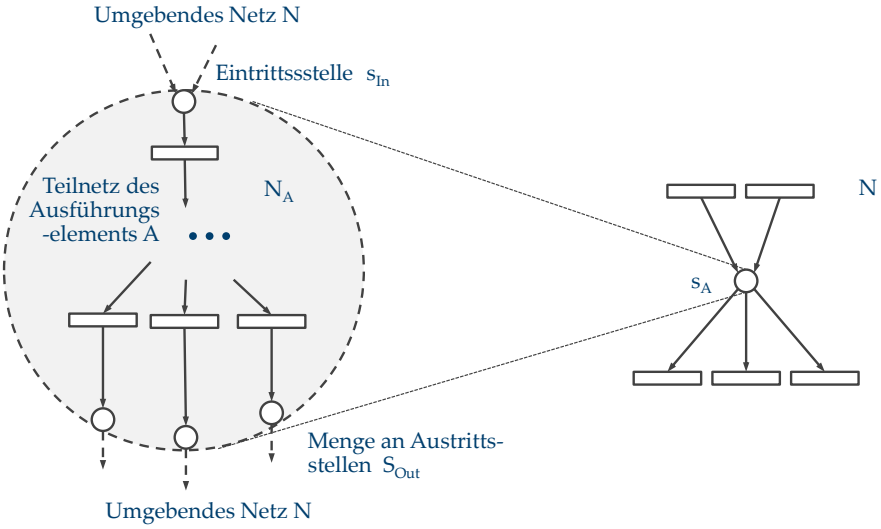


Abbildung 13: Rand des Netzes eines Ausführungselements

Def. Ein **Modellnetz** $N = (S, T, F, s_{In}, S_{Out})$ *eines Ausführungselements ist ein Netz mit*

S *: die Menge der Stellen*

T *: die Menge der Transitionen* $T \cap S = \emptyset$

F *: die Flussrelation* $F \subseteq (S \setminus S_{Out} \times T) \cup (T \times S \setminus s_{In})$

$s_{In} \in S$ *: die Eintrittsstelle,* $\bullet s_{In} \notin T$

$S_{Out} \subset S$*: die Menge der Austrittstellen,* $\forall s \in S_{Out} : s\bullet \notin T$

Dabei wird angenommen, dass das Modellnetz N eines Ausführungselements in ein Gesamtnetz G des Gesamtprogramms eingebettet ist. Insofern sind – wie bei der Definition zum Rand eines Teilnetzes – der Vorbereich der Eintrittsstelle und der Nachbereich der Austrittsstellen bezogen auf das Gesamtnetz zu verstehen. Definitionsbedingt ist dieser Vor- und Nachbereich bezogen auf das (Teil)Netz N leer. Da alle Ausführungselemente mit Ausnahme des Ausführungselements „Programm“ in das Gesamtnetz eingebettet sind, bildet der Vorbereich der Eintrittsstelle den Fluss in das Ausführungselement und der Nachbereich der Austrittsstellen den Fluss aus dem Ausführungselement heraus. Die Eintritts- und Austrittsstellen der Teilnetze werden auch als Randstellen (oder nur als *Rand*) des Netzes eines Ausführungselements bezeichnet.

Eine wesentliche Eigenschaft der GBT-Modellnetze ist, dass Teilnetze, wie beispielsweise das Netz einer primitiven Anweisung, nur an den Randstellen mit dem übergeordneten Netz verbunden werden. Der für die Teilnetze dargestellte „Rahmen“ ist damit so zu verstehen, dass es bei einer Einbettung des (Teil-)Netzes in ein übergeordnetes (Gesamt-)Netz G außer zu den Randstellen keinen Fluss „durch“ diesen Rahmen gibt. Eine wichtige Eigenschaft der Teilnetze ist, dass sie stellenberandet und markentreu sind. Damit können die Teilnetze im übergeordneten Netz zu einer (Super-)Stelle abstrahiert werden, ohne dass sich das Schaltverhalten im übergeordneten verändert [Ra85]. Der Nachweis, dass ein Teilnetz markentreu ist, wird im Folgenden auf zwei Weisen erfolgen: Erstens dadurch, dass das Teilnetz ein S-Netz ist, und zweitens über die Erreichbarkeit. Hierzu darf es für jede (erreichbare) Anzahl von Marken der Eintrittsstelle innerhalb des Teil-

netzes nur Pfade im Erreichbarkeitsgraphen geben, die sicherstellen, dass die gleiche Anzahl von Marken im Austrittsbereich verfügbar wird.

5.3 Anweisungen und Anweisungsblöcke

Das allgemeine Modell für Anweisungen ist das nach Abbildung 14 dargestellte stellenberandete Teilnetz. Zwar wäre ein Kreis als Umrahmung (wie in Abbildung 13 dargestellt ist) für diese stellenberandeten und markentreuen Teilnetze anschaulicher, die Darstellung als Rechteck wie in Abbildung 14 ist aber einfacher, und daher wird im Folgenden die Umrahmung auch mit einem Rechteck vorgenommen. Die Randstellen werden in der Darstellung auf den Rand des Teilnetzes platziert.

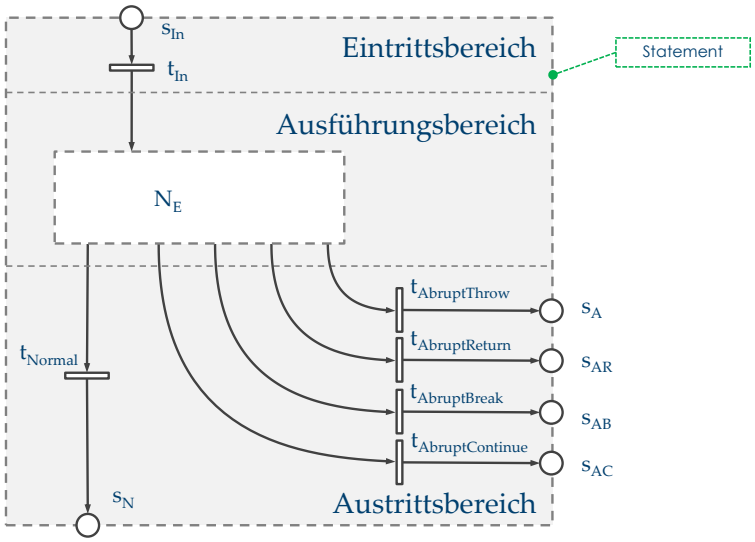


Abbildung 14: Allgemeines Modellnetz einer Anweisung

Das Netz der Anweisung besteht aus drei Abschnitten: Dem Eintrittsbereich, dem Ausführungsbereich und dem Austrittsbereich. Eintrittsbereich und Austrittsbereich sind bei allen Netzen der verschiedenen Anweisungen gleich und bilden mit den Stellen s_{In}, s_N sowie den Stellen für abruptes Beenden s_A, s_{AR}, s_{AB} und s_{AC} den *Rand* des Teilnetzes der Anweisung. Zur besseren Lesbarkeit der Netze werden die Austrittsstellen für abruptes Beenden am rechten Rand des Teilnetzes angeordnet, die Stelle für normales Beenden am unteren Rand. Die Transitionen t_{In}, t_{Normal}, $t_{AbruptThrow}$, $t_{AbruptReturn}$, $t_{AbruptBreak}$ und $t_{AbruptContinue}$ im Eintritts- und Austrittsbereich sind als „Hülle“ um den Ausführungsbereich der Anweisung zu verstehen: Bevor eine Marke in den Ausführungsbereich fließen kann, schaltet die Transition t_{In} im Eintrittsbereich, und, wenn die Marke das Netz des Ausführungsbereichs wieder verlässt, schaltet im Austrittsbereich entweder t_{Normal} oder eine der Transitionen für abruptes Beenden. Diese Transitionen werden im Folgenden auch als Eintritts-Transition bzw. Austritts-Transitionen bezeichnet. Das Schalten der Eintritts- und Austritts-Transitionen wird bei der Erhebung der GBT-Metriken in Kapitel 5.8 genutzt.

Die Austrittsstellen für abruptes Beenden sind so zu verstehen, dass

- in s_A eine Marke gestellt wird, wenn das Ausführungselement durch eine Ausnahme endet,
- in s_{AR} eine Marke gestellt wird, wenn das Ausführungselement durch „return“ endet,
- in s_{AB} eine Marke gestellt wird, wenn das Ausführungselement durch „break“ endet und
- in s_{AC} eine Marke gestellt wird, wenn das Ausführungselement durch „continue“ endet.

Abbildung 14 und Abbildung 15 zeigen das Modell der Anweisung auf verschiedenen Abstraktionsstufen: Abbildung 14 zeigt die Verfeinerung und Abbildung 15 in der linken Darstellung die Abstraktion, die das Teilnetz auf den Rand und den Bezeichner vergröbert, sowie in der rechten Darstellung die Abstraktion, die das Teilnetz zu einer Stelle vergröbert.

Bei der Abstraktion eines stellenberandeten und markentreuen Teilnetzes zu einer einzelnen Stelle wird in der Literatur [Re85, Ba90] von Superstellen gesprochen. Diese Superstellen werden im Modellnetz in einigen Fällen zur besseren Veranschaulichung mit doppelt gezogener Randlinie dargestellt. Superstellen können in der Betrachtung des Netzes aber als „normale“ Stellen angesehen werden. Damit kann z. B. der Erreichbarkeitsbaum für ein Netz unabhängig davon gebildet werden, ob Superstellen im Netz enthalten sind. Der Typ des Ausführungselements wird zur besseren Lesbarkeit durch den gestrichelt umrandeten Text angezeigt. Die vergröberte Darstellung des (Teil-)Netzes der Anweisungen wird im Folgenden dann verwendet, wenn die Anweisung in andere Ausführungselemente eingebettet genutzt wird. Die Abstraktion eines Teilnetzes zu einer Superstelle wird für formale Netzanalysen wie z. B. für den Nachweis der Markentreue genutzt.

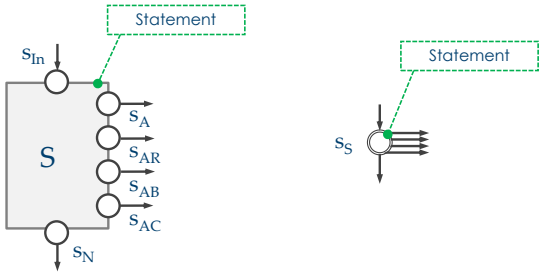


Abbildung 15: Abstrakte Darstellung eines Anweisungs-Netzes

Da die Flüsse für abruptes Beenden für „return“, „break“ und „continue“ in vielen Netzen gleich verlaufen, werden diese zur besseren Übersicht auch gebündelt mit doppelt gezogenem Fluss dargestellt. Die drei Flüsse sind aber immer als einzeln verlaufend mit eigenen Austrittsstellen anzunehmen. Abbildung 16 zeigt diesen gebündelt dargestellten Fluss.

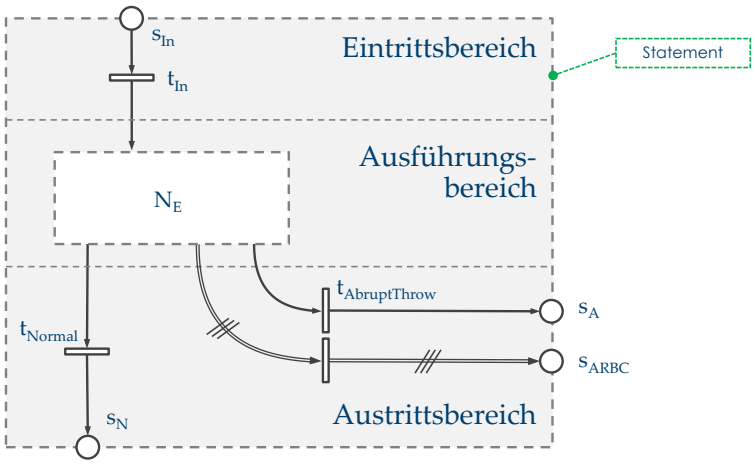


Abbildung 16: Gebündelte Darstellung der Flüsse für abruptes Beenden

Die Stelle s_{ARBC} steht stellvertretend für die drei Stellen s_{AR}, s_{AB} und s_{AC}. Der Fluss für abruptes Beenden mit „throw“, der eine geworfene Ausnahme modelliert, wird dagegen separat dargestellt.

5.3.1 Primitive Anweisung

Das Modellnetz einer primitiven Anweisung modelliert eine Anweisung des realen Programms, die von möglichen Ausnahmen (Exceptions) abgesehen keinen Einfluss auf den Kontrollfluss des Programms hat. Der Ausführungsbereich dieses Netzes abstrahiert die Ausführung der realen Anweisung. Ein Beispiel einer solchen Abstraktion einer Java-Anweisung zeigt Abbildung 17. Der Fluss in dieses Netz N_E erfolgt aus dem Eintrittsbereich, und der Fluss heraus führt in den Austrittsbereich des Modellnetzes der Anweisung. In Abbildung 18 ist das vollständige Modellnetz der primitiven Anweisung dargestellt.

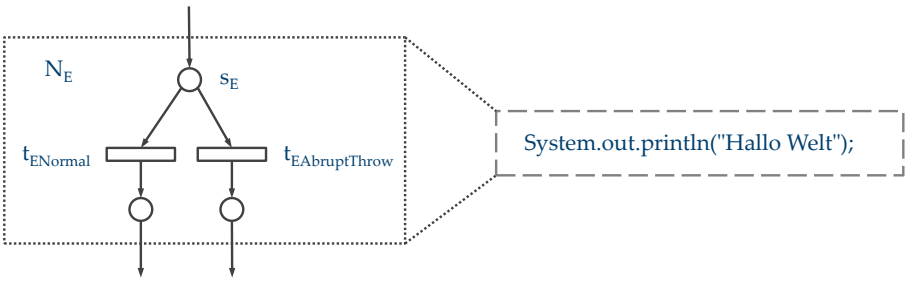


Abbildung 17: Teilnetz zur Abstraktion einer realen Anweisung

Der Konflikt, der durch die beiden Transitionen $t_{ENormal}$ und $t_{EAbruptThrow}$ mit der gemeinsamen Eingangsstelle s_E entsteht, bildet den Nichtdeterminismus des Ausführungsverhaltens der realen Anweisung. Es ist im Allgemeinen nicht entscheidbar, ob die reale Anweisung bei einer Ausführung normal oder abrupt endet.

Im GBT-Modell kann dies aber „beobachtet“ werden, indem bei normalem Beenden der Anweisung die Transition $t_{ENormal}$ schaltet, ansonsten die Transition $t_{EAbruptThrow}$. Damit wird im Modell einer primitiven Anweisung von Abbildung 18 bei Ausführung

mit normalem Beenden der realen Anweisung der Weg über $t_{ENormal}$ und t_{Normal} eingeschlagen, sodass in die Stelle s_N eine Marke gestellt wird. Endet die reale Anweisung abrupt, indem eine Ausnahme geworfen wird, wird in die Stelle s_A eine Marke gestellt.

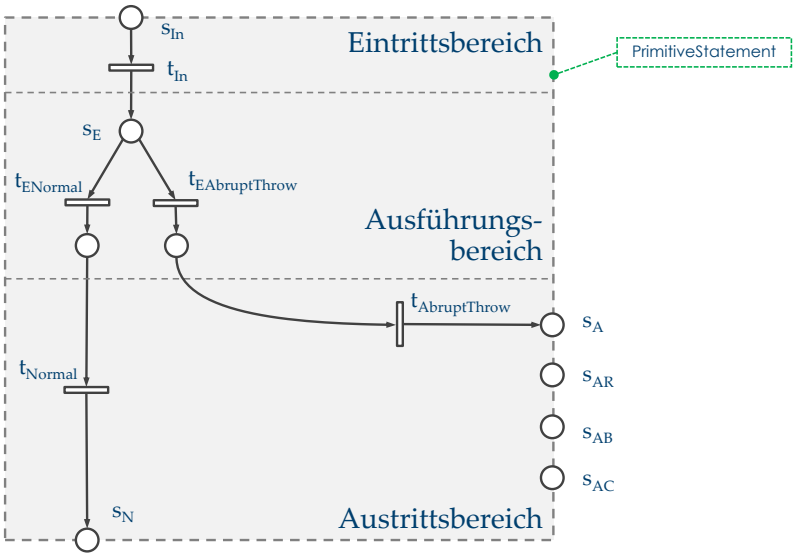


Abbildung 18: Modellnetz einer primitiven Anweisung

Da primitive Anweisungen definitionsbedingt keine anderen Anweisungen enthalten, ist abruptes Beenden mit „return“, „break“ oder „continue“ nicht möglich. Dementsprechend ist auch kein Fluss in diese Austrittsstellen vorgesehen.

Für das Netz nach Abbildung 18 gilt für alle Transitionen t: $|\bullet t| = 1 = |t\bullet|$; damit ist das Netz der primitiven Anweisung ein S-Netz.

Die primitiven Anweisungen realer Programme (wie z. B. Zuweisungsanweisungen) können auch eingebettete boolesche oder bedingte Ausdrücke enthalten. Solche Anweisungen werden in Kapitel 5.5.3 behandelt.

5.3.2 Normales und abruptes Beenden

Ein Ausführungselement endet entweder normal oder abrupt; dies entspricht der Ausführungssemantik der gängigen Programmiersprachen (z. B. vgl. [Go05, Kapitel 14.1]). Für ein Ausführungselement kann es die folgenden Gründe für ein abruptes Beenden geben:

- Bei der realen Ausführung einer (primitiven) Anweisung oder eines (primitiven) Ausdrucks wird eine Exception geworfen.
- Bei der realen Ausführung wird eine der Anweisungen „throw“, „break“, „continue“ oder „return“ ausgeführt.

Zum einen enden diese (primitiven) Anweisungen dann selbst abrupt, zum anderen werden auch die umschließenden Ausführungselemente, wie z. B. ein Anweisungsblock oder eine if-Anweisung, abrupt enden.

Abbildung 19 zeigt stellvertretend für die *TerminateStatements* das Modellnetzt für „return“.

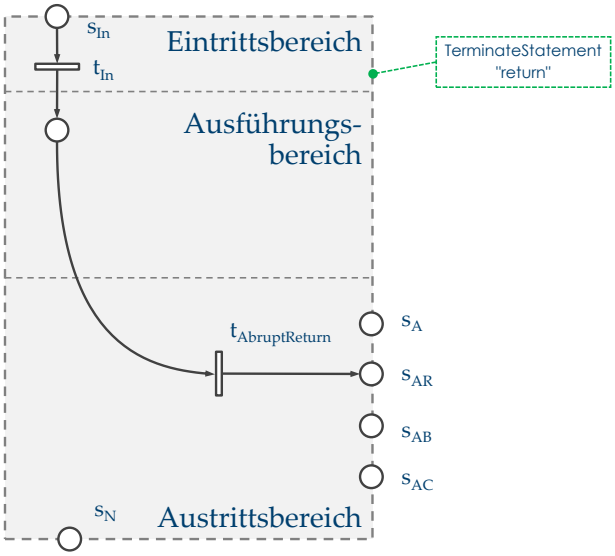


Abbildung 19: Modellnetz für „return“

Die Anweisung „return“ endet definitionsbedingt immer abrupt und stellt bei jeder Ausführung eine Marke in die Austrittsstelle s_{AR}. Die im nachfolgenden beschriebenen Anweisungsblöcke und Verbundanweisungen, die das „return“ umschließen, enden dann ebenso abrupt.

In den gängigen Programmiersprachen kann bei „return“ auch ein Ausdruck als Rückgabewert angegeben werden. Dieser Rückgabewert wird als Subausdruck in Abschnitt 5.5.3 auf Seite 98 behandelt.

Die Terminierungsanweisungen der RPR werden zwar in Java- oder C++-ähnlicher Syntax geschrieben, sollen aber exemplarisch für die entsprechenden Operationen der verschiedenen Programmiersprachen verstanden werden. Wie die RPR-Ausführungssemantik von „break“ und „continue“ zu verstehen ist, wird in Abschnitt 5.7.2 beschrieben.

5.3.3 Anweisungsblock

Anweisungsblöcke bilden die unverzweigte Sequenz von Anweisungen. Ein Anweisungsblock B1 mit *n* Anweisungen hat in RPR die Form

```
B1: { S1: stmt [] S2: stmt []  ... Sn: stmt [] }
```

Einen leeren Anweisungsblock bildet

```
B2: { }
```

Anweisungsblöcke werden im GBT-Modell beispielsweise als Schleifenkörper, then- oder else-Blöcke genutzt. Auch stellt eine Prozedur- oder ein Methodenkörper des realen Programms im Modell einen Anweisungsblock dar. Eine Folge dieser Struktur ist, dass Anweisungen nur in Anweisungsblöcken eingebettet auftreten können und damit aus einer

vollständigen Ausführung der Blöcke auf die vollständige Ausführung der Anweisungen geschlossen werden kann.

Grammatikausschnitt 16 zeigt den Teil der RPR-Grammatik der Anweisungsblöcke.

```
StatementBlock     = "{" StatementList "}".

StatementList      = Statement StatementList | empty.
```

Grammatikausschnitt 7: Grammatikausschnitt von Anweisungsblock

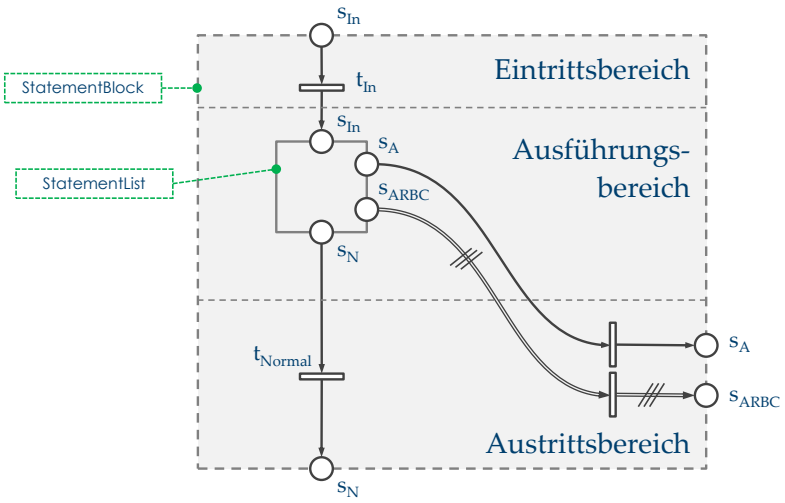


Abbildung 20: Modellnetz des Anweisungsblocks

Das Netz des Anweisungsblocks zeigt Abbildung 20. Eintritts- und Austrittsbereich sind genau gleich wie bei der Anweisung angelegt; auch der Rand des Netzes ist gleich wie bei Anweisungen.

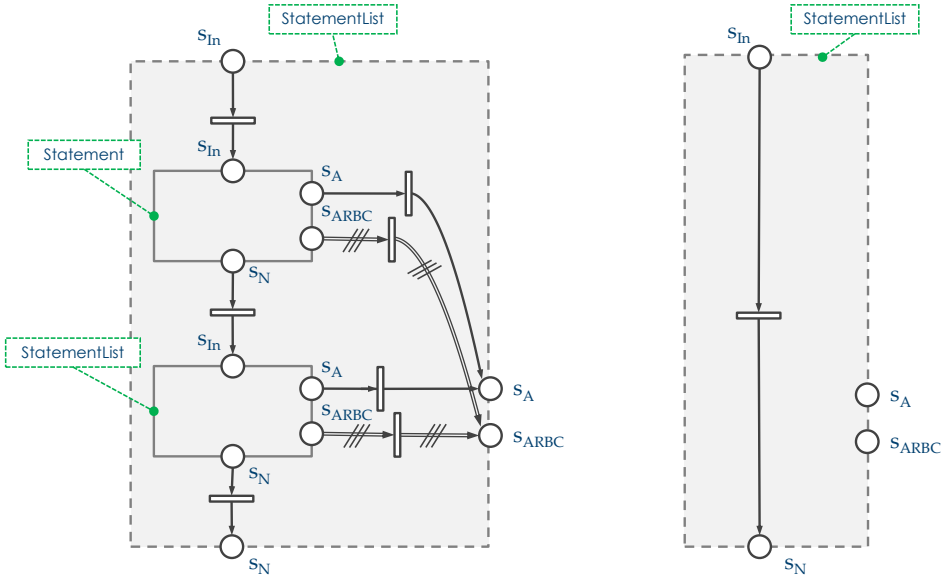


Abbildung 21: Modellnetze von StatementList

Gemäß der RPR-Grammatik besteht ein Anweisungsblock aus einer *StatementList*, die entweder leer ist oder aus einer Anweisung, gefolgt von einer weiteren *StatementList*. Abbildung 21 zeigt die beiden Netze: Links die *StatementList*, bestehend aus einer Anweisung und einer weiteren *StatementList*, und rechts die leere *StatementList*.

Satz: Das Netz eines beliebigen Anweisungsblocks mit beliebigen eingebetteten (Teil-)S-Netzen der Anweisungen ist ein S-Netz.

Beweis: Für alle Transitionen t, die zu den Netzen von Abbildung 20 und Abbildung 21 gehören, gilt: $| \bullet t | = 1 = | t\bullet |$. Da die Einbettung der Anweisung oder der *StatementList* nur am Rand der Teilnetze, also an den Stellen s_{In}, s_N und s_A erfolgt, kann keine Kante zu einer Transition der eingebetteten Teilnetze hinzukommen. Da die Teilnetze der eingebetteten Anweisungen definitionsgemäß S-Netze sind, ist damit auch das (Gesamt-)Netz jedes Anweisungsblocks ein S-Netz. Da für die im Folgenden beschriebenen Netze die Einbettung der Teilnetze immer an den Randstellen erfolgt, wird auch der Nachweis, dass die Netze S-Netze sind, in der gleichen Form stattfinden.

Wie bei den realen Programmiersprachen üblich werden die Anweisungen als Sequenz, d. h. nacheinander ausgeführt. Folglich wird mit der Ausführung der *StatementList* (Abbildung 21, links) erst dann begonnen, wenn die vorausgehende Anweisung normal beendet ist. Endet eine Anweisung abrupt, endet auch die *StatementList* und der Anweisungsblock mit dieser Anweisung abrupt. Ein Anweisungsblock entspricht aber nicht der verzweigungsfreien Folge von Knoten des CFG, weil jede Anweisung des Anweisungsblocks abrupt enden kann. Wenn in einem Anweisungsblock die Anweisungen S_1, S_2 und S_3 nacheinander folgen, gilt nicht, dass aus der Ausführung von S_1 auf die Ausführung von S_3 geschlossen werden kann. Der Beweis ist einfach, denn wenn z. B. S_2 immer abrupt endet (z. B. durch eine Division durch null), wird S_3 nie ausgeführt.

Die Netze der Anweisungsblöcke lassen sich genau wie die der Anweisungen auf die Ränder vergröbern sowie als Superstelle abstrahieren.

5.4 Boolesche Ausdrücke

Die GBT-Modellsprache unterscheidet zwischen booleschen Einzeltermen und zusammengesetzten booleschen Ausdrücken. Grammatikausschnitt 8 zeigt den Ausschnitt der RPR-Grammatik für die booleschen Ausdrücke. Wie bei den Netzen der verschiedenen Anweisungen werden sich die Netze der beiden Ausdruckstypen *Condition* und *CompoundExpression* nur im Ausführungsbereich unterscheiden, d. h. Eintritts- und Austrittsbereich und der Rand der Netze sind gleich; auch die Abstraktion der verschiedenen booleschen Ausdruckstypen sind gleich.

```
BoolExpression     = Identifier
                     ( Condition | CompoundExpression ).

Condition          = "expr" SubExpressions.
```

```
CompoundExpression = ("andThen"|"orElse"|"and"|"or" )
                     ( BoolExpression , BoolExpression ).
```

Grammatikausschnitt 8: Boolescher Ausdruck

Ausdrücke können im Unterschied zu Anweisungen nur durch eine Ausnahme abrupt enden. „return“, „break“ oder „continue“ sind im Ausdruck in den realen Sprachen (und in RPR) nicht möglich. Aus diesem Grund haben Ausdrücke auch nur eine Austrittsstelle für abruptes Beenden.

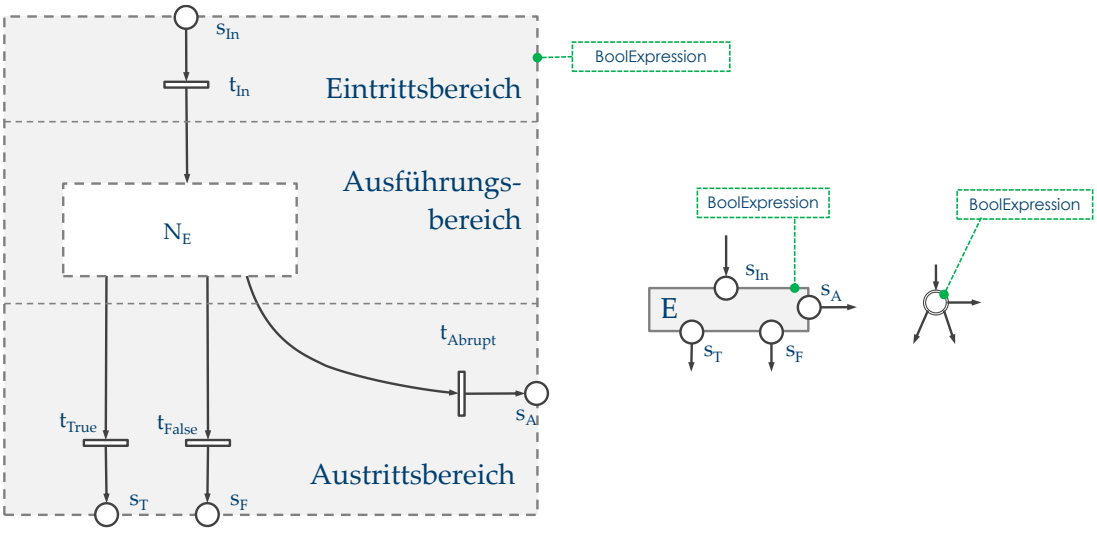


Abbildung 22: Allgemeines Modellnetz eines booleschen Ausdrucks

Das verfeinerte Netz für Ausdrücke nach Abbildung 22 entspricht weitgehend dem Netz der Anweisungen, nur dass für normales Beenden zwei Stellen vorgesehen sind: s_T, s_F für das true- und das false-Resultat des Ausdrucks. So kann der boolesche Ausdruck abhängig vom Resultat eine Verzweigung bilden. Wie bei den Anweisungen ist das Netz N_E, das den Ausführungsbereich des Ausdrucks bildet, bei den verschiedenen Ausdruckstypen unterschiedlich und wird im Folgenden einzeln ausführlich beschrieben. Wie bei den Anweisungen bilden auch beim booleschen Ausdruck die Transitionen t_{In}, t_{True} , t_{False} und t_{Abrupt} eine Hülle um das Netz des Ausführungsbereichs. Jeder Markenfluss in den Ausführungsbereich führt über t_{In}, und jeder Markenfluss aus dem Ausführungsbereich heraus führt über eine der Transitionen t_{True}, t_{False} oder t_{Abrupt}. Für alle (Teil-)Netze der Ausdrücke wird wie bei den Netzen der Anweisungen der Nachweis geliefert, dass die Netze S-Netze oder zumindest markentreu sind.

5.4.1 Einzelterm

Das GBT-Element *Einzelterm* abstrahiert einen booleschen Ausdruck der realen Programmiersprachen, der keine booleschen Operationen und keine Subausdrücke enthält. Das Teilnetz N_E eines Einzelterms, das die Ausführung des realen Ausdrucks modelliert, ist in

Abbildung 23 dargestellt, und das vollständige Modellnetz des Einzelterms ist in Abbildung 24 dargestellt.

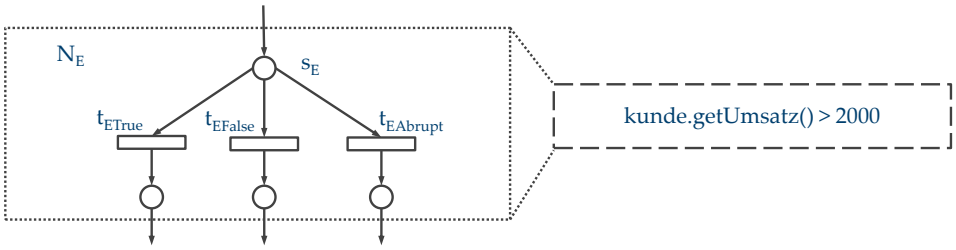


Abbildung 23: Teilnetz zur Abstraktion eines realen Ausdrucks

Der Konflikt, der durch die drei Transitionen t_{EFalse}, t_{ETrue} und $t_{EAbrupt}$ mit der gemeinsamen Eingangsstelle s_E entsteht, bildet den Nichtdeterminismus des Ausführungsverhaltens eines Einzelterms. Es ist im Allgemeinen nicht entscheidbar, welches boolesche Resultat der reale Ausdruck liefert oder ob er abrupt endet. Wie bei den Anweisungen kann im GBT-Modell dies aber „beobachtet" werden, indem abhängig vom Resultat der Ausdrucksausführung eine der drei Transitionen schaltet.

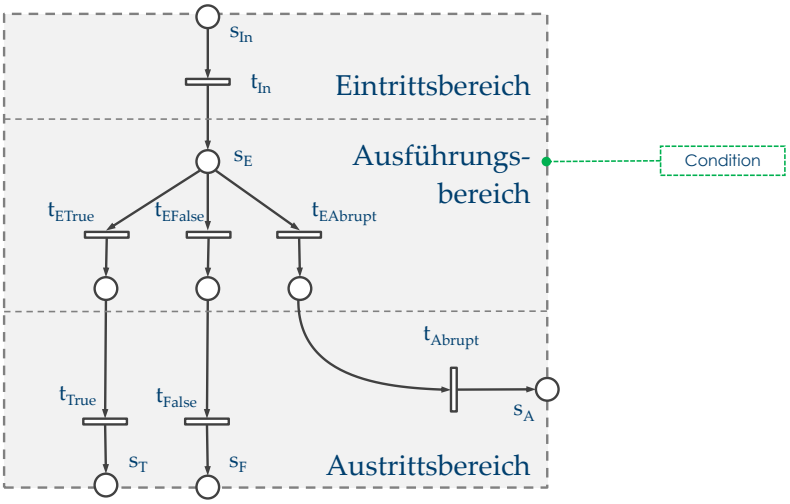


Abbildung 24: Modellnetz eines primitiven booleschen Ausdrucks

Das vollständige Netz eines primitiven booleschen Ausdrucks zeigt Abbildung 24. Wie am Netz von Abbildung 24 leicht zu erkennen ist, hat jede Transition genau eine Eingangs- und genau eine Ausgangsstelle. Das Netz des Einzelterms ist damit ein S-Netz.

5.4.2 Zusammengesetzter boolescher Ausdruck (mit Kurzschlusssemantik)

Genau wie bei den gängigen Programmiersprachen können boolesche Ausdrücke des GBT-Modells über binäre Operationen strukturerhaltend zu zusammengesetzten Ausdrücken verbunden werden. Das neu gebildete Netz eines zusammengesetzten Ausdrucks hat damit wieder den gleichen Rand wie die Netze der eingebetteten booleschen Ausdrücke. Entsprechend können Netze von zusammengesetzten Ausdrücken selbst als Teilnetze in weiteren Einbettungen verwendet werden. Den Netzen der zusammengesetzten

Ausdrücke liegt im Folgenden die Annahme zugrunde, dass, wie in Java, eine definierte Ausführungsreihenfolge der beiden eingebetteten Ausdrücke vorliegt. [Go05, Kapitel 15.7.1] definiert beispielsweise *„The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated"*. In der Modellsprache wird die Und-Operation mit Kurzschlusssemantik als

```
E: andThen(E1: expr [], E2: expr [])
```

formuliert; der Vereinfachung wegen werden in diesem Beispiel die beiden Teilausdrücke als Einzelterme angenommen. Der Gesamtausdruck bildet ein „eigenes" Ausführungselement (im Beispiel mit dem Bezeichner E). Abbildung 25 zeigt das Teilnetz der andThen-Operation.

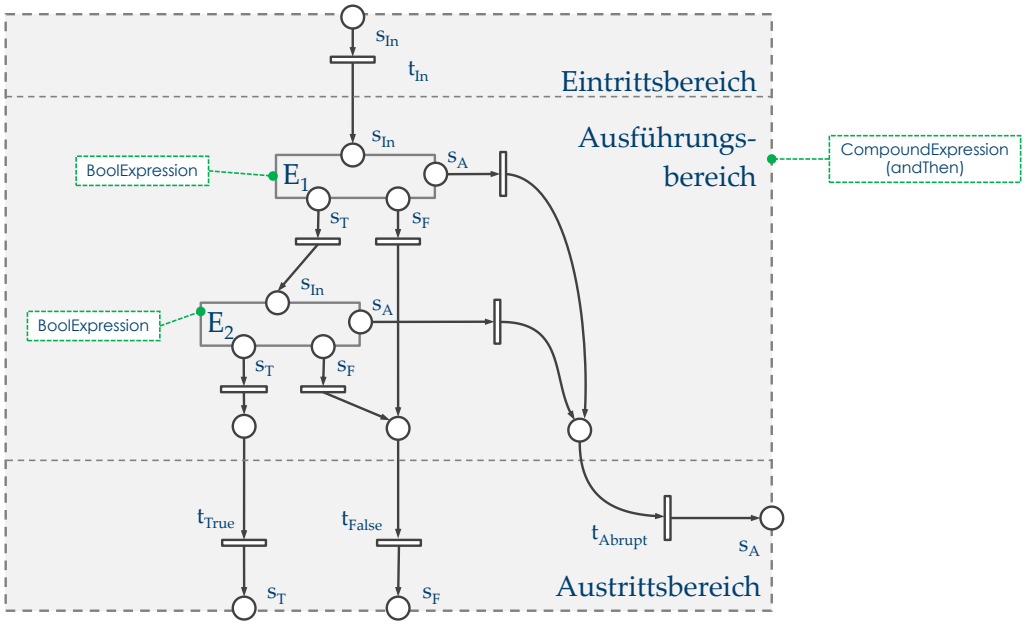


Abbildung 25: Modellnetz einer Operation mit Kurzschlusssemantik

Eintritts- und Austrittsbereich des Netzes eines zusammengesetzten Ausdrucks entsprechen genau dem des allgemeinen Ausdrucks von Kapitel 5.4 sowie auch dem des Einzelterms. Abbildung 25 zeigt auch die Fallunterscheidung der Kurzschlusssemantik: Der Teilausdruck E_2 wird nur wirksam (d. h. eine Marke wird in die Eingangsstelle von E_2 gestellt), wenn E_1 true ergibt und damit eine Marke in die Ausgangsstelle s_T von E_1 stellt. Das Netz der orElse-Operation (orElse(E1, E2)) ist im Prinzip gleich aufgebaut wie das Netz der andThen-Operation. Der Teilausdruck E_2 wird aber nur dann wirksam (d. h. eine Marke wird in die Eingangsstelle von E_2 gestellt), wenn E_1 false ergibt und damit eine Marke in die Ausgangsstelle s_F von E_1 stellt; d. h. es geht vom Netz des ersten Operanden genauso eine Verzweigung aus.

Satz: Das Netz des zusammengesetzten Ausdrucks für beliebige eingebettete (Teil-)S-Netze der Operanden ist ein S-Netz.

Beweis: Genau wie beim Anweisungsblock bildet das Netz des zusammengesetzten Ausdrucks mit den Randstellen der eingebetteten Teilnetze ein S-Netz. Da per Definition die eingebetteten Teilnetze S-Netze sind, ist auch das Gesamtnetz ein S-Netz.

5.4.3 Zusammengesetzter boolescher Ausdruck (ohne Kurzschlusssemantik)

Das Modellnetz eines zusammengesetzten Ausdrucks mit einem Operator ohne Kurzschlusssemantik zeigt Abbildung 26. Für normales Beenden des ersten Operanden wird unabhängig von dessen Resultat immer auch der zweite Operand ausgeführt. Damit wird das Netz der and-Operation auch deutlich größer als das Netz der andThen-Operation. Grund dafür ist die Tatsache, dass mit dem Beenden des zweiten Operanden das Resultat des ersten Operanden zur Bestimmung des Gesamtresultats berücksichtigt werden muss – bei der andThen- bzw. orElse-Operation ist dies nicht erforderlich. Um nach Beenden des zweiten Operanden das Gesamtresultat zu bestimmen, werden die Stellen s_{E1T} und s_{E1F} ausgewertet, die das Resultat des ersten Operanden widerspiegeln. Die folgenden Schaltvorgänge sind hierzu möglich:

- t_{AE1F} oder t_{AE1T} schaltet, wenn E_2 abrupt endet, und „sammelt“ dabei die Marken von s_{E1T} oder s_{E1F} ein (ohne dass das Resultat von E_1 dann eine Rolle spielt)
- t_{TE1T} schaltet, wenn E_1 und E_2 das true-Resultat liefern
- eine der Transition t_{TE1F} , t_{FE1T} oder t_{FE1F} schaltet, wenn entweder E_1 oder E_2 (oder beide) das false-Resultat liefern.

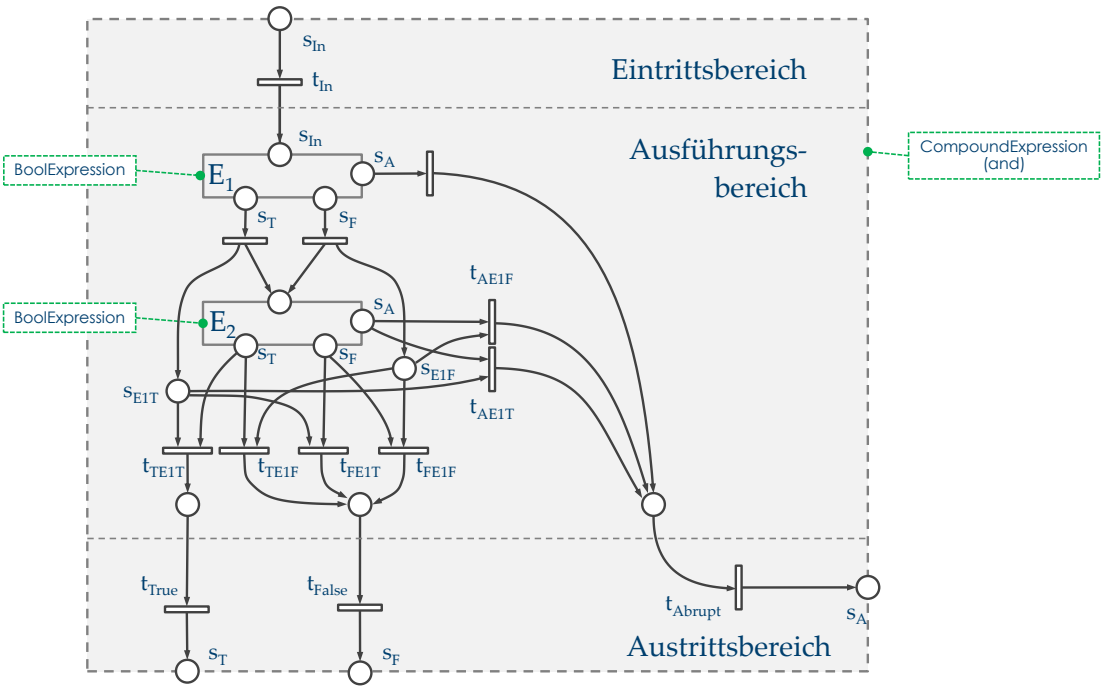


Abbildung 26: Modellnetz der and-Operation

Abbildung 27 zeigt das Modellnetz der and-Operation, wobei die eingebetteten Teilnetze der Operanden als Superstellen abstrahiert sind. Das Netz des and-Ausdrucks ist, wie man leicht sieht, kein S-Netz. Der Nachweis, dass das Teilnetz aber markentreu ist, kann aber über die Erreichbarkeit erfolgen. Über den Erreichbarkeitsbaum des Netzes von Abbildung 27 lässt sich zeigen, dass für jede Markierung der Eintrittsstelle s_{In} nur Endmarkierungen existieren, bei denen genau eine der Austrittstellen s_T, s_F oder s_A markiert ist. Damit ist das Netz markentreu und kann selbst wieder zu einer Stelle abstrahiert werden.

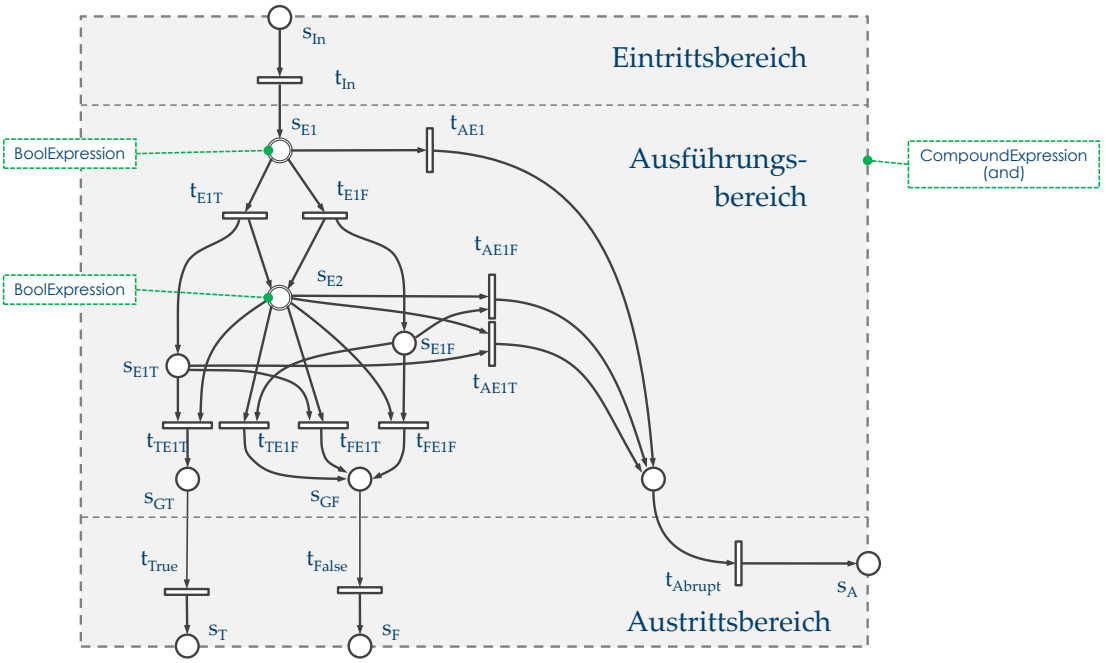


Abbildung 27: Modellnetz der and-Operation mit Superstellen

5.5 Subausdrücke

Eingebettete Subausdrücke verwendet RPR für Einzelterme und Anweisungen sowie für die Alternativausdrücke des bedingten Ausdrucks. Den betreffenden Ausschnitt der RPR-Grammatik für die Subausdrücke zeigt Grammatikausschnitt 9.

```
SubExpressions     = "[" ExpressionList "]".

ExpressionList     = Expression ";" ExpressionList | empty.
```

Grammatikausschnitt 9: Grammatikausschnitt für Subausdrücke

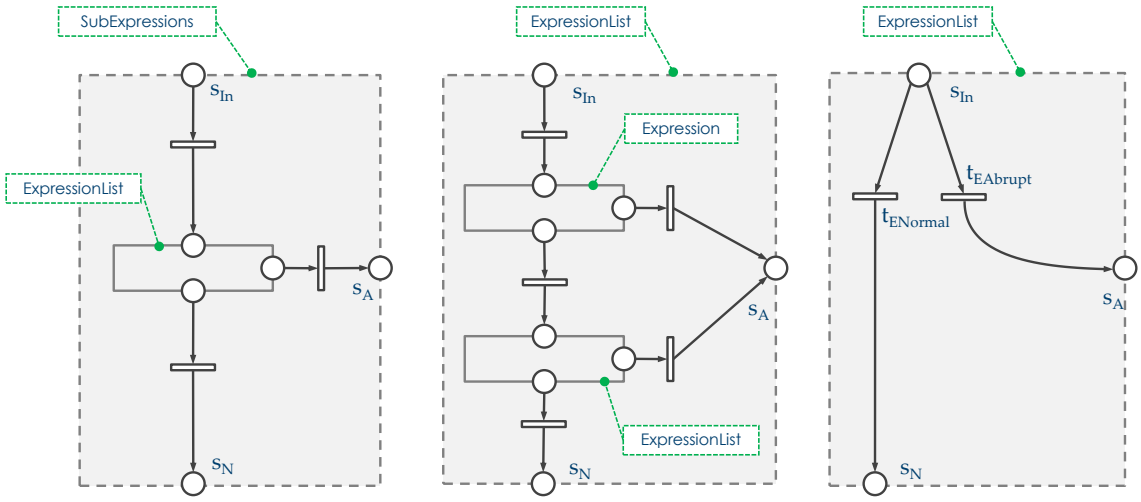


Abbildung 28: Modellnetze von SubExpression und ExpressionList

Der Aufbau und die Ausführungssemantik der in *SubExpressions* eingebetteten *ExpressionList* entspricht weitgehend der *StatementList* (Abbildung 21 auf Seite 89); lediglich die eingebetteten Ausführungselemente haben einen anderen Typ: die *StatementList* bildet eine Liste von Anweisungen, die *ExpressionList* eine Liste von Ausdrücken. In Abbildung 28 (links) ist das Modellnetz von *SubExpressions* dargestellt, das lediglich eine *ExpressionList* enthält. Das mittlere und das rechte Modellnetz zeigen die beiden möglichen Netze für *ExpressionList:* Im mittleren Netz enthält *ExpressionList* eine *Expression*, gefolgt von einer weiteren *ExpressionList*. Das rechte Netz bildet den Abschluss der *ExpressionList*.

Ausdrücke (Expressions) sind in RPR boolesche oder bedingte Ausdrücke, alle anderen Ausdrücke des Originalprogramms werden präteriert. Das folgende Programmbeispiel zeigt einen solchen präterierten Originalausdruck:

```
if( eineMethode( a / b ) ) { ...  }
```

Übertragen in RPR ergibt sich:

```
S1: if ( E1: expr [] ) { ... }
```

Wie man sieht, wird der numerische Original-Teilausdruck „a / b“, der zu einer Division durch null führen kann (und folglich zu einem abrupten Beenden des Gesamtausdrucks), im Modell nicht abgebildet. Da aber dennoch abruptes Beenden durch eine Exception möglich ist, hat das rechte Netz von Abbildung 28, das die „leere“ *ExpressionList* modelliert, dennoch einen Fluss für abruptes Beenden. Bei der *StatementList* ist dies für den leeren Anweisungsblock nicht erforderlich, weil alle Original-Statements im Modell abgebildet werden, sodass nicht mit einer Ausnahme „aus dem Nichts“ zu rechnen ist.

Die Ausdrücke der *ExpressionList* werden auch wie die Anweisungen der *StatementList* in fest definierter Reihenfolge ausgeführt. [Go05, Kapitel 15.7.4] definiert hierzu beispielsweise für die Argumente eines Methodenaufrufs: *“ … each argument expression appears to be fully evaluated before any part of any argument expression to its right”*. Auch das Ablaufverhalten bei abruptem Beenden ist bei *StatementList* und *ExpressionList* gleich: Endet

einer der Ausdrücke der *ExpressionList* abrupt, enden die *ExpressionList* und die *SubExpressions* abrupt, d. h. auf den abrupt endenden Ausdruck folgende Ausdrücke werden nicht ausgewertet. [Go05, Kapitel 15.7.4] definiert hierzu für die Argumente eines Methodenaufrufs: *„If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated".*

5.5.1 Allgemeiner Ausdruck (Expression)

Der allgemeine Ausdruck (Expression) bildet eine Generalisierung von booleschem und bedingtem Ausdruck. Den betreffenden Ausschnitt der RPR-Grammatik zeigt Grammatikausschnitt 10.

```
Expression          = BoolExpression | ConditionalExpression.
```

Grammatikausschnitt 10: Grammatikausschnitt für Expression

Falls ein boolescher Ausdruck als Subausdruck auftritt (linkes Netz von Abbildung 29), spielt das Resultat des Ausdrucks in diesem Zusammenhang keine Rolle; dementsprechend münden die beiden Ausgänge für true- und false-Resultat in dieselbe Austrittsstelle.

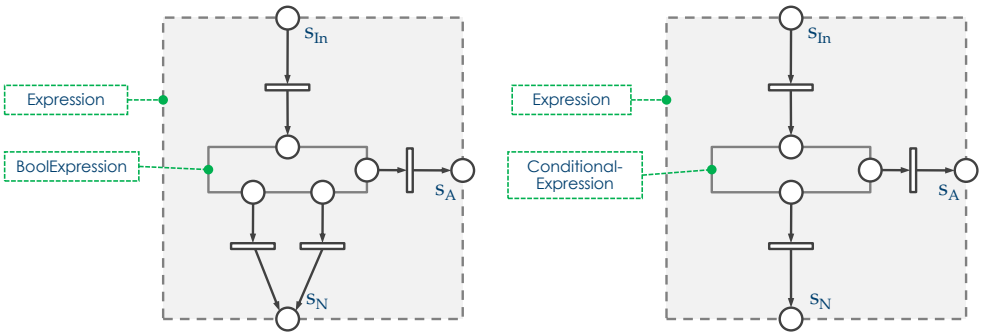


Abbildung 29: Modellnetz von Expression

5.5.2 Einzelterm mit Subausdrücken

Wie in Abschnitt 4.3.2 auf Seite 71 ausführlich begründet wurde, kann ein Einzelterm (Condition) Subausdrücke enthalten. Den betreffenden Ausschnitt der RPR-Grammatik zeigt Grammatikausschnitt 11.

```
Condition           = "expr" SubExpressions.
```

Grammatikausschnitt 11: Grammatikausschnitt für Einzelterme mit Subausdrücken

Im Modellnetz von Abbildung 30 ist das Netz der Subausdrücke in das Netz des primitiven Ausdrucks ergänzt. Die ergänzten Netzteile sind hervorgehoben dargestellt. Das Modellnetz der Subausdrücke liegt vor der Ausdrucksauswertung. Endet einer der Subausdrücke abrupt, wird kein Resultat des Gesamtausdrucks bestimmt und der Gesamtausdruck endet ebenso abrupt.

Das Modellnetz des Einzelterms mit Subausdrücken ist, wie an leicht zu erkennen ist, ein S-Netz.

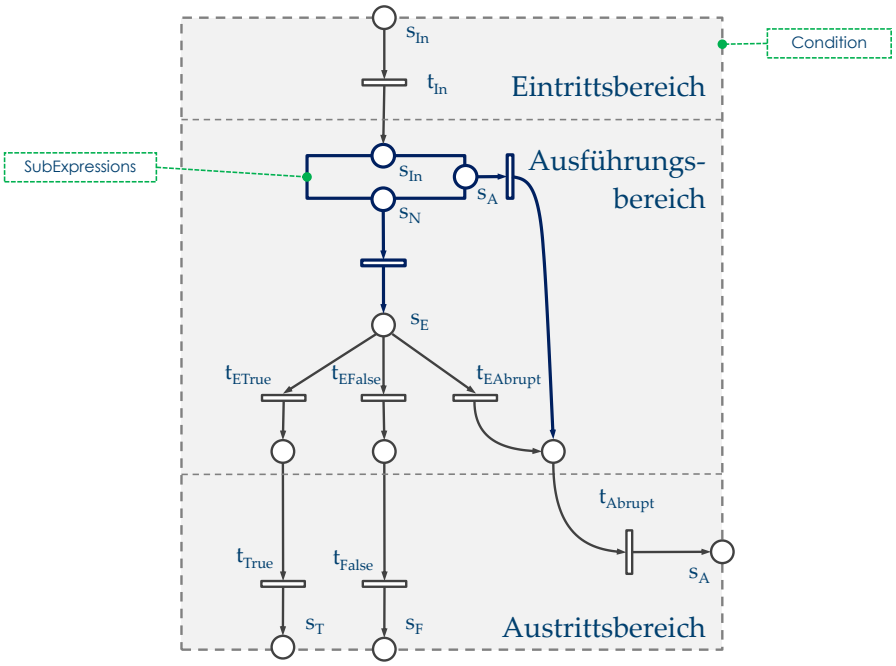


Abbildung 30: Modellnetz eines Einzelterms mit Subausdrücken

5.5.3 Anweisungen mit Subausdrücken

Wie die Einzelterme kann prinzipiell jede RPR-Anweisung eingebettete Subausdrücke enthalten. Grammatikausschnitt 12 zeigt den entsprechenden Teil der Grammatik.

```
Statement           = Identifier
                      ( PrimitiveStatement
                      | TerminateStatement
                      | WhileStatement
                      | IfStatement
                      | SwitchStatement
                      | TryStatement )
                      SubExpressions.
```

Grammatikausschnitt 12: Grammatikausschnitt von Statement

Abbildung 31 zeigt das so erweiterte Modellnetz der primitiven Anweisungen mit Subausdrücken. Die ergänzten Subausdrücke sind hervorgehoben dargestellt. Subausdrücke einer Anweisung werden wie beim Einzelterm immer vor der Anweisung selbst ausgeführt. Das entspricht auch der Ausführungssemantik der gängigen Sprachen (vgl. z. B. [Go05, Kapitel 15.7.4]).

Die Erweiterung der Anweisungs-Netze um Subausdrücke erfolgt für alle RPR-Anweisungen genau gleich, wie es hier für die primitive Anweisung beschrieben ist.

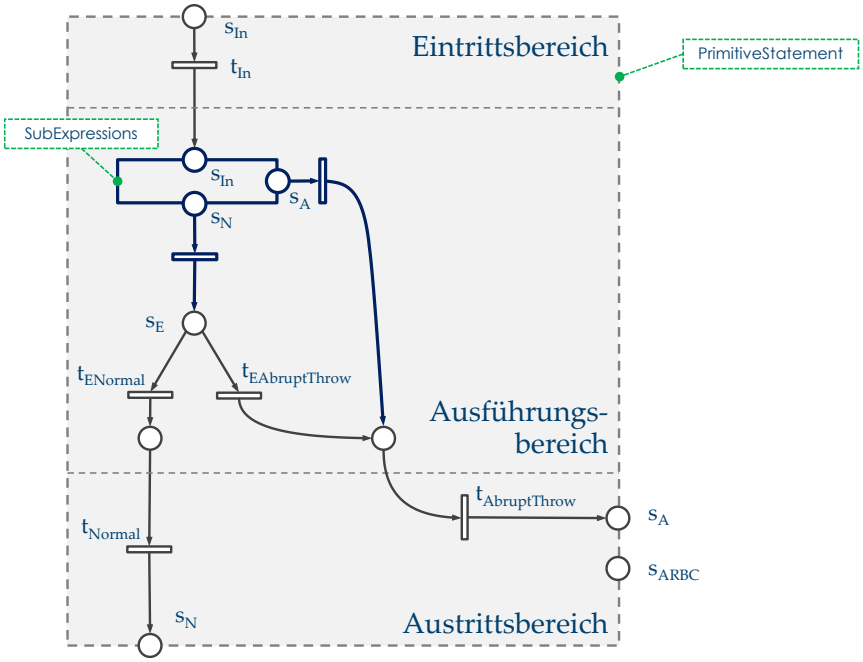


Abbildung 31: Modellnetz der primitiven Anweisung mit Subausdrücken

5.6 Bedingter Ausdruck

Der bedingte Ausdruck besteht aus der Bedingung und den beiden Alternativausdrücken. Grammatikausschnitt 13 zeigt diesen Teil der RPR-Grammatik.

```
ConditionalExpression = Identifier BoolExpression "?"
                        SubExpressions ":" SubExpressions.
```

Grammatikausschnitt 13: Grammatikausschnitt des bedingten Ausdrucks

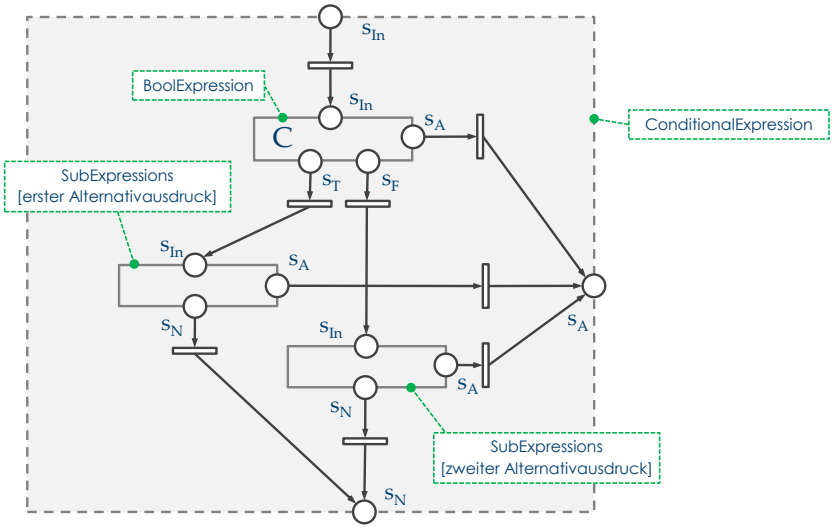


Abbildung 32: Modellnetz des bedingten Ausdrucks

Der Gesamtausdruck endet abrupt, wenn die Bedingung oder der ausgeführte Alternativausdruck abrupt endet, ansonsten endet der bedingte Ausdruck normal. Das Resultat des bedingten Ausdrucks spielt für den GBT keine Rolle; der bedingte Ausdruck bildet damit auch keinen booleschen Ausdruck, sondern „nur“ einen *Ausdruck*, d. h. *Expression*, und hat auch im Modellnetz nach Abbildung 32 für normales Beenden nur eine Ausgangsstelle.
Mit der gleichen Argumentation wie bei den vorhergehenden Netzen ist das Netz des bedingten Ausdrucks ein S-Netz.

5.7 Verbundanweisungen

RPR definiert die Verbundanweisungen Entscheidung, Fallunterscheidung, Schleife und die try-Anweisung. Im Gegensatz zur primitiven Anweisung sind die Verbundanweisungen nicht atomar aufgebaut, sondern bilden eine Komposition von anderen eingebetteten Ausführungselementen.

Die Ausführungssemantik der Verbundanweisungen wird im Folgenden beschrieben und jeweils als Modellnetz spezifiziert. Zur besseren Übersicht wird dazu auch der betreffende Teil der RPR-Grammatik mit angegeben. Alle genannten Verbundanweisungen bilden wie die primitive Anweisung eine Spezialisierung der bereits beschriebenen allgemeinen Anweisung. Die Netze dieser Anweisungen haben alle den gleichen Rand.

5.7.1 Entscheidungsanweisung

In Grammatikausschnitt 14 ist der Teil der RPR-Grammatik der Entscheidungsanweisung dargestellt. Die Entscheidungsanweisung besteht demnach aus einem Bedingungsausdruck sowie den beiden then- und else-Blöcken. In Abbildung 33 ist das Modellnetz der Entscheidungsanweisung dargestellt.

```
IfStatement = "if" "(" BoolExpression ")"
              "then" StatementBlock "else" StatementBlock.
```

Grammatikausschnitt 14: Grammatikausschnitt der Entscheidungsanweisung

Eine Entscheidungsanweisung endet abrupt, wenn der Bedingungsausdruck oder der ausgeführte Anweisungsblock abrupt endet. Ansonsten endet die Entscheidungsanweisung normal.

Aus Abbildung 33 ist leicht zu erkennen, dass das Modellnetz einer if-Anweisung ein S-Netz ist: Alle Transitionen haben genau eine Stelle im Vor- und eine im Nachbereich, und die Teilnetze sind nur an den Randstellen verbunden.
Wenn der ausgeführte Anweisungsblock mit „return“, „break“ oder „continue“ endet, dann endet die if-Anweisung ebenso abrupt. Dieser Fluss ist aus Übersichtsgründen „gebündelt“ dargestellt und so anzunehmen, dass die drei Flüsse je einzeln verlaufen und auch in einzelne Austrittsstellen münden (vgl. Abschnitt 5.3.1 auf Seite 86).

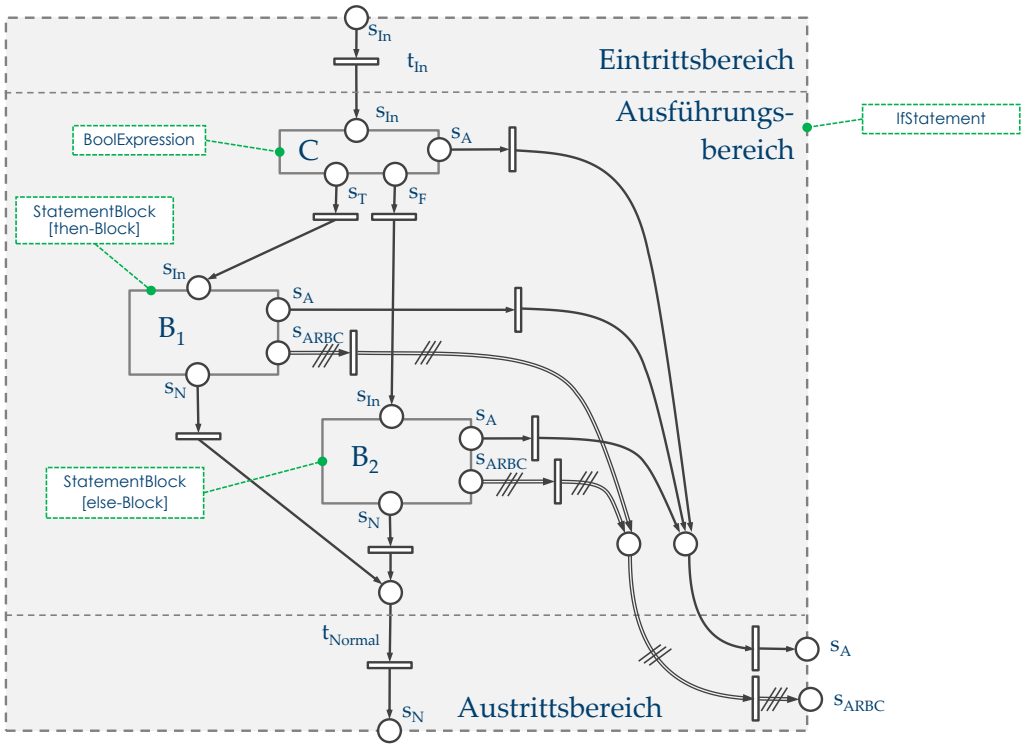


Abbildung 33: Modellnetz einer Entscheidungsanweisung

5.7.2 Schleifenanweisung

Die Schleifenanweisung ist aus einer Wiederholbedingung und dem Schleifenkörper aufgebaut. Grammatikausschnitt 15 enthält hierzu die RPR-Grammatik, und Abbildung 34 zeigt das Modellnetz der Schleifenanweisung.

```
WhileStatement = "while" "(" BoolExpression ")"
                             StatementBlock.
```

Grammatikausschnitt 15: Grammatikausschnitt der Schleifenanweisung

Eine Besonderheit der Schleifenanweisung bildet die Behandlung des abrupten Beendens des Schleifenkörpers durch „break“ oder „continue“. Durch den Fluss aus den Stellen s_{AB} und s_{AC} des Schleifenkörpers wird das bei den realen Sprachen übliche Ablaufverhalten nachgebildet. Dass eine Schleifenanweisung dennoch mit „break“ und „continue“ abrupt enden kann, liegt an der Möglichkeit der realen Sprachen, diese Abbrüche mit einem sogenannten Label zu versehen, das auch eine weiter außerhalb liegende Schleifenanweisung angeben kann.

In den gängigen Programmiersprachen gibt es neben den ablehnenden Schleifen auch annehmende Schleifen und Zählschleifen. Die Zählschleifen können gleich wie die ablehnende Schleife in das GBT-Modell abgebildet werden, wobei die Wiederholbedingung der Zählschleife in die Wiederholbedingung des GBT-Modells übertragen wird und die Ausdrücke des Initialisierungs- und Updatebereichs, sofern diese GBT-relevant sind, in die Subausdrücke abgebildet werden. Die sogenannte forEach-Schleife, die über eine Kollek-

tion iteriert, kann wie die for-Schleife in die GBT-Modellschleife abgebildet werden, enthält aber keinen booleschen Ausdruck als Wiederholbedingung. Dieser wird durch einen Stellvertreter „Die Kollektion enthält weitere Elemente“ ersetzt.

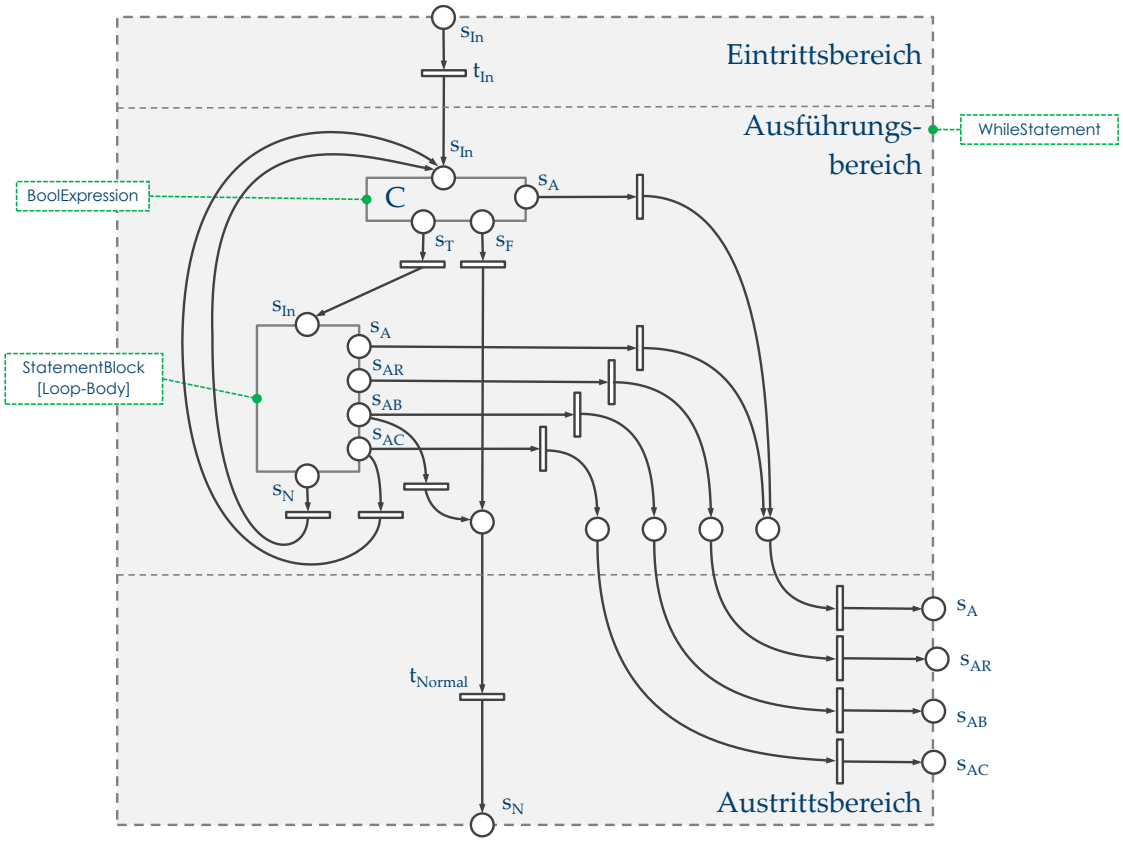


Abbildung 34: Modellnetz der Schleifenanweisung

Annehmende Schleife	Umformung

do {
...
} while(condition);
	boolean firstLoop = true;
	while(firstLoop || condition) {
	  ...
	  firstLoop = false;
	}

Tabelle 11: Umformung der annehmenden Schleife in eine ablehnende Schleife

Die annehmenden Schleifen werden nach der in Tabelle 11 in Java-Syntax angegebenen Umformung in ablehnende Schleifen umgeformt und so ins GBT-Modell übertragen. Die ergänzten und fett gedruckten Programmteile wirken sich natürlich nicht auf die Überdeckungsmetriken aus.

In Kapitel 6.3.5 wird zudem noch auf eine Sonderbehandlung der annehmenden Schleife bei der Bestimmung der Schleifenüberdeckung beschrieben. Diese Sonderbe-

handlung ist dadurch begründet, dass ein Überspringen des Schleifenkörpers bei annehmenden Schleifen keine sinnvolle Forderung ist.

5.7.3 Fallunterscheidung

In Grammatikausschnitt 16 ist die Grammatik der Fallunterscheidung angegeben, und Abbildung 35 zeigt das Modellnetz.

```
SwitchStatement    = "switch" CaseHandler.

CaseHandler        = "case" StatementBlock CaseHandler
                   | "default" StatementBlock.
```

Grammatikausschnitt 16: Grammatikausschnitt der Fallunterscheidung

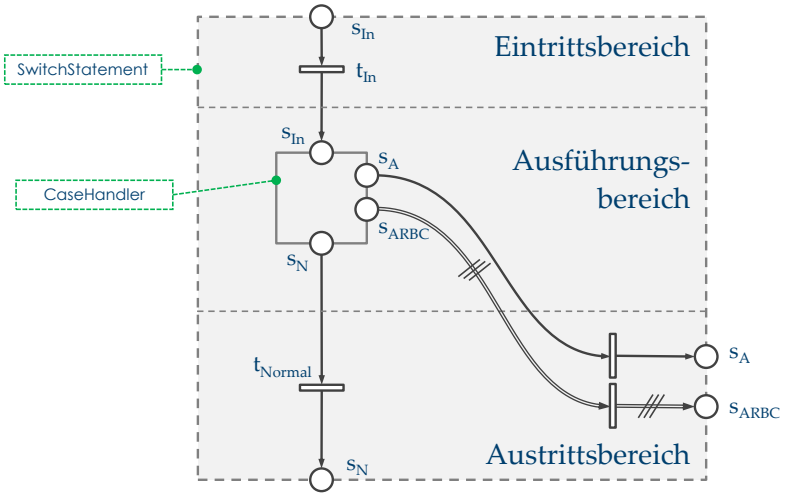


Abbildung 35: Modellnetz der Fallunterscheidung

Eine Fallunterscheidung (die switch-Anweisung) endet abrupt, wenn einer der case-Blöcke oder der default-Block abrupt endet. Ansonsten endet die switch-Anweisung normal. Da in den Original-Programmen der Ausdruck, der den Einsprung in den jeweiligen case-Block steuert, kein boolescher Ausdruck ist, wird er im GBT-Modell präteriert; wenn dort GBT-relevante Ausdrücke enthalten sind, werden diese über die Subausdrücke abgebildet. Ebenso werden die sogenannten switch-Labels, die als Konstanten die einzelnen Fälle definieren, präteriert. Das Netz der switch-Anweisung nach Abbildung 35 enthält damit im Ausführungsbereich lediglich einen *CaseHandler*, der den ersten case-Block sowie ggf. weitere folgende case-Blöcke modelliert.

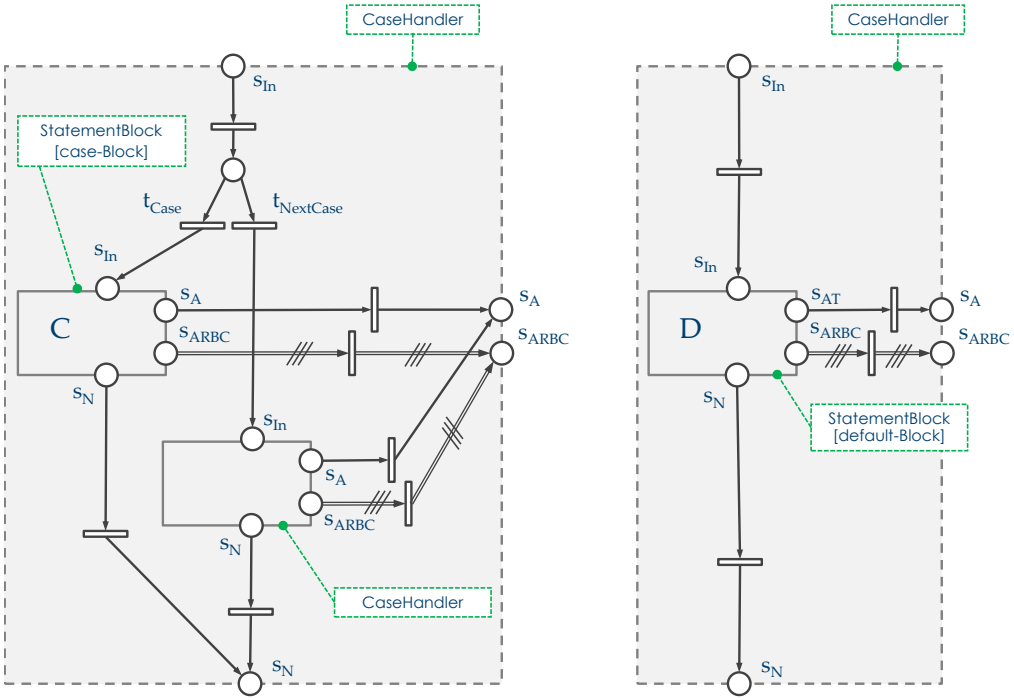


Abbildung 36: Die beiden Modellnetze von CaseHandler

Für *CaseHandler* zeigt Abbildung 36 die beiden möglichen Teilnetze: Im linken Netz schaltet t_{Case}, wenn der Anweisungsblock C der auszuführende Fall ist. Ansonsten schaltet $t_{NextCase}$, und der nächste *CaseHandler* wird ausgeführt. Dies endet, wenn der default-Block ausgeführt wird (rechtes Netz).

An dieser Stelle sei noch darauf hingewiesen, dass für switch-Anweisungen ohne default-Block das Ergänzen eines leeren default-Blocks im GBT-Modell zu einem nicht ausführbaren Anweisungsblock (d. h. zu totem Code) führen kann. Ein Beispiel in Java-Syntax zeigt Programmcode 1.

```
enum X { a, b }
...

X x = ...
switch(x) {
  case a: /* ... */ break;
  case b: /* ... */ break;
  default: /* das ist toter Code */
}
```

Programmcode 1: switch-Anweisung mit Aufzählungstyp

Für den switch-Ausdruck *x*, der vom Typ des Aufzählungstyps *X* ist, sind die möglichen Werte mit den switch-Labels erschöpfend aufgeführt. D. h. es gibt keinen x-Wert (auch nicht null), der zur Ausführung des default-Blocks führen könnte. Ist der switch-Ausdruck nicht von einem Aufzählungstyp, ist eine solche erschöpfende Auflistung der switch-Labels natürlich praktisch kaum möglich. Für den Fall also, dass die switch-Labels

erschöpfend die möglichen Werte des switch-Ausdrucks auflisten, wird eine 100 prozentige Zweig- oder Blocküberdeckung nicht zu erreichen sein.

5.7.4 try-Anweisung

Die try-Anweisung dient der Ausnahmebehandlung und bildet in der GBT-Modellsprache ebenso wie die if-, die while- oder die switch-Anweisung eine Spezialisierung der allgemeinen Anweisung. Den Grammatikausschnitt der try-Anweisung zeigt Grammatikausschnitt 17, und Abbildung 37 zeigt das Modellnetz.

```
TryStatement      = "try" StatementBlock CatchHandler.

CatchHandler      = "catch" StatementBlock CatchHandler
                  | empty.
```

Grammatikausschnitt 17: Grammatikausschnitt der try-Anweisung

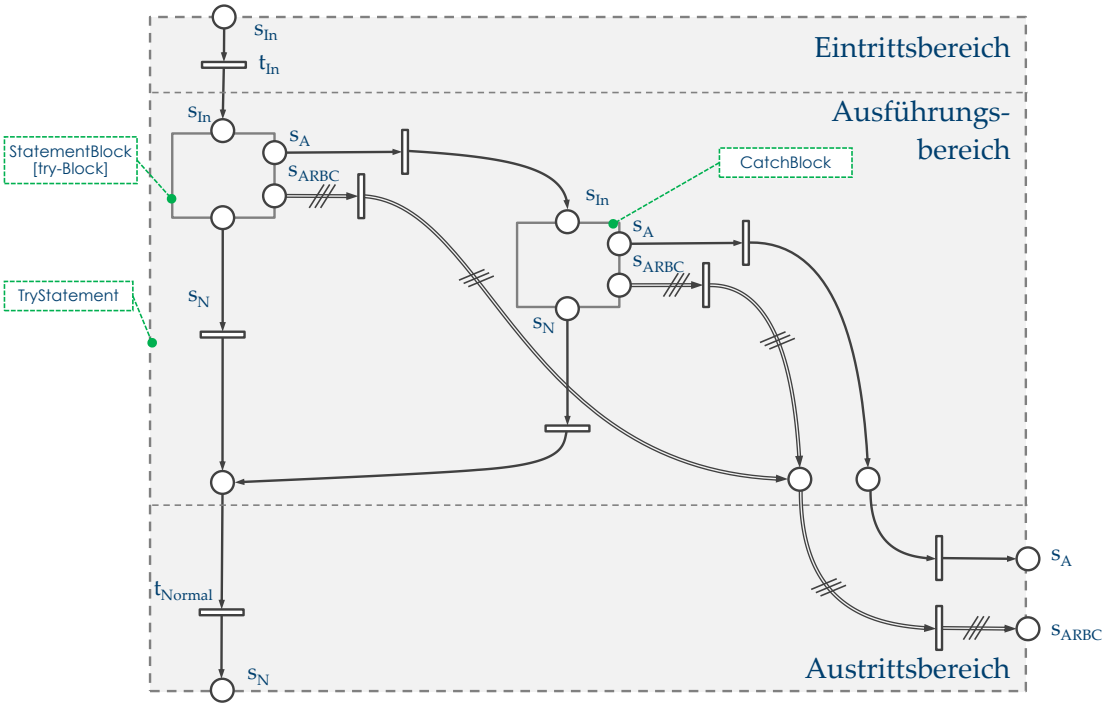


Abbildung 37: Modellnetz der try-Anweisung

Die try-Anweisung wird durch den try-Block sowie den geschachtelt angeordneten *CatchHandlern* gebildet, die den catch-Blöcken des Originalprogramms entsprechen. Welcher der catch-Blöcke angesprungen wird, hängt bei den gängigen Programmiersprachen vom Typ der geworfenen Ausnahme ab. Da dieser Ausdruck der realen Sprachen kein boolescher Ausdruck ist, wird er im Modell unterdrückt. Die beiden alternativen Netze für *CatchHandler* zeigt Abbildung 38.

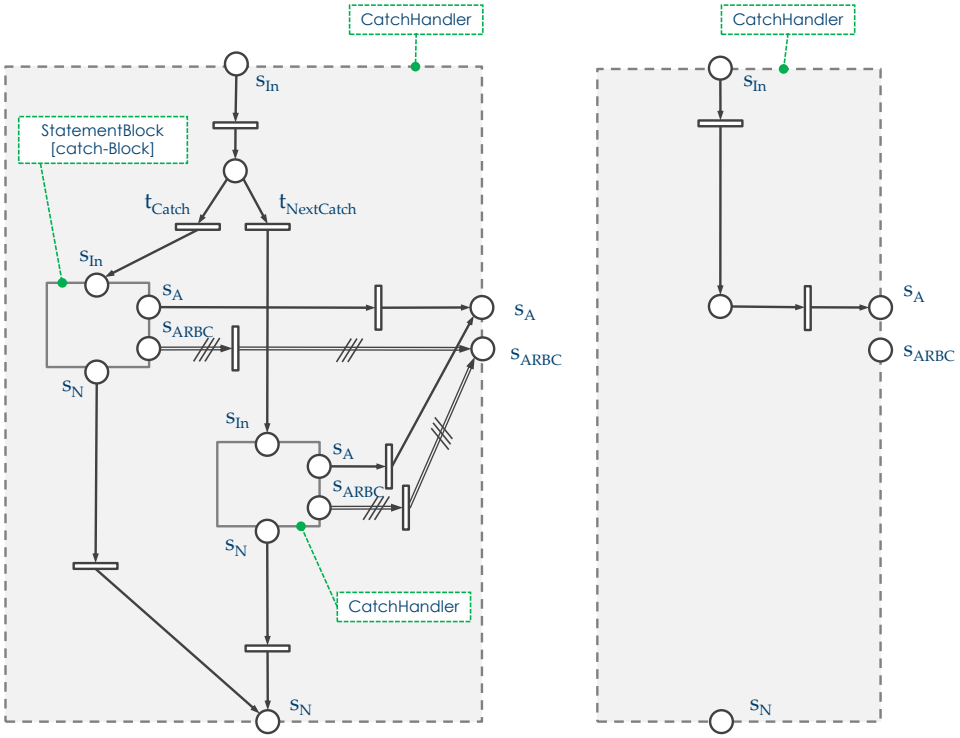


Abbildung 38: Die beiden Modellnetze von CatchHandler

Im linken Netz wird über den Konflikt bei t_{Catch} und $t_{NextCatch}$ entschieden, ob der catch-Block zur Ausnahmebehandlung ausgeführt wird oder aber die Suche nach einem anderen passenden *CatchHandler* fortgesetzt werden soll. Das rechts in Abbildung 38 dargestellte Netz bildet den leeren *CatchHandler*. Wird kein „passender" catch-Block gefunden, dann gilt die Ausnahme als unbehandelt und wird „weitergeworfen".

Wie bei den realen Programmiersprachen üblich, endet die try-Anweisung der RPR damit normal, wenn der try-Block normal endet oder die Ausnahme „gefangen" wird, also der catch-Block normal endet. Ansonsten endet die try-Anweisung abrupt.

Da im Netz der try-Anweisung und im Netz von *CatchHandler* alle Transitionen genau eine Stelle im Vor- und eine im Nachbereich haben, ist das Netz der try-Anweisung für beliebige eingebettete S-Netze der try- und catch-Anweisungsblöcke ein S-Netz.

5.8 Zählerstellen

Die Modellnetze der Ausführungselementtypen Anweisung, Anweisungsblock und boolescher Ausdruck haben im Eintritts- und Austrittsbereich die Eintritts- und Austritts-Transitionen (vgl. Kapitel 5.3, Seite 84). Alle Flüsse zum Ausführungselement hin führen definitionsgemäß über die Eintritts-Transition, und über eine der Austritts-Transitionen führt der Fluss wieder aus dem Netz des Ausführungselements heraus.

Es wird nun derart eine Netztransformation vorgenommen, dass diesen Eintritts- und Austritts-Transitionen im Nachbereich eine sogenannte Zählerstelle zugeordnet wird. Zählerstellen haben selbst einen leeren Nachbereich und sind damit so angelegt, dass dort so viele Marken liegen, wie es von einer leeren Anfangsmarkierung ausgehend Schaltvorgänge der im Vorbereich liegenden Eintritts- oder Austrittstransitionen gibt.

Def. **Zählerstellen** *eines GBT-Modellnetzes sind Stellen mit leerem Nachbereich, die nicht Austrittsstellen sind.* $isCounter(s) \Leftrightarrow s \in S / S_{Out} \wedge s\bullet = \emptyset$.

Nach [Ba90] bilden diese Zählerstellen den absoluten Rand eines Netzes (vgl. Abschnitt 5.2.1 auf Seite 80).

Für einige Betrachtungen ist es vorteilhaft, eine Netztransformation derart vorzunehmen, dass alle Zählerstellen des Netzes N im Netz N' ausgeblendet, also entfernt werden.

Def.: *Seien* $N = (S, T, F)$ *ein Netz und* $N' = (S', T', F')$ *das Netz N ohne Zählerstellen. Eine Abbildung* $N \rightarrow N'$ *heißt dann* **Zählerentfernung**, *wenn* $T' = T$ *und* $S' = S \setminus \{ s \in S \mid isCounter(s) \}$ *und* $F' = F \setminus \{ (x, y) \in F \mid isCounter(y) \}$

Def.: *Sei* s_C *eine Zählerstelle, dann ist* $c = \mid M(s_C) \mid$ *der* **Wert** *des (Ausführungs- oder Resultats-)***Zählers**.

5.8.1 Modellnetz des booleschen Ausdrucks mit Zählerstellen

Abbildung 39 zeigt diese hervorgehoben dargestellten Zählerstellen im Modellnetz des booleschen Ausdrucks (die Flüsse, die bereits in Abbildung 22 auf Seite 91 beschrieben wurden, sind dünn gestrichelt dargestellt).

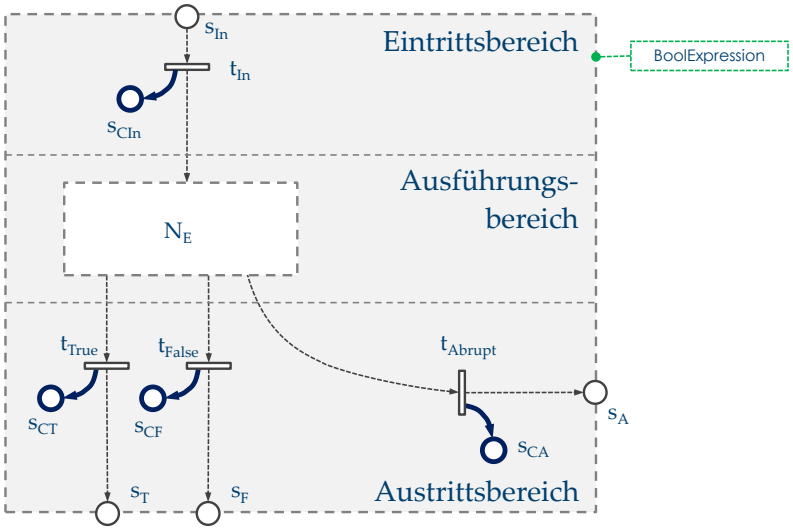


Abbildung 39: Modellnetz des booleschen Ausdrucks mit Zählerstellen

Die Stellen s_{CIn}, s_{CT}, s_{CF} und s_{CA} sind damit als Ausführungszähler zu verstehen: Werden diese bei der Anfangsmarkierung leeren Stellen in einer Endmarkierung betrachtet, dann geben

$c(E) = \mid M(s_{CIn}) \mid$	die Anzahl der Auswertungen von E,
$cT(E) = \mid M(s_{CT}) \mid$	die Anzahl der Auswertungen von E mit dem Resultat true,

cF(E) = ∣ M(s_{CF}) ∣	die Anzahl der Auswertungen von *E* mit dem Resultat false, und
ca(E) = ∣ M(s_{CA}) ∣	die Anzahl der Ausführungen von *E*, die abrupt beendet wurden,

wieder. Sowohl die Netze der Einzelterme als auch die der zusammengesetzten booleschen Ausdrücke werden um die genannten Zählerstellen erweitert. Da die Markierung der Zählerstellen s_{CT} und s_{CF} die jeweilige Anzahl der ermittelten (booleschen) Resultate des Ausdrucks widerspiegelt, werden diese beiden Stellen im Folgenden auch als Resultatsstellen bezeichnet. Die Markierung der Zählerstellen wird im Folgenden auch als Zählerstand (oder Wert des Zählers) bezeichnet.

5.8.2 Modellnetz für Anweisung und Anweisungsblock mit Zählerstellen

Die Netzerweiterung für die Modellnetze der Anweisungen sowie der Anweisungsblöcke berücksichtigt die zusätzlich zum abrupten Beenden enthaltenen Austrittstransitionen für „return“, „break“ und „continue“. Zum einen werden alle zugehörenden Austrittstransitionen um die Zählerstellen ergänzt, zum anderen soll ein Beenden einer Anweisung oder eines Anweisungsblocks auf diese Weise aber als „normal“-Beenden bewertet werden. Das folgende Beispiel kann dies begründen:

```
1 boolean gefunden = false;
2 for(i = 0; i < arrayEingaben.length; i++) {
3    if(arrayEingaben[i] == gesuchterWert) {
4       gefunden = true;
5       break;
6    }
7 }
```

Programmcode 2: Abruptes und normales Beenden

Weder beim Anweisungsblock von Zeile 4 und 5 noch bei der if-Anweisung von Zeile 3 wird ein Programmierer von „abruptem Beenden“ vergleichbar einer geworfenen Ausnahme sprechen. D. h. auch wenn „formal“ abruptes Beenden vorliegt, soll im GBT das Beenden dennoch als „normal“ bewertet werden. Aus diesem Grund fließt auch dann eine Marke in die Zählerstelle für normales Beenden, wenn eine Anweisung oder ein Anweisungsblock abrupt mit „return“, „break“ oder „continue“ endet. Im Folgenden wird dieses Beenden auch als *schwach abruptes* Beenden bezeichnet. Abruptes Beenden mit „throw“ wird als *streng abruptes* Beenden bezeichnet, und normales Beenden wird als *streng normal* bezeichnet, wenn die Austrittstransition t_{Normal} schaltet.

Abbildung 40 zeigt das entsprechend um Zählerstellen erweiterte Modellnetz einer Anweisung oder eines Anweisungsblocks. In die Zählerstelle s_{CN} wird sowohl bei streng normalem als auch bei schwach abruptem Beenden eine Marke gestellt. In die Zählerstelle s_{CNF} wird nur bei streng normalem Beenden eine Marke gestellt, und in die Zählerstellen

s_{CA} , s_{CAR} , s_{CAB} und s_{CAC} wird abhängig vom Grund des abrupten Beendens eine Marke gestellt.

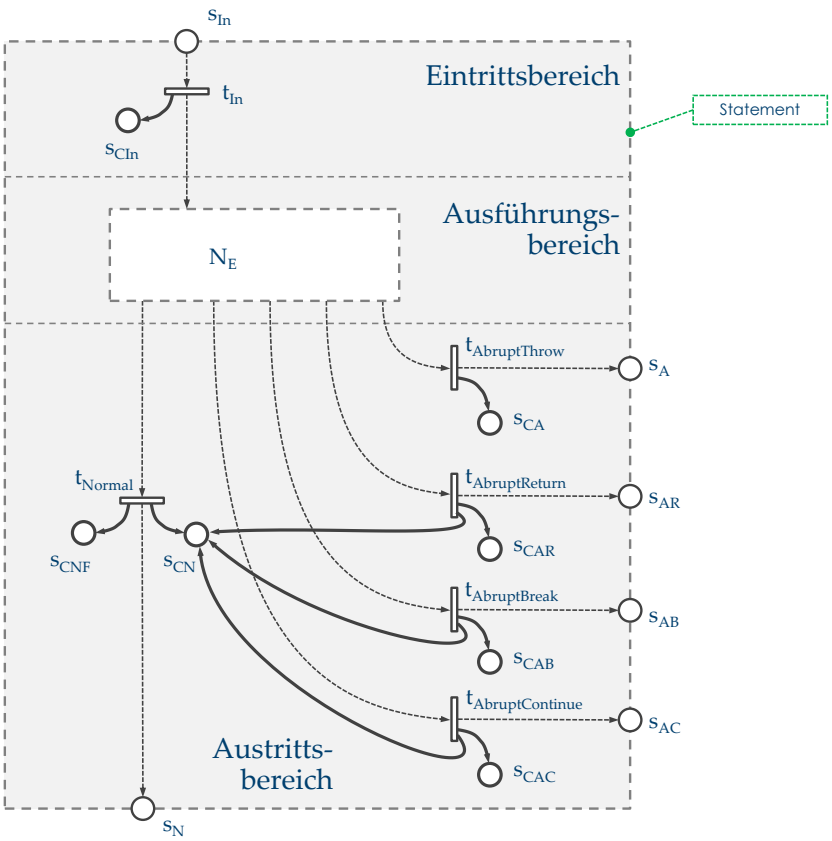


Abbildung 40: Modellnetz der Anweisung mit Zählerstellen

Wenn für ein Ausführungselement S (eine Anweisung oder einen Anweisungsblock) die in der Anfangsmarkierung leeren Stellen s_{CIn}, s_{CN}, s_{CA} , s_{CAR} , s_{CAB} und s_{CAC} in einer Endmarkierung betrachtet werden, dann geben

$c(S) = \mid M(s_{CIn}) \mid$	die Anzahl der Ausführungen von *S*,
$cn(S) = \mid M(s_{CN}) \mid$	die Anzahl der Ausführungen von *S*, die streng normal oder schwach abrupt beendet wurden (im Folgenden einfach als „normal Beendet“ bezeichnet),
$cnf(S) = \mid M(s_{CNF}) \mid$	die Anzahl der Ausführungen von *S*, die streng normal beendet wurden,
$car(S) = \mid M(s_{CAR}) \mid$	die Anzahl der Ausführungen von *S*, die mit „return“ beendet wurden,
$cab(S) = \mid M(s_{CAB}) \mid$	die Anzahl der Ausführungen von *S*, die mit „break“ beendet wurden,

$cac(S) = \mid M(s_{CAC}) \mid$	die Anzahl der Ausführungen von *S*, die mit „continue“ beendet wurden, und
$ca(S) = \mid M(s_{CA}) \mid$	die Anzahl der Ausführungen von *S*, die abrupt (durch eine Ausnahme) beendet wurden,

wieder. Die Zähler *c(S), cn(S), cnf(S), car(S), cab(S), cac(S)* und *ca(S)*, die den Markierungen der Zählerstellen entsprechen, sind als Attribute einer Anweisung oder eines Anweisungsblocks S zu verstehen.

In Abbildung 40 ist leicht zu erkennen, dass ausgehend von einer leeren Anfangsmarkierung und nach Erreichen der Endmarkierung für beliebige Ausführungen die folgenden Gleichungen gelten:

$$\mid M(s_{CN}) \mid = \mid M(s_{CNF}) \mid + \mid M(s_{CAR}) \mid + \mid M(s_{CAB}) \mid + \mid M(s_{CAC}) \mid \quad (1)$$

$$\mid M(s_{CIn}) \mid = \mid M(s_{CN}) \mid + \mid M(s_{CA}) \mid \quad (2)$$

(1) gilt definitionsbedingt und bringt die Anforderung an die Zählerstelle für normales Beenden zum Ausdruck, wonach sowohl streng normales als auch schwach abruptes Beenden gezählt werden soll. (2) bringt das Prinzip der Modellnetze zum Ausdruck, wonach ein Ausführungselement entweder normal streng abrupt endet. Der Beweis hierzu wird im folgenden Abschnitt noch nachgereicht.

Die beschriebene Netzerweiterung um Zählerstellen ist deswegen auf die Netze der Ausführungselementtypen Anweisung, Anweisungsblock und boolescher Ausdruck begrenzt, weil nur deren Ausführungen in die nachfolgend beschriebenen Überdeckungsmetriken einfließen. So wird z. B. das Netz der *SubExpressions* (vgl. Kapitel 5.5, Seite 95) nicht um die Zählerstellen erweitert. Das ist dadurch begründet, dass *SubExpressions* als „Ganzes“ bei keiner Metrik eine Rolle spielen. Lediglich die einzelnen Elemente der Sequenz wie z. B. die booleschen Ausdrücke fließen in Überdeckungsmetriken ein.

Dass beim bedingten Ausdruck ebenfalls keine Zählerstellen ergänzt werden, begründet sich ähnlich wie bei *SubExpressions*. Zur Bewertung der vollständigen Ausführung eines bedingten Ausdrucks wird allein die Bedingung betrachtet. Der bedingte Ausdruck wird demnach dann im Sinne der Entscheidungsüberdeckung als vollständig ausgeführt bewertet, wenn die Bedingung mindestens einmal true und einmal false bewertet wurde (eine ausführliche Betrachtung hierzu findet in Abschnitt 6.3.4 statt). Das Resultat des gesamten bedingten Ausdrucks fließt aber in keine Überdeckungsmetrik ein – und wird im Ausführungsmodell auch entsprechend unterdrückt.

5.9 Programm

Program bildet das Startsymbol eines GBT-Modell-Programms, das vergleichbar einer Java-Methode einen Programmkörper enthält, der aus einem Anweisungsblock gebildet wird. Abbildung 41 zeigt das Netz eines RPR-Programms, das nun das vollständige Original-Programm für den GBT modelliert.

Wie in Abbildung 41 zu erkennen ist, endet das Programm auch dann normal, wenn der Programmkörper abrupt mit „return“ endet. Die Begründung hierzu ist einfach, weil für einen Programmierer offensichtlich die beiden Java-Methoden *f1()* und *f2()* von Programmcode 3 als semantisch gleich angesehen werden und damit beide auch normal enden.

```
void f1() {
  // ...
}

void f2() {
  // ...
  return;
}
```

Programmcode 3: Beenden eines Programms mit return

Ein Programm endet somit nur dann abrupt, wenn der Programmkörper durch eine Ausnahme abrupt endet. „break“ und „continue“ treten nur innerhalb von Schleifen auf und müssen hier nicht weiter berücksichtigt werden.

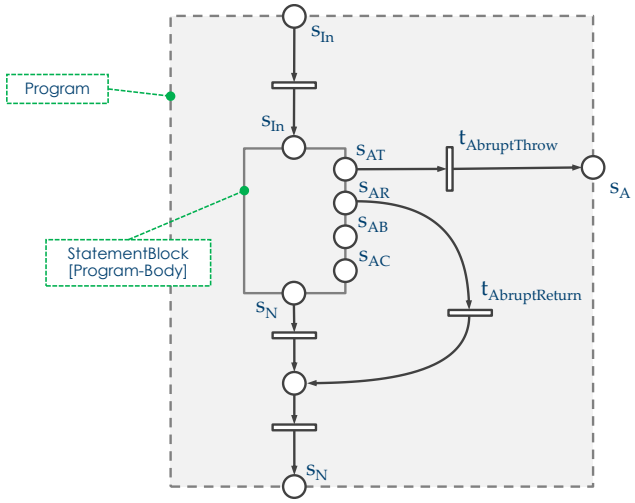


Abbildung 41: Modellnetz eines Programms

Ein sehr einfaches GBT-Modell-Programm P mit einem Anweisungsblock K (dem Programmkörper), der eine primitive Anweisung S1 enthält, zeigt das folgende Beispiel:

```
P: K: { S1: stmt }
```

Da bereits gezeigt wurde, dass das Netz eines beliebigen zählerbefreiten Anweisungsblocks immer ein S-Netz ist und das Netz des Programmkörpers im Netz des Programms nur an den Austrittsstellen verbunden ist, bildet auch das Netz jedes GBT-Modell-Programms ein S-Netz. Nach [DE95, Seite 41] gilt: Ist $N = (S, T, F)$ ein S-Netz mit einer Anfangsmarkierung M_0 und M eine für N und M_0 erreichbare Folgemarkierung, dann gilt

$$| M_0(S) | = | M(S) |.$$

D. h. alle für M_0 erreichbaren Markierungen enthalten genau dieselbe Zahl an Marken wie M_0. Betrachtet wird nun ein GBT-Modellnetz $N = (S, T, F, s_{In}, S_{Out})$ eines Programms P mit der Anfangsmarkierung $M_0 = \{ s_{In} \rightarrow 1,\ s \in S \setminus s_{In} \mid s \rightarrow 0 \}$. D. h. die Stelle s_{In} des Programms wird mit einer Marke markiert, alle anderen Stellen des Gesamtnetzes sind nicht markiert. Damit ist die Gesamtzahl aller Marken des Systems $| M_0(S) | = 1$. Die Programmausführung wird nun wie folgt modelliert:

1. Ausgangspunkt bildet die Anfangsmarkierung M_0.
2. Es wird solange geschaltet, bis keine Transition mehr aktivierbar ist. Wie der Nichtdeterminismus an den Verzweigungsstellen behandelt wird, wird im nächsten Kapitel beschrieben.
3. Die so erreichte Folgemarkierung sei M_E (Endmarkierung)

Hierbei wird von einer Programmausführung ausgegangen, die terminiert, also keine Endlosschleife enthält.

Satz: Wenn beim GBT-Modellnetz N mit der Anfangsmarkierung M_0 eine Folgemarkierung erreicht wird, in der eine der Austrittsstellen des Programms markiert ist, sind alle Transitionen von N tot.

Beweis: Da alle zählerbefreiten GBT-Modellnetze S-Netze sind, gilt: $| M_E(S) | = | M_0(S) | = 1$. Liegt nun eine Marke im Austrittsbereich des Programms, kann ansonsten im zählerbefreiten GBT-Modellnetz keine weitere Marke liegen. D. h. im zählerbefreiten GBT-Modellnetz sind alle Transitionen tot. Daran ändert sich auch nichts, wenn die Zählerstellen wieder hinzugenommen werden: Alle Zählerstellen haben per Definition einen leeren Nachbereich, können also auch keinen Nachbereich aktivieren.

Damit bilden zwei Folgemarkierungen das Ende einer Programmausführung: die Endmarkierung $M_N = \{ s_N \rightarrow 1,\ s \in S \setminus s_N \mid s \rightarrow 0 \}$ für normales Beenden des Programms und die Endmarkierung $M_A = \{ s_A \rightarrow 1,\ s \in S \setminus s_A \mid s \rightarrow 0 \}$ für abruptes Beenden des Programms.

5.10 Dominanzrelation

Für einige der folgenden Überlegungen zu den Modellnetzen wird die Dominanz-Beziehung zwischen Ausführungselementen genutzt. Anschaulich formuliert dominiert ein Ausführungselement A ein Ausführungselement B direkt, wenn das Netz von B im Netz von A direkt eingebettet ist. So ist z. B. der then- oder else-Block in einer if-Anweisung direkt eingebettet oder der Schleifenkörper in der Schleifenanweisung.

Aus der Direkt-Dominanzbeziehung der Ausführungselemente entsteht der Dominanzbaum, dessen Wurzel das Programm bildet. Die Blätter des Dominanzbaums bilden

die primitiven Ausführungselemente wie primitive Anweisung oder primitiver Ausdruck.

Def. Seien A, B, C Ausführungselemente

A ***direkt-dominiert*** B Geschrieben: A ***ddom*** B	⇔	Das Netz von B ist im Netz von A (direkt) eingebettet Definitionsbedingt hat jedes Ausführungselement eines GBT-Modellprogramms außer dem Startsymbol *Program* genau einen direkten Dominator. Man schreibt: A = ddom(B) ⇔ A ddom B
A ***dominiert*** B Geschrieben: A ***dom*** B	⇔	*A ddom B ∨ (A ddom C ∧ C dom B)* *Die Relation dom bildet die transitive Hülle von ddom* *Die Dominanzbeziehung ist transitiv:* *A dom B ∧ B dom C ⇒ A dom C* *Für direkte Dominanz gilt:* *X ddom Y ⇔* *X dom Y ∧ ¬∃ Z : X dom Z ∧ Z dom Y*
B ∈ ***items***(A)	⇔	A dom B

Am folgenden RPR-Programm lassen sich diese Relationen beispielhaft beschreiben:

```
S1: if(E1: and(E2: expr, E3: expr)) …
```

Es gilt: *S1 dom E1*, weil E1 (der Bedingungsausdruck) direkt in S1 eingebettet ist, und *E1 dom E2*, weil *E2* (ein Teilausdruck) direkt in *E1* eingebettet ist. Durch die Transitivität gilt auch *S1 dom E2.*

5.11 Programmausführung

5.11.1 Bestimmung der Ausführungszähler

Die Modellnetze sind stark von den nichtdeterministischen Verzweigungen geprägt, die sich auf das Ausführungsverhalten des realen Programms beziehen. Eine Simulation der realen Programmausführung ist daher im Modell nicht möglich. An den nichtdeterministischen Verzweigungen ist auf Grundlage des Modells nicht entscheidbar, welche der Transitionen schalten soll. In diesem Punkt unterscheiden sich die Modellnetze im Übrigen nicht vom CFG, der ebenso nicht eigenständig „simuliert“ werden kann. Zur Ausführung des Modells müsste „von Hand“ bei jeder Verzweigung entscheiden werden, welchen Pfad die Marke einschlagen soll. Bei sehr kleinen Programmen ist dies zwar durchaus möglich (wie z. B. beim Referenzprogramm aus Abschnitt 6.6), bei Programmen der industriellen Praxis aber völlig ausgeschlossen.

Im Unterschied zu den nichtdeterministischen Verzweigungen des Modells wird in der realen Programmausführung deterministisch ermittelt, ob z. B. eine Anweisung normal oder abrupt endet.

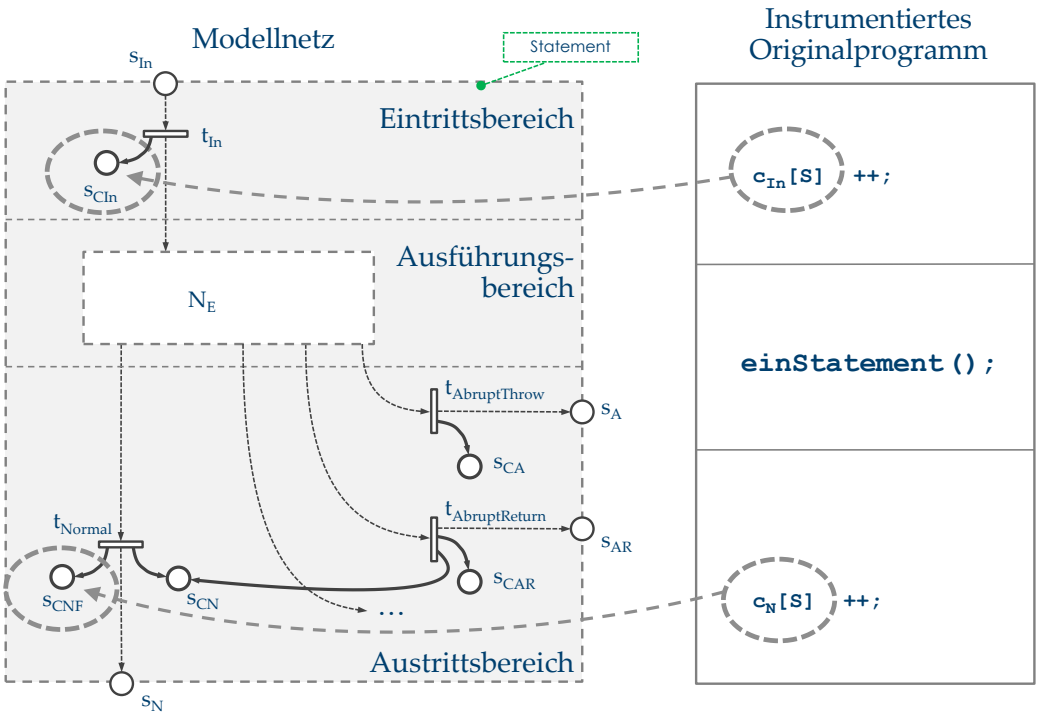


Abbildung 42: Modellnetz und Programmcode

In das Originalprogramm werden nun Ausführungszähler so eingefügt, dass sich aus deren Werten die Markierungen der Zählerstellen des Modells (exakt) bestimmen lassen. D. h. das reale Programm und die reale Programmausführung werden dazu genutzt, um (exakt) die Ausführungszählerwerte zu liefern, die einer Ausführung des Modells entsprechen.

In Abbildung 42 ist für eine RPR-Anweisung *S* dieser Zusammenhang dargestellt. Der nichtdeterministischen Verzweigung des Modellnetzes ist der deterministische reale Pro-

grammcode gegenübergestellt. Die markierten und mit Pfeilen verbundenen Modell- sowie Programm-Elemente sind so zu verstehen, dass deren Werte nach Ende einer Programmausführung und damit mit Erreichen einer Endmarkierung des Modellnetzes als (exakt) gleich angenommen werden. Die Pfeilrichtung ist für die praktische Testausführung als Richtung des Datenflusses zu verstehen: Das instrumentierte reale Programm liefert so unmittelbar einen Teil der Ausführungszähler des Modellnetzes.

Es wird nun die Anweisung des Originalprogramms von Abbildung 42 betrachtet. Unmittelbar vor dieser Anweisung wird ein Ausführungszähler ergänzt. Dieser wird immer vor der Ausführung der Anweisung inkrementiert. Hat dieser Zähler vor der Programmausführung den Wert null und die Zählerstelle s_{CIn} des Modellnetzes die leere Anfangsmarkierung, dann gilt nach Erreichen einer Endmarkierung für alle Programmausführungen:

$$| M(s_{CIn}) | = c_{In}[S],$$

wobei S als Bezeichner der korrespondierenden RPR-Anweisung angenommen wird.

Unmittelbar nach der Anweisung wird ein Anweisungszähler ergänzt, der immer nach normalem Beenden der Originalanweisung inkrementiert wird. Entsprechend gilt für den Ausführungszähler für streng normales Beenden:

$$| M(s_{CNF}) | = c_N[S]$$

Damit sind aber noch nicht alle Markierungen der Zählerstellen des Modellnetzes bestimmt. Ein Problem dabei bilden die Ausführungszähler für abruptes Beenden, deren Gesamtsumme nach Erreichen einer Endmarkierung durch Gleichung (1) und (2) aus Abschnitt 5.8 angegeben werden kann:

$$| M(s_{CA}) | + | M(s_{CAR}) | + | M(s_{CAB}) | + | M(s_{CAC}) | = | M(s_{CIn}) | - | M(s_{CNF}) | \quad (3)$$

Anschaulich besagt diese Gleichung, dass ein Ausführungselement immer dann abrupt endet, wenn es ausgeführt wird und nicht streng normal endet.

Um die Zählerwerte für abruptes Beenden einzeln bestimmen zu können, wird als Beispiel eine break-Anweisung betrachtet. Abbildung 43 zeigt das Modellnetz und den gegenübergestellten Original-Programmcode. Im Modellnetz lässt sich leicht erkennen, dass bei leerer Anfangsmarkierung die Zählerstelle im Eintrittsbereich s_{CIn} und die Zählerstelle für abruptes Beenden s_{CAB} für beliebige Ausführungen die gleiche Markierung haben. Der Ausführungszähler vor der realen break-Anweisung spiegelt diesen Wert wider. Für die break-Anweisung B gilt damit bei leerer Anfangsmarkierung der Zählerstellen und dem Wert null des Ausführungszählers für beliebige Ausführungen:

$$| M(s_{CIn}) | = | M(s_{CAB}) | = c_{In}[B]$$

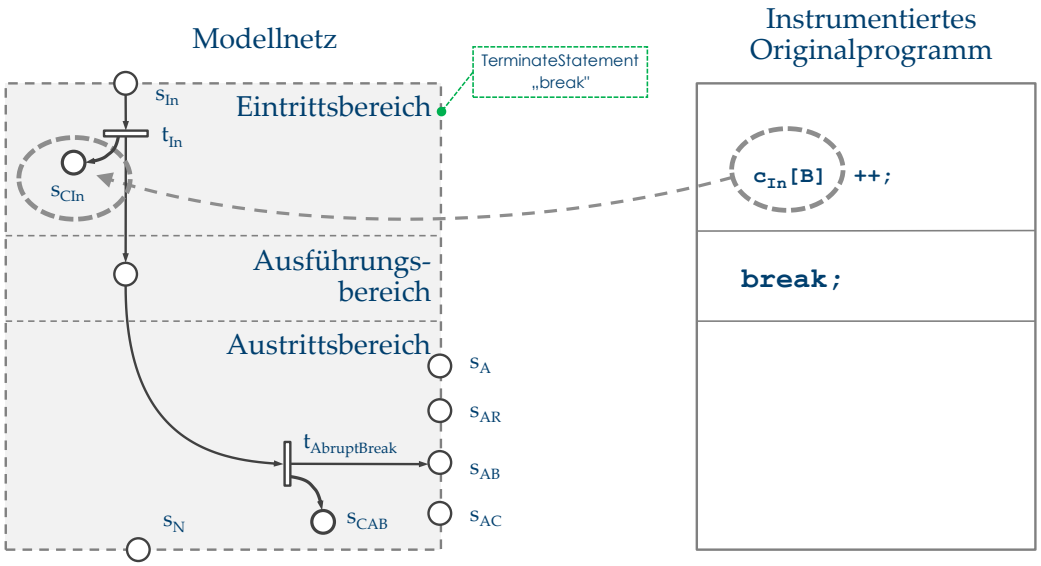


Abbildung 43: Modellnetz der break-Anweisung

Für die weiteren Überlegungen wird der folgende Java-Programmcode betrachtet.

```
1  int[][] arrayWerte = ...;
2  int sucheNach = ...;
3  boolean gefunden = false;
4  suche: for (int i = 0; i < arrayWerte.length; i++) {
5    for (int j = 0; j < arrayWerte[i].length; j++) {
6      if (arrayWerte[i][j] == sucheNach) {
7          gefunden = true;
8          break suche;
9      }
10   }
11 }
```

Programmcode 4: break-Anweisung mit Label

Es ist leicht zu sehen, dass eine break-Anweisung bei allen übergeordneten Strukturen genauso oft abruptes Beenden bewirkt, wie sie selbst Ausführungen hat. Dies gilt für alle umschließenden Strukturen bis zu der, die das break behandelt. Im Beispiel ist das die Schleifenanweisung von Zeile 4 mit der Label-Angabe „suche“. Wenn also im Beispiel das break von Zeile 8 *n* mal ausgeführt wurde, dann bewirkt dies für jede umschließende Struktur *n* mal abruptes Beenden durch dieses „break“.

Das „Umschließen“ entspricht in RPR der *ddom*-Relation. Im folgenden Algorithmus sei *A.id* der Identifier des Ausführungselements *A*. *B* sei eine break-Anweisung, und mit *B.loopID* wird der Identifier der vom break referenzierten Schleife angenommen. In der Regel ist das die „nächste“ umschließende Schleife oder, wie im Beispiel, eine weiter außen liegende Schleife, die über ein sogenanntes Label identifiziert wird.

Der Algorithmus ist so angelegt, dass ausgehend von der break-Anweisung der direkte Dominator gewählt wird. Für diesen direkten Dominator wird der Zähler für abruptes Beenden durch „break“ um die Anzahl der Ausführungen der break-Anweisung erhöht.

Dies findet für alle weiteren direkten Dominatoren statt, bis die durch die break-Anweisung angegebene Schleife erreicht ist.

```
Für alle break-Anweisungen B
  A = ddom(B);

  Wiederhole bis A.id = B.loopID
    // die Ausführungen von B den „break“-
    // Ausführungen von A hinzuzählen
    cab(A) += c(B);

    // die nächste „Umschließung“ wählen
    A = ddom(A);
  End
End
```

Für „continue“ wird gleich wie für „break“ verfahren. Für „return“ ändert sich der Ablauf geringfügig, da nicht bis zur angegebenen Schleife, sondern bis zum Programm iteriert wird.

Damit lassen sich für alle Ausführungselemente des Programms die Markierungen der Zählerstellen s_{CAR} , s_{CAB} *und* s_{CAC} bestimmen. Der abschließend noch nicht bestimmte Wert $| M(s_{CA}) |$ kann durch Umformung von Gleichung (3) berechnet werden:

$$| M(s_{CA}) | = | M(s_{CIn}) | - | M(s_{CNF}) | - | M(s_{CAR}) | - | M(s_{CAB}) | - | M(s_{CAC}) |$$

5.11.2 Testfallausführung

Da beim Testen Programmausführungen immer durch die Eingabedaten eines Testfalls bestimmt werden, wird im Folgenden statt „Programmausführung“ von „Testfallausführung“ gesprochen. D. h. wie in Abschnitt 5.9 für Programmausführung dargestellt wurde, beginnt eine Testfallausführung für ein Programm mit dem Modellnetz N mit der Anfangsmarkierung M_0 und endet mit einer Endmarkierung M_E. Die Eingabedaten, die den Kontrollfluss steuern, stammen dann aus der Spezifikation eines Testfalls. Da von deterministischen Prüflingen ausgegangen wird, sind die Schaltfolgen ausgehend von der Anfangsmarkierung für einen Testfall immer gleich. Es wird die beim Testen gebräuchliche Formulierung genutzt: „Ein Programm mit einem Testfall ausführen“. Wenn mit dieser Testfallausführung eine Endmarkierung des Modellnetzes des Programms erreicht ist, dann geben die Markierungen der Zählerstellen Auskunft über die Ausführung der Ausführungselemente für den ausgeführten Testfall.

Def. *Seien P ein Programm, A ein Ausführungselement von P, T eine Testsuite für P und* $t \in T$ *ein Testfall. Das Programm wird mit t ausgeführt. Die nachfolgend angegebenen Stellen beziehen sich auf das Modellnetz von A. Dann gilt:*

$c(A, t) = | M(s_{In}) |$ Der Ausführungszähler im Eintrittsbereich von A lie-

	fert die Anzahl der Ausführungen von A. Dieser Zähler ist in Abbildung 42 als Variable c_{In} angegeben.
$exe(A, t) \Leftrightarrow c(A, t) > 0$	A wird von t mindestens einmal ausgeführt.
$ca(A, t) = \mid M(s_{CA}) \mid$	Der Ausführungszähler für abruptes Beenden von A.
$exeA(A, t) \Leftrightarrow ca(A, t) > 0$	A wird mit t ausgeführt und endet mindestens einmal abrupt.

Falls A eine Anweisung oder ein Anweisungsblock ist, dann gilt:

$cn(A, t) = \mid M(s_{CN}) \mid$	Der Ausführungszähler für normales Beenden von A.

Falls A ein boolescher Ausdruck ist, dann gilt:

$cT(A, t) = \mid M(s_{CT}) \mid$	Der Ausführungszähler für Auswertungen von A mit dem Resultat true.
$cF(A, t) = \mid M(s_{CF}) \mid$	Der Ausführungszähler für Auswertungen von A mit dem Resultat false.
$cn(A, t) = cT(A, t) + cF(A, t)$	Ein berechneter Ausführungszähler für normales Beenden von A.
$exeT(A, t) \Leftrightarrow cT(A, t) > 0$	A wird mit t ausgeführt und liefert mindestens einmal das Resultat true.
$exeF(A, t) \Leftrightarrow cF(A, t) > 0$	A wird mit t ausgeführt und liefert mindestens einmal das Resultat false.

Für alle Ausführungselemente gilt:

$exeN(A, t) \Leftrightarrow cn(A, t) > 0$	A wird mit t ausgeführt und mindestens einmal normal beendet.

5.12 Bewertung des Ablaufmodells

Die Ausführungssemantik der RPR-Elemente mit Petri-Netzen zu modellieren hat eine Reihe von Vorteilen: Das Modell ist präzise und allgemein lesbar. Die für den GBT relevante Ausführung der Modellelemente ist so präzise beschrieben. Insbesondere ist auch die Ausführungssemantik für abruptes Beenden im Modell abgebildet. Das GBT-Referenzmodell liefert so eine präzise Spezifikation, die vorgibt, ob und wie eine reale Programmiersprache in das GBT-Modell übertragen werden kann.

Ein weiterer großer Vorteil ist, dass die Inkrementoperationen der Ausführungszähler präzise im Modell angegeben sind. Damit liegt für die die Implementierung eines GBT-Werkzeugs eine präzise Spezifikation vor. Auch erlaubt das formale Modell den mathematischen Nachweis, dass der Zähler für streng abruptes Beenden aus den anderen Zäh-

lern berechnet werden kann. Für die praktische Implementierung eines Werkzeugs ist das ein großer Vorteil.

Ein Vorteil der Petri-Netze gegenüber einer informellen Technik besteht auch darin, dass Werkzeuge genutzt werden können. Werkzeuge können z. B. zur Simulation wie beispielsweise zur Bestimmung des Erreichbarkeitsgraphen genutzt werden. Für die Netzerweiterung zur Bestimmung der Termüberdeckung wird im folgenden Kapitel davon auch Gebrauch gemacht.

5.13 Die Transformation von Java-Programmen

Die GBT-Modellsprache RPR und die reale Programmiersprache Java [Go05] haben viele Gemeinsamkeiten. So entsprechen sich zu großen Teilen die GBT-relevanten Anweisungstypen wie Entscheidung, Fallunterscheidung, Schleife oder die Ausnahmebehandlung. Allerdings ist die Java-Syntax deutlich vielfältiger als RPR. Zudem enthält Java eine ganze Reihe von Besonderheiten, die in RPR nicht vorkommen. Diese stellen zwar kein großes Problem für den GBT dar, müssen aber wie die Abbildung der Anweisungen einzeln betrachtet werden. Einen weiteren zu berücksichtigenden Aspekt bilden die Ausführungszähler des GBT-Modells, die dort inhärent in den Zählerstellen der Ausführungselemente angenommen werden. In Java müssen diese Zähler im Programmcode ergänzt werden.

5.13.1 Bezeichner der Ausführungselemente

Im GBT-Modell hat jede Anweisung, jeder Anweisungsblock und jeder Ausdruck einen eindeutigen Bezeichner. Dieser Bezeichner wird genutzt, um beispielsweise für ein Ausführungselement *A* mit *exe(A, t)* auszudrücken, dass *A* vom Testfall *t* ausgeführt wird. Es wird auch für Java-Programme zur GBT-Auswertung vorteilhaft sein, die Programmkonstrukte derart bezeichnen zu können, nur gibt es bei Java-Programmen diese Bezeichner nicht. Aus diesem Grund werden automatisch beim Parsen des Programms die Bezeichner vergeben – verbunden mit der Problematik, dass GBT-Resultate bei Programmänderungen nicht übertragbar sind, da diese temporären Bezeichner immer nur für eine Programmversion gelten.

5.13.2 Java-Methoden

Eine Java-Methode entspricht einem RPR-Programm. Während die Signatur und die Modifier (die Sichtbarkeit, static oder final) in RPR unterdrückt werden, wird der Methodenkörper in einen Anweisungsblock abgebildet. Das folgende Beispiel zeigt die Abbildung:

Java	RPR

static private int foo(int a) {
...
}
	M1: B1: {
	...
	}

Die hierarchische Struktur von Java mit Klassen und Paketen kommt im GBT-Modell nicht zum Ausdruck, diese Information wird ergänzend zum GBT-Modellprogramm abgelegt. Für den GBT ist diese Information auch nur so weit bedeutend, wie Überdeckungen für einzelne Hierarchien ausgewertet werden (z. B. eine Auswertung der Überdeckung für eine Klasse oder ein Paket). Die Unterscheidung von Java in statische und nicht-statische Methoden wird im GBT-Modell ebenfalls nicht vorgenommen; für den GBT ist dies auch nicht wichtig. Der sogenannte static-Initializer [Go05, Kapitel 8.7] bildet einen Block, der vergleichbar einer statischen Methode von der Java-Laufzeitumgebung aufgerufen wird, sobald die Klasse für die Benutzung geladen und initialisiert wird. Da sich der static-Initializer aus GBT-Sicht wie eine Methode verhält, erfolgt die Behandlung gleich wie bei Methoden. Abstrakte Methoden oder Schnittstellen (Interfaces) spielen für den GBT dagegen keine Rolle, da diese nicht ausführbar sind und damit im GBT nicht betrachtet werden können.

5.13.3 Java-Attribute

Die Attributdeklarationen im Klassenrumpf für statische oder nicht-statische Attribute werden in Java dann als Anweisung gewertet, wenn eine Zuweisung vorgenommen wird. Diese Unterscheidung wird deswegen vorgenommen, weil Deklarationen mit Zuweisungen zur Laufzeit ausgeführt werden (vgl. [Go05, Kapitel 8.3.2.3]); Deklarationen ohne Zuweisungen werden dagegen nicht ausgeführt. Im GBT-Modell werden sowohl die „zuweisenden“ als auch „nicht-zuweisenden“ Deklarations-Anweisungen unterdrückt, weil diese Anweisungen in keinen Kontrollfluss eingebettet sind, sondern unweigerlich dann ausgeführt werden, wenn ein Objekt erzeugt wird. Bei lokalen Variablen-Deklarationen innerhalb von Methoden unterscheidet die Java-Syntax nicht in zuweisende und nicht-zuweisende Deklarationen. Im GBT-Modell werden diese Deklarationen daher auch als primitive Anweisungen aufgenommen.

5.13.4 Anweisungen

Da es in der Modellsprache für viele der Java-Anweisungen direkt korrespondierende Anweisungen gibt, erfolgt die Beschreibung der Abbildung von Java ins GBT-Modell anhand der Gegenüberstellung der jeweils betroffenen Grammatikausschnitte. Es wird der rechte Teil der Produktionsregeln jeweils einander gegenübergestellt. Die Notation und die Darstellung der genutzten BNF entspricht der, die in Tabelle 10 (Seite 70) beschrieben ist. Zudem werden noch die Option […] und die Wiederholung (beliebig oft, auch keinmal) { … } verwendet. Die hier verwendete Java-Syntax stammt aus [Go05].

if-Anweisung

Der then- und else-Teil einer if-Anweisung sind nach der Java-Grammatik je eine Anweisung, wobei der else-Teil optional ist. Im GBT-Modell sind then- und else-Teil dagegen Anweisungsblöcke, und der else-Teil ist nicht optional. Die Abbildung der if-Anweisung in das GBT-Modell erfolgt damit derart, dass then- und else-Teil je in Anweisungsblöcke abgebildet werden und, falls der else-Block im Java-Programm nicht vorhanden ist, dieser als leerer else-Block im GBT-Modell ergänzt wird.

Java	RPR
"if" "(" Expression ")" Statement ["else" **Statement**]	"if" "(" BoolExpression ")" "then" StatementBlock "else" StatementBlock

Schleifenanweisungen

Eine while-Schleife von Java kann unmittelbar in die while-Schleife des GBT-Modells übertragen werden. Der Schleifenkörper der Java-Schleife wird in einen Anweisungsblock des GBT-Modells übertragen, und der Ausdruck der Wiederholbedingung des Originalprogramms wird in die *BoolExpression* des GBT-Modells übertragen.

Java	RPR

"while" "(" **Expression** ")"
  Statement
 	
"do" **Statement** "while"
  "(" **Expression** ")" ";"
 	
"for" "(" **ForControl** ")"
  Statement
	"while"
	"(" BoolExpression ")"
	StatementBlock

Eine etwas ausführlichere Betrachtung verlangt die einfache for-Schleife. Für diese hat *ForControl* nach [Go05, Kapitel 14.14] die Syntax

```
[ ForInit ] ";" [ Expression ] ";" [ ForUpdate ]
```

mit *ForInit* und *ForUpdate* als *StatementExpressionList*, das sind mit Komma getrennte *StatementExpression*, d. h. Ausdrucksanweisungen wie z. B. Zuweisungen oder Inkrement-Ausdrücke. Das folgende Beispiel 2 enthält eine solche mit Komma getrennte Liste. In der Praxis wird man aber ganz überwiegend die for-Schleifen nach Beispiel 1 vorfinden, also mit genau einem Initialisierungsausdruck und einem Update-Ausdruck. Alle drei Angaben, also *ForInit*, *Expression* und *ForUpdate* sind optional, können also auch weggelassen werden. Beispiel 3 zeigt einen solchen Fall.

```
// Beispiel 1
for(int i = 0; i < 10; i++) { … }

// Beispiel 2
for(int i = 0, j = 2; j + i < 10; i++, j += i) { … }

// Beispiel 3
for(;;) { … }
```

Die *Expression* (sofern angegeben) ist nach der Java-Spezifikation ein boolescher Ausdruck. Die Abbildung der for-Schleife in das GBT-Modell erfolgt nun derart, dass *Expression* in die Wiederholbedingung der GBT-Modell-Schleife abgebildet wird. Ist *Expression* leer, wird true als Bedingung angenommen. Falls in *ForInit* oder *ForUpdate* GBT-relevante Ausdrücke (also boolesche oder bedingte Ausdrücke) enthalten sind, werden diese im Modell als *SubExpressions* der Schleifenanweisung abgebildet.

try-Anweisung

Die Ausnahmebehandlung des GBT-Modells entspricht bereits weitgehend der Ausnahmebehandlung von Java. Insofern werden die try-Anweisungen des Originalprogramms praktisch unverändert in try-Anweisungen des GBT-Modells übertragen. Folgende Abweichungen sind zu berücksichtigen:

- Die catch-Argumente (die Typ-Bezeichner der „zu fangenden“ Ausnahmen) werden im GBT-Modell unterdrückt.
- finally-Blöcke werden im GBT-Modell unterdrückt, weil keinerlei Fallunterscheidung oder Alternativausführung vorliegt. Der finally-Block des Java-Programms wird im GBT-Modell als „verschmolzen“ mit dem try-Block angenommen.

Java	RPR

"try" Block
{
"catch" "(" FormalParameter ")"
Block
}
"finally" Block
	"try" StatementBlock
	{
	"catch" StatementBlock
	}

Fallunterscheidung – switch

Bei der Java-Fallunterscheidung werden für einen switch-Ausdruck mehrere Fälle – sogenannte switch-Labels – mit einem entsprechend auszuführenden Anweisungsblock festgelegt. Die Java-Syntax bezeichnet einen solchen Fall als *SwitchBlockStatementGroup*: Ein oder mehrere switch-Label, gefolgt von einer Sequenz von Anweisungen. Steht am Ende eines solchen Anweisungsblockes eine break-Anweisung, ist die switch-Anweisung normal beendet. Endet ein Block ohne break, führt der Kontrollfluss die weiteren Anweisungsblöcke der folgenden *SwitchBlockStatementGroup* ebenso aus. Im GBT-Modell wird

jede dieser *SwitchBlockStatementGroups* als Anweisungsblock abgebildet; die Blöcke tragen dementsprechend zur Blocküberdeckung bei. Damit besteht der Java-Programmcode von Programmcode 5 aus drei Anweisungsblöcken mit den switch-Labels 1, 2 und 3.

```
switch(x) {
  case 1: S;
  case 2: S; break;
  case 3: S; break;
}
```

Programmcode 5: switch-Anweisung mit Struktur-Anomalie

Die Anweisung *S* steht wieder für eine beliebige Java-Anweisung. Bei Fall 1 (case 1) steht am Ende des Anweisungsblocks kein break, daher wird mit Abschluss der Ausführung unmittelbar der Anweisungsblock von Fall 2 weiter ausgeführt, ohne dass das switch-Label hierzu geprüft wird. Damit werden mit dem Wert x = 1 mehrere Blöcke ausgeführt. Da derartige Programmkonstrukte kaum genutzt werden und auch in den Programmierrichtlinien in der Regel davon abgeraten wird, wird im GBT-Modell hierfür keine adäquate Abbildung bereitgestellt. Faktisch wird im GBT-Resultat dann die Zweigüberdeckung falsch ausgewiesen, weil der „case-2“-Block als ausgeführt bewertet wird (und damit die Kante zu diesem Block als durchlaufen), obwohl die Kante zu diesem Block möglicherweise nicht durchlaufen wurde.

Der switch-Ausdruck (im Beispiel das x) kann im Java-Programm ganzzahlig, ein Aufzählungstyp oder eine (ab Java 7) eine Zeichenkette sein. Natürlich kann dieser Ausdruck auch GBT-relevante Unterausdrücke enthalten. Diese werden als *SubExpressions* der Anweisung behandelt.

Java	RPR

"switch" "(" **Expression** ")"
{
{ "case" **ConstantExpression** ":" }
BlockStatements
}
["default" **BlockStatements**]
	"switch"
	{
	"case"
	StatementBlock
	}
	"default" StatementBlock

break-Anweisungen des Java-Programms, die Teil einer Fallunterscheidung sind, werden nicht in RPR übertragen. Der „Fall“ wird als Anweisungsblock abgebildet, wodurch sich in RPR das „break“ am Blockende erübrigt. Zudem ist in RPR „break“ für Schleifenabbrüche reserviert.

Weitere Java-Anweisungen

Zuweisungs-Anweisungen werden in die primitive Anweisung der Modellsprache übertragen. Die Anweisungen für abruptes Beenden haben in RPR jeweils eine namensgleiche Abbildung.

Eine Ausnahme bildet die Java-„Semikolon“-Anweisung (das sogenannte empty statement), die im Modell präteriert wird.

Java	RPR
"return" [**Expression**] ; "throw" **Expression** ; "break" [**Identifier**] "continue" [**Identifier**] ; StatementExpression ;	"return" [SubExpressions] "throw" [SubExpressions] "break" Identifier "continue" Identifier "stmt" [SubExpressions]

Für den Fall, dass bei diesen Anweisungen im Originalprogramm GBT-relevante Ausdrücke enthalten sind, werden diese als *SubExpressions* ins GBT-Modell übertragen. So werden z. B. GBT-relevante Parameter einer „return“-Anweisung als Subausdrücke behandelt.

5.13.5 Ausdrücke

Wie in Abschnitt 4.3.2 auf Seite 71 schon dargestellt wurde, werden nur boolesche und bedingte Ausdrücke in das GBT-Modell übertragen. Alle anderen Ausdrücke (wie z. B. numerische Ausdrücke, Zeichenketten oder Felder) werden im Modell unterdrückt. Einige Anweisungen des GBT-Modells sehen einen Bedingungsausdruck ausdrücklich als Teil der eigenen Struktur vor (z. B. die Schleife und die if-Anweisung), darüber hinaus kann zu jeder Anweisung des GBT-Modells eine Ausdruckssequenz abgebildet werden (vgl. Kapitel 4.3.2). Für die Abbildung dieser Ausdrücke in das GBT-Modell sind die folgenden Fälle zu betrachten:

1. Der Ausdruck ist ein **boolescher Ausdruck:**
   Der (binäre) Baum, der den booleschen Ausdruck des Java-Programms repräsentiert, wird in einen strukturgleichen Baum in das GBT-Modell übertragen. Die (binären) Operationen des Original-Programms werden dabei in die entsprechenden Ausführungselemente der zusammengesetzten Ausdrücke des GBT-Modells übertragen. Um die möglicherweise bestehenden Subausdrücke der Einzelterme auszuwerten, werden diese weiter nach Fall 3 analysiert.
2. Der Ausdruck ist ein **bedingter Ausdruck**:
   Im GBT-Modell wird der Repräsentant des bedingten Ausdrucks angelegt. Die Ausdrucksauswertung wird für den (booleschen) Bedingungsausdruck nach Fall 1 und für die beiden Alternativausdrücke nach Fall 3 fortgesetzt.
3. Sonst (z. B. ein numerischer Ausdruck):
   Den Ausgangspunkt bildet der (Teil-)Ableitungsbaum, der den (Rest-)Ausdruck repräsentiert. Von der Wurzel dieses Teilbaums ausgehend werden die Unterknoten in Vorordnung durchlaufen. Repräsentiert ein Knoten einen booleschen oder

bedingten Ausdruck, dann werden im GBT-Modell die Repräsentanten dieser booleschen und bedingten Ausdrücke angelegt und in die SubExpression-Liste gestellt. Für diese Ausdrücke werden die Ausdrucks-Analyse nach Fall 1 und 2 fortgesetzt, bis der Baum des Originalausdrucks vollständig ausgewertet ist

Zur praktischen Umsetzung dieser Baumauswertung bietet sich das sogenannte Visitor-Pattern an [Ga95]. Die drei oben gelisteten Fälle der Ausdrucksauswertung bilden für das Visitor-Pattern je einen Knotentyp, zu dem die Visitor-Klasse eine entsprechende Implementierung zum Traversieren des Ableitungsbaums enthält.

5.13.6 Ausführungszähler

Im GBT-Modell werden die Ausführungszähler als inhärenter Bestandteil der Ausführungselemente angenommen. In Java sind diese Zähler natürlich nicht enthalten. Für eine Übertragung der Eigenschaften des GBT-Modells auf den „realen" Prüfling müssen somit die Ausführungszähler im Java-Programmcode ergänzt werden; in der Literatur wird hier von Instrumentierung gesprochen (vgl. Abschnitt 3.3 und 0). Grundsätzlich können die Ausführungszähler des GBT-Modells in Java durch Integer-Variablen implementiert werden. Falls von sehr vielen Ausführungen auszugehen ist, muss der Integer-Überlauf berücksichtigt werden. Dabei wird der Ausführungszähler dann nicht weiter erhöht, wenn der Maximalwert für den verwendeten Integer-Datentyp erreicht ist. Ausführungszähler für Anweisungen lassen sich ansonsten relativ einfach durch ergänzte Inkrement-Anweisungen erheben. Bei booleschen Ausdrücken ist die Platzierung des Zählerinkrements etwas aufwändiger. Im Folgenden wird die Instrumentierung für Anweisungen, Anweisungsblöcke und Ausdrücke ausführlich beschrieben.

Ausführungszähler für Anweisungen

Vor und nach einer Anweisung des Java-Programms werden Inkrement-Anweisungen der Ausführungszähler im Programmcode ergänzt. Das Zählerinkrement vor einer Anweisung wird im Folgenden als Pre-Zähler bezeichnet. Das Zählerinkrement nach der Anweisung (im Folgenden als Post-Zähler bezeichnet) hat den Zweck, die normal beendeten Ausführungen zu erfassen. Aus der Differenz zwischen dem Post-Zähler und dem Pre-Zähler lässt sich so die Anzahl von abrupt beendeten Ausführungen ermitteln (vgl. Kapitel 0 auf Seite 112). Für Anweisungen, die innerhalb eines Anweisungsblocks folgen, ist der Post-Zähler der i. Anweisung auch der Pre-Zähler der Anweisung i + 1. In Java gibt es allerdings einige Anweisungen, die keine direkt folgende Anweisung und damit auch keinen Post-Zähler zulassen. Konkret sind dies die Anweisungen return, break, continue und throw. Alle diese Anweisungen führen nach Java-Spezifikation zu abruptem Beenden. Für alle anderen Anweisungen sind Pre- und Post-Ausführungszähler möglich. Zur konkreten Umsetzung der Pre- und Post-Zähler bietet sich pro Java-Klasse ein Array an, das groß genug ist, um alle erforderlichen Zähler aufzunehmen. Diese Umsetzung hat zwei Vorteile: Zum einen bleibt die Anzahl der ergänzten Zählerattribute gering – auch bei sehr vielen erforderlichen Ausführungszählern ist nur ein Array erforderlich – und zum anderen muss zur Auswertung der Zählerstände nur ein Array und nicht viele ein-

zelne Variablen je Klasse betrachtet werden. Jede Anweisung des Prüflings wird mit einem eindeutigen Index versehen, der den Zugriff auf das Array steuert. Für eine Klasse A wird somit die folgende fett gedruckte Instrumentierung vorgenommen:

```
class A {
  final int AMOUNT_COUNTER = ...
  long exeCounter[] = new long[AMOUNT_COUNTER];

  void foo() {

    ...
    exeCounter[i]++;        // Pre-Zähler
    stmt;                   // das i. Statement
    exeCounter[i + 1]++;    // Post-Zähler
    ...
  }
}
```

Programmcode 6: Instrumentierung der Anweisungen

Ausführungszähler für Anweisungsblöcke: Als Ausführungszähler für nicht leere Blöcke können aus Effizienzgründen die Pre-Zähler der ersten Anweisung und der Post-Zähler der letzten Anweisung mitgenutzt werden. Leere Blöcke erhalten nur einen Pre-Zähler; da abruptes Beenden nicht vorkommen kann, erübrigt sich der Post-Zähler.

Die abrupt endenden Anweisungen break, continue, return oder throw erhalten nur einen Pre-Zähler (vgl. Abschnitt 5.8 auf Seite 106).

Ausführungszähler für Ausdrücke

Die Ausführungszähler für Ausdrücke sind deutlich schwieriger in Java-Programmcode zu integrieren als die Ausführungszähler für Anweisungen oder Blöcke. Während bei Anweisungen die bereits beschriebenen Pre- und Post-Zähler einfach als weitere Anweisung ergänzt werden können, kann innerhalb eines booleschen Ausdrucks nicht ohne Weiteres ein Zählerinkrement als Anweisung eingefügt werden. Es können aber Operationen in einen Ausdruck eingefügt werden. Wie ein boolescher Ausdruck um die erforderlichen Ausführungszähler erweitert wird, sodass sein Wert und die Ausführungssemantik unverändert bleiben, wird in Folgenden schrittweise beschrieben. Hierzu wird ein boolescher (Original-)Ausdruck

```
A
```

mit einem angenommenen RPR-Identifier *ID_A* in einem ersten Schritt zu einem *A'* erweitert:

```
A' = exeCounter(ID_A) & A
```

Anstelle von *A* kann ein beliebiger boolescher Ausdruck angenommen werden wie z. B. *„kunde.getUmsatz() > 2000“*.

exeCounter ist als eine Methode anzunehmen, die einen Ausführungszähler inkrementiert und als Rückgabewert immer true zurückliefert. Da der &-Operator (logischer Und-Operator) ohne Kurzschlusssemantik arbeitet und die Ausführungsabfolge von links nach rechts gilt, wird *exeCounter* immer vor A ausgeführt, *exeCounter* bildet damit den Pre-Zähler des Ausdrucks. Wie man leicht sieht, gilt für den Wert von *A'* bei *exeCounter(ID_A) = true*:

```
A' = exeCounter(ID_A) & A                                    (1)
   = true & A
   = A
```

Damit ändert dieses Einfügen des Pre-Zählers den Ausdruckswert nicht. Allerdings wird so nur erhoben, ob *A* ausgeführt wird, nicht aber, welchen booleschen Wert der Ausdruck *A* hat. Hierzu dient ein weiterer ergänzter Operator, um den der Ausdruck *A'* erweitert wird:

```
A'' = A' && trueCounter(ID_A)
```

Da der &&-Operator das logische Und mit Kurzschlusssemantik durchführt, wird der Operand *trueCounter(ID_A)* nur dann ausgeführt, wenn der erste Operand *A'* den Wert true hat. Auf diese Weise werden alle Ausführungen mit *A' = true* gezählt. Da *trueCounter* immer den Wert true zurückliefert, gilt für *A''*:

```
A'' = A' && trueCounter(ID_A)                                (2)
    = A' && true
    = A'
```

Und mit (1) ergibt sich *A'' = A*; d. h. der Ausdruckswert ändert sich durch die ergänzten Ausdrücke nicht. Um auch bei false-Resultaten von A einen Zähler inkrementieren zu können, wird ein weiterer Operator ergänzt:

```
A''' = A'' || falseCounter(ID_A)
```

Da der ||-Operator das logische Oder mit Kurzschlusssemantik durchführt, wird der Operand *falseCounter(ID_A)* nur dann ausgeführt, wenn der erste Operand *A''* den Wert false hat. Auf diese Weise werden alle Ausführungen mit *A'' = false* gezählt. Da *falseCounter* immer den Wert false zurückliefert, gilt für *A'''*:

```
A''' = A'' || falseCounter(ID_A)                             (3)
     = A'' || false
     = A''
```

Da die Methoden *exeCounter*, *trueCounter* und *falseCounter* lediglich den entsprechenden Ausführungszähler für den angegebenen Ausdruck inkrementieren müssen, kann aus Effizienzgründen statt des Methodenaufrufs dieses Zählerinkrement auch direkt den

Original-Ausdruck integriert werden. Die RPR-ID des Ausdrucks wird dann als Index eines Arrays aufgefasst. Das folgende Beispiel kann dies veranschaulichen: Der Original-Java-Programmcode

```
class A {
   ...
  void foo() {
    ...
    if(a) {  ... }
   }
}
```

wird damit in den instrumentierten Programmcode nach Programmcode 7 transformiert.

```
class A {
  final int AMOUNT_EXPR = ...;
  long exeCounter[] = new long[AMOUNT_EXPR];
  long trueCounter[] = new long[AMOUNT_EXPR];
  long falseCounter[] = new long[AMOUNT_EXPR];

  void foo() {
    ...
    if(
     (exeCounter[ID_A]++ >= 0) &
     (
     a
     && (trueCounter[ID_A]++ >= 0)
     )
     || (falseCounter[ID_A]++ < 0)
     ) { ... }
   }
}
```

Programmcode 7: Instrumentierung der booleschen Ausdrücke

Die Vergleichsoperationen „>= 0“ und „< 0“ sind deswegen zu der Inkrementausdrücken ergänzt worden, damit der Ausdruck syntaktisch als boolescher Ausdruck gewertet wird und das oben angegebene boolesche Resultat zurückgeliefert wird.

Abbildung 44 zeigt den bereits in Abbildung 42 auf 114 beschriebenen Zusammenhang zwischen den Zählerstellen des Modellnetzes eines Ausdrucks und den in den Ausdruck eingefügten Ausführungszählern.

Die Datenflussrichtung der Ausführungszähler ist wie in Abbildung 42 vom instrumentierten Programm in das Modellnetz. D. h. die Ausführung des instrumentierten Programms liefert die Auskunft über die Markierung der Zählerstellen des Modells.

Von großer Bedeutung ist aber, dass die Vorgabe für die Platzierung der Ausführungszähler aus dem Modellnetz stammt. Die Ausführungszähler werden ausgehend von den Modellnetz-Anforderungen exakt so in das Originalprogramm eingefügt, dass der Zählerwert gebildet wird, der der Ausführung des Modells entspricht.

Gelingt es für eine Programmiersprache nicht, einen Zähler in der geforderten Zählweise in das Programm einzufügen, dann sind die Schlussfolgerungen, die auf der jeweiligen Zählerstelle des Modells basieren, für diese Programmiersprache unzulässig.

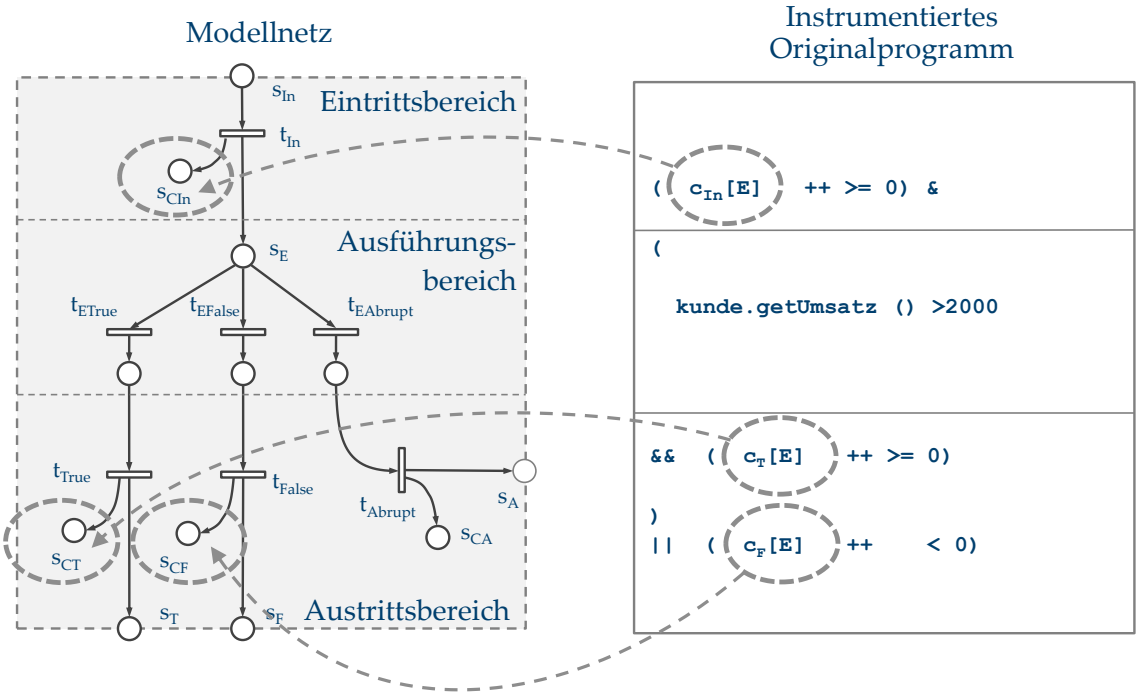


Abbildung 44: Modellnetz des Ausdrucks mit instrumentiertem Programmcode

6 Definition der Überdeckungsmetriken für den Glass-Box-Test

Auf Grundlage des GBT-Referenzmodells werden in diesem Kapitel die populären GBT-Überdeckungsmetriken sowie weitere neu entwickelte Überdeckungsmetriken präzise definiert. Hierzu wird zunächst eine allgemeine Definition für Überdeckungsmetrik und Überdeckung geliefert. Danach wird ein Regelwerk aufgestellt, mit dem Überdeckungsmetriken auf Plausibilität, Differenziertheit und Vergleichbarkeit geprüft werden können. Anschließend werden die verschiedenen Überdeckungsmetriken definiert und anhand des Regelwerks geprüft.

6.1 Eine allgemeine Definition

Auch wenn die verschiedenen GBT-Überdeckungsmetriken unterschiedliche Programmkonstrukte und deren unterschiedliches Ausführungsverhalten betrachten, ist das der jeweiligen Metrikdefinition zugrundeliegende Prinzip sehr ähnlich. Alle GBT-Metriken dieser Arbeit basieren auf Ausführungszählern der Anweisungen oder Blöcke, Resultatszählern der Ausdrücke oder speziellen Zählern, wie sie z. B. zur Ermittlung der Schleifenüberdeckung genutzt werden. Auf Grundlage dieser Zähler ist eine allgemeine Definition von Überdeckungsmetrik und Überdeckung möglich.

Seien

- P ein Programm
- T eine Testsuite zum Test von P
- π die Menge der Zähler im Modell von P
- $\pi_m \subseteq \pi$ eine Teilmenge der Zähler von P
- $\omega \in \pi_m$ ein Zähler von P

Dann ist $\varphi(P, T, \omega)$ definiert durch

$$\varphi(P, T, \omega) = \begin{cases} 0, wenn\ \omega = 0 \\ 1, wenn\ \omega > 0 \end{cases}$$

nach Ausführung von P mit T. Die Überdeckung $c_m(P, T)$ ist dann

$$c_m(P, T) = c(P, T, \pi_m) = \frac{\sum_{\omega \in \pi_m} \varphi(P, T, \omega)}{|\pi_m|}$$

Darin ist $|\pi_m|$ die Kardinalität von π_m. Eine Metrik m ergibt sich folglich durch eine bestimmte Wahl von $\pi_m \subseteq \pi$.

Def: Eine Testsuite T heißt für eine Metrik m und ein Programm P **redundanzfrei**, *wenn kein Testfall aus T entnommen werden kann, ohne die Überdeckung der Metrik m zu reduzieren. T ist bezogen auf P und m redundanzfrei* $\Leftrightarrow\ \forall t \in T : c_m(P, T \setminus t) < c_m(P, T)$

Def. Bezogen auf eine *Metrik m* führt ein Testfall $t \in T$ *einen Zähler* $\omega \in \pi_m$ **exklusiv** *aus* $\Leftrightarrow\ \varphi(P, \{ t \}, \omega) = 1 \wedge \forall t' \in T, t' \neq t: \varphi(P, \{ t' \}, \omega) = 0$

6.2 Plausibilitätsregeln für Glass-Box-Test-Überdeckungsmetriken

Mit dem Ziel, Überdeckungsmetriken auf Plausibilität, Differenziertheit und Vergleichbarkeit theoretisch validieren zu können, werden im Folgenden fünf Plausibilitätsregeln aufgestellt, die vergleichbar dem Axiomensystem von Weyuker [We88] angelegt sind.

Für die nachfolgend beschriebenen Plausibilitätsregeln gibt es jeweils gute (man könnte sagen „handfeste“) Gründe, die auch argumentiert werden. Es ist die These, dass Überdeckungsmetriken, die die Plausibilitätsregeln nicht erfüllen, praktische Nachteile haben. Die Liste dieser Plausibilitätsregeln ist aber keinesfalls abgeschlossen.

1. Null-Überdeckung bei leerer Testsuite
   Die leere Testsuite führt zu keiner Überdeckung.

   $$T = \emptyset \Rightarrow c_m(P, T) = 0$$

   Mit $T = \emptyset$ wird das Programm nicht ausgeführt und es kann folglich auch zu keiner Überdeckung kommen.

2. Normierung
   Um eine Vergleichbarkeit der Überdeckungen zu ermöglichen, soll die Überdeckung normiert sein:

$$0 \leq c_m(P, T) \leq 1$$

Die Angabe der Überdeckung erfolgt damit üblicherweise in Prozent.

3. Nicht fallende Monotonie
Seien T_1 und T_2 je eine Testsuite für das Programm *P*. Dann kann die Testsuite T_1, die eine Teilmenge von T_2 bildet, nie eine höhere Überdeckung erzielen als T_2.

$$T_1 \subseteq T_2 \Rightarrow c_m(P, T_1) \leq c_m(P, T_2)$$

Ohne diese schwache Monotonie könnte das Hinzufügen eines Testfalls in eine Testsuite die Überdeckung verringern, oder das Entnehmen eines Testfalls könnte die Überdeckung erhöhen, was nicht plausibel wäre. Gleichwohl gibt es keine Garantie, dass ein weiterer Testfall zur Erhöhung der Überdeckung beiträgt. Die nicht fallende Monotonie bedeutet aber auch, dass eine Teilmenge T_T von T im günstigsten Fall die gleiche Überdeckung wie T erreicht. Damit liegt die obere Grenze der Überdeckung für beliebige Teilmengen von T fest.

4. Erfüllbarkeit
Es muss ein Programm P mit einer Testsuite *T* geben, sodass eine vollständige Überdeckung erzielt wird.

$$\exists T: c_m(P, T) = 1$$

Bei der sogenannten Pfadüberdeckung kann, wenn Schleifen im Programm enthalten sind, die Menge der ausführbaren Pfade unendlich groß werden. Eine vollständige Überdeckung ist dann mit keiner (endlichen) Testsuite zu erzielen.

5. Wachsende Monotonie
Eine Überdeckungsmetrik $c_m(P, T)$ soll monoton wachsen. Dies ist dann erfüllt, wenn jede Teilmenge einer redundanzfreien Testsuite wieder redundanzfrei ist.

Überdeckungsmetriken, die nicht monoton wachsend sind, haben praktische Nachteile. Das Beispiel einer speziellen Zweigüberdeckung kann dies veranschaulichen: Für einen Verzweigungsknoten mit zwei ausgehenden Zweigen soll die Überdeckung so lange mit null bewertet werden, bis beide Zweige durchlaufen sind. Der Testfall, der den ersten Zweig ausführt, würde die Überdeckung damit unverändert bei null belassen. Erst ein zweiter Testfall, der den zweiten Zweig durchläuft, führt zur Erhöhung der Überdeckung. Eine solche Überdeckungsmetrik wäre nicht plausibel und zudem unpraktisch, weil der Tester mit dem ersten Testfall keine Erhöhung der Überdeckung angezeigt bekommt und so keinerlei Hinweis darauf hat, dass der Testfall einen Beitrag zum angestrebten Testziel leistet.

Natürlich ist die beschriebene Definition der Zweigüberdeckung abwegig – die stark verbreitete MC/DC-Überdeckung hat aber sehr ausgeprägt das beschriebene nicht streng monotone Verhalten.

6.3 Kontrollflussbasierte Überdeckungsmetriken

Im Folgenden werden die populären kontrollflussbasierten Überdeckungsmetriken der Literatur auf Grundlage des GBT-Modells definiert. Soweit möglich, wird auf die Definitionen der Literatur genau Bezug genommen, wobei gerade bei den einfachen Metriken deren mangelnde Präzision im Wege steht.

6.3.1 Anweisungsüberdeckung

Die Anweisungsüberdeckung ist eine der ältesten GBT-Überdeckungsmetriken und wird in der Literatur in der Regel auf Grundlage des CFG definiert. Dann entspricht ein Knoten des CFG einer Anweisung des Programms, sodass die Anweisungsüberdeckung mit einer Knotenüberdeckung gleichgesetzt werden kann. Das GBT-Modell dieser Arbeit enthält dagegen viele Stellen und Transitionen, die gar nicht oder nicht unmittelbar mit einer Anweisung korrespondieren wie z. B. Anweisungsblöcke oder Ausdrücke. Aus diesem Grunde wird zunächst die Gesamtmenge aller Anweisungen eines Programms definiert: Unter Anweisungen werden die primitiven Anweisungen und die Verbundanweisungen verstanden, also: if-Anweisungen, switch-Anweisungen, while-Anweisungen, primitive Anweisungen und try-Anweisungen. Und zu dieser Menge von Anweisungen eines Programms kann die Teilmenge ausgeführter Anweisungen bestimmt werden.

Def.: Seien P ein Programm und T die Testsuite für P. **stmts(P)** *ist die Menge aller Anweisungen von P*

exeStmts(P, T) $\subseteq stmts(P)$ *sei die Menge der ausgeführten Anweisungen von P mit T.*

$A \in exeStmts\ (P, T) \Leftrightarrow \exists t \in T : exe(A, t)$

Dann ist die **Anweisungsüberdeckung** *der Anteil der ausgeführten Anweisungen:*

$$stmtCov(P, T) = \frac{| exeStmts(P, T) |}{| stmts(P) |}$$

In [Li02] wird die Anweisungsüberdeckung ebenfalls als Quotient der Anzahl der ausgeführten Anweisungen und der Anzahl aller Anweisungen definiert. Die Grundlage bildet der Kontrollflussgraph, für den allerdings keine präzise Herleitung aus einer Programmiersprache erfolgt. Da aber generell zwei Knoten für Beginn und Ende des Programms ergänzt werden, wird sich der Wert für die Anweisungsüberdeckung nach [Li02] vom hier berechneten Wert entsprechend unterscheiden. Für vollständige Überdeckung sind beide Definitionen aber gleich. Auch stimmt die Definition für vollständige Überdeckung mit der Definition für Anweisungsüberdeckung nach [ZHM97] überein: „*(Statement*

Coverage Criterion). A set P of execution paths satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path p in P such that node n is on the path p."

Eine deutlich präzisere Definition für „Anweisung" liefert [FAA07, Kapitel 4.1]. Auch sie stimmt für vollständige Überdeckung mit der Definition in dieser Arbeit überein: "*The term statement itself is not defined in DO-178B, but it is relatively well understood within the software community. […] In procedural languages often used in airborne applications, such as C, C++, and Ada, a simple delimiter, such as a semicolon, is used to set simple statements apart. When flow of control must be selected based on current data values, compound statements, such as if, case, and loop statements, are used. Coverage for compound statements is understood to be coverage for the portion of the statement that evaluates the flow-controlling expression(s) and coverage for all other statements (applied recursively) contained within the compound statement.* Explizit wird damit auch auf die Überdeckung der Verbundanweisungen eingegangen. [FAA07] definiert allerdings nur die vollständige Anweisungsüberdeckung. Aus diesem Grund wird für die Überdeckung einer Verbundanweisung (compound statement, wie z. B. if- oder while-Anweisungen) die Überdeckung aller eingebetteten Anweisungen gefordert. Nach der Definition dieser Arbeit werden auch die von [FAA07] genannten Verbundanweisungen zur Bewertung der Anweisungsüberdeckung einbezogen. Eine Verbundanweisung gilt allerdings bereits dann schon als überdeckt, wenn sie nur teilweise ausgeführt wird. Am folgenden Beispiel lässt sich dies leicht veranschaulichen:

```
S1: if( ... ) B1: {
  S2: stmt
} else B2: {
  S3: stmt
}
```

Für eine Testausführung, bei der die if-Anweisung S1 mit der Anweisung des then-Blocks (S2) ausgeführt wird, ergibt sich folgende Überdeckung: Wir zählen insgesamt drei Anweisungen (die if-Anweisung zählt auch als Anweisung) wobei für die angenommene Testausführung die if-Anweisung S1 sowie S2 ausgeführt sind. Es ergibt sich folglich eine Überdeckung von 2/3. Eine Forderung wie in [FAA07], wonach eine Verbundanweisung erst dann als überdeckt gilt, wenn sämtliche (rekursiv) eingebetteten Anweisungen ebenso überdeckt sind, ist für eine Überdeckungsmetrik nicht sinnvoll. Dass Verbundanweisungen zur Anweisungsüberdeckung beitragen sollen, kann leicht am folgenden praxisüblichen Code-Beispiel der Sprache C demonstriert werden:

```
...
for(p = start; *p != NULL; p++);
...
```

Die Beispielanweisung "bewegt" einen Zeiger von einem Startwert auf einen Endwert und hat damit offensichtlich die Charakteristik einer Anweisung.
Das abrupte Beenden der Anweisungen stellt dagegen ein Problem aller genannten Definitionen der Anweisungsüberdeckung dar. In der Definition dieser Arbeit gilt eine An-

weisung dann als ausgeführt und damit zur Anweisungsüberdeckung beitragend, wenn die Ausführung „begonnen“ wurde, also entweder normal oder abrupt endet. Der Grund hierfür sind die immer abrupt endenden Anweisungen. Beispielsweise enden in Java die Anweisungen return, break, continue oder throw immer abrupt, d. h. diese Anweisungen enden nie normal und würden ansonsten nie als ausgeführt bewertet. Eine 100 prozentige Anweisungsüberdeckung wäre sonst unmöglich.

Untersuchung der Metrik

Im Folgenden wird untersucht, ob die Plausibilitätsregeln, die für Überdeckungsmetriken in Kapitel 6.2 aufgestellt wurden, gelten. Sei P ein GBT-Modellprogramm und T eine Testsuite zum Test von P.

Plausibilitätsregel 1: Null-Überdeckung bei leerer Testsuite	Die Regel gilt. Für T = Ø gilt *exeStmts(P, T)* = Ø. Daraus folgt stmtCov(P,T) = 0
Plausibilitätsregel 2: Normierung	Die Regel gilt. Definitionsbedingt gilt *exeStmts(P, T)* $\subseteq$ stmts(P) Daraus folgt $0 \leq$ stmtCov(P,T) ≤ 1
Plausibilitätsregel 3: Nicht fallende Monotonie	Die Regel gilt. Wenn $T_1 \subseteq T_2$, dann gilt *exeStmts(P,* T_1*)* $\subseteq$ *exeStmts(P,* T_2*).* Daraus folgt stmtCov (P, T_1) $\leq$ stmtCov(P, T_2)
Plausibilitätsregel 4: Erfüllbarkeit	Die Regel gilt für Programme ohne „toten“ Code. Für alle realen Programme gilt $0 < \mid stmts(P) \mid < \infty$. Wenn alle Anweisungen ausgeführt werden, gilt *exeStmts(P, T)* = stmts(P) und damit stmtCov(P,T) = 1
Plausibilitätsregel 5: Wachsende Monotonie	Die Regel gilt. Eine Testsuite T ist nur dann hinsichtlich der Anweisungsüberdeckung redundanzfrei, wenn jeder Testfall $t \in T$ exklusiv mindestens eine Anweisung $S \in stmtCov(P,T)$ ausführt. Diese Eigenschaft der „Exklusiv-Ausführung“ bleibt für alle Teilmengen von T erhalten. Daraus folgt, dass jede Teilmenge $T_T \subseteq$ T einer redundanzfreien Testsuite T wieder redundanzfrei ist.

6.3.2 Zweigüberdeckung

Die Definition der Zweigüberdeckung (branch coverage) orientiert sich an der Definition von Riedemann [Rie97], der die Zweigüberdeckung auf Grundlage des CFG definiert. Nach Riedemann tragen nur die sogenannten Entscheidungskanten zur Zweigüberde-

ckung bei. Entscheidungskanten sind die Kanten im CFG, die von Verzweigungsknoten ausgehen, also von Knoten mit mehr als einer ausgehenden Kante. Er definiert die Zweigüberdeckung als den Anteil der im CFG durchlaufenen Entscheidungskanten. Boolesche Ausdrücke, die nicht das Prädikat einer Verzweigung bilden, wirken sich bei Riedemann nicht auf die Zweigüberdeckung aus. Aus diesem Grund lässt sich die Zweigüberdeckung nicht einfach auf Grundlage der booleschen Ausdrücke des GBT-Referenzmodells definieren. Auch wird die Verzweigung der Fallunterscheidungen im GBT-Referenzmodell nicht von booleschen Ausdrücken bestimmt. Zur Definition der Zweigüberdeckung auf Grundlage des GBT-Referenzmodells werden daher zunächst die Verzweigungsstellen eines GBT-Modellprogramms festgelegt. Im zweiten Schritt wird definiert, wann eine Kante, die von einer Verzweigungsstelle ausgeht, im Test als „durchlaufen" bewertet wird. Verzweigungsstellen in diesem Sinne liegen bei den Verbundanweisungen Entscheidung, Fallunterscheidung und Schleife vor. Der Algorithmus zur Berechnung der Zweigüberdeckung ist so angelegt, dass jede verzweigende Kontrollstruktur *K* ein Gewicht $w(K) \rightarrow \mathbb{N} \setminus \{ 1 \}$ zugewiesen bekommt. Das Gewicht entspricht der Anzahl der ausgehenden Kontrollflusskanten, wobei die Kante für abruptes Beenden nicht eingerechnet wird; da zur Vollständigkeit der Zweigüberdeckung natürlich nicht das abrupte Beenden verlangt wird. Zudem erhält jede Kontrollstruktur *K* für eine Testsuite *T* einen Ausführungswert $e(K, T) \rightarrow \mathbb{N}$ mit $0 \leq e(K, T) \leq w(K)$, der die Anzahl der für *T* ausgeführten Zweige angibt. Da boolesche Ausdrücke im Folgenden ganz entscheidend den Ausführungswert der Kontrollstrukturen bestimmen, wird der Ausführungswert $e(C, T)$ eines booleschen Ausdrucks wie folgt definiert:

$$e(C,T) = \begin{cases} 2, & \exists\, t_1 \in T : exeT(C,t_1) \;\wedge\; t_2 \in T : exeF(C,t_2) \\ 1, & \exists\, t_1 \in T : exeT(C,t_1) \;\wedge\; \neg\exists\, t_2 \in T : exeF(C,t_2) \\ 1, & \neg\exists\, t_1 \in T : exeT(C,t_1) \;\wedge\; \exists\, t_2 \in T : exeF(C,t_2) \\ 0, & sonst \end{cases}$$

Für einen Testfall *t* und einen booleschen Ausdruck *C* ist $exeF(C, t)$ bzw. $exeT(C, t)$ dann erfüllt, wenn *C* bei der Ausführung mit den Eingabedaten von *t* mindestens einmal das Resultat false bzw. true hat (vgl. Definition aus Abschnitt 0 auf Seite 112).

Für eine Entscheidungsanweisung *I* mit der Bedingung *C* gilt:

$$w(I) = 2$$
$$e(I, T) = e(C, T)$$

Für eine Schleife *S* mit der Wiederholbedingung *C* gilt:

$$w(S) = 2$$
$$e(S, T) = e(C, T)$$

Für eine Fallunterscheidung *S* mit n case-Blöcken C_i ($1 \leq i \leq n$) und einem default-Block D gilt:

$$w(S) = n + 1$$
$$e(S,T) = \sum_{i=1}^{n} \begin{cases} 1, & \exists\, t \in T : exe(C_i, t) \\ 0, & sonst \end{cases} + \begin{cases} 1, & \exists\, t \in T : exe(D, t) \\ 0, & sonst \end{cases}$$

Während die bislang betrachteten verzweigenden Anweisungen leicht zu berücksichtigen waren, stellt die Ausnahmebehandlung für die Zweigüberdeckung ein Problem dar: Da es im try-Block in der Regel keinen bestimmten Verzweigungsknoten zu den Ausnahmebehandlungen gibt, würde nach der bislang geltenden Definition die try-Anweisung nicht zu den verzweigenden Anweisungen zählen. Allerdings könnte dann eine vollständige Zweigüberdeckung erzielt werden, ohne dass die catch-Blöcke ausgeführt wären. Aus diesem Grund werden die try-Anweisungen in die Verzweigungsanweisungen mit aufgenommen. Der try-Block selber gilt aber nicht als Verzweigung.

Für eine try-Anweisung S mit n catch-Blöcken C_i ($1 \le i \le n$) gilt damit:

$$w(S) = n$$
$$e(S,T) = \sum_{i=1}^{n} \begin{cases} 1, & \exists\, t \in T : exe(C_i, t) \\ 0, & sonst \end{cases}$$

Abweichend zu dieser Definition beziehen die Definitionen zur Zweigüberdeckung der Literatur (z. B. [Rie97, Li02, ZHM97]) die Ausnahmebehandlung nicht ein, da das dort zugrundeliegende Modell keine Ausnahmen kennt.

Die Zweigüberdeckung für ein Programm P und eine Testsuite T ergibt sich dann aus der Summe der Ausführungswerte der verzweigenden Kontrollstrukturen, normiert auf deren Gesamtgewicht.

Def. Sei forkStmts(P) die Menge aller verzweigenden Verbundanweisungen des Programms P.

Dann ist die **Zweigüberdeckung** *einer Testsuite T für ein Programm P:*

$$branchCov(P,T) = \frac{\sum_{i \in forkStmts(P)} e(i,T)}{\sum_{i \in forkStmts(P)} w(i)}$$

Anmerkung zu continue- und break-Zweigen

Abbrüche wie beispielsweise in der Sprache Java durch die Anweisungen continue und break führen in Schleifen zu abruptem Beenden des Schleifenkörpers, sodass beispielsweise im CFG für eine continue-Anweisung des Programms eine Kante zum Knoten des Schleifenkopfs gezogen wird. Abbildung 45 zeigt ein solches Beispielprogramm mit zugehörigem CFG. Nach der Definition von [Rie97] tragen diese Zweige aber nicht zur Zweigüberdeckung bei, weil diese Zweige keine Entscheidungszweige sind. Wie in Abbildung 45 leicht zu erkennen ist, verzweigt sich der Kontrollfluss im continue-Knoten nicht. Bereits [NS73] weisen darauf hin, dass es kein unbedingtes break oder continue in einer Schleife geben kann, weil der Programmcode nach dem break oder continue ansons-

ten nicht erreichbar wäre. Am Beispielprogramm ist dies leicht zu erkennen: Die auskommentierte Anweisung S2 wäre nicht erreichbar, und die Übersetzer der gängigen Programmiersprachen erkennen das auch als Fehler. Damit können die Anweisungen für abruptes Beenden (break, continue, throw oder return) nur am Ende eines „bedingten" Blockes wie z. B. eines then- oder else-Blocks stehen. Diese der Abbruch-Anweisung vorausgehende Verzweigung (im Beispiel die Verzweigung (if(E2) …) trägt zur Zweigüberdeckung bei, nicht aber die abbrechende Anweisung selbst. Das bedeutet, dass break-, continue-, throw oder return-Anweisungen nicht bei der Bestimmung der Zweigüberdeckung berücksichtigt werden müssen.

```
while(E1) {
  if(E2) {
    S1;
    continue;
    // S2;
  } else {
    S3;
  }
  S4;
}
```

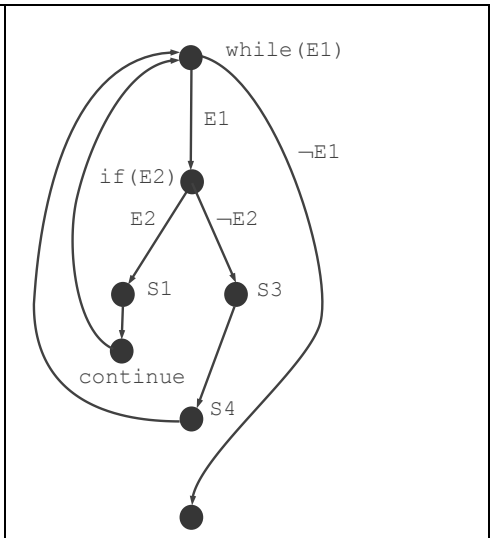


Abbildung 45: Beispielprogramm mit continue-Anweisung

Anmerkung zur Fallunterscheidung

Viele der gängigen Programmiersprachen erlauben die Angabe von mehreren der sogenannten switch-Labels für einen switch-Block. Das Beispiel in Abbildung 46 zeigt einen solchen Fall. Die Labels 2, 3 und 4 münden in denselben Anweisungsblock *Block2*.

```
switch(x) {
  case 1: /* Block1 */ ; break;
  case 2:
  case 3:
  case 4: /* Block2 */ ; break;
}
```

Abbildung 46: switch-Anweisung

Nach der Definition dieser Arbeit wirken sich diese einzelnen Labels aber nicht auf die Zweigüberdeckung aus. Es wird lediglich die Ausführung von *Block2* berücksichtigt und es wird nicht berücksichtigt, über welche der Labels diese Ausführung zustande kam. Übertragen auf einen CFG liegt ausgehend vom switch-Ausdruck nur eine Kante zu *Block2* vor, die mit der Bedingung ($x = 2 \lor x = 3 \lor x = 4$) attributiert ist. Ein CFG, der aus-

gehend vom switch-Ausdruck drei Kanten zu *Block2* vorsieht, wäre aber genauso plausibel. Leider liefert die Literatur hierzu keinen Anhaltspunkt. An diesem Punkt ist aber eine zukünftige Modellerweiterung denkbar. Es wird dann die Anzahl der Verzweigungen der switch-Anweisung nicht durch die Zahl der switch-Blöcke, sondern durch die Zahl der switch-Labels bestimmt, und der Ausführungswert ergibt sich durch die im switch-Ausdruck ermittelten Werte. Die Definition der Metrik bleibt aber ansonsten unverändert.

Untersuchung der Metrik

Im Folgenden wird untersucht, ob die Regeln, die für Überdeckungsmetriken in Kapitel 6.2 aufgestellt wurden, gelten. Sei *P* ein GBT-Modellprogramm, das Verzweigungen enthält (d. h. $| forkStmts(P) | > 0$) und *T* eine Testsuite zum Test von *P*.

Plausibilitätsregel 1: Null-Überdeckung bei leerer Testsuite	Die Regel gilt. Für T = Ø gilt für alle A ∈ forkStmts(P): e(A, T) = 0 und damit $\sum_{i \in forkStmts(P)} e(i,T) = 0$ Und daraus folgt $branchCov(P,T) = 0$
Plausibilitätsregel 2: Normierung	Die Regel gilt. Definitionsbedingt gilt ∀A ∈ forkStmts(P) : 0 ≤ e(A, T) ≤ w(A) ∧ w(A) > 0. Damit gilt $0 \leq \sum_{i \in forkStmts(P)} e(i,T) \leq \sum_{i \in forkStmts(P)} w(i)$ Und daraus folgt 0 ≤ branchCov(P,T) ≤ 1
Plausibilitätsregel 3: Nicht fallende Monotonie	Die Regel gilt. Ein Testfall einer Testsuite kann immer nur erhöhend auf den Ausführungswert der verzweigenden Kontrollstrukturen wirken. Damit gilt für $T_1 \subseteq T_2$: ∀A ∈ forkStmts(P): e(A, T_1) ≤ e(A, T_2). Und damit auch $\sum_{i \in forkStmts(P)} e(i,T_1) \leq \sum_{i \in forkStmts(P)} e(i,T_2)$ Daraus folgt $branchCov(P,T_1) \leq branchCov(P,T_2)$
Plausibilitätsregel 4: Erfüllbarkeit	Die Regel gilt für Programme ohne „toten" Code. Für alle realen Programme gilt $0 < | forkStmts(P) | < \infty$, und

	das Gewicht jeder Verzweigungsstelle ist endlich. Wenn für eine Testsuite T und alle Verzweigungsstellen A des Programms gilt e(A, T) = w(A), dann gilt auch $branchCov(P,T) = 1$.
Plausibilitätsregel 5: Wachsende Monotonie	Die Regel gilt. Eine Testsuite T ist nur dann hinsichtlich der Zweigüberdeckung redundanzfrei, wenn jeder Testfall $t \in T$ exklusiv mindestens einen Zweig ausführt. Diese Eigenschaft der „Exklusiv-Ausführung“ bleibt für alle Teilmengen von T erhalten.

6.3.3 Blocküberdeckung

Die Blocküberdeckung gibt den Anteil der ausgeführten Anweisungsblöcke des Programms wider. Anweisungsblöcke bilden in den Programmiersprachen (und im GBT-Modell) eine zentrale Programmstruktur; Beispiele für Anweisungsblöcke sind der then- oder else-Block der if-Anweisung, die case-Blöcke der Fallunterscheidung oder der Schleifenkörper.

Die Blocküberdeckung hat im Gegensatz zur Zweigüberdeckung ihren Ursprung weniger in der Literatur als darin, dass einige GBT-Werkzeuge Überdeckungswerte zur Blocküberdeckung liefern – allerdings ohne dass die Werkzeuge eine Definition für diese Metrik mitliefern. Wie aus Tabelle 5 auf Seite 46 zu erkennen ist, verwenden die GBT-Werkzeuge auch keine übereinstimmende Definition.

Die Blocküberdeckung dieser Arbeit ist in der Zielsetzung der Zweigüberdeckung ähnlich und soll wie diese die möglichen Verzweigungen des Programms berücksichtigen und die Anweisungsüberdeckung subsumieren. Generell sind Block- und Zweigüberdeckung sehr ähnlich, weil nahezu alle Zweige in einen Block münden und umgekehrt. Unterschiede bestehen bei Schleifenanweisungen, weil der Zweig aus der Schleife heraus in keinen Block mündet und beim Prozedurkörper, der einen Block bildet, dem aber keine Verzweigung vorausgeht. Gegenüber der Zweigüberdeckung hat die Blocküberdeckung einige praktische Vorteile:

- Es wird für die Blocküberdeckung nicht die Verzweigung in den Block, sondern die Ausführung des Blocks bewertet. Damit ist die Blocküberdeckung einfacher zu bestimmen und als die Zweigüberdeckung.
- Für die Blocküberdeckung ist es klar, dass die catch-Blöcke einbezogen werden, während die meisten Definitonen der Zweigüberdeckung keine Ausnahmebehandlung berücksichtigen.
- Im Unterschied zu Verzweigungen haben Blöcke eine konkrete Repräsentation im Programmcode. Damit ist die Visualisierung der Blocküberdeckung im Programmcode einfacher und verständlicher.

Bei den Überlegungen, ob die Anweisungsblöcke der try-Anweisung zur Blocküberdeckung beitragen sollen, ist das Beispiel von Abbildung 47 hilfreich. Einer exemplarischen Implementierung in der Modellsprache ist die vergleichbare Implementierung in der Pro-

grammiersprache C [KR77] gegenübergestellt. Da die Programmiersprache C keine Ausnahmebehandlung kennt, ist es üblich, den Erfolg oder den Misserfolg der Öffnen-Operation einer Datei über eine Entscheidung abzuprüfen. In der GBT-Modellsprache – als Beispiel einer Programmiersprache, die Ausnahmebehandlung kennt – wird der Misserfolg einer Datei-Öffnen-Operation im catch-Block behandelt. Abbildung 47 zeigt das Beispiel mit und ohne Ausnahmebehandlung nebeneinander. Wir zählen also rechts im C-Programm zwei Anweisungsblöcke (den then- und den else-Block). Im linken Programmcode wirft die Öffnen-Operation im Fehlerfall eine Ausnahme, und der catch-Anweisungsblock wird ausgeführt. Ein getrenntes Abprüfen wie im C-Programm ist damit nicht erforderlich (und nicht möglich). Um für Überdeckungsmetriken bei Sprachen mit und ohne Ausnahmebehandlung vergleichbare Überdeckungswerte zu erhalten, ist es somit sinnvoll, wenn der try-Block sowie der catch-Block zur Blocküberdeckung beitragen. Die Zweigüberdeckung ist von diesen Überlegungen allerdings nicht betroffen: Im C-Beispielprogramm liegt eine Verzweigung vor, im Programm mit Ausnahmebehandlung nicht.

„Datei auslesen“
mit Ausnahmebehandlung

```
try {
  // 1. Datei öffnen
  // ...

  // 2. Datei auslesen
} catch {
  // 3. Fehler behandeln
}
```

„Datei auslesen“
ohne Ausnahmebehandlung

```
// 1. Datei öffnen
File* f = fopen(...);

if(f != null) {
  // 2. Datei auslesen
} else {
  // 3. Fehler behandeln
}
```

Abbildung 47: Programmbeispiel für „Datei auslesen“

Def.: **blocks(P)** *ist die Menge aller Anweisungsblöcke eines Programms P.*

exeBlocksN(P, T) $\subseteq$ *blocks(P) sei die Menge der mit T normal ausgeführten Anweisungsblöcke von P.* $B \in exeBlocksN(P, T) \Leftrightarrow B \in blocks(P) \wedge \exists t \in T : exeN(B, t)$

Die **Blocküberdeckung** *ist der Anteil der ausgeführten Blöcke:*

$$blockCov(P, T) = \frac{| exeBlocksN(P, T) |}{| blocks(P) |}$$

Die Blocküberdeckung subsumiert die Anweisungsüberdeckung, weil Anweisungen nur in Blöcken auftreten können. Die Blocküberdeckung ist aber nicht mit der Anweisungsüberdeckung gleichzusetzen, weil auch leere Blöcke (z. B. leere else-Blöcke) als Block gewertet werden, die keine Anweisung enthalten.

Untersuchung der Metrik:
Da die Definition der Blocküberdeckung als Anteil der ausgeführten Blöcke der Definition der Anweisungsüberdeckung entspricht, gelten die Regeln 1 – 5 in gleicher Weise.

6.3.4 Entscheidungsüberdeckung (decision coverage)

[FAA07] definiert das Kriterium "Entscheidungsüberdeckung" wie folgt: *"Decision coverage: Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken on all possible outcomes at least once"*. Ausdrücklich grenzen die Autoren die Entscheidungsüberdeckung von der Zweigüberdeckung ab und beziehen neben den Verzweigungen des Kontrollflusses auch alle Verzweigungen ein, die durch bedingte Ausdrücke und die Kurzschlusssemantik bei zusammengesetzten Ausdrücken entstehen (vgl. Tabelle 9 von Seite 65): *"For evaluating structural coverage analysis tools for decision coverage, the short-circuit operators and the C conditional expression should be considered as splitting a decision in which they occur into separate decisions, each of which must be covered separately"*.

Wie generell bei Zertifizierungsrichtlinien üblich, definiert [RTCA, FAA07] keine Metrik mit Abstufungen, sondern ein Kriterium, das nur vollständig erfüllt werden kann. Neben den Kontrollflüssen in den genannten Ausdrücken wird auch ausdrücklich die Ausführung der Blöcke zur Ausnahmebehandlung gefordert [FAA07, Kapitel 4.2.4] *„the requirement to invoke every point of entry and exit in the program should also include exception handlers"*. Diese Anforderung ist bereits durch die Zweigüberdeckung berücksichtigt.

Da die Entscheidungsüberdeckung auch wie die Zweigüberdeckung die Verzweigungen des Kontrollflusses betrachtet, bilden die dort definierten Verzweigungen den Ausgangspunkt zur Definition der Entscheidungsüberdeckung. Wie bei der Definition der Zweigüberdeckung werden den Ausdrücken, die im Sinne der Entscheidungsüberdeckung zu Verzweigungen führen, ebenso ein Gewicht $w(K) \rightarrow \mathbb{N}$ zugewiesen, und jeder Ausdruck K erhält für eine Testsuite T einen Ausführungswert $e(K, T) \rightarrow \mathbb{N}$ mit $0 \leq e(K, T) \leq w(K)$, der die Anzahl der für T ausgeführten Verzweigungen angibt. Alle Verzweigungen werden gleich gewichtet; d. h. ein nicht vollständig ausgeführter bedingter Operator wirkt sich auf die Entscheidungsüberdeckung gleich aus wie eine nicht vollständig ausgeführte if-Anweisung.

Verzweigungen des Kontrollflusses
Es gelten die Definitionen der Zweigüberdeckung.

Ausdrücke mit Kurzschlusssemantik
Ausgangspunkt zur Behandlung der Kurzschlusssemantik in der Entscheidungsüberdeckung bildet Tabelle 9 von Seite 65. Daraus ist ersichtlich, dass für einen zusammengesetzten Ausdruck *E1 andThen E2* (sowie *E1 orElse E2*) allein der Teilausdruck E1 die Verzweigung steuert.

$$w(E) = 2$$
$$e(E, T) = e(E1, T)$$

D. h. zur Bestimmung des Ausführungswerts wird nur der erste Teilausdruck E1 betrachtet, der für eine vollständige Ausführung mindestens einmal zu true und einmal zu false ausgewertet werden muss.

Bedingter Ausdruck
Genau wie bei der Entscheidungsanweisung steuert beim bedingten Ausdruck der Bedingungsausdruck allein den Ausführungswert. Für einen bedingten Ausdruck E mit der Bedingung E1 gilt damit:

$$w(E) = 2$$
$$e(E, T) = e(E1, T)$$

Def. Sei decisionItems(P) die Menge aller verzweigenden Ausführungselemente des Programms P. Damit enthält decisionItems(P) alle Entscheidungsanweisungen, alle Fallunterscheidungen, alle Schleifenanweisungen, alle try-Anweisungen, alle zusammengesetzten Ausdrücke mit einem Kurzschlusssemantik-Operator sowie alle bedingten Ausdrücke eines Programms P.

Dann ist die **Entscheidungsüberdeckung** *einer Testsuite T für ein Programm P:*

$$decisionCov(P,T) = \frac{\sum_{i \in \text{decisionItems}(P)} e(i,T)}{\sum_{i \in decisionItems(P)} w(i)}$$

Untersuchung der Metrik
Für die Entscheidungsüberdeckung gelten die Plausibilitätsregeln 1 – 5. Der Nachweis erfolgt wie bei der Zweigüberdeckung.

Damit die Metrik bestimmt werden kann, sind natürlich Programme mit verzweigenden Ausführungselementen Voraussetzung, d. h. es muss für den sinnvollen Einsatz der Entscheidungsüberdeckung gelten $| decisionItems(P) | > 0$. Für reale Programme ist dies natürlich keine relevante Einschränkung.

6.3.5 Schleifenüberdeckung

Die Überlegungen zur Schleifenüberdeckung basieren auf dem Boundary-Interior-Test nach Howden [Ho75] und sind in Abschnitt 2.5.4 auf Seite 27 beschrieben. Vollständige Schleifenüberdeckung ist demnach dann erreicht, wenn der Schleifenkörper bei Ausführung der Schleife jeweils mindestens einmal

- nicht ausgeführt (d. h. „übersprungen“) wird,
- genau einmal ausgeführt wird und
- mehrfach (d. h. wiederholend) ausgeführt wird.

Die zur Bestimmung der Schleifenüberdeckung nötigen Zählerstellen sind im GBT-Modellnetz allerdings nicht vollständig enthalten. Zwar gibt es die Zählerstellen der Schleifenanweisung im Eintritts- und im Austrittsbereich sowie die Zählerstellen des

Schleifenkörpers. Aus diesen Markierungen kann aber nicht auf die Anzahl der Schleifendurchläufe der einzelnen Schleifenausführungen geschlossen werden.

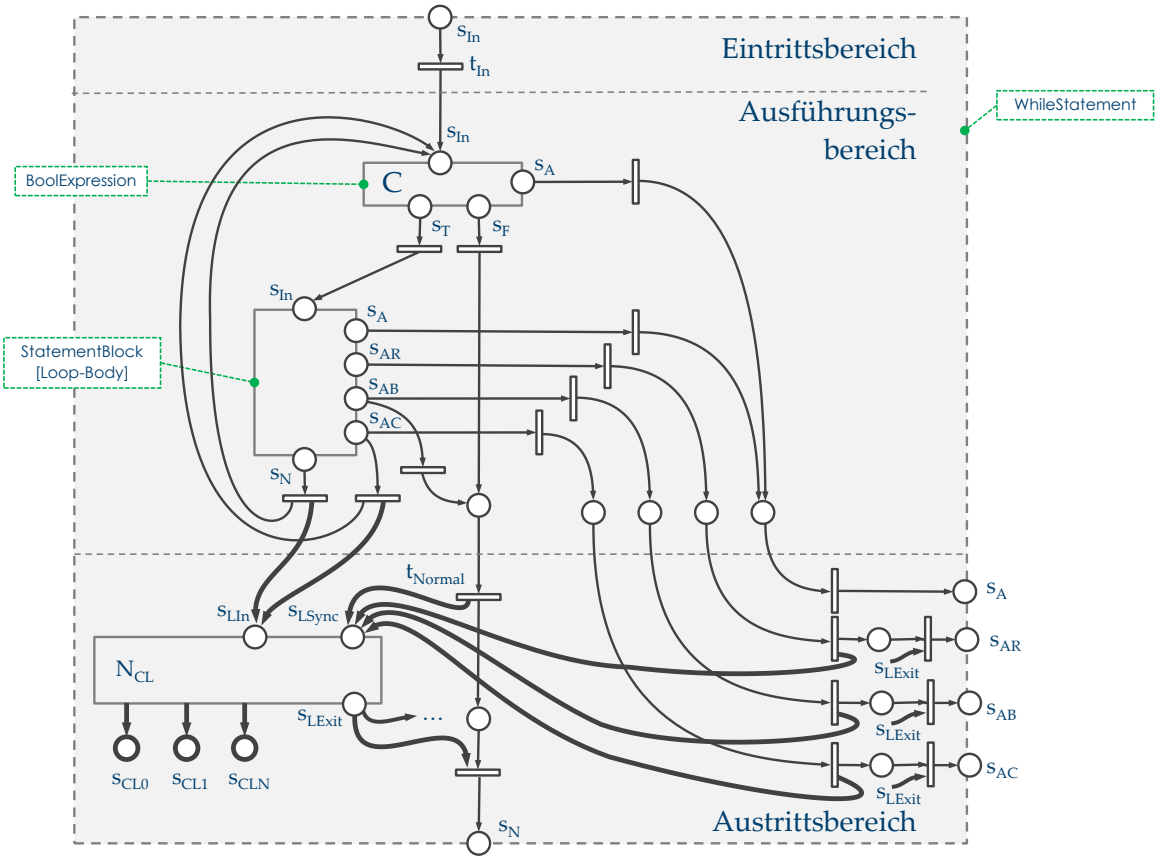


Abbildung 48: Erweitertes Modellnetz der Schleifenanweisung

Aus diesem Grund erhält die Schleifenanweisung zur Ermittlung der Schleifenüberdeckung zusätzlich noch die Zählerstellen s_{CL0}, s_{CL1} und s_{CLN}, deren Vorbereich mit dem Teilnetz N_{CL} verbunden ist. Abbildung 48 zeigt das so erweiterte Gesamtnetz der Schleifenanweisung, wobei die zur Erhebung der Schleifenüberdeckung hinzugekommenen Flüsse hervorgehoben dargestellt sind (vgl. Abbildung 34 auf Seite 102). Die ergänzten Zählerstellen sind so zu verstehen, dass $|M(s_{CL0})|$ die Anzahl für „Übersprungen“ liefert, $|M(s_{CL1})|$ die Anzahl der „Einmalausführung“ und $|M(s_{CLN})|$ die Anzahl der „Mehrfachausführungen“.

Die Randstellen s_{LIn} und s_{LSync} des Teilnetzes N_{CL} werden mit Schalten der Eintrittstransition t_{In} der Schleifenanweisung als leer angenommen. Mit dem Schalten einer der Austrittstransitionen t_{Normal}, $t_{AbruptReturn}$, $t_{AbruptBreak}$ oder $t_{AbruptContinue}$ (d. h. bei normalem Beenden der Schleifenanweisung) wird eine Marke in die Synchronisationsstelle s_{LSync} von N_{CL} gestellt. Die Stelle s_{LIn} enthält dann so viele Marken, wie der Schleifenkörper Ausführungen hatte. Ausgehend von der Markierung dieser Stelle s_{LIn} ist das Netz N_{CL} so angelegt, dass entsprechend dem oben angegebenen Regelwerk in eine der Zählerstellen s_{CL0}, s_{CL1} und s_{CLN} eine Marke gestellt wird. Danach werden die Randstellen s_{LIn}

und s_{LSync} des Teilnetzes N_{CL} wieder geleert, und in die Stelle s_{LExit} wird eine Marke gestellt, sodass die Schleifenanweisung beenden kann.

Für eine Schleife W gibt damit

$cl0(W) = \mid M(s_{CL0}) \mid$	die Anzahl der Ausführungen von *W*, bei denen der Schleifenkörper nicht ausgeführt wird,
$cl1(W) = \mid M(s_{CL1}) \mid$	die Anzahl der Ausführungen von *W*, bei denen der Schleifenkörper genau einmal ausgeführt wird und
$clN(W) = \mid M(s_{CLN}) \mid$	die Anzahl der Ausführungen von *W*, bei denen der Schleifenkörper mehr als einmal ausgeführt wird,

an.

Def.: **loopStmts(P)** $\subseteq$ *stmts(P) ist die Menge aller Schleifenanweisungen von P und bildet eine Teilmenge der Anweisungen von P.*

Schleifen mit nicht ausgeführtem Schleifenkörper (Boundary-Tests):
exeLoopStmts0(P, T) $\subseteq$ *loopStmts(P) ist die Menge von Schleifenanweisungen, deren Schleifenkörper mindestens einmal übersprungen (also nicht ausgeführt) wird:*
$exeLoopStmts0(P, T) = \{ W \mid W \in loopStmts(P) \wedge cl0(W) > 0 \}$

Schleifen mit genau einmal ausgeführtem Schleifenkörper:
exeLoopStmts1(P, T) $\subseteq$ *loopStmts(P) sei die Menge von Schleifenanweisungen, deren Schleifenkörper mindestens einmal einfach (d. h. nicht wiederholend) ausgeführt wird:*
$exeLoopStmts1(P, T) = \{ W \mid W \in loopStmts(P) \wedge cl1(W) > 0 \}$

Schleifen mit mehrfach ausgeführtem Schleifenkörper:
exeLoopStmtsN(P, T) $\subseteq$ *loopStmts(P) sei die Menge von Schleifenanweisungen, deren Schleifenkörper mindestens einmal wiederholt ausgeführt wird.*
$exeLoopStmtsN(P, T) = \{ W \mid W \in loopStmts(P) \wedge clN(W) > 0 \}$

Die **Schleifenüberdeckung** *berechnet sich damit wie folgt:*

$$loopCov(P, T) = \frac{\left| exeLoopStmts0(P, T) \right| + \left| exeLoopStmts1(P, T) \right| + \left| exeLoopStmts2(P, T) \right|}{3 \cdot \mid loopStmts(P) \mid}$$

Die gängigen Programmiersprachen enthalten mit der do-while-Schleife eine annehmende Schleife, die es im GBT-Modell nicht gibt. Um die Schleifenüberdeckung auch bei diesem Schleifentyp nutzen zu können, muss natürlich auf die Forderung der null-fachen Ausführung des Schleifenkörpers verzichtet werden. Annehmende Schleifen sind dann bezogen auf die Schleifenüberdeckung vollständig getestet, wenn die Ausführung des Schleifenkörpers mindestens einmal nicht wiederholt wird und mindestens einmal wie-

derholt wird. Damit ist die Wiederholbedingung der Schleife auch mindestens einmal unmittelbar false und mindestens einmal true. Annehmende Schleifen wirken sich damit geringfügig anders auf die Schleifenüberdeckung aus als die ablehnende Schleife. Annehmende Schleifen haben mit der ersten Ausführung bereits 50 % Überdeckung, während ablehnende Schleifen mit der ersten Ausführung nur 33 % Überdeckung erreichen.

Zählschleifen mit fester Anzahl von Durchläufen können dagegen keinen sinnvollen Beitrag zur Schleifenüberdeckung liefern. So ist z. B. für eine Schleife der Form

```
for(int i = 0; i < 10; i++) { … }
```

eine 0- und einfache Anzahl Schleifendurchläufen natürlich nicht möglich. Zählschleifen werden aus diesem Grund nicht in die Schleifenüberdeckung eingezogen.

Untersuchung der Metrik

Im Folgenden wird untersucht, ob die Regeln, die für Überdeckungsmetriken in Kapitel 6.2 aufgestellt wurden, für die Schleifenüberdeckung gelten. Sei *P* ein GBT-Modellprogramm, das Schleifen enthält, und *T* eine Testsuite zum Test von *P*.

Plausibilitätsregel 1: Null-Überdeckung bei leerer Testsuite	Die Regel gilt. Für $T = \emptyset$ wird keine Schleife ausgeführt. Daraus folgt loopCov(P,T) = 0
Plausibilitätsregel 2: Normierung	Die Regel gilt. Definitionsbedingt gilt *exeLoopStmts0(P, T)* $\subseteq$ loopStmts (P) $\wedge$ *exeLoopStmts1(P, T)* $\subseteq$ loopStmts (P) $\wedge$ *exeLoopStmtsN(P, T)* $\subseteq$ loopStmts (P) Der obere Grenzwert der Schleifenüberdeckung ist damit $\text{loopCov}_{\text{Max}}(P, T) = \frac{1+1+1}{3} = 1$ Daraus folgt $0 \leq \text{loopCov}(P,T) \leq 1$
Plausibilitätsregel 3: Nicht fallende Monotonie	Die Regel gilt. Ein Testfall kann definitionsbedingt immer nur erhöhend auf die Schleifenüberdeckung wirken.
Plausibilitätsregel 4: Erfüllbarkeit	Die Regel gilt.
Plausibilitätsregel 5: Wachsende Monotonie	Die Regel gilt. Eine Testsuite *T* ist nur dann hinsichtlich der Schleifenüberdeckung redundanzfrei, wenn jeder Testfall $t \in T$ „exklusiv“ mindestens einmal zum null-, ein- oder mehrfachen

	Ausführen einer Schleife führt. Diese Eigenschaft der „Exklusiv-Ausführung“ bleibt für alle Teilmengen von T erhalten.

6.4 Bedingungsüberdeckungen

In Abschnitt 2.5.3 auf Seite 24 wurden die verschiedenen Bedingungsüberdeckungen bereits ausführlich vorgestellt. Bei den Bedingungsüberdeckungen werden zusammengesetzte boolesche Ausdrücke betrachtet, und eine vollständige Überdeckung ist dann erreicht, wenn jeder primitive Teilausdruck mindestens einmal das Gesamtresultat des Ausdrucks „bestimmt“ hat. Der Nachweis dieser Einflussnahme ist aber für die im Folgenden vorgestellte Termüberdeckung und die MC/DC unterschiedlich, was sich auch auf die Modellanforderungen auswirkt. In diesem Abschnitt werden die Anpassungen an die Modellnetze beschrieben, die erforderlich sind, um die Überdeckungen erheben zu können

6.4.1 Termüberdeckung

Die Termüberdeckung nach [LL10] liefert durch die sogenannte true- und false-Wirksamkeit eines primitiven booleschen Ausdrucks (im Folgenden als Einzelterm oder nur als Term bezeichnet) den Nachweis dafür, dass ein einzelner Term den Wert des Gesamtausdrucks bestimmt.

Ein Term ist dann true-wirksam, wenn er den Wert true hat und eine Wertänderung dieses Terms auf false das Resultat des Gesamtausdrucks verändert. Entsprechend ist der Term false-wirksam, wenn ein Wechsel auf true das Gesamtresultat verändert. Der entscheidende Unterschied zu MC/DC ist, dass die Wirksamkeit nicht durch ein Paar an Termbelegungen nachgewiesen wird, sondern bereits für eine einzelne Termbelegung bestimmt werden kann. Hierzu wird der Baum betrachtet, der den Gesamtausdruck repräsentiert (vgl. Abschnitt 2.5.3 auf Seite 24).

Die im Folgenden beschriebene Erweiterung der Modellnetze der booleschen Ausdrücke bildet zusammen mit der Metrikdefinition eine formale Spezifikation der Termüberdeckung. Der Nachweis, dass diese Spezifikation der Definition nach [LL10] entspricht, erfolgt durch Erreichbarkeitsanalyse für die Modellnetze des primitiven booleschen Ausdrucks sowie der zusammengesetzten Ausdrücke für „And“ und „Or“, jeweils mit- und ohne Kurzschlusssemantik. Für die möglichen Termbelegungen dieser zusammengesetzten Ausdrücke werden dazu durch Erreichbarkeitsanalyse die Markierungen an den für die Termüberdeckung relevanten Zählerstellen ausgewertet und mit den Vorgaben aus [LL10] verglichen.

Der Nachweis, dass dann auch beliebige zusammengesetzte Ausdrücke die Termüberdeckung nach [LL10] liefern, erfolgt durch Induktion: Da alle Modellnetze der booleschen Ausdrücke am Rand das gleiche Verhalten haben und immer die gleichen Kompositionsregeln genutzt werden, genügt der Nachweis für ein Netz eines zusammengesetzten Ausdrucks.

Ausgangspunkt der Überlegung für die Modellerweiterung bildet der in [LL10] beschriebene zweistufige Ablauf: In einem ersten Durchlauf werden die Werte der Teilausdrücke bestimmt, in einem zweiten erfolgt die abschließende Wirksamkeitsbestimmung. Die im Folgenden vorgestellten Modellnetze erweitern den ersten Durchlauf, indem nicht nur der Wert des Ausdrucks, sondern im übergeordneten Ausdruck auch eine „vorläufige" Wirksamkeit der beiden eingebetteten Teilausdrücke bestimmt wird. Zeigt sich dann für den übergeordneten Ausdruck im zweiten Durchlauf eine „tatsächliche" Wirksamkeit, dann wird die vorläufig festgestellte Wirksamkeit an die Teilausdrücke „weitergereicht". Handelt es sich beim eingebetteten Ausdruck um einen Term, dann speichert dieser die Wirksamkeit in einem Wirksamkeitszähler ab.

Allerdings sind solche Wirksamkeitszählerstellen im Modellnetz des Terms nicht enthalten. Es gibt zwar die Zählerstellen für das true- und false-Resultat, diese sind aber immer unabhängig von der Wirkung des Terms auf das Gesamtresultat (vgl. Abbildung 39 auf Seite 107). Genauso fehlen dem Modellnetz der Verbundausdrücke die Stellen, um die vorläufige Wirksamkeit abspeichern zu können.

Das bislang entwickelte GBT-Modellnetz der booleschen Ausdrücke wird nun in drei Bereichen erweitert:

1. Zum Rand der Netze der booleschen Ausdrücke werden drei weitere Eintrittsstellen und zwei weitere Austrittsstellen hinzugefügt. Die hinzugefügten Eintrittsstellen nehmen bildlich gesprochen den Wirksamkeits-Entscheid des übergeordneten Ausdrucks entgegen. Die hinzugefügten Austrittsstellen liefern das für die Wirksamkeitsbestimmung erforderliche boolesche Resultat eines Ausdrucks. Die Netze bleiben aber stellenberandet.
2. Zum Netz der Ausdrücke kommen Netzerweiterungen hinzu, die die Wirksamkeitsbestimmung modellieren. Diese Erweiterungen sind für die verschiedenen booleschen Ausdrücke unterschiedlich und werden im Folgenden jeweils für den primitiven Ausdruck und die verschiedenen Verbundausdrücke schrittweise beschrieben. Auch die Netze der Ausdrücke mit und ohne Kurzschlusssemantik (wie z. B. AndThen und And) unterscheiden sich z. T. deutlich.
3. Um einen booleschen Gesamtausdruck mit Wirksamkeitsbestimmung in das bestehende GBT-Modellnetz einbetten zu können, muss das Netz des Ausdrucks, der die Wurzel des Ausdruckbaums repräsentiert, über einen Adapter in das Gesamtnetz eingebettet werden.

Die für die Modellnetze aller booleschen Ausdrücke ergänzten Eintritts- und Austrittsstellen sind beispielhaft in Abbildung 49 beim primitiven booleschen Ausdruck (Term) dargestellt. Die sogenannten Wirksamkeitsstellen s_{Tw}, s_{Fw} und s_u liegen am Rand des Netzes und werden genau dann markiert, wenn der Ausdruck vom hierarchisch übergeordneten Ausdruck als true- oder false-wirksam oder unwirksam ermittelt wurde.

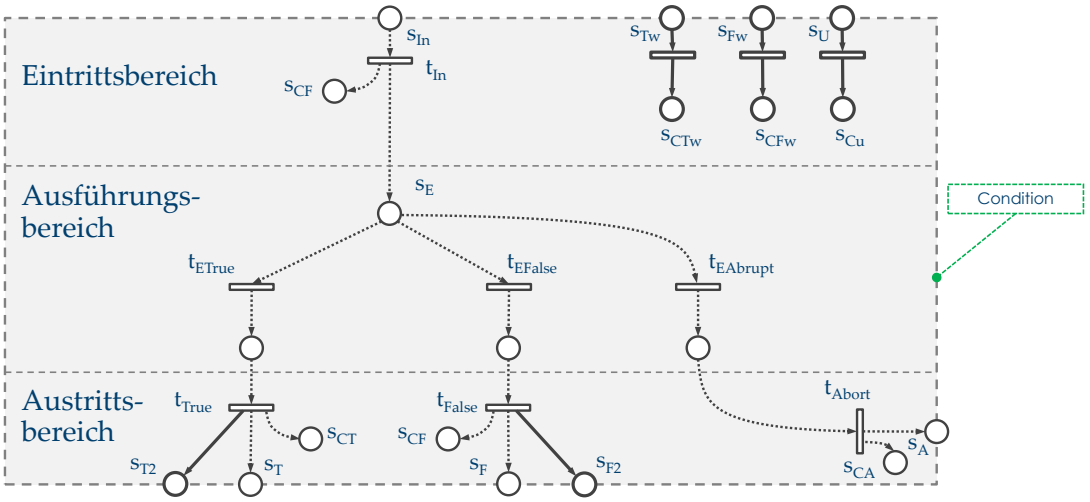


Abbildung 49: Netz-Erweiterungen für primitive boolesche Ausdrücke

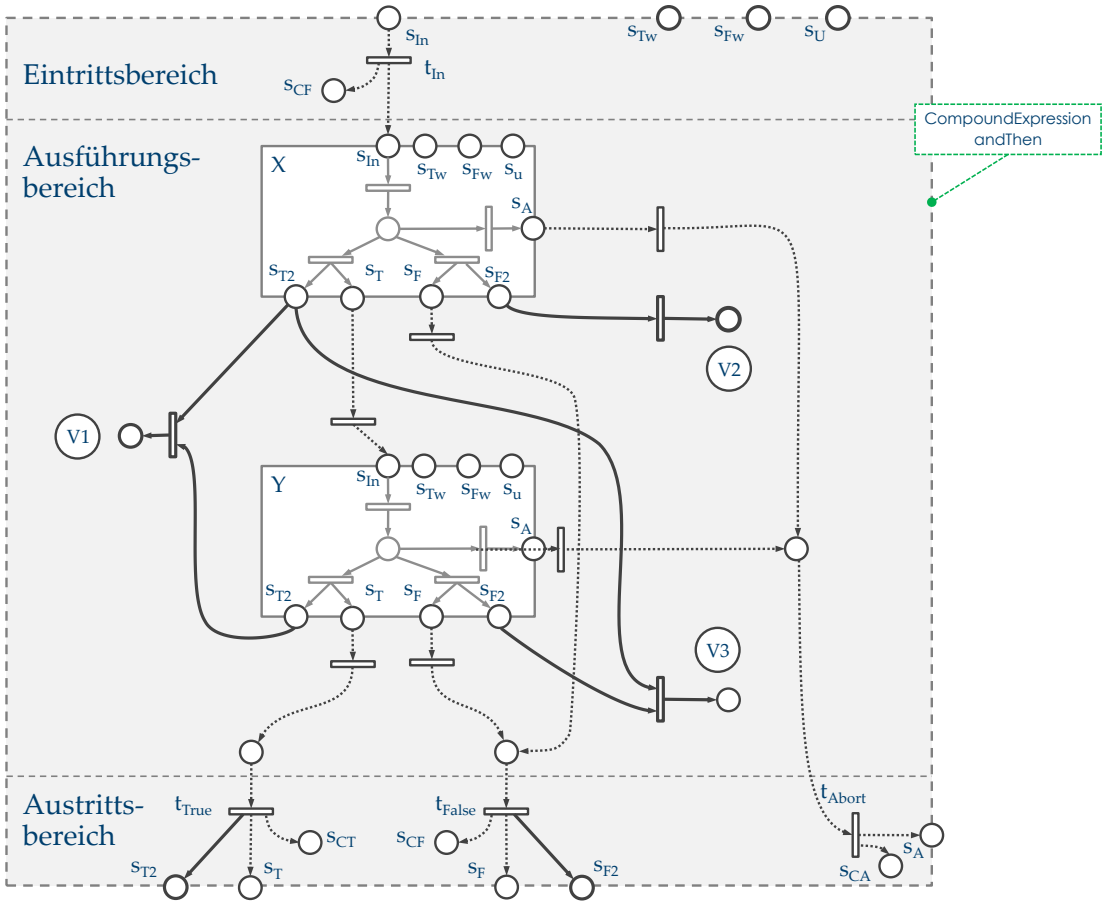


Abbildung 50: Netz-Erweiterungen für Verbundausdrücke (erste Fassung)

Beim primitiven booleschen Ausdruck fließen diese Marken nach Abbildung 49 direkt in eine Zählerstelle. Die Markierung der Stelle s_{CTw} gibt damit an, wie oft der Term true-wirksam war, entsprechend liefert die Markierung der Stelle s_{CFw} die Anzahl der false-Wirksamkeiten. Wie oft ein Term ausgeführt wurde und dabei unwirksam geblieben ist, liefert die Markierung der Stelle s_{Cu}. Diese Angabe spielt allerdings für die Termüberde-

ckung keine Rolle. Die ergänzten Austrittsstellen s_{T2} und s_{F2} erhalten genau wie die Resultatsstellen abhängig vom Resultat des Ausdrucks eine Marke.

Betrachtet man nun die hervorgehobenen Erweiterungen des Modellnetzes eines andThen-Ausdrucks in Abbildung 50, dann sind zum einen die gleichen Randstellen wie beim Term zu nennen, zum anderen die bereits genannten Stellen zur Speicherung einer vorläufigen Wirksamkeit der eingebetteten Ausdrücke.

Für die mit V1, V2 und V3 gekennzeichneten „vorläufig"-Wirksamkeitsstellen lässt sich für eine leere Anfangsmarkierung und einen RPR-Gesamtausdruck *andThen(X, Y)* das Folgende feststellen:

- Die mit V1 gekennzeichnete Stelle ist dann markiert, wenn gilt: $X \wedge Y$. Nach Tabelle 4 auf Seite 26 sind dann beide Teilausdrücke X und Y true-wirksam.
- Die mit V2 gekennzeichnete Stelle ist dann markiert, wenn gilt: $\neg X$. Der Teilausdruck X ist dann false-wirksam.
- Die mit V3 gekennzeichnete Stelle ist dann markiert, wenn gilt: $X \wedge \neg Y$. Der Teilausdruck Y ist dann false-wirksam.

Ausgehend von dieser ersten, noch unvollständigen Fassung des Modellnetzes für den andThen-Ausdruck wird nun das vollständige Modellnetz entwickelt. Dieses vollständige Netz des AndThen-Ausdrucks *AndThen(X, Y)* zeigt Abbildung 51 mit den eingebetteten Teilausdrücken X und Y.

Um übersichtlich auf die Netzerweiterungen zur Wirksamkeitsbestimmung eingehen zu können, sind in Abbildung 51 Hinweisnummern (wie z. B. ①) platziert, damit die einzelnen Netzteile gezielt beschrieben werden können.

Der von den Wirksamkeitsstellen s_{Tw}, s_{Fw} und s_u bei ① ausgehende Fluss mündet nicht wie beim primitiven booleschen Ausdruck in eine Zählerstelle, sondern fließt über den Label-Bezeichner *U* und *W* in die Wirksamkeitsbestimmung der Teilausdrücke *X* und *Y* ein. Eine solche Wirksamkeitsbestimmung liegt beispielsweise bei ③ vor. X und Y haben hier jeweils das Resultat true, wodurch beide zu true-wirksam werden, sofern der Gesamtausdruck wirksam ist. Ein Fluss mit der Angabe $s_{Tw}(X)$ ist dabei so zu verstehen, dass beim Schalten der Transition eine Marke zur Wirksamkeitsstelle s_{Tw} des Teilnetzes von X fließt.

Ist der Gesamtausdruck unwirksam, wird den beiden Teilausdrücken X und Y jeweils in die beiden (Un-)Wirksamkeitsstellen $s_u(X)$ und $s_u(Y)$ eine Marke gestellt. Dieses Prinzip liegt auch bei ④ und bei ⑥ vor, wo jeweils die Wirksamkeitsstellen des Gesamtausdrucks ausgelesen und damit auch geleert werden. Bei ⑥ ist anzumerken, dass hier X das Resultat true und Y das Resultat false hat und damit Y true-wirksam und X unwirksam wird. D. h. die (Un-)Wirksamkeitsstelle $s_u(X)$ erhält auch bei Wirksamkeit des Gesamtausdrucks eine Marke.

Für das Verständnis der Wirksamkeitsstellen s_{Tw}, s_{Fw} oder s_u eines Ausdrucks A ist anzumerken, dass genau eine von diesen Stellen nur dann eine Marke erhält, wenn A normal endet. Endet der Ausdruck abrupt, ist er in jedem Fall unwirksam, und keine der genannten Wirksamkeitsstellen erhält eine Marke. Dieses Prinzip ist bei Hinweis ② zu erkennen: Der Gesamtausdruck wird abrupt enden, und folglich wird auch keine Marke

über U und W bereitgestellt. Bei Hinweis ⑤ wurde Ausdruck X zuvor bereits normal beendet. Folglich „wartet“ dieser nun auf eine Wirksamkeitsmarke, die er auch in die Stelle $s_u(X)$ erhält.

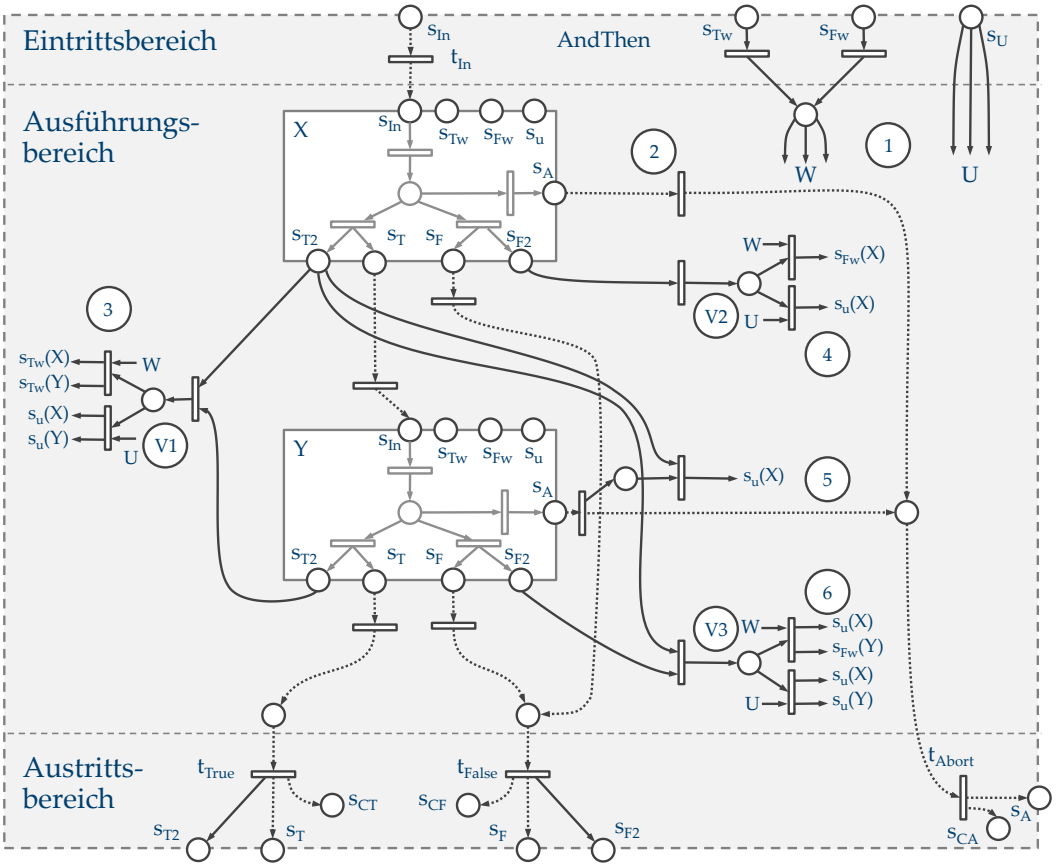


Abbildung 51: Netz-Erweiterungen für einen AndThen-Ausdruck

Das Netz des And-Operators zeigt Abbildung 52. Auch in dieser Darstellung sind die Flüsse, die den logischen Ausdruck modellieren, gestrichelt dargestellt. Eine ausführliche Beschreibung zu diesen Flüssen befindet sich in Kapitel 5.4.3.

Die Netzerweiterungen zur Wirksamkeitsbestimmung unterscheiden sich von denen des AndThen-Ausdrucks an einigen Stellen: Bei Hinweis ⑦ wird X nur dann false-wirksam, wenn Y true ist. Der Fall mit X = false und Y = false wird bei Hinweis ⑧ behandelt. Hier sind X und Y immer unwirksam. Bei Hinweis ⑨ wird das abrupte Beenden von Y behandelt. Für X = true und X = false werden die zur Wirksamkeitsbestimmung bereits bereitgestellten Marken „eingesammelt“, und X erhält eine Marke in die Stelle $s_u(X)$.

Vergleichbare Netze zur Erweiterung des OrElse- und des Or-Ausdrucks, die die Wirksamkeit bestimmen, können nach dem gleichen Prinzip wie beim AndThen- und And-Ausdruck aufgestellt werden. Aus Platzgründen wird an dieser Stelle aber darauf verzichtet

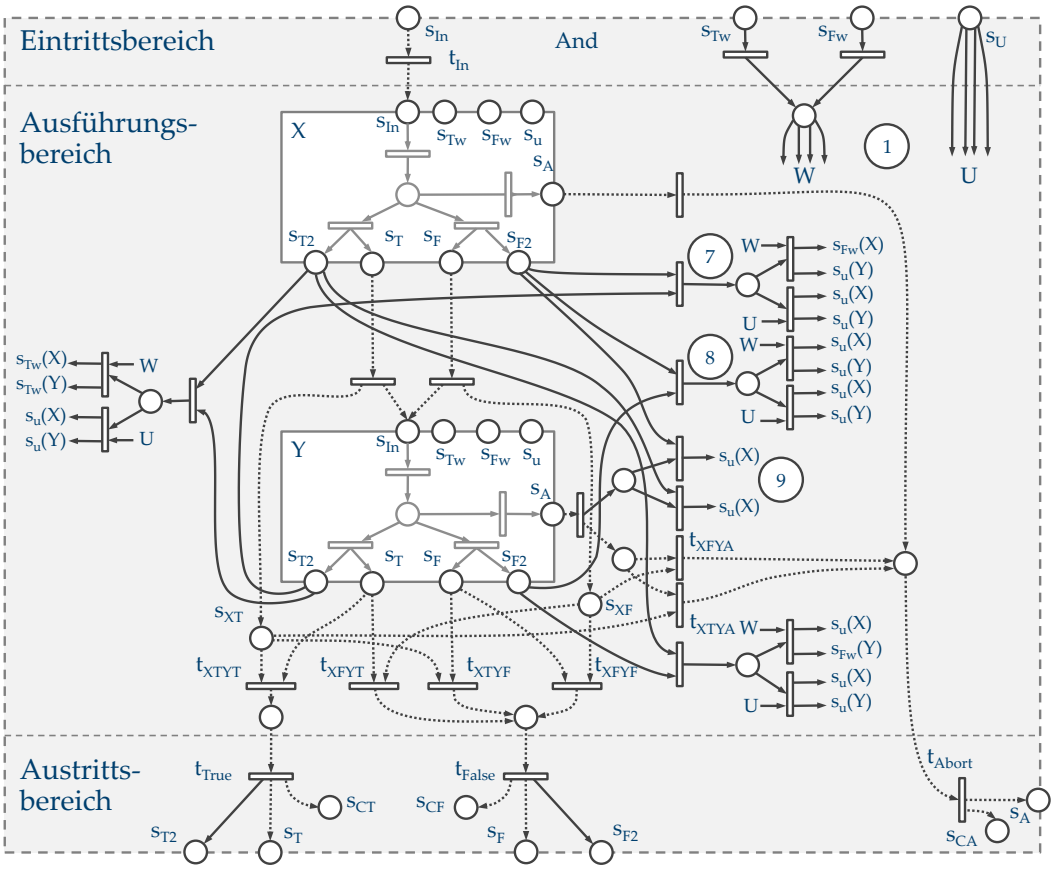


Abbildung 52: Netz-Erweiterungen für einen And-Ausdruck

Evaluation der Netze

Die Evaluation dieser (komplizierten) Netze erfolgt über die Erreichbarkeitsanalyse, die mit dem Werkzeug WoPeD [WoPeD, VA00] durchgeführt wurde. Die Modellnetze der Verbundausdrücke mit eingebetteten primitiven booleschen Ausdrücken wurden dazu im Werkzeug erfasst. Mit dem Werkzeug wurden dann für die relevanten Anfangsmarkierungen die Endmarkierungen ermittelt, und diese wurden (ohne Werkzeugunterstützung) mit den Soll-Werten verglichen. Die Anfangsmarkierungen sind dadurch bestimmt, dass nur die Eintrittsstelle s_{In} sowie eine der Wirksamkeitsstellen s_{Tw}, s_{Fw} oder s_U markiert sind, wobei s_{Tw} und s_{Fw} gemeinsam durch die mit W gekennzeichnete Stelle betrachtet werden können. D.h. es werden zwei Anfangsmarkierungen für jedes Netz betrachtet. Abbildung 53 zeigt beispielhaft das Modellnetz des AndThen-Ausdrucks mit der Anfangsmarkierung, bei der nur die W- und die Eintrittsstelle eine Marke enthalten (d. h. der Gesamtausdruck ist damit wirksam). Die möglichen Endmarkierungen ergeben sich durch die möglichen Schaltvorgänge an den nicht-deterministischen Verzweigungen der eingebetteten primitiven Ausdrücke (d. h. durch deren boolesches Resultat). Das Werkzeug liefert schließlich im Erreichbarkeitsgraphen für jede Endmarkierung den Markierungsvektor. Dieser enthält zum einen die Markierungen der Austrittsstellen (d. h. das boolesche Resultat des Gesamtausdrucks) sowie die Markierung der Wirksamkeitsstellen der eingebetteten Terme. Diese Markierung entspricht der Wirksamkeit im Sinne der Termüberdeckung.

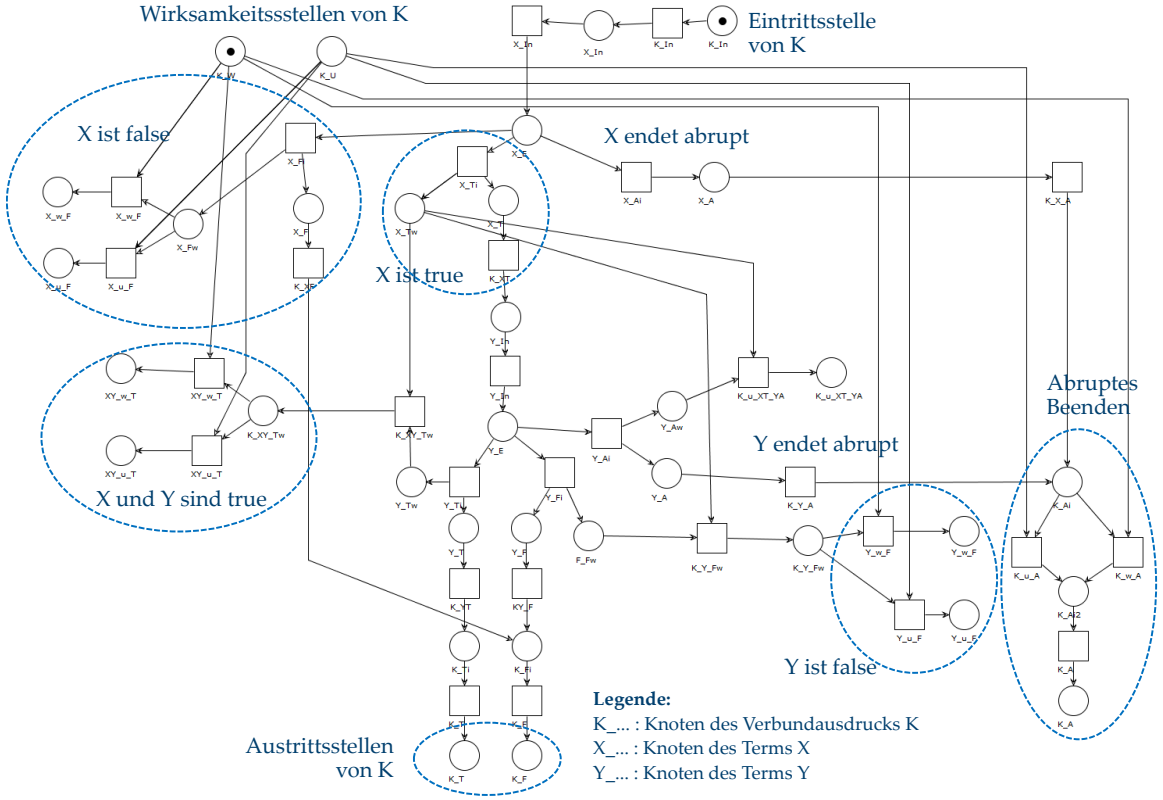


Abbildung 53: Netz des AndThen-Ausdrucks mit dem Werkzeug WoPeD [VA00]

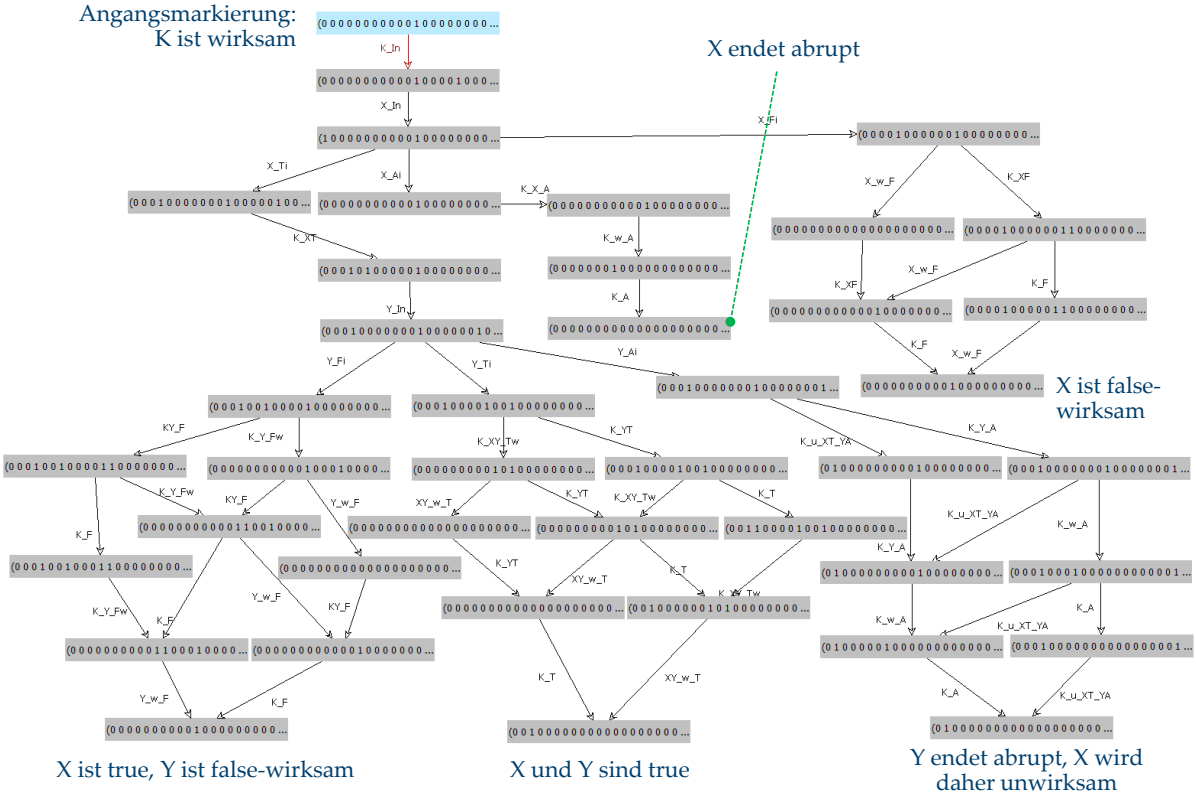


Abbildung 54: Erreichbarkeitsgraph für das Netz des AndThen-Ausdrucks

Abbildung 54 zeigt den Erreichbarkeitsgraph für das Netz mit der Anfangsmarkierung von Abbildung 53. Die vom Werkzeug berechneten Endmarkierungen sind leicht dadurch zu erkennen, dass es keine ausgehenden Kanten gibt, d. h. das Netz tot ist.

RPR-Adaption

Eine geringfügige Veränderung der Kompositionsregeln und damit der Grammatik der Modellsprache ist allerdings noch erforderlich, da das Netz des Ausdrucks, der die Wurzel des Ausdrucksbaum bildet, durch die beschriebenen Erweiterungen einen anderen Rand hat, als es die Einbettung z. B. in einer if-Anweisung vorsieht. Aus diesem Grund wird für diesen Wurzel-Ausdruck eine Produktion *Decision* ergänzt und die Grammatik von Kapitel 4.3 nach Grammatikausschnitt 18 modifiziert.

```
Decision                = BoolExpression.

IfStatement             = "if" "(" Decision ")" ...

WhileStatement          = "while" "(" Decision ")" ...

ConditionalExpression   = Decision "?"
                          SubExpressions ":" SubExpressions.

Expression              = Decision | ConditionalExpression.
```

Grammatikausschnitt 18: Grammatik der GBT-Modellsprache mit Decision

Das Netz für Decision zeigt Abbildung 55. Wesentliches Merkmal ist, dass genau ein Netz des Wurzelausdrucks *BoolExpression* eingebettet ist, der immer dann wirksam wird, wenn er normal endet.

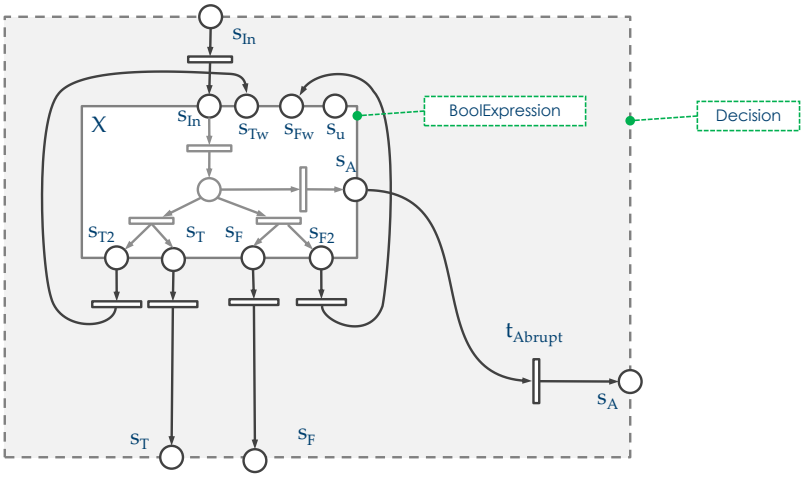


Abbildung 55: Decision als Adapter

Dieser „Rückfluss“ der Marke aus den Stellen s_{Ttc}, s_{Ftc} in die Wirksamkeitsstellen beschreibt das wesentliche Ablaufprinzip des Modellnetzes eines Gesamtausdrucks: Nachdem die Werte der Terme und Teilausdrücke bestimmt sind, erfolgt die Wirksamkeitsbestimmung, indem ein zweites Mal das Modellnetz durchlaufen wird. Wie am Modelnetz für *Decision* leicht erkannt werden kann, endet die *Decision*, also der Repräsentant des Ge-

samtausdrucks bereits, bevor die Wirksamkeitsbestimmung abgeschlossen ist. D. h. nach Feststellen der Teilresultate kann die Wirksamkeitsbestimmung parallel und asynchron mit dem weiteren Ablauf stattfinden. Diese Eigenschaft wird sich auch in der noch folgenden Implementierung widerspiegeln. Die Markentreue des Adapter-Netztes lässt sich anschaulich gut begründen. Zwar liefern die eingebetteten Teilnetze bei normalem Beenden über die ergänzten Austrittsstellen s_{T2} und s_{F2} zunächst eine Marke mehr, als in die Eintrittsstelle s_{In} eingestellt wurde, aber diese Marke wird in die Wirksamkeitsstellen s_{Tw}, s_{Fw} und s_u zurückgestellt. Über den Erreichbarkeitsgraphen lässt sich diese Markentreue auch formal für die komplizierteren Netze nachweisen. Für die Berechnung der Termüberdeckung werden schließlich nur die Wirksamkeits-Zählerstellen s_{CTw} und s_{CFw} der Terme betrachtet. Für einen Term A gibt bei einer leeren Anfangsmarkierung die Markierung der Wirksamkeits-Zählerstellen die Anzahl der true- und false-Wirksamkeit an:

$WT(A) = | M(s_{CTw}(A)) |$ die Anzahl der true-Wirksamkeit von A

$WF(A) = | M(s_{CFw}(A)) |$ die Anzahl der false-Wirksamkeit von A

Auf dieser Grundlage kann nun die Termüberdeckung definiert werden:

Def.: Sei P ein Programm, T eine Testsuite zum Test von P, D eine Bedingung und E ein Term. Dann ist **terms(P)** *die Menge aller Terme des Programms P und* **terms(D)** $\subseteq$ *terms(P) ist die Menge der Terme einer Bedingung D.*

Die true-Wirksamkeit von E wird definiert durch

$$Tw(P, T, E) = \begin{cases} 0, wenn\ WT(E) = 0 \\ 1, wenn\ WT(E) > 0 \end{cases}$$

nach Ausführung von P mit T. Entsprechend wird die false-Wirksamkeit eines Terms E definiert durch

$$Fw(P, T, E) = \begin{cases} 0, wenn\ WF(E) = 0 \\ 1, wenn\ WF(E) > 0 \end{cases}$$

Für eine Bedingung D ergibt sich die **Termüberdeckung** *zu:*

$$\boldsymbol{termCov(D, T)} = \frac{\sum_{E \in \text{terms(D)}} (Tw(P,T,E) + Fw(P,T,E))}{2 \cdot | \text{terms(D)} |}$$

Für das Programm P ergibt sich die Termüberdeckung zu:

$$termCov(P, T) = \frac{\sum_{E \in \text{terms(P)}} (Tw(P,T,E) + Fw(P,T,E))}{2 \cdot | \text{ terms(P) } |}$$

Die Autoren von [AOH03] empfehlen, nur solche Bedingungen in die Bedingungsüberdeckung einzubeziehen, die aus mehr als einem Term bestehen.

Untersuchung der Metrik

Im Folgenden wird untersucht, ob die Regeln, die für Überdeckungsmetriken in Kapitel 6.2 aufgestellt wurden, für die Termüberdeckung gelten. Sei *P* ein GBT-Modellprogramm und *T* eine Testsuite zum Test von *P*.

Plausibilitätsregel 1: Null-Überdeckung bei leerer Testsuite	Die Regel gilt. Für $T = \emptyset$ wird kein boolescher Ausdruck ausgeführt, und für alle Terme E des Programms gilt Tw(P, T, E) = 0 und Fw(P, T, E) = 0. Daraus folgt termCov (P,T) = 0
Plausibilitätsregel 2: Normierung	Die Regel gilt. Definitionsbedingt gilt bei n Termen eines Programms für den oberen Grenzwert der Termüberdeckung: $termCov_{Max}(P, T) = \frac{n \cdot 2}{2 n} = 1$ Daraus folgt $0 \leq termCov(P,T) \leq 1$
Plausibilitätsregel 3: Nicht fallende Monotonie	Die Regel gilt. Ein Testfall kann definitionsbedingt immer nur erhöhend auf die Termüberdeckung wirken.
Plausibilitätsregel 4: Erfüllbarkeit	Die Regel gilt. Für alle realen Programme gilt $0 < \mid terms(P) \mid < \infty$, und man kann induktiv für beliebige zusammengesetzte Ausdrücke zeigen, dass eine vollständige Wirksamkeit erzielt werden kann.
Plausibilitätsregel 5: Wachsende Monotonie	Die Regel gilt. Eine Testsuite *T* ist nur dann hinsichtlich der Termüberdeckung redundanzfrei, wenn jeder Testfall $t \in T$ exklusiv mindestens bei einem Term zu einer Wirksamkeit führt. Diese Eigenschaft bleibt für alle Teilmengen von T erhalten.

6.4.2 Implementierung der Termüberdeckung

Das Modellnetz zur Bestimmung der Termüberdeckung erhebt diese „on-the-fly“, es wird also unmittelbar nach Ausführung des Gesamtausdrucks die Wirksamkeit der Terme festgestellt. Im Modellnetz gibt es hierzu zwei „Durchläufe“: In einem ersten Durchlauf werden die Werte der Teilausdrücke sowie eine „vorläufige“ Wirksamkeit der eingebetteten Teilausdrücke bestimmt. Dieser erste Durchlauf endet, wenn das Gesamtresultat des Ausdrucks feststeht. Im zweiten „Durchlauf“ erfolgt dann die abschließende Wirksamkeitsbestimmung. Der im Folgenden vorgestellte Algorithmus entspricht diesem Ablauf des Modellnetzes in großen Teilen. Allerdings findet aus praktischen Erwägungen heraus nur der erste Durchlauf eingebettet im Originalausdruck statt. Der zweite Durchlauf erfolgt in einer zusätzlich bereitgestellten Baumstruktur, die aber genau der Ausdrucksstruktur entspricht.

Die Implementierung des Algorithmus erfordert eine deutlich aufwändigere Instrumentierung als die, die zur Erhebung des Resultats eines booleschen Ausdrucks erforderlich ist (vgl. Abschnitt 5.13.6 auf Seite 126). Gegenüber dieser Instrumentierung sind zwei wesentliche Erweiterungen erforderlich: Erstens muss die Wertebelegung der Teilausdrücke für eine Auswertung des Gesamtausdruckes festgestellt werden. Und zweitens muss der Algorithmus zur Bestimmung der Wirksamkeiten in den Prüfling integriert werden. Hierzu wird die Baumstruktur des Ausdrucks in das Programm eingefügt und entsprechend der Vorgabe von [LL10] ausgewertet. Zur Abbildung der Baumstruktur des Ausdrucks müssen zudem die hierfür notwendigen Knotentypen bereitgestellt werden.

Der im Folgenden präsentierte Implementierungsvorschlag ist zwar in Java-Syntax beschrieben, lässt sich aber relativ leicht auf andere objektorientierte Programmiersprachen übertragen. Die Beschreibung erfolgt nun in drei Schritten:

1. Definition der Knotentypen zur Abbildung der Baumstruktur des Ausdrucks.
2. Beschreibung, wie die Wertebelegung der Teilausdrücke für eine Auswertung eines (Gesamt-)Ausdrucks festgestellt wird. Hier wird weitgehend die Ausführung des instrumentierten Originalausdrucks genutzt.
3. Beschreibung, wie die ergänzte Baumstruktur der Ausdrücke aufgebaut wird und wie die abschließende Wirksamkeitsbestimmung der Terme erfolgt.

Definition der Knotentypen

Für die Struktur des Baumes, der den Gesamtausdruck repräsentiert, werden die Knotentypen der RPR verwendet. Grammatikausschnitt 19 wiederholt den betreffenden RPR-Teil.

```
BoolExpression     = Identifier
                     ( Condition | CompoundExpression ).

Condition          = "expr" SubExpressions.

CompoundExpression = ("andThen"|"orElse"|"and"|"or" )
                       ( BoolExpression , BoolExpression ).
```

Grammatikausschnitt 19: Grammatikausschnitt für boolesche Ausdrücke

Übertragen in Java-Programmcode bedeutet dies, dass eine abstrakte Klasse *BoolExpression* mit den beiden Kindklassen *Condition* und *CompoundExpression* entsteht. Das so entwickelte UML-Klassendiagramm [RJB04] ist in Abbildung 56 dargestellt. Klasse *BooleanExpression* abstrahiert den primitiven und zusammengesetzten Ausdruck; generell haben beide einen booleschen Wert (das Attribut *value*) und können im Zusammenhang der Kurzschlusssemantik inaktiv werden (das Attribut *active*). Mit der Methode *setValues* werden der Wurzel eines Ausdrucksbaums die Resultate der Teilausdrücke einer Ausführung mitgeteilt.

In der Klasse *Condition* sind die beiden Attribute zur Speicherung der true- und false-Wirksamkeit enthalten. Da die Wirksamkeit nur für die Terme und nicht für die zusammengesetzten Ausdrücke bestimmt wird, sind die Attribute hierzu (*effectiveTrue* und *effectiveFalse*) auch nur in der Klasse *Condition* enthalten. Die Klasse *CompoundExpression* referenziert den (binären) Operator sowie den linken und den rechten Operanden (die Assoziationen *lhs* und *rhs*). Der Aufzählungstyp Operator umfasst die in diesem Algorithmus vorgesehenen booleschen Operationen.

Die zentrale Wirksamkeitsbestimmung findet in der Methode *calcEffectiveness* statt. Da für *CompoundExpression* und *Condition* unterschiedlich vorzugehen ist, erfolgt die Beschreibung auch getrennt. Zur Wirksamkeitsbestimmung bei Verbundausdrücken (*CompoundExpression*) wird der (Teil-)Ausdrucksbaum in Vorordnung durchlaufen.

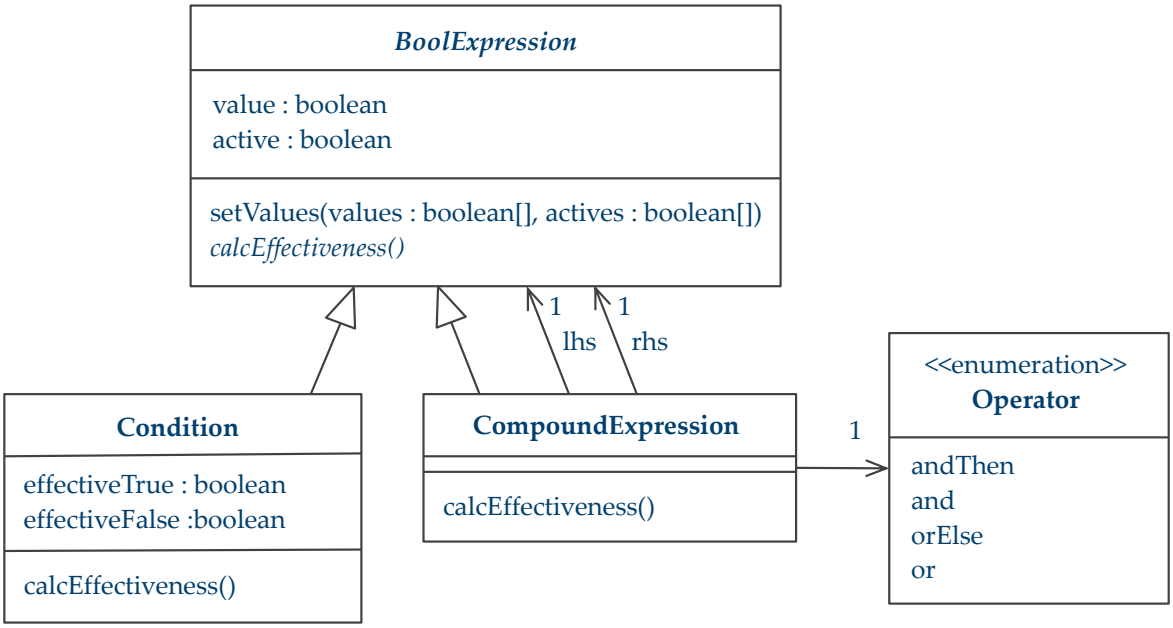


Abbildung 56: UML-Klassendiagramm BoolExpression

Für den linken und den rechten Teilausdruck wird die Wirksamkeit bezogen auf den Operator ermittelt. Diese Auswertung bezieht sich auf Tabelle 4 auf Seite 26. Aus Effizienzgründen wird statt einer Tabellenauswertung für X und Y der ebenso auf Seite 26 angegebene (und trickreiche) Ausdruck verwendet:

Wirksam(X) = (Y == operation);

Wirksam(Y) = (X == operation);

Dabei nimmt die boolesche Variable *operation* für den Fall der *and*-Operation zwischen X und Y den Wert true und für den Fall der *or*-Operation den Wert false an.

Ist ein Teilausdruck ein Verbundausdruck, dann wird abhängig von der Wirksamkeit des linken und des rechten Operanden der Ausdrucksbaum weiter durchlaufen.

```
// Klasse CompoundExpression
@Override
void calcEffectiveness() {
  // Wirksamkeiten der eingebetteten Ausdrücke setzen
  // vgl. Tabelle 4 auf Seite 26

  // linker Ausdruck
  if(this.rhs.value ==
    (this.op == Operation.and ||
              this.op == Operation.andThen)) {
    this.lhs.calcEffectiveness();
  }

  // rechter Ausdruck
  if(this.rhs.active && this.lhs.value ==
    (this.op == Operation.and ||
      this.op == Operation.andThen)) {
    this.rhs.calcEffectiveness();
    }
  }
}
```

Ist ein Teilausdruck ein Term, dann kann die Wirksamkeit unmittelbar bestimmt werden (vgl. Modellnetz in Abbildung 49 auf Seite 150). Terme mit dem Wert true sind true-wirksam und Terme mit dem Wert false sind false-wirksam.

```
// Klasse Condition
@Override
void calcEffectiveness() {
  // Wirksamkeit des primitiven Ausdrucks setzen
  if(this.value) {
    effectiveTrue = true;
  } else {
    effectiveFalse = true;
  }
}
```

Erhebung der Wertebelegung der Teilausdrücke

Die Feststellung der Werte der Teilausdrücke für eine Auswertung eines Gesamtausdrucks erfolgt nach einem ähnlichen Muster wie das Zählen der true- und false- Resultate, das in Abschnitt 5.13.5 auf Seite 124 beschrieben wird. Allerdings wird dort das Resultat der Ausdrücke nur einzeln und zusammenhanglos festgestellt, es wird keine Wertebelegung der Teilterme für eine Auswertung eines Gesamtausdrucks erhoben. Um diese Wer-

tebelegung der Teilterme festzustellen, wird der Gesamtausdruck als (binärer) Baum betrachtet. Beim Durchlaufen dieses Baumes in Nachordnung wird der i. Teilausdruck T_i zu

```
(Ti & (actives[i] = true) && (results[i] = true))
```

instrumentiert. Hierbei ist *actives* und *results* je ein Array von booleschen Werten, das vor der Ausdrucksauswertung mit false initialisiert wird. Das mit dem &-Operator verbundene (*actives[i] = true*) wird immer dann ausgeführt, wenn T_i ausgeführt wird und dabei normal endet. Durch die Vorbelegung mit false werden so alle ausgeführten Teilbäume festgestellt (und damit natürlich auch alle nicht ausgeführten). Bedingt durch die Kurzschlusssemantik des &&-Operators wird der Ausdrucksteil *(results[i] = true)* nur dann ausgeführt, wenn der Term T_i den Wert true hat. Diese Instrumentierung wird für alle Teilbäume des Gesamtausdrucks mit Ausnahme der Wurzel vorgenommen. Die Wurzel muss deswegen hier nicht berücksichtigt werden, weil das Resultat des Wurzelausdrucks für die Wirksamkeitsbestimmung nicht relevant ist.

Für den Java-Originalausdruck

```
if(A && B) { ... }
```

mit der RPR-Abstraktion des Bedingungsausdrucks

```
K1: andThen(A: expr [], B: expr [])
```

ergibt sich damit eine Instrumentierung wie folgt:

```
boolean[] results = new boolean[2];
boolean[] actives = new boolean[2];

boolean expressionResult =
   (A & (actives[0] = true) && (results[0] = true))
   &&
   (B & (actives[1] = true) && (results[1] = true));

// ... jetzt die Wirksamkeit bestimmen

if(expressionResult) {
   // ...
}
```

Der Wert des Gesamtausdrucks ist deswegen in der Variablen *expressionResult* abgespeichert, um die Ausdrucksauswertung von der Entscheidungsanweisung zu entkoppeln.

Baumstruktur des Ausdrucks

Im nächsten Schritt wird aus den bereits beschriebenen Knotentypen der Baum für den betrachteten Ausdruck aufgebaut. Die Beschreibung erfolgt wieder am Beispielausdruck

K1: andThen(A: expr [], B: expr []). Im folgenden Programmcode repräsentiert *decision* die Wurzel des Ausdrucksbaums *K1*.

```
BooleanExpression decision =
  new CompoundExpression("K1",
    new Condition("A"), new Condition("B"),
    Operation.andThen);
```

Sobald der Baum aufgebaut ist, werden der Wurzel (*decision)* die Werte der Teilausdrücke übergeben, und mit *calcEffectiveness*() wird abschließend die Wirksamkeit der Terme in der bereits beschriebenen Weise bestimmt.

```
decision.setValues(results, actives);
decision.calcEffectiveness();
```

Da sich die Struktur eines Ausdrucksbaumes im Programmverlauf zweifellos nicht verändert, ist es aus Effizienzgründen sinnvoll, die Wurzel des Baumes an zentraler Stelle im Programm abzulegen. Es kann dann bei Folgeaufrufen auf den bereits fertigen Baum zu gegriffen werden. Als eindeutigen Bezeichner der Wurzel kann der RPR-Bezeichner genutzt werden.

Für Programmiersprachen, bei denen die in der Implementierung genutzte Vererbung und Polymorphie nicht zur Verfügung stehen (wie z. B. bei der Programmiersprache C) kann die Implementierung nicht so elegant wie beim präsentierten objektorientierten Entwurf erfolgen. Gleichwohl lassen sich dort auch entsprechende Datenstrukturen der Knotentypen definieren und die polymorphen Methodenaufrufe durch Fallunterscheidungen der Knotentypen mit dann statisch gebundenen Funktionsaufrufen ersetzen.

6.4.3 MC/DC

Die Bedingungsüberdeckung MC/DC wurde bereits in Kapitel 2.5.3 auf Seite 24 ausführlich vorgestellt. Aber auch wenn die dort angegebene Definition relativ einfach und knapp erscheint, ist eine Abbildung in das Ablaufmodell sehr aufwändig. Das größte Problem dabei ist, dass eine Wertebelegung nicht wie bei der Termüberdeckung unmittelbar („on-the-fly“) in Inkremente der Wirksamkeitszähler übertragen werden kann, sondern zunächst unbewertet abgespeichert werden muss. Erst der paarweise Vergleich mit einer weiteren Wertebelegung zeigt die Wirksamkeit für den Hauptterm an. Und da MC/DC erst dann erfüllt ist, wenn die Wirksamkeit für alle Terme nachgewiesen ist, müssen alle aufgetretenen Wertebelegungen solange abgespeichert bleiben.

Zur Abbildung in das GBT-Modell dieser Arbeit müsste im Netz des Gesamtausdrucks für jede theoretisch mögliche Wertekombination der enthaltenen Terme jeweils eine Zählerstelle vorgesehen werden. In diese Zählerstelle würde dann eine Marke gestellt, wenn die entsprechende Wertbelegung der Terme tatsächlich auftritt. Damit ergeben sich bei n eingebetteten Termen eines Ausdrucks 2^n Zählerstellen mit entsprechend aufwändiger Flussrelation. Auf eine Darstellung als Petri-Netz wird daher verzichtet.

Es wird dagegen das Ausführungselement, das den Gesamtausdruck repräsentiert, mit sogenannten Termkombinationsflags attributiert. Da ein Gesamtausdruck D mit *n* eingebetteten Termen 2^n mögliche Wertekombinationen zulässt, wird der Gesamtausdruck entsprechend mit 2^n Termkombinationsflags (*fTC(D, 0) … fTC(D, 2^n - 1)*) für alle theoretisch möglichen Wertekombinationen attributiert. Diese Flags werden beim Programmstart (d. h. mit der Anfangsmarkierung des Programms) mit *false* initialisiert. Tritt eine Wertekombination *k* eines Gesamtausdrucks *D* in der Testausführung auf, wird das Flag *fTC(D, k) = true* gesetzt.

Für Gesamtausdrücke mit vielen Einzeltermen kann es aus Effizienzgründen natürlich sinnvoller sein, das Flag für eine Wertekombination erst dann anzulegen, wenn sie tatsächlich im Test aufgetreten ist. Gerade bei Ausdrücken mit vielen Einzeltermen ist anzunehmen, dass bei Weitem nicht alle Wertekombinationen beim Test tatsächlich auftreten werden.

Def.: Sei D ein Bedingungsausdruck mit der Menge C von Einzeltermen und einer Wertebelegung k für C. Wert(k, t) sei der boolesche Wert des Terms $t \in C$ und Wert(k, D) der boolesche Wert des Gesamtresultats von D für k.
Ein Term $t_j \in C$ mit zwei Wertebelegungen k_1 und k_2 heißt **bestimmend** *für D, wenn $Wert(k_1, t_j) \neq Wert(k_2, t_j) \wedge \forall t_i \in C, 1 \leq i \leq \mid C \mid \wedge i \neq j : Wert(k_1, t_i) = Wert(k_2, t_i) \wedge Wert(D, k_1) \neq Wert(D, k_2)$. Eine Testsuite* **T erfüllt für ein Programm P das Kriterium MC/DC**, *wenn alle Terme von P für ihren Bedingungsausdruck mindestens einmal bestimmend sind.*

MC/DC liefert in [RTCA, FAA01, FAA07] nur eine Definition für vollständiges Erfüllen, Abstufungen mit Erfüllungsgraden gibt es dort nicht. Dieser Erfüllungsgrad kann aber zudem über den Anteil der bestimmenden Terme definiert werden:

Def.: **terms(P)** *sei die Menge aller Einzelterme der Bedingungen von P.*

Bestimmende Terme: *$determineTerms(P, T) \subseteq terms(P)$ sei die Menge der Einzelterme, die für ihren Bedingungsausdruck mindestens einmal bestimmend sind.*

Für das Programm P ergibt sich die **MC/DC-Überdeckung** *zu:*

$$mcdcCov(P, T) = \frac{\mid determineTerms(P,T) \mid}{\mid terms(P) \mid}$$

Untersuchung der Metrik

Im Folgenden wird untersucht, ob die Regeln, die für Überdeckungsmetriken in Kapitel 6.2 aufgestellt wurden, für das MC/DC-Überdeckungskriterium gelten. Sei *P* ein GBT-Modellprogramm und *T* eine Testsuite zum Test von *P*.

Plausibilitätsregel 1: Normierung	Die Regel gilt. Definitionsbedingt gilt *determineTerms* (P, T) $\subseteq$ *terms (P)* Daraus folgt $0 \leq$ mcdcCov(P,T) ≤ 1
Plausibilitätsregel 2: Null-Überdeckung bei leerer Testsuite	Die Regel gilt. Für $T = \emptyset$ wird kein boolescher Ausdruck ausgeführt und folglich kann kein Term bestimmend sein. Daraus folgt mcdcCov (P,T) = 0
Plausibilitätsregel 3: Nicht fallende Monotonie	Die Regel gilt. Ein Testfall kann definitionsbedingt immer nur erhöhend auf die MC/DC-Überdeckung wirken.
Plausibilitätsregel 4: Erfüllbarkeit	Die Regel gilt. Für alle realen Programme gilt $0 < \mid terms(P) \mid < \infty$, und man kann induktiv für beliebige zusammengesetzte Ausdrücke mit voneinander unabhängigen Termen zeigen, dass eine vollständige Wirksamkeit erzielt werden kann.
Plausibilitätsregel 5: Monoton wachsend	Die Regel gilt nicht. Damit ein Term bestimmend für einen Gesamtausdruck D ist, sind zwei Wertebelegungen k_1 und k_2 erforderlich. Sei eine Testsuite $T_A = \{ t_1 , t_2 \}$ und t_1 führt zu einer Wertebelegung k_1 und t_2 zu einer Wertebelegung k_2. Zu anderen Wertebelegungen führen t_1 und t_2 nicht. Dann ist T_A redundanzfrei. Eine Testsuite $T_B = \{ t_1 \}$ (d. h. T_B ist eine Teilmenge von T_A) ist nicht redundanzfrei, weil t_1 „alleine“ zu keiner MC/DC-Überdeckung führt und so aus T_B auch entnommen werden kann, ohne dass sich die MC/DC-Überdeckung von T_B verändert.

Die Implementierung von MC/DC unterscheidet sich von der der Termüberdeckung darin, dass für MC/DC im Prüfling lediglich die Erhebung der Wertebelegung der Terme erforderlich ist (Teil 2 der Implementierung der Termüberdeckung auf Seite 160). Die so festgestellten Termbelegungen werden abgespeichert. Eine „on-the-fly“-Auswertung wie bei der Termüberdeckung ist nicht möglich, weil immer ein paarweiser Vergleich der Belegungen stattfinden muss. Dieser findet dann nach Ende des Tests bei der Auswertung des GBT-Protokolls statt.

6.5 Neue Überdeckungsmetriken

6.5.1 Catch-Überdeckung

In einigen empirischen Untersuchungen zum GBT (z. B. [EBI06, BWK07, MND09]) wird darüber berichtet, dass besonders Programmcode zur Fehlerbehandlung eine anfangs (d. h. vor Einführung des GBT) geringe Überdeckung hat und durch den GBT bedingt neue Testfälle ergänzt werden, die zu einer deutlichen Erhöhung dieser Fehlerbehandlungs-Überdeckung führen. Um beim Test gezielt eine Aussage über die Vollständigkeit der geworfenen – und behandelten – Ausnahmen zu machen, ist damit eine catch-Überdeckung nützlich. Die catch-Überdeckung gibt den Anteil der ausgeführten catch-Blöcke an.

Def.: Seien P ein Programm und T die Testsuite für P. **catchBlocks(P)** *ist die Menge aller catch-Blöcke der try-Anweisungen von P.* **exeCatchBlocks(P, T)** $\subseteq$ *catchBlocks (P) sei die Menge der mit T ausgeführten catch-Blöcke der try-Anweisungen von P.*

$C \in exeCatchBlocks\ (P, T) \Leftrightarrow C \in catchBlocks\ (P) \wedge \exists t \in T : exe(C, t)$

Die catch-Überdeckung ist der Anteil der ausgeführten catch-Blöcke:

$$catchCov(P, T) = \frac{| exeCatchBlocks(P, T) |}{| catchBlocks(P) |}$$

6.5.2 Leere-else-Überdeckung

Leere else-Blöcke entstehen in den realen Programmiersprachen in aller Regel dadurch, dass der else-Block zur if-Anweisung weggelassen wird. Da die Ausführung eines leeren else-Blocks natürlich nicht zur Anweisungsüberdeckung beiträgt, werden Testfälle, die zur Ausführung des leeren else-Blocks führen, von Testern leicht übersehen. Die leere-else-Überdeckung gibt gezielt den Anteil der ausgeführten leeren else-Blöcke an.

Def.: Seien P ein Programm und T die Testsuite für P. **emptyElseBlocks(P)** $\subseteq$ *blocks(P) sei die Menge aller else-Blöcke der if-Anweisungen, die keine Anweisung enthalten.*
exeEmptyElseBlocks(P, T) $\subseteq$ *emptyElseBlocks(P) sei die Menge der mit T ausgeführten leeren else-Blöcke von P.*
$B \in exeEmptyElseBlocks\ (P, T) \Leftrightarrow B \in emptyElseBlocks(P) \wedge \exists t \in T : exe(B, t)$
Die **Leere-else-Überdeckung** *ist dann der Anteil der ausgeführten leeren else-Blöcke:*

$$blockCov(P, T) = \frac{| exeEmptyElseBlocks\ (P, T) |}{| emptyElseBlocks(P) |}$$

6.5.3 Bedingter-Ausdrucks-Überdeckung

Nach den Metrikdefinitionen dieser Arbeit wirkt sich die Überdeckung des bedingten Ausdrucks in der Entscheidungsüberdeckung aus. Eine von den anderen Verzweigungen

isolierte Betrachtung ist aber durchaus nützlich, weil in der Regel weit weniger bedingte Ausdrücke als if-Anweisungen verwendet werden und so die Überdeckung des bedingten Ausdrucks nur stark „verdünnt“ sichtbar wird. Die nun folgende Überdeckungsmetrik basiert ausschließlich auf der Überdeckung des bedingten Ausdrucks.

Def.: Seien P ein Programm und T die Testsuite für P. **condExpr(P)** *sei die Menge aller bedingter Ausdrücke des Programms P.*
exeCondExpr1(P, T) $\subseteq$ *condExpr(P) sei die Menge von bedingten Ausdrücken, bei denen für T der erste Alternativausdruck wirksam wurde.*
exeCondExpr2(P, T) $\subseteq$ *condExpr (P) sei die Menge von bedingten Ausdrücken, bei denen für T der zweite Alternativausdruck wirksam wurde.*
Sei C der Bedingungsausdruck eines bedingten Ausdrucks E, dann gilt
$E \in exeCondExpr1(P, T) \Leftrightarrow E \in condExpr(P) \wedge \exists t \in T : exeT(C, t)$ *und*
$E \in exeCondExpr2(P, T) \Leftrightarrow E \in condExpr(P) \wedge \exists t \in T : exeF(C, t)$

Die **Bedingter-Ausdrucks-Überdeckung** *ist dann wie folgt:*

$$condExprCov(P, T) = \frac{| exeCondExpr1(P, T) | + | exeCondExpr2(P, T) |}{| 2 \cdot condExpr(P) |}$$

6.6 Bewertung des Referenzprogramms

Das folgende Java-Programm liegt den Auswertungen von Tabelle 5 auf Seite 46 zugrunde und zeigt die nach Übertragung in das GBT-Modell ermittelte Anweisungs-, Zweig- und Blocküberdeckung. Die Testausführung wird immer mit a = 42 und b = 7 vorgenommen. Wie man leicht erkennen kann, dient das Referenzprogramm nur dazu, möglichst viele Sonderfälle der GBT-Metrikenerhebung abzudecken; eine sinnvolle Funktion erfüllt das Programm nicht. Die Angaben x / y in den drei rechten Spalten sind so zu verstehen, dass y die Anzahl der Anweisungen, Zweige oder Blöcke angibt und x die dafür erzielte Überdeckung.

	Anw.	Zweig	Block
`public static void main(String[] args) {`			1 / 1
`  // Fall 0: Standardanweisung, Zuweisung`			
`  int a = Integer.valueOf(args[0]), b = …;`	1 / 1		
`  // Fall 1: then wird ausgeführt, kein else`			
`  if(a == 42) {`	1 / 1		
`    System.out.println("1");`	1 / 1	1 / 2	1 / 1
`  }`			0 / 1
`  // Fall 2: nur then wird ausgeführt, befüllter else`			
`  if(a == 42) {`	1 / 1		
`    System.out.println("2 - then");`	1 / 1	1 / 2	1 / 1
`  } else {`			
`    System.out.println("2 - else");`	0 / 1		0 / 1
`  }`			
`  // Fall 3: leerer else wird ausgeführt`			
`  if(b == 42) {`	1 / 1		

`  System.out.println("3");`	0 / 1	1 / 2	1 / 1
`}`			0 / 1
`// Fall 4: befüllter else wird ausgeführt`			
`if(b != 7) {`	1 / 1		
`  System.out.println("4 - then");`	0 / 1	1 / 2	0 / 1
`} else {`			
`  System.out.println("4 - else");`	1 / 1		1 / 1
`}`			
`// Fall 5: einer der case-Blöcke wird ausgeführt`			
`switch(a) {`	1 / 1		
`  case 42: System.out.println("5 - 42"); break;`	2 / 2	1 / 4	1 / 1
`  case 9: System.out.println("5 - 9"); break;`	0 / 2		0 / 1
`  case 1: System.out.println("5- 1"); break;`	0 / 2		0 / 1
`  default: System.out.println("5 -default");`	0 / 1		0 / 1
`}`			
`// Fall 6: der default-Block wird ausgeführt`			
`switch(a) {`	1 / 1		
`  case 99: System.out.println("6 - 99"); break;`	0 / 2	1 / 2	1 / 1
`  default: System.out.println("6 -default");`	1 / 1		0 / 1
`}`			
`// Fall 7: leerer default wird ausgeführt`			
`switch(a) {`	1 / 1		
`  case 99: System.out.println("7 - 99"); break;`	0 / 2	1 / 2	0 / 1
`}`			1 / 1
`// Fall 8: Schleifenkörper wird mehrfach ausgeführt`			
`int x = b; // a = 42, b = 7`	1 / 1		
`while(x < a) {`	1 / 1		
`  x++;`	1 / 1	2 / 2	1 / 1
`  System.out.println("8 - x  = " + x);`	1 / 1		
`}`			
`// Fall 9: Schleifenkörper wird nicht ausgeführt`			
`x = b; // a = 42, b = 7`	1 / 1		
`while(x > a) {`	1 / 1		
`  x++;`	0 / 1	1 / 2	0 / 1
`  System.out.println("9 - x = " + x);`	0 / 1		
`}`			
`// Fall 10: Division durch null - Exception`			
`try {`	1 / 1		
`  b = b / (b - 7);`	1 / 1		1 / 1
`  System.out.println("10 - b = " + b);`	0 / 1		
`} catch(Exception e) {`			
`  System.out.println("10 - " + e.getMessage());`	1 / 1	1 / 1	1 / 1
`}`			
`finally {`			
`  System.out.println("10 - finally");`	1 / 1		
`}`			
`// Fall 11: keine Exception`			
`try {`	1 / 1		
`  b = b / (b - 2);`	1 / 1		1 / 1
`  System.out.println("11 - b = " + b);`	1 / 1		
`} catch(Exception e) {`			
`  System.out.println("11 - " + e.getMessage());`	0 / 1	0 / 1	0 / 1
`}`			
`}`			
	27 / 43 (62,8 %)	11 / 22 (50 %)	12 / 23 (52 %)

7
Ein Werkzeug für den Glass-Box-Test

In diesem Kapitel werden die Anforderungen an ein GBT-Werkzeug beschrieben, die sich durch die Modell- und Metrikendefinition aus Kapitel 4, 5 und 6 ergeben. Anschließend wird das GBT-Werkzeug CodeCover vorgestellt, das viele dieser Anforderungen implementiert. Es folgt ein Abgleich der CodeCover-Funktionen mit den aufgestellten Anforderungen. Daran schließt sich die Beschreibung an, wie CodeCover das testfallgenaue Ausführungsprotokoll erzeugt. Mit einer Evaluation des Werkzeuges endet das Kapitel.

7.1 Anforderungen an ein Werkzeug für den Glass-Box-Test

Aus den Kapiteln 4, 5 und 6 zur Modell- und Metrikendefinition ergeben sich drei zentrale Anforderung an ein GBT-Werkzeug:

1. Programmcode-Abstraktion
   Das Werkzeug bildet entsprechend den Vorgaben aus Kapitel 4 und 5 aus dem Originalprogramm eine Abstraktion, die die Ausführungselemente Anweisung, Anweisungsblock und die GBT-relevanten Ausdrücke entsprechend der RPR-Definition enthält.
2. Testfallgenaues Ausführungsprotokoll
   Das Werkzeug liefert für ein Ausführungselement A (eine Anweisung oder ein Anweisungsblock) bei einer Programmausführung und mit einem Testfall t die Ausführung:

   $exe(A, t) \Leftrightarrow A$ wird von t mindestens einmal ausgeführt

   Für einen booleschen Ausdruck C liefert das Werkzeug die Ausführungen mit den Resultaten true und false:

   $exeT(C, t) \Leftrightarrow C$ wird bei der Ausführung mit t mindestens einmal true
   $exeF(C, t) \Leftrightarrow C$ wird bei der Ausführung mit t mindestens einmal false

3. Metrikenbestimmung
   Das GBT-Werkzeug bestimmt die Überdeckungsmetriken entsprechend den Definitionen aus Kapitel 6. D. h. es erfolgt die Metrikenberechnung auf Grundlage der Ausführungselemente und deren Ausführungen für die betrachtete Testsuite.

Nur für Werkzeuge zum Test sicherheitskritischer Software (vgl. [IEC61506, ISO26262, RTCA]):

4. Bestimmung der MC/DC oder der Termüberdeckung
   Mit den unter 2. genannten Auswertungen ist es nicht möglich, die Termüberdeckung oder MC/DC zu bestimmen (vgl. Abschnitt 6.4). Falls das Werkzeug zum Test von sicherheitskritischer Software eingesetzt werden soll, dann kommt als Anforderung noch hinzu, dass MC/DC oder die Termüberdeckung nach Abschnitt 6.4 bestimmt werden können.

Zweifellos lassen sich noch weitere Anforderungen nennen, die den praktischen Einsatz des Werkzeugs betreffen. So ist es nach [YLW09] für ein GBT-Werkzeug natürlich vorteilhaft, wenn es sich benutzerfreundlich in die Entwicklungs- und Test-Umgebung integrieren lässt. Zudem soll das Werkzeug eine erzielte Überdeckung möglichst direkt im Programmcode der Entwicklungsumgebung visualisieren und zusätzlich zur Archivierung in einer separaten Datei ablegen. Diese weiteren Anforderungen werden hier aber als nachrangig betrachtet und zunächst nicht weiter verfolgt.

7.2 Das Glass-Box-Test-Werkzeug CodeCover

CodeCover ist ein Open-Source-GBT-Werkzeug, das an der Universität Stuttgart im Rahmen eines Studienprojekts entwickelt wurde. CodeCover unterstützt die Programmiersprachen Java, C und COBOL und liefert die in Kapitel 6 definierten Überdeckungsmetriken Anweisungs-, Zweig-, Block- und Schleifenüberdeckung sowie die Termüberdeckung.

7.2.1 Entstehung

CodeCover wurde von Herbst 2007 bis Herbst 2008 im Rahmen eines Studienprojekts in der Abteilung Software Engineering des Instituts für Softwaretechnik der Universität Stuttgart entwickelt. Eine ausführliche Übersicht zu diesem Studienprojekt liefern [CC08a, CC08b]. CodeCover ist damit zwar nicht das direkte Resultat dieser Arbeit, wurde aber ganz wesentlich von Vorarbeiten zu dieser Arbeit mitgeprägt, und umgekehrt flossen auch Erkenntnisse aus der Implementierung sowie dem praktischen Einsatz von CodeCover in diese Arbeit ein.

Obwohl bereits seit 2008 stabile Release-Versionen zur Verfügung stehen, wurde CodeCover laufend weiterentwickelt und erweitert. Hier sind u. a. die Diplomarbeiten von Schumm [Schu09] und Ebert [Ebe11] entstanden. Die CodeCover-Funktionen, die eine Visualisierung der Überdeckung verschiedener Testfälle betreffen, sind 2009 entstanden und in [KGM09] ausführlich beschrieben. Diese Arbeiten bilden überwiegend neue

Auswertungen der in CodeCover vorhandenen Überdeckungsdaten. Durch Einflüsse dieser Arbeit wurden aber auch das CodeCover-interne Datenmodell und die Metrikenberechnung verändert und erweitert, damit diese (weitgehend) den Definitionen dieser Arbeit entsprechen.

7.2.2 Funktionsübersicht

Eine wichtige Zielsetzung von CodeCover ist, dass es sowohl in der Lehre als auch in der industriellen Praxis eingesetzt werden kann. D. h. es soll einerseits für relativ kleine Programme und mit möglichst geringem Einarbeitungsaufwand gut zu benutzen sein. Zur Benutzerfreundlichkeit trägt auch die Integration in die Entwicklungsumgebung Eclipse bei. Andererseits soll CodeCover auch in Industrieprojekten, also bei großen Programmen und in einer heterogenen Integrationsumgebung, einsetzbar sein. Hierfür kann CodeCover über die Skriptsprache Apache Ant[4] in einen Build- und Testprozess integriert werden.

Mit der CodeCover-Entwicklung wird des Weiteren das Ziel verfolgt, ein für zukünftige Erweiterungen offenes GBT-Werkzeug bereitzustellen. Diese offenen Schnittstellen bilden bei CodeCover ein ganz wesentliches Entwurfsmerkmal und betreffen mehrere Bereiche des Werkzeugs:

- **Frontend**: Das Frontend liest den Programmcode des Prüflings ein und bildet das Original-Programm in das Werkzeug-interne Modell ab. Hier war von Anfang an das Ziel, mehrere Programmiersprachen zu unterstützen: Neben der Programmiersprache Java wurde auch ein Frontend für die Programmiersprache COBOL implementiert. Die Frontend-Schnittstelle ist so beschaffen, dass mit angemessenem Aufwand weitere Programmiersprachen integriert werden können. So wurde z. B. im Rahmen der Bachelor-Arbeit von Hanikel [Ha13] ein Frontend für die Programmiersprache C implementiert.
- **Metriken-Berechnung:** CodeCover verfügt über eine Schnittstelle, über die weitere Überdeckungsmetriken ergänzt werden können. So wurde z. B. die Blocküberdeckung, die Bedingter-Ausdruck-Überdeckung (vgl. Abschnitt 6.5.3 auf Seite 165) und die sogenannte synchronized-Überdeckung ergänzt, die Auskunft über die Behandlung von gegenseitigem Ausschuss gibt (vgl. Abschnitt 4.2.8 auf Seite 68).
- **Reporting-Funktion**: Die Überdeckungsberichte lassen sich individuell anpassen. Zudem lassen sich weitere Berichte über die Reporting-Schnittstelle integrieren.

Für das CodeCover-Frontend wurde neben der Programmiersprache Java auch die Programmiersprache COBOL gewählt, weil es 2007 kein anderes Open-Source-GBT-Werkzeug für COBOL gab. Zudem unterscheidet sich COBOL von Java erheblich, was bei der Festlegung der Werkzeug-internen programmiersprachenneutralen Abstraktion des Programmcodes – und in der Folge auch für die RPR von Kapitel 4 – ein Vorteil war.

Vor der Entwicklung von CodeCover fanden eine Reihe von Vorarbeiten statt. Lösch untersucht beispielsweise in [Lö05], wie die bei objektorientierten Programmen übliche

[4] Apache Ant ist ein Produkt der Apache Software Foundation (vgl. http://ant.apache.org/, letzter Zugriff 30.09.2013)

dynamische Bindung im GBT behandelt werden kann (vgl. Abschnitt 4.2.7). Beschreibungen von Vorarbeiten zur Behandlung von COBOL-Programmen und zur Instrumentierung von Ausdrücken befinden sich in [Schm04, Schm06]. Zur Implementierung des sogenannten Frontends wird der Scanner- und Parser-Generator des JavaCC [JCC] verwendet.

CodeCover steht unter EPL-Lizenz; der Quelltext, die Dokumentation und die aktuelle Version sowie der Zugriff auf das Versionskontrollsystem sind unter www.codecover.org verfügbar.

7.2.3 Benutzungsschnittstelle

CodeCover kann als sogenanntes Plugin in die Entwicklungsumgebung Eclipse integriert werden. Der wesentliche Vorteil gegenüber der Konsolen-Oberfläche oder der Verwendung der Ant-Skriptsprache ist die einfache Installation und Bedienung; zudem werden einige Auswertungen (wie z. B. die Visualisierung der Überdeckung) unmittelbar im Programmcode dargestellt. Allerdings steht diese Eclipse-Integration derzeit nur für die Programmiersprache Java zur Verfügung. Ein typisches Bild der CodeCover-Eclipse-Oberfläche zeigt Abbildung 57. Um CodeCover in einem Eclipse-Projekt in der dargestellten Form nutzen zu können, sind lediglich die folgenden drei Schritte erforderlich:

1. In den Projekteigenschaften wird auf der Seite „CodeCover“ die CodeCover-Funktion aktiviert, und die zu erhebenden Metriken werden ausgewählt.
2. Im Eclipse „Project Explorer” werden die Java-Pakete oder Klassen markiert, die in die GBT-Auswertung einbezogen werden sollen. Im Popup-Menü wird hierzu “Use For Coverage Measurement” ausgewählt.
3. Falls als Testtreiber JUnit verwendet wird, erfolgt die Ausführung der Testfälle über “CodeCover Measurement for JUnit”. Ansonsten wird in der Ausführungskonfiguration auf der CodeCover-Seite “Run with CodeCover” gewählt.

Sind die Einstellungen vorgenommen und der Prüfling ausgeführt worden, dann werden die im ausgewählten Ausführungsprotokoll gespeicherten Testfälle mit dem jeweiligen Ausführungszeitpunkt in der „Test Sessions“-Sicht angezeigt (der untere linke Bereich von Abbildung 57). Falls – wie im Beispiel von Abbildung 57 – ein Unit-Test mit dem Werkzeug JUnit durchgeführt wurde, dann werden die Bezeichner der JUnit-Testfälle als Testfall-Bezeichner verwendet.

Die in dieser Sicht ausgewählten Testfälle steuern die Überdeckungsbestimmung, die z. B. zur Visualisierung der Überdeckung im Programmcode herangezogen wird. Wird beispielsweise nur einer der Testfälle ausgewählt (wie in Abbildung 57 der Testfall „t2“), dann wird nur diejenige Überdeckung für die Auswertungen berücksichtigt, die vom ausgewählten Testfall ausgeht. Diese Funktion basiert auf dem testfallgenauen GBT-Protokoll. Der für GBT-Werkzeuge typische GBT-Bericht, der die Zusammenfassung der Überdeckung enthält, ist in Abbildung 57 (rechts unten) und in Abbildung 58 dargestellt.

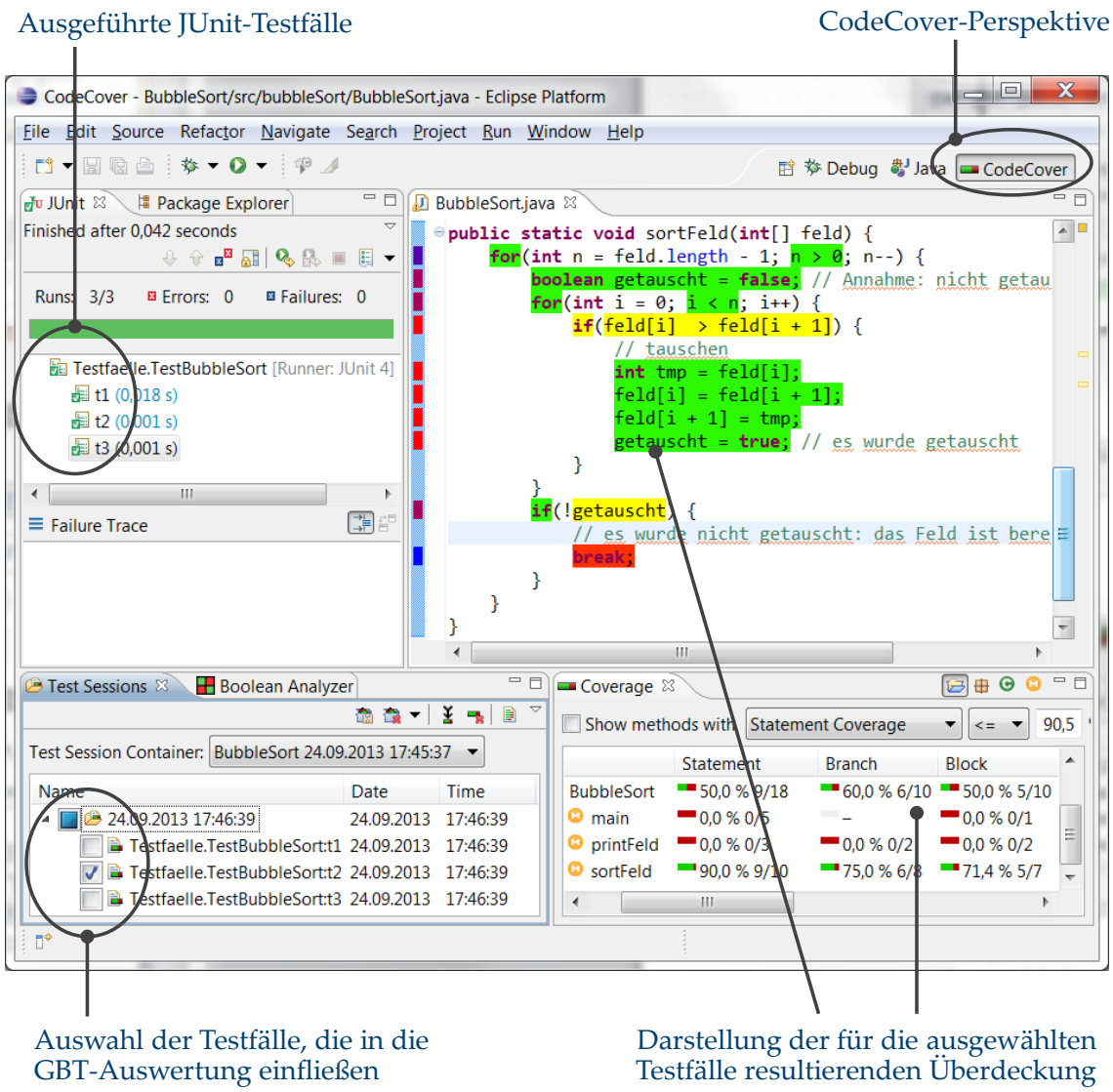


Abbildung 57: CodeCover mit der Eclipse-Oberfläche

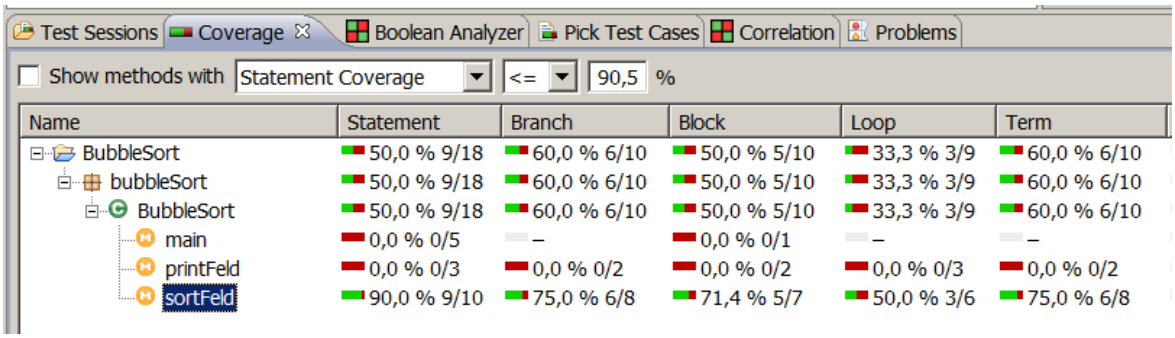


Abbildung 58: Überdeckungsbericht in der Eclipse-Oberfläche

Die Angabe zur Anweisungsüberdeckung der Methode „sortFeld“ in Abbildung 58 ist so zu verstehen, dass 9 von 10 Anweisungen ausgeführt wurden. Die nicht ausgeführte

break-Anweisung ist auch in Abbildung 57 durch die Rotfärbung gut zu erkennen. Entsprechend lässt sich aus den Angaben der Zweig- und Blocküberdeckung ablesen, dass 6 von 8 Zweigen bzw. 5 von 7 Blöcken ausgeführt wurden. Diese Angaben entsprechen exakt den Definitionen aus Kapitel 6.

Eine Unterstützung für den Tester bei der Analyse der Überdeckung von zusammengesetzten booleschen Ausdrücken kann der sogenannte „Boolean Analyser“ liefern, der in Abbildung 59 (im unteren Bereich) dargestellt ist. Zu jedem zusammengesetzten booleschen Ausdruck werden die Resultate der Einzelterme sowie die durch eine Termbelegung erzielte Termüberdeckung angegeben. Pro Zeile wird eine im Test aufgetretene Termbelegung angezeigt. In der rechten Spalte werden auch die Bezeichner der Testfälle angegeben, die zu der Termbelegung geführt haben. Die in der Darstellung markierten (grün hinterlegten) Werte sind wirksame Termwerte.

In Abbildung 59 (im oberen Bereich) wird die Termüberdeckung im Programmcode visualisiert. Die Gelbfärbung von Term A und D zeigt an, dass die Termüberdeckung nicht vollständig erreicht wurde. Die Grünfärbung von Term B, C und D zeigt eine vollständige Termüberdeckung an. Der Beispielausdruck und die Testfälle stammen aus [LL10, Seite 519].

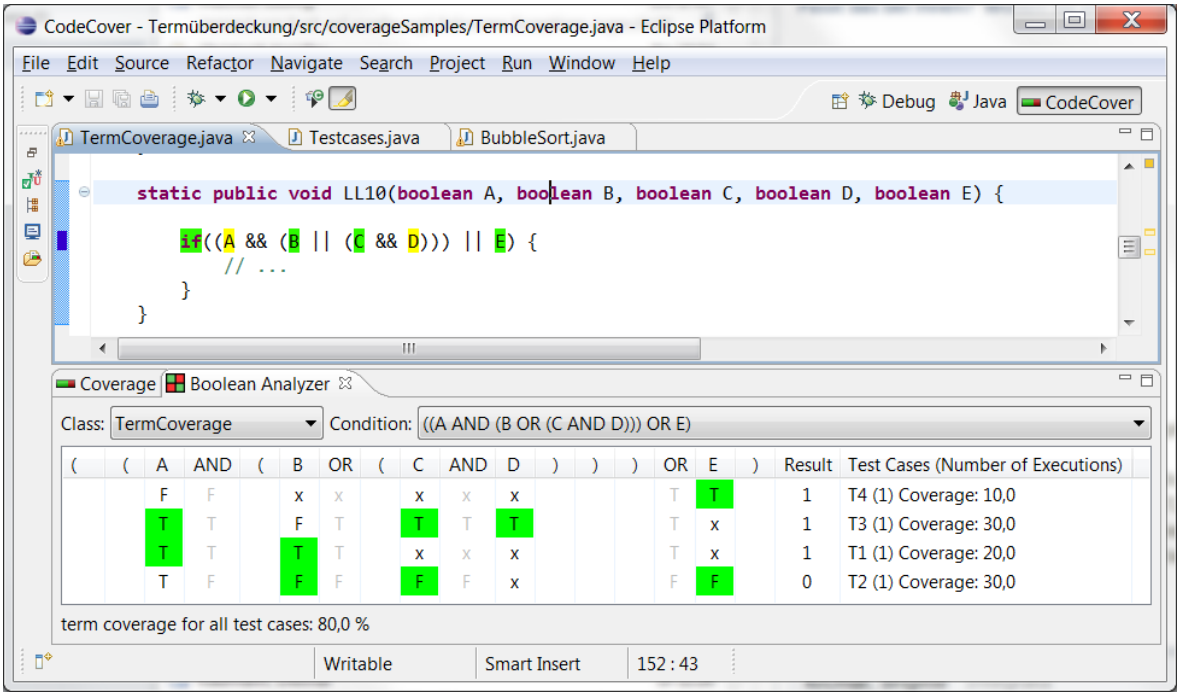


Abbildung 59: „Boolean Analyzer“ in der Eclipse-Oberfläche

7.2.4 Datenfluss

Abbildung 60 zeigt den Datenfluss, der in dieser Form dann für den Benutzer sichtbar ist, wenn die CodeCover-Konsolenoberfläche genutzt wird. Schematisch werden in dieser Abbildung das GBT-Modell und das GBT-Protokoll als zwei Artefakte dargestellt. In CodeCover fließt beides in eine gemeinsame Datei, den sogenannten TestSessionContainer (TSC), der zudem eine vollständige Kopie des gesamten betrachteten Programmcodes

enthält. Damit kann ein GBT-Bericht vollständig auf Grundlage des TSC erstellt werden. Insgesamt unterscheidet sich der CodeCover-Datenfluss kaum von dem anderer GBT-Werkzeuge, die ebenfalls Quelltext-Instrumentierung verwenden. Eine Besonderheit bildet allerdings der gestrichelt dargestellte Datenfluss von der Testsuite über die Testausführung in das GBT-Protokoll. Dieser Datenfluss ist durch das testfallgenaue GBT-Protokoll (vgl. Kapitel 3.3.4) begründet. Das GBT-Protokoll von CodeCover ist nach den Testfällen untergliedert, die zur jeweiligen Überdeckung geführt haben. Im GBT-Protokoll befinden sich auch die Referenzen auf die jeweils ausgeführten Testfälle. Als Referenz werden hierzu eindeutige Testfall-Bezeichner genutzt. In Kapitel 7.4 wird ausführlich beschrieben, wie CodeCover das testfallgenaue GBT-Protokoll erhebt.

Wenn die Eclipse-Integration von CodeCover eingesetzt wird, liegt der dargestellte Datenfluss in gleicher Weise vor, ist aber für den Nutzer nicht sichtbar. Das Instrumentieren, Kompilieren und die Ausführung des instrumentierten Programmcodes sind dann in den Eclipse-Build-Prozess und die Eclipse-Ausführung integriert. Auch wird der TSC im gleichen Format wie bei Verwendung der Konsolenoberfläche erstellt.

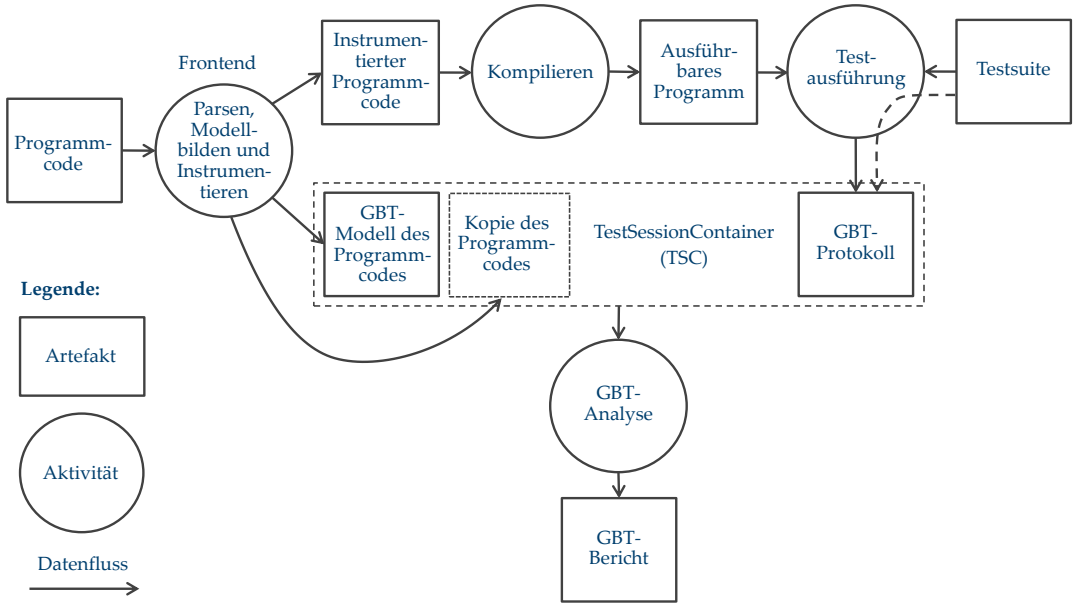


Abbildung 60: CodeCover-Artefakte und -Datenfluss

Ein TSC ist nur dann konsistent, wenn das GBT-Protokoll und das GBT-Modell des Programmcodes von derselben Programmversion stammen. Schumm zeigt in [Schu08], dass auch bei geringen Programmänderungen die zuvor erhobenen GBT-Protokolle obsolet werden können. Aus diesem Grund werden von CodeCover im GBT-Modell und im GBT-Protokoll Versionskennungen eingetragen, mit denen die Konsistenz sichergestellt wird.

7.2.5 Metrikenberechnung

Das CodeCover-Modell eines Programms bildet einen Baum, in dem die hierarchische Struktur des Gesamtprogramms abgebildet ist. Bei Java-Programmen wird so die Paketstruktur mit den Klassen widergespiegelt, und das „default“-Paket bildet die Wurzel. Die Java-Methoden bilden schließlich die Programme im Sinne des Referenzmodells, die als

Teilbäume unterhalb der Klassen-Knoten im CodeCover-Modell abgebildet sind. Um für einen TSC (d. h. für ein Programm sowie ein GBT-Protokoll) eine GBT-Überdeckung zu berechnen, wird dieser Baum ausgehend von der Wurzel in Nachordnung traversiert, und es wird abhängig vom Typ des Knotens die Überdeckung für die ausgewählten Testfälle des GBT-Protokolls ermittelt. CodeCover implementiert für dieses Traversieren das Visitor-Entwurfsmuster (vgl. [Ga95]). Der besondere Vorteil des Visitor-Entwurfsmusters in diesem Fall ist, dass die Berechnung der Überdeckung nicht Teil des GBT-Modells ist, sondern Teil der Visitor-Implementierung. Einerseits können so für ein GBT-Modell verschiedene Metriken erhoben werden, und andererseits lassen sich relativ leicht weitere Metriken durch Implementierung weiterer Visitor-Klassen ergänzen.

Zur Überdeckungsberechnung wird in CodeCover pro Ausführungselement ein sogenanntes Resultatstupel bestimmt. Um beispielsweise für eine Anweisung des GBT-Modells die Überdeckung zu bestimmen, wird ein Tupel (1/1) geliefert, wenn die Anweisung für die gegebenen Testfälle ausgeführt wurde, ansonsten das Tupel (0/1). Dieses Resultatstupel liefert damit das Gewicht und den Ausführungswert (vgl. Kapitel 6.3.2 auf Seite 136) für ein Ausführungselement sowie die betrachtete Metrik. Bei Anweisungen und der Anweisungsüberdeckung hat dieses Gewicht den Wert 1, und der Ausführungswert beträgt 1, wenn die Anweisung ausgeführt wurde, ansonsten 0.

7.3 Abdeckung der Anforderungen durch CodeCover

CodeCover abstrahiert den Original-Programmcode weitgehend in der Form, wie es in Kapitel 4 und 5 vorgeschlagen wird. Auch werden die Ausführungszähler so in den Originalprogrammcode eingefügt, dass deren Werte den Markierungen der Zählerstellen der Modellnetze von Kapitel 5 entsprechen. In den meisten Fällen, in denen es Modellunterschiede gibt, erfolgt in der Metrikenberechnung eine entsprechende Korrektur, sodass CodeCover die Anweisungs-, Zweig- , Block- und Schleifenüberdeckung exakt nach den Definitionen von Kapitel 6 liefert.

Auch wird die Termüberdeckung genau nach der Definition dieser Arbeit (und damit der Definition nach [LL10]) geliefert. Die Implementierung entspricht allerdings (noch) nicht dem in Abschnitt 6.4.2 beschriebenen Algorithmus mit der „on-the-fly“-Bestimmung der Termüberdeckung. In der aktuellen CodeCover-Version werden die Wertebelegungen der Terme eines Ausdrucks „uninterpretiert“ im GBT-Protokoll abgespeichert. Die Auswertung erfolgt dann nach Testende.

Dagegen fehlt in der aktuellen CodeCover-Version die Entscheidungsüberdeckung. Die dazu notwendigen Modellmerkmale wie die verzweigenden Kontrollstrukturen, der bedingte Ausdruck und die booleschen Verbundausdrücke sind im CodeCover-Modell aber prinzipiell enthalten, sodass eine Implementierung dieser Metrik ohne großen Aufwand noch erfolgen kann.

Zu den genannten Modellunterschieden, die nicht während der Metrikenberechnung korrigiert werden können, zählen solche booleschen Ausdrücke, die nicht Bedingungsausdruck von Entscheidung, Schleife oder bedingtem Ausdruck sind. Diese sollen nach der Definition von Kapitel 6 zur Entscheidungs- und Termüberdeckung beitragen. Aller-

dings werden diese in CodeCover in der aktuellen Version nicht berücksichtigt. Eine entsprechende Überarbeitung ist geplant.

7.4 Testfallgenaues Glass-Box-Test-Protokoll

Das testfallgenaue GBT-Protokoll ist erforderlich, um die GBT-Überdeckung getrennt für einzelne Testfälle bestimmen zu können (vgl. Kapitel 3.3.4). Für die Programmiersprache Java sind im Werkzeug CodeCover hierzu zwei Techniken vorgesehen: Eine JMX-Schnittstelle, die speziell beim Systemtest genutzt werden kann, und eine JUnit-Schnittstelle für den Unit-Test.

7.4.1 JMX-Schnittstelle

Im Rahmen der Instrumentierung kann in den Prüfling ein sogenanntes MBean eingewoben werden. Dieses MBean erlaubt es über ein standardisiertes Protokoll der Java-VM (die sogenannte JMX-Schnittstelle [Go05]), Mitteilungen an den Prüfling zu übermitteln. Damit ist es möglich, dass während der (Test-)Ausführung des Prüflings Botschaften an den Prüfling gesendet werden können. Das durch CodeCover in den Prüfling eingewobene MBean bietet zwei wesentliche Funktionen an: Erstens „Initialisieren“, womit die Zählerstände der Ausführungszähler auf null gesetzt werden. Und zweitens „Auslesen“, womit die Zählerstände der Ausführungszähler zusammen mit der Testfall-Kennung außerhalb des Prüflings abgespeichert werden können.

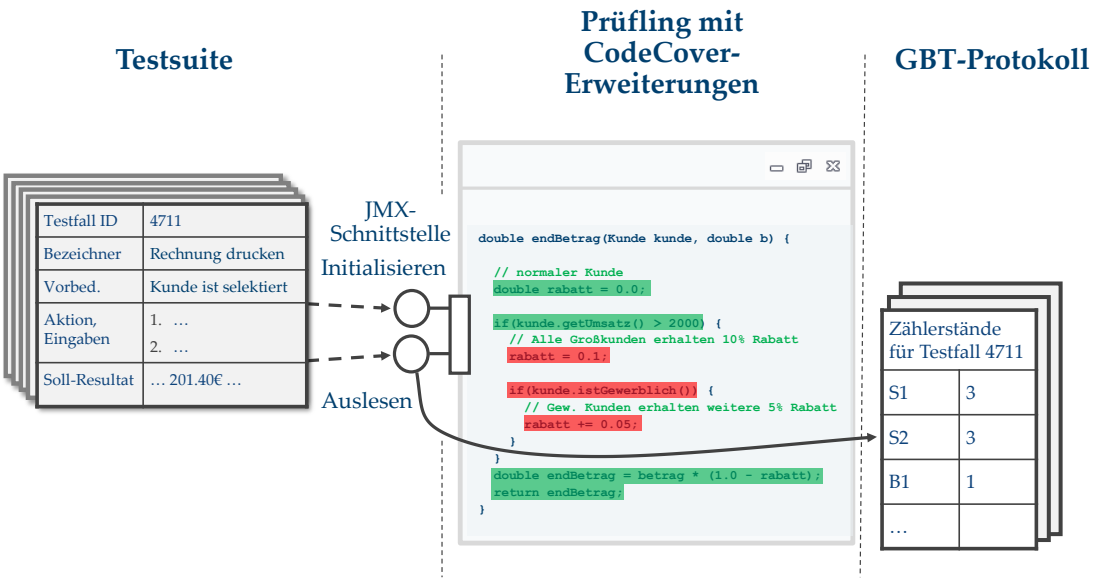


Abbildung 61: Prüfling mit JMX-Schnittstelle

Abbildung 61 zeigt schematisch diesen Ablauf. Mit dem Start der Testfallausführung wird die Initialisieren-Botschaft gesendet, und mit Ende dieser Ausführung folgt die Auslesen-Botschaft. Die Zählerstände werden dann im GBT-Protokoll abgespeichert.

Beide Botschaften können vom Tester über die CodeCover-Oberfläche aufgerufen werden. Alternativ kann das Testfall-Verwaltungswerkzeug Justus [Justus] verwendet werden, das, genau wie in Abbildung 61 dargestellt ist, im Verlauf der Testausführung

die entsprechenden Botschaften an den Prüfling aussendet, ohne dass der Tester dabei eingreifen muss [Schm11].

7.4.2 JUnit-Integration

JUnit [JUnit] ist ein in der industriellen Praxis weit verbreitetes Unit-Test-Treiberprogramm (vgl. Kapitel 2.3.1, Seite 15 ff). Kennzeichnend für JUnit ist, dass eine mit der Annotation „@Test“ versehene Methode als ein Testfall interpretiert wird. Das Werkzeug CodeCover fügt sich derart in die JUnit-Testausführung ein, dass der Beginn und das Ende eines Unit-Testfalls an das in den Prüfling eingewobene MBean übermittelt werden. Die Behandlung dieser Ereignisse findet genau wie bei der oben beschriebenen JMX-Schnittstelle statt. Auf diese Weise lässt sich der Überdeckungsanteil jedes einzelnen Unit-Testfalls auswerten und darstellen. In [KGM09] wird diese Funktion beispielsweise dazu genutzt, um für die Unit-Testfälle paarweise die sogenannte Redundanz festzustellen, also den Grad, zu dem zwei Unit-Testfälle in der Ausführung überlappen.

7.5 Evaluation des Werkzeugs CodeCover

Über den Einsatz von CodeCover in Industrieprojekten wird in [Schm11] sowie von Ebert [Ebe11] berichtet. Die dort beschriebenen Java-Projekte sind 50 und 160 kLOC groß. Zur Instrumentierung wird in beiden Projekten die Batch-Schnittstelle von CodeCover genutzt. Das Instrumentieren von Projekten dieser Größe wird als problemlos beschrieben und dauert nur wenige Sekunden. Die dabei generierte TSC-Datei, die das RPR-Modell sowie den Original-Programmcode enthält, wird dann zwar relativ groß (etwa 50 MB bei 160 kLOC Programmcode), es lassen sich aber alle Auswertungen normal ausführen. In beiden Projekten werden je etwa 200 Testfälle im GBT ausgeführt, deren Ausführung testfallgenau gespeichert und ausgewertet wird.

Einen Vergleich von CodeCover mit anderen GBT-Werkzeugen, der speziell den Einsatz in der Lehre betrachtet, liefert [Kle09]. Weitere Berichte über den Einsatz von CodeCover in der Lehre finden sich bei [KGM09, Ga10, CK11, KMT12]. Generell wird in diesen Artikeln die Brauchbarkeit und speziell die Bedienbarkeit von CodeCover als sehr gut bewertet. Die Autoren von [KMT12] vergleichen CodeCover mit dem Werkzeug Emma [Emma] und betrachten neben der Bedienbarkeit auch die von den Werkzeugen gelieferten Überdeckungsmetriken. Abschließend stellen sie fest: *„[…] we conclude that CodeCover tool reports a more accurate coverage information than Emma [...]“*.

7.5.1 Funktionstest

Beim Funktionstest eines GBT-Werkzeuges bilden die Programme mit ihren Programmausführungen die Test-Eingabedaten. Das Soll-Resultat eines solchen Testfalls ist die erwartete Überdeckung. Diese erwartete Überdeckung kann aber letztlich nur „von Hand“ ermittelt werden, weil es kein Referenz-GBT-Werkzeug gibt, das als Orakel herangezogen werden könnte. Damit bleibt ein Test, der die vom Werkzeug gelieferte Überdeckung prüft, immer auf kleine Programme begrenzt.

Der Test von CodeCover ist vergleichbar angelegt wie der Test, den die Autoren von [FAA007] beschreiben: Für die verschiedenen Ausführungselemente wird mindestens ein Testfall gewählt, in dem das Ausführungselement nicht überdeckt sein soll, und mindestens ein Testfall, in dem die Überdeckung erwartet wird. Diese Testfälle liegen bei CodeCover vollständig als Unit-Tests vor.

Das Referenzprogramm aus Abschnitt 6.6 auf Seite 166 kann auch als ein Testfall aufgefasst werden. Abbildung 62 zeigt, wie CodeCover die Überdeckung für dieses Programm mit der vorgegebenen Programmausführung visualisiert. In Abbildung 63 ist der GBT-Bericht für die Referenzausführung angegeben. Die von CodeCover angezeigten Werte entsprechen genau den „von Hand“ auf Grundlage des Modells ermittelten Soll-Resultaten.

```
// Fall 0 -> Standardanweisung, Zuweisung; a = 42, b =
int a = Integer.valueOf(args[0]), b = Integer.valueOf(

// Fall 1 -> then wird ausgeführt, kein else
if(a == 42) {
    System.out.println("1");
}

// Fall 2 -> nur then wird ausgeführt, befüllter else
if(a == 42) {
    System.out.println("2 - then");
} else {
    System.out.println("2 -else");
}

// Fall 3 -> leerer else wird ausgeführt
if(b == 42) {
    System.out.println("3");
}

// Fall 4 -> befüllter else wird ausgeführt
if(b != 7) {
    System.out.println("4 - then");
} else {
    System.out.println("4 - else");
}

// Fall 5 -> einer der cases wird ausgeführt
switch(a) {
    case 42: System.out.println("5 - 42"); break;
    case 9: System.out.println("5 - 9"); break;
    case 1: System.out.println("5- 1"); break;
    default: System.out.println("5 -default");
}

// Fall 6 -> default wird ausgeführt
switch(a) {
    case 99: System.out.println("6 - 99"); break;
    default: System.out.println("6 -default");
}
```

```
// Fall 7 -> keiner der cases, kein default wird ausgeführ
switch(a) {
    case 99: System.out.println("7 - 99"); break;
}


// Fall 8 -> Schleifenkörper wird N-fach ausgeführt
int x = b; // a = 42, b = 7
while(x < a) {
    x++;
    System.out.println("8 - x  = " + x);
}


// Fall 9 -> Schleifenkörper wird nicht ausgeführt
x = b; // a = 42, b = 7
while(x > a) {
    x++;
    System.out.println("9 - x = " + x);
}


// Fall 10 -> Division durch Null Exception
try {
    b = b / (b - 7);
    System.out.println("10 - b = " + b);
} catch(Exception e) {
    System.out.println("10 - " + e.getMessage());
}
finally {
    System.out.println("8 - finally");
}

// Fall 11 -> keine Exception
try {
    b = b / (b - 2);
    System.out.println("11 - b = " + b);
} catch(Exception e) {
    System.out.println("11 - " + e.getMessage());
}
```

Abbildung 62: Visualisierung der GBT-Überdeckung

Name	Statement	Branch	Block	Loop	Term
GBTToolsTest	62,8 % 27/43	50,0 % 11/22	52,2 % 12/23	33,3 % 2/6	58,3 % 7/12

Abbildung 63: GBT-Bericht der Ausführung des Referenzprogramms

7.5.2 Prüfung der Instrumentierung

Eine korrekte Instrumentierung ist für die korrekte Ermittlung der Überdeckungsmetriken Grundvoraussetzung. Wie für Java-Programme die Instrumentierung zu erfolgen hat, ist für Anweisungen ausführlich in Abschnitt 5.11.1 auf Seite 114 und für Ausdrücke in Abschnitt 5.13.6 auf Seite 125 beschrieben.

Im Folgenden wird für Anweisungen und Ausdrücke die Instrumentierung durch CodeCover an zwei Beispielen gezeigt. Als Beispiel zur Instrumentierung von Anweisungen wird das folgende Original-Programm betrachtet:

```
static void f1() {
  statementA();
  statementB();
}
```

Die Instrumentierung durch CodeCover ergibt:

```
static void f1() {
  CCCounter$a7tm3igr.statements[1]++;
  statementA();
  CCCounter$a7tm3igr.statements[2]++;
  statementB();
  CCCounter$a7tm3igr.statements[3]++;
}
```

Die (fett gedruckte) Instrumentierung entspricht damit im Wesentlichen der in Abschnitt 5.13.6 beschriebenen Form. Das Array der Ausführungszähler *statements* ist öffentliches Attribut einer sogenannten Counter-Klasse, die, um Namenskonflikte zu vermeiden, einen generierten (und hier gekürzt wiedergegebenen) Bezeichner hat. Pro Java-Klasse wird eine Counter-Klasse generiert. Der Index für den Zugriff in das *statements*-Array wird über die Identifier des RPR-Modells bestimmt.

Die Instrumentierung der booleschen Ausdrücke entspricht vom Grundsatz her ebenso der in Abschnitt 5.13.6 sowie in Abschnitt 6.4.2 beschriebenen Form. Aus Effizienzgründen wird anstelle des dort vorgeschlagenen booleschen Arrays eine (einzelne) Integervariable verwendet, die aber bitweise betrachtet wird. Es wird je ein Bit für „Ausdruck ist aktiv“ und „Ausdruck ist true“ verwendet. Damit können bei einem 32-Bit-Integer immerhin Gesamtausdrücke mit 16 Einzeltermen abgebildet werden. Falls längere Ausdrücke auftreten, werden weitere Integervariablen hinzugenommen. Z. B. wird der Ausdruck

```
if(A && B) {
  // ...
}
```

von CodeCover wie folgt instrumentiert:

```
int CCHelper_C1;  // die Integervariable, die als
```

```
                    // „Bit-Array“ betrachtet wird
if(
   (
      (
      ((CCHelper_C1 |= (8)) == 0 || true) &&
      ((A) && ((CCHelper_C1 |= (4)) == 0 || true))
      )
      &&
      (((CCHelper_C1 |= (2)) == 0 || true) &&
      ((B) && ((CCHelper_C1 |= (1)) == 0 || true)))
   ) /* 1 */
) {
   // ...
}
```

An der mit / * 1 */ gekennzeichneten Stelle folgt noch der ergänzte Programmcode, um die festgestellte Wertekombination, die nun in der Variablen *CCHelper_C1* abgelegt ist, abzuspeichern.

CodeCover implementiert allerdings in der aktuellen Version (noch) nicht die „on-the-fly“- Ermittlung der Termüberdeckung, wie es im Algorithmus von Abschnitt 6.4.2 vorgeschlagen wird. Diese Erweiterung von CodeCover ist aber in Planung.

7.5.3 Glass-Box-Test

Beim Test von CodeCover bietet sich selbstverständlich auch ein GBT an, der in diesem Fall mit CodeCover selbst vorgenommen wird. Wie bei jedem größeren System hat der GBT auch bei CodeCover eine Reihe von Besonderheiten:

- CodeCover ist modular aufgebaut, und es bietet sich eine modulweise Betrachtung der Überdeckung an. So wird man z. B. die Java-, C- und COBOL-Frontends getrennt betrachten, weil auch die Testsuiten getrennt gehalten werden.
- Es gibt in CodeCover automatisch generierten Programmcode. Die Pakete „parser“ und „syntaxtree“ (siehe Abbildung 64) sind beispielsweise ganz überwiegend durch den Javacc-Parsergenerator generiert worden und bilden für das Modul „instrumentation-java“ einen erheblichen Anteil am gesamten Programmcode (in etwa 2/3). Hier führt defensiver (und z. T. überflüssiger) Programmcode auch bei gründlichem Test nur zu einer geringen Überdeckung.
- Da die Generierung eines CodeCover-Releases über Ant-Skripte erfolgt, wird nicht die (komfortable) Integration in die Eclipse-Umgebung zur Instrumentierung und Ausführung genutzt. Stattdessen ist die Instrumentierung in die Ant-Sktipte integriert, und das Einlesen des GBT-Protokolls für die einzelnen Testfälle erfolgt über die Import-Funktion. Für die Auswertung kann dann aber wieder die Eclipse-Oberfläche genutzt werden.

Das Instrumentieren des Referenzprogramms von Abschnitt 6.6 auf Seite 166 ergibt den Überdeckungsbericht, der als Ausschnitt in Abbildung 64 dargestellt ist.

	Statement	Branch	Block
parser	29,2 % 2127/7275	22,9 % 1078/4715	25,6 % 1358/5306
syntaxtree	25,3 % 593/2347	14,8 % 135/912	18,5 % 375/2024
visitor	22,9 % 641/2801	41,4 % 89/215	27,1 % 230/848
DepthFirstVisitor	17,6 % 51/289	70,0 % 7/10	29,4 % 35/119
DepthFirstVisitorWithExcepI	30,8 % 89/289	80,0 % 8/10	43,7 % 52/119
EndOffset	100,0 % 5/5	–	100,0 % 3/3
GJDepthFirst	0,0 % 0/520	0,0 % 0/10	0,0 % 0/120
GJNoArguDepthFirst	0,0 % 0/520	0,0 % 0/10	0,0 % 0/120
GJNoArguVisitor	–	–	–
GJVisitor	–	–	–
GJVoidDepthFirst	0,0 % 0/295	0,0 % 0/10	0,0 % 0/119
GJVoidVisitor	–	–	–
InstrBasicBooleanVisitor	66,7 % 16/24	60,0 % 6/10	64,3 % 9/14
InstrumentationVisitor	54,6 % 441/807	39,7 % 54/136	53,1 % 103/194

Abbildung 64: GBT-Bericht zu „Instrumentieren des Referenzprogramms“

Für die Klasse, die im Wesentlichen das Einlesen des Prüflings vornimmt (die Klasse *InstrumentationVisitor*, diese ist in Abbildung 64 hervorgehoben), ergibt sich (nur) eine Anweisungsüberdeckung von knapp 55 %. Dies liegt daran, dass im Referenzprogramm einige der Java-Programmkonstrukte fehlen, die in CodeCover behandelt werden. Da aber keines der anderen untersuchten Werkzeuge Term- oder Schleifenüberdeckung erheben kann, sind im Referenzprogramm auch keine darauf zugeschnittenen Testfälle enthalten.

Die beim Programmcode von CodeCover enthaltenen Unit-Tests zum Test des Frontends und der Instrumentierung führen aber zu einer vollständigen Überdeckung der *InstrumentationVisitor*-Klasse.

8
Die Generierung von Testfall-Hinweisen mit CodeCover

In diesem Kapitel wird ein in CodeCover implementiertes Verfahren zur Unterstützung beim Testfallentwurf vorgestellt. Aus den Resultaten des GBT werden dem Tester sogenannte Testfall-Hinweise geliefert, die einen praktikablen Ausgangspunkt zum Entwurf neuer Testdaten bilden. Der Tester kann auf dieser Grundlage systematisch Testfälle entwickeln, die zu einer Erhöhung der GBT-Überdeckung führen. Da für Projekte der industriellen Praxis mit sehr vielen dieser Testfall-Hinweise zu rechnen ist, wird zur Priorisierung eine Heuristik vorgestellt. Auf diese Weise wird eine wichtige Forderung der neueren Literatur zum GBT berücksichtigt: Neben der Erhöhung der GBT-Überdeckung werden beim Testfallentwurf weitere Kriterien mit einbezogen, die die Effizienz und Effektivität der Testfälle betreffen.

8.1 Einführung

Der Literatur folgend (z. B. [EBI06, BWK07, MND09]), bildet der GBT eine Art „Motivator“, der den Tester so lange auffordert, neue Testfälle zu bilden, bis eine angestrebte Mindestüberdeckung erreicht ist. Mockus et al. stellen in [MND09] hierzu fest: *“By introducing Coverage we have observed that developers who did not write unit tests have started doing so”*. Das hier zugrundeliegende Kalkül besteht darin, dass die neuen Testfälle zwar zunächst mit dem Ziel der Erhöhung der Überdeckung gebildet werden. Aber mit dieser Erhöhung werden unweigerlich Eingabedaten gefunden, die bislang ungetestete Funktionen des Programms ansprechen. Auch kann man annehmen, dass das erstmalige Ausführen eines Programmcodebereichs mehr Chancen zur Fehleraufdeckung bietet als die erneute Ausführung mit variierten Eingaben. Die in Kapitel 2.7 vorgestellten Untersuchungen zur Wirksamkeit des GBT bestätigen diese Annahme: Die Ausführung von Testfällen, mit denen eine Erhöhung der GBT-Überdeckung einhergeht, hat eine signifikant höhere Fehlerentdeckungsrate als die Ausführung von Testfällen, die keinen Beitrag zur Erhöhung der Überdeckung leisten. Damit bleibt aber die Frage unbeantwortet, *wie* der Tester aus den Resultaten des GBT neue Testfälle entwickelt, die zu einer Erhöhung der Überdeckung führen. Das Hauptproblem dabei ist, dass der Tester die Programmausführung nicht gezielt zum nicht ausgeführten Programmcode führen kann, sondern dies über passend gewählte Eingabedaten erreichen muss. Bei kleinen Prüflingen wie beispielsweise

beim Modultest ist der Zusammenhang zwischen Programmausführung und Eingabedaten in vielen Fällen noch einfach, und der Tester kann vergleichsweise leicht auf Eingabedaten schließen, die zur Erhöhung der GBT-Überdeckung führen. Nicht zuletzt wird aus diesem Grund der GBT in der Testliteratur fast ausschließlich beim Modultest beschrieben. Beim Systemtest großer Programme ist es aber ungleich schwieriger, von gezielt auszuführendem Programmcode auf die entsprechenden Eingaben zu schließen. Einen weiteren wichtigen Aspekt beim Entwurf der GBT-Testfälle nennen Berner, Weber und Keller in [BWK07]. Sie weisen darauf hin, dass die Erhöhung der Überdeckung nur ein technisches Hilfsmittel darstellt, um neue Testfälle entwickeln zu können. Bei diesem Testfallentwurf muss aber immer die Fehlersensitivität mit berücksichtigt werden. Allein aus der Erhöhung der Überdeckung wird nach Aussage der Autoren nicht der entscheidende Qualitätsgewinn erreicht, sondern die neu entwickelten Testfälle müssen auch möglichst gezielt die schwerwiegenden Fehler des Programms anzeigen. In [BWK07] wird dieses Problem des GBT wie folgt zusammengefasst: *„ The coverage rate may increase, but the tests are not improved in quality".* Eine systematische Anleitung, wie die Tester aus den GBT-Resultaten neue Testfälle entwickeln sollen, liefern die Autoren aber nicht (vgl. Kapitel 3.2.4).

Damit ergeben sich zwei zentrale Anforderungen an ein Verfahren zur Entwicklung neuer Testfälle: Erstens soll der Tester dabei unterstützt werden, Eingabedaten zu finden, die zur Ausführung von bislang nicht ausgeführtem Programmcode führen. Auf diese Weise wird die GBT-Überdeckung erhöht. Und zweitens soll die Fehlersensitivität dieser Testfälle berücksichtigt werden. Da in der Regel keine vollständige Überdeckung erreicht wird, soll die Entwicklung solcher Testfälle bevorzugt werden, die eine größere Chance haben, einen schwerwiegenden Fehler anzuzeigen. Im Folgenden wird ein solches Verfahren vorgestellt.

8.2 Grundgedanke

Gegeben sei ein Java-Programm nach Abbildung 65, zu dem ein Test mit den Testfällen des Black-Box-Tests durchgeführt und die GBT-Überdeckung mit einem GBT-Werkzeug erhoben wird. Die hell hinterlegten Anweisungen sind bei diesem Test ausgeführt worden, die dunkel hinterlegten Anweisungen nicht. So ist leicht zu erkennen, dass der then-Block der if-Anweisung von Zeile 4 nicht ausgeführt wird. Die if-Anweisung mit dem Entscheidungsausdruck wird ausgeführt – der Entscheidungsausdruck wird für die gewählten Testfälle aber nicht zu true. Für den Tester stellt sich nun die Frage, mit welchen Eingabedaten er das Resultat des Entscheidungsausdrucks der if-Anweisung so verändert, dass damit die Zeilen 6 bis 9 ausgeführt werden. Der Programmpfad von der Eingabe der Test-Eingabedaten bis an die Programmstelle in Zeile 6 ist bei großen Programmen allerdings oft kaum nachvollziehbar. Gezielt mit Eingabedaten eine nicht ausgeführte Programmstelle wie in Zeile 6 zu erreichen, ist damit praktisch ausgeschlossen. Abhilfe schaffen aber die Testfälle, die die Verzweigungsstelle in Zeile 4 ausführen: Mit einem dieser Testfälle als Ausgangspunkt müssen lediglich diejenigen Eingabedaten variiert werden, die Einfluss auf die Bedingung der betrachteten Verzweigungsstelle haben.

```
 1 double berechneEndBetrag(Kunde kunde, double betrag) {
 2   // normaler Kunde
 3   double rabatt = 0.0;
 4   if(kunde.getUmsatz() > 2000) {
 5     // Alle Großkunden erhalten 10% Rabatt
 6     rabatt = 0.1;
 7     if(kunde.istGewerblich()) {
 8       // Gewerbliche Kunden erhalten weitere 5% Rabatt
 9       rabatt += 0.05;
10     }
11   }
12   double endBetrag = betrag * (1.0 - rabatt);
13   return endBetrag;
14 }
```

Abbildung 65: Beispiel Java-Programm mit Visualisierung der Überdeckung

8.2.1 Definitionen

Def. *Seien P ein Programm, A ∈ items(P) ein Ausführungselement, T eine Testsuite zum Test von P und t ∈ T ein Testfall. Dann ist* **testcases(A)** *⊆ T die Menge von Testfällen, die zur Ausführung von A führen.*

t ∈ testcases(A)	⇔	exe(A, t)

Satz: Für zwei Ausführungselemente A und B eines Programms P und einen Testfall t ∈ T gilt:

A dom B ∧ exe(B, t)	⇒ exe(A, t)	(1)
A dom B ∧ ¬exe(A, t)	⇒ ¬exe(B, t)	(2)

Beweis: Wegen A dom B (d. h. das Netz von B ist indirekt eingebettet im Netz von A) führt definitionsbedingt jeder Markenfluss zur Eintrittsstelle von B über die Eintrittsstelle von A. Zu (1): Wenn ausgehend von der Anfangsmarkierung M_0 des Programms die Transition im Eintrittsbereich von B mindestens einmal geschaltet hat, d. h. $| M(s_{In}(B)) | > 0$, dann hat auch die Transition im Eingangsbereich von A mindestens einmal geschaltet; d. h. es gilt damit $| M(s_{In}(A)) | > 0$. Zu (2): Wenn ebenso ausgehend von M_0 gilt $| M(s_{In}(A)) | = 0$ und jeder Markenfluss zur Eintrittsstelle von B durch die Eintrittsstelle von A führt, muss auch $| M(s_{In}(B) | = 0$ gelten.

Ergänzend sei noch anzumerken, dass Aho et al. in [ASU99] die Dominanzbeziehung auf Grundlage des CFG nach (1) definieren: „Ein Knoten d eines CFG dominiert einen Knoten *n*, geschrieben *d dom n*, falls jeder Pfad vom Startknoten des CFG nach *n* durch *d* führt. Nach dieser Definition dominiert jeder Knoten sich selbst, und der Eingang einer Schleife dominiert alle Knoten in der Schleife.“ Auch wird in der Literatur zum Übersetzerbau

zwischen Dominanz und strikter Dominanz unterschieden. Die Dominanz ist dort reflexiv (d. h. es gilt *n dom n*); für die strikte Dominanz *sdom* gilt $n\ sdom\ m \Leftrightarrow n\ dom\ m \wedge n \neq m$. Der direkte Dominator wird auch als *immediate dominator* bezeichnet, und die Relation wird als *idom* geschrieben. Die Konstruktion des Dominanzbaums ist allerdings durch die flexible Struktur des CFG aufwändig. Eine ausführliche Beschreibung hierzu befindet sich in [ASU99, Ag99, Gi05]. In dieser Arbeit wird die Dominanzbeziehung durch die Komposition der Ausführungselemente definiert, und (1) und (2) bilden aus dieser Komposition Schlussfolgerungen.

Es finden sich auch einige Artikel zu GBT (z. B. [Ag99, MB03, Gi05]), die Schlussfolgerungen aus (1) dazu nutzen, möglichst effektive Testfälle (hinsichtlich der GBT-Überdeckung) zu entwickeln. Es soll mit möglichst wenigen Testfällen eine hohe Überdeckung erzielt werden. Die Autoren versuchen dazu, die Blätter des Dominanzbaums gezielt durch Testfälle zu erreichen. Für das im Folgenden vorgestellte Verfahren zum Entwurf neuer Testfälle bildet dagegen die Schlussfolgerung aus (2) die Grundlage: Aus praktischen Erwägungen heraus wird bei zwei Ausführungselementen *A* und *B* mit *A dom B* die Überdeckung von *B* nur dann angestrebt, wenn *A* bereits überdeckt ist. D. h. es erfolgt ein schrittweises „Vortasten“ der Überdeckung, von der Wurzel des Dominanzbaums ausgehend.

8.3 Anleitung zum Testfallentwurf

Die Darstellung des in Abbildung 65 auf Seite 185 gezeigten Programmcodes als GBT-Modell zeigt Abbildung 66. Das äußere Ausführungselement entspricht der if-Anweisung von Zeile 4. Das nicht ausgeführte und mit dick gezogenem Rand dargestellte Ausführungselement ist der then-Block, der in Zeile 5 beginnt. Die Testfälle, die die if-Anweisung ausführen, dann aber den else-Zweig nehmen – und den then-Block dabei „tangieren“ – sind durch den dick gezogenen Pfeil dargestellt. Das zugrunde liegende Muster lässt sich für ein Programm *P* und eine Testsuite *T* so beschreiben: Es gibt ein Ausführungselement *A* aus *P* (hier die if-Anweisung), das von Testfällen von *T* ausgeführt wird (der Ausführungszählerstand ist folglich größer null), und ein Ausführungselement *B* aus *P*, das von *A* direkt dominiert wird, aber von den Testfällen von *T* nicht ausgeführt wird (der then-Block mit dem Ausführungszählerstand null).

Den Arbeiten zur automatischen Testdatengenerierung [Ko90, GMS00, WBS01, Ost07] folgend wird dieser hervorgehobene then-Block als Testziel bezeichnet. Dieses Muster wird im Folgenden als Ausgangspunkt zur Entwicklung neuer Testfälle genutzt. Dabei spielen zwei Merkmale eine entscheidende Rolle: Erstens die in [WBS01] genannte maximale Annäherungsstufe: Die bereits ausgeführten Testfälle sind nur durch einen Verzweigungsknoten vom nicht ausgeführten Programmcode – dem Testziel – getrennt. Und zweitens die „tangierenden“ Testfälle, die den direkten Dominator des Testziels ausführen. Für den neuen Testfall, der zur Ausführung des Testziels führt, soll einer dieser Testfälle die Vorlage bilden.

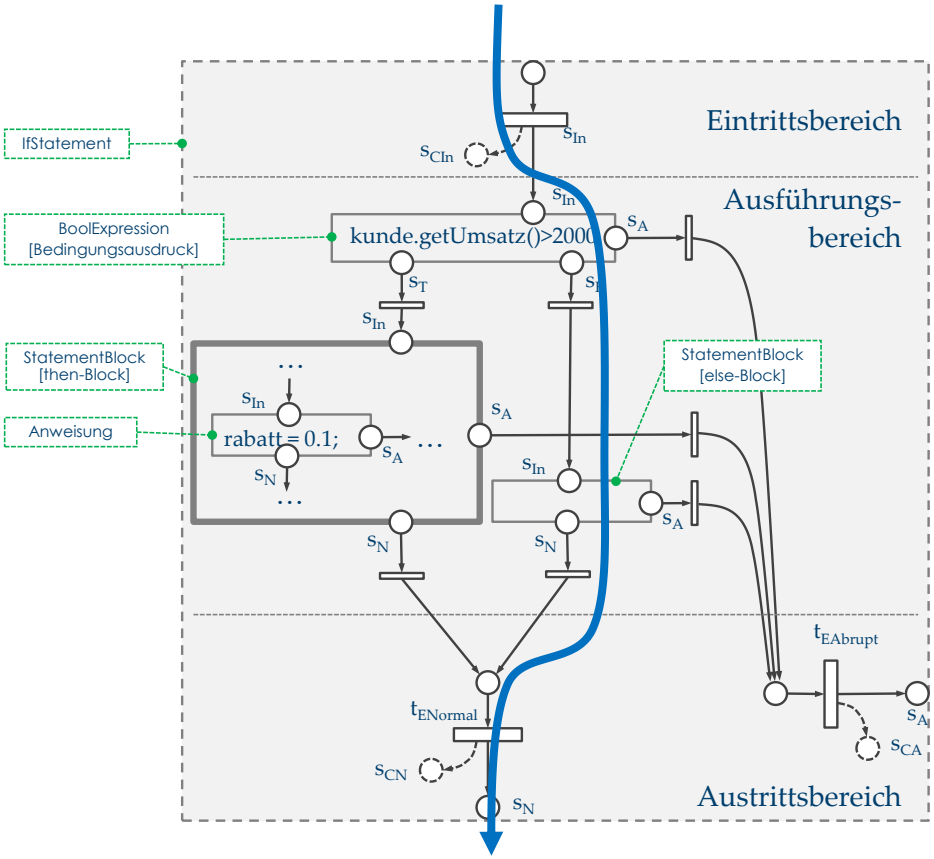


Abbildung 66: Modellnetz des Beispielprogramms zum Testfallentwurf

Def. *Ein Testfall* $t \in T$ **tangiert** *ein Ausführungselement A:*
t *tangiert* $A \Leftrightarrow \neg exe(A, t) \wedge exe(ddom(A), t)$

D. h. ein Testfall t tangiert ein Ausführungselement A, wenn A von t nicht ausgeführt wird, der direkte Dominator von A aber von t ausgeführt wird. Den Ausgangspunkt zur Herleitung neuer Testfälle bilden die Ausführungselemente, die von keinem der Testfälle der Testsuite ausgeführt, aber tangiert werden. Gesucht werden somit Ausführungselemente A mit

$$| \text{testcases}(A) | = 0 \ \wedge \ | \text{testcases}(\text{ddom}(A)) | > 0 \quad (3)$$

Zur Herleitung eines neuen Testfalls hat das Tupel

$$\{ \ A, \ \text{testcases}(\text{ddom}(A)) \ \}$$

für Ausführungselemente nach (3) einen großen praktischen Nutzen: Aus dem direkten Dominator von A – z. B. einer if- oder Schleifenanweisung – kann der logische Ausdruck der Verzweigungsstelle zusammen mit einem der tangierenden Testfälle dem Tester als Anhaltspunkt für einen neuen Testfall vorgeschlagen werden. Als Resultat der Auswer-

tung ergibt sich eine Tabelle, die pro Zeile einen sogenannten Testfall-Hinweis enthält und dem Ausführungselement die tangierenden Testfälle des direkten Dominators, gegenüberstellt. Diese könnte beispielsweise wie Tabelle 12 aussehen. Diese Tabelle wird auch von CodeCover geliefert, ein Beispiel zeigt Abbildung 67 auf Seite 195.

Nicht ausgeführtes Ausführungselement A (das Testziel)	testcases(ddom(A)) (die tangierenden Testfälle)
then-Block Codezeile: 4 kunde.getUmsatz() > 2000	Testfall 4711: Kunde bearbeiten Testfall 4712: Rechnung drucken
Schleifenkörper Codezeile 222 artikelIterator.hasNext()	Testfall 4712: Rechnung drucken
…	

Tabelle 12: Liste an Tesfall-Hinweisen

Def. Seien P ein Programm und T eine Testsuite zum Test von P. Ein **Testfall-Hinweis** *h ist ein Tupel h = { A, D, p, T_T } mit*
dem Testziel A ∈ items(P) : | testcases(A) | = 0,
dem direkten Dominator D ∈ items(P) : D ddom A ∧ | testcases(D) | > 0,
das Prädikat p von D, das die Ausführung von A steuert,
und die A tangierenden Testfällen T_T = testcases(D)

Defintionsgemäß gilt für die tangierenden Testfälle $T_T \subseteq T \wedge | T_T | > 0$.

Um einen neuen Testfall aus dem Testfall-Hinweis zu gewinnen, wird der Tester einen der tangierenden Testfälle wählen und dessen Eingabedaten so variieren, dass der Wert des Prädikats *p* sich verändert und damit das Testziel ausgeführt wird. Im Beispiel von Tabelle 12 würde der Tester z. B. zum Testfall „4711“ Eingabedaten für den Fall „kunde.getUmsatz() > 2000“ suchen. Das Soll-Resultat des neuen Testfalls kann natürlich nicht aus dem Programmcode entnommen werden, sondern muss aus der Spezifikation – oder der Domänenkenntnis des Testers – ermittelt werden. Bei diesen Überlegungen wird davon ausgegangen, dass die Prädikate der Verzweigungsstellen einen Ausdruck enthalten, der vom Tester über die Eingabedaten beeinflusst werden kann. Besteht zwischen dem Prädikat und den praktisch möglichen Eingabedaten kein erkennbarer Zusammenhang, ist der Testfall-Hinweis nutzlos. Ein weiteres Problem sind technisch formulierte oder andere, für den Tester unverständliche Prädikate. Diese Testfall-Hinweise werden entweder unmittelbar als nutzlos aussortiert, oder der Tester muss die Entwickler bei seinen Überlegungen hinzuziehen.

Das beschriebene Vorgehen kann neben den im Beispiel genannten Testfall-Hinweisen zur Ausführung von then- oder else-Blöcken genauso Testfall-Hinweise für alle Ausführungselemente des GBT-Modells ermitteln. Beispielsweise kann der Ausdruck

```
K1: andThen(A: expr, K2: orElse(B: expr, C: expr))
```

betrachtet werden, zu dem der erste Operand *A* mit den Testfällen der Testsuite *T* nur mit Wert false ausgewertet wird. D. h. der Gesamtausdruck *K1* wird von den Testfällen der Testsuite *T* ausgeführt und ergibt false, der Teilausdruck *K2* wird wegen der Kurzschlusssemantik nicht ausgeführt. Es gilt in diesem Fall für den Teilausdruck *K2* Gleichung (3), weil *K2* in *K1* direkt eingebettet ist, d. h. es gilt *K1 ddom K2*. Nach der Definition des Testfall-Hinweises wird *K2* zum Testziel, *K1* zum direkten Dominator und *A* zum Prädikat. Die Testfälle, die *K1* zur Ausführung führen, sind die tangierenden Testfälle.

Da der Tester ausgehend von einem tangierenden Testfall den neuen Testfall entwickelt, spielen die tangierenden Testfälle für den praktischen Nutzen eines Testfall-Hinweises eine zentrale Rolle. Damit diese tangierenden Testfälle zur Verfügung stehen, muss das GBT-Werkzeug in der Lage sein, das GBT-Protokoll testfallgenau (vgl. Kapitel 3.3.4) bereitzustellen. Der GBT ohne die testfallgenaue GBT-Protokollierung kann die Spalte mit den nicht ausgeführten Ausführungselementen (den Testzielen) von Tabelle 12 auch ermitteln, aber nicht die zugehörigen tangierenden Testfälle, auf deren Grundlage der Tester den neuen Testfall entwerfen kann.

8.4 Spezifische und unspezifische tangierende Testfälle

Für die Testfall-Hinweise lassen sich zwei Ausprägungen der tangierenden Testfälle unterscheiden, die im weiteren Verlauf für die Priorisierung eine Rolle spielen:

- Für einen Testfall-Hinweis sind sich die tangierenden Testfälle inhaltlich sehr ähnlich, betreffen also gleiche oder sehr ähnliche Funktionen des Programms. Die tangierenden Testfälle werden dann als *spezifisch* für den Testfall-Hinweis bezeichnet. Speziell wenn im Bezug zur gesamten Testsuite nur wenige Testfälle tangieren, kann in der Praxis von spezifischen tangierenden Testfällen ausgegangen werden. Zudem können hierarchisch strukturierte Testsuiten einen Anhaltspunkt für die Ähnlichkeit von Testfällen liefern. Da die Struktur in vielen Fällen durch die Programmfunktionen bestimmt wird, können Testfälle, die gemeinsam in einer Ordnerstruktur abgelegt sind, als ähnlich gelten.
- Die tangierenden Testfälle haben keine erkennbare inhaltliche Überlappung. Das ist insbesondere dann der Fall, wenn sehr viele oder alle Testfälle der Testsuite tangieren. Diese Testfälle werden dann für den Testfall-Hinweis als *unspezifisch* bezeichnet.

In den Fallstudien hat sich deutlich gezeigt, dass bei spezifischen tangierenden Testfällen der praktische Nutzen eines Testfall-Hinweises größer ist, da der Tester einen konkreten

Anhaltspunkt für den neuen Testfall hat. Er kann so die Spezifikation oder Fachexperten zu einer konkreten Programmfunktion heranziehen. Bei unspezifischem tangierenden Testfällen, insbesondere wenn es sich hier um nahezu die gesamte Testsuite handelt, fehlt dieser konkrete Anhaltspunkt.

8.4.1 Experiment

Im Rahmen eines studentischen Projekts soll die These überprüft werden:

„Testfall-Hinweise mit spezifischen, d. h. sehr wenigen tangierenden Testfällen sind zur Herleitung eines neuen Testfalls besser geeignet als solche mit unspezifischen Testfällen“

Zu diesem Zweck werden für ein bereits von Studierenden mit Black-Box-Testfällen getestetes Open-Source-Programm [CF13] die Testfall-Hinweise entsprechend Tabelle 12 generiert. Aus insgesamt 84 Testfall-Hinweisen zu nicht ausgeführten then-Blöcken (den Testzielen) werden zufällig ausgewählt:

- je vier Testfall-Hinweise mit klar spezifischem tangierenden Testfall (nur ein Testfall tangiert),
- vier Testfall-Hinweise mit unspezifisch tangierenden Testfällen (nahezu alle, etwa 120 Testfälle tangieren) und
- vier Testfall-Hinweise mit etwa 10 tangierenden Testfällen (also weder spezifisch noch unspezifisch; im Folgenden als halbspezifisch bezeichnet)

Zu den Testfall-Hinweisen wird anstelle aller tangierenden Testfälle nur ein zufällig gewählter Testfall angegeben. Zudem werden die Testfall-Hinweise in zufälliger Reihenfolge in einer Liste angeordnet. So ist es für den Leser nicht mehr erkennbar, welcher der Testfall-Hinweise einen spezifischen, unspezifischen oder halbspezifisch tangierenden Testfall enthält. Die Studierenden haben nun die Aufgabe, aus den Testfall-Hinweisen neue Testfälle zu entwickeln und dabei anzugeben, ob der Testfall-Hinweis eine Unterstützung beim Entwurf des neuen Testfalls geleistet hat. Der Grad an Unterstützung soll in einer Intervallskala wie folgt von A bis D angegeben werden:

A. Eingabedaten für den neuen Testfall lassen sich leicht finden, der Hinweis war gut
B. Mit etwas Mühe finde ich die Eingabedaten des neuen Testfalls, der Hinweis hat mir dabei geholfen
C. Mit großer Mühe kann ich einen Testfall finden, der Hinweis war besser als nichts
D. Nein, der Hinweis nützt mir nichts; den neuen Testfall finde ich ohne den Hinweis schneller

Ergebnisse

Die Befragten geben den Grad an Unterstützung für die zwölf gelieferten Testfall-Hinweise wie folgt an:

	A.	B.	C.	D.
Spezifische Testfall-Hinweise	3	1		
Halbspezifische Testfall-Hinweise	4			
Unspezifische Testfall-Hinweise			3	

Ein Testfall-Hinweis mit unspezifischem tangierenden Testfall kann von den Studierenden nicht ausgewertet werden, daher gibt es nur drei Angaben für diese Gruppe von Testfall-Hinweisen, die alle nur mit „C“ bewertet werden. Damit zeigt sich hier deutlich, dass die Testfall-Hinweise mit unspezifisch tangierenden Testfällen erkennbar schlechter bewertet werden als die spezifischen und die halbspezifischen Testfall-Hinweise.

Die Teilnehmer an dem Experiment sind drei Studierende der Softwaretechnik (7. Semester), die die Software selbst geschrieben haben und sowohl den Programmcode als auch die Anwendungsdomäne gut kennen.

8.5 Priorisierungen von Testfall-Hinweisen

Für Programme der industriellen Praxis ist mit sehr vielen Testfall-Hinweisen durch die beschriebenen Auswertungen zu rechnen. Bei der Untersuchung von Ebert [Ebe11] ergeben sich beispielsweise für ein mittelgroßes Programm der Industrie (50 kLOC) 1.700 Testfall-Hinweise. Ähnliche Werte sind auch in [Schm11] angegeben. Nur bei sehr kleinen Programmen (< 5 kLOC) mit sehr hoher Überdeckung (> 80 %) bleibt die Menge der Testfall-Hinweise in einem praktikablen Rahmen. Obwohl prinzipiell jeder der Hinweise geeignet ist, als Grundlage für einen neuen Testfall genutzt zu werden, zeigen sich in der Praxis erhebliche Aufwands- und Effizienzunterschiede. Damit kommt einer Priorisierung der Hinweise eine große praktische Bedeutung zu. Vergleichbar argumentieren auch die Autoren von [Fi11]. Sie weisen darauf hin, dass es in vielen Fällen nicht wirtschaftlich ist, eine 100-%-GBT-Überdeckung anzustreben. Die Autoren zweifeln auch an, dass es sinnvoll ist, einen bestimmten Überdeckungswert (z. B. 70 %) anzustreben, weil es dadurch keine Sicherheit gibt, dass mit den so gewonnenen Testfällen die Teile der Anwendung angesprochen werden, die die höchste Fehlerdichte haben. Sie empfehlen daher eine Fokussierung auf eine Erhöhung der Überdeckung bei Programmcode, der eine überdurchschnittliche Fehlerdichte aufweist. Hierzu zählen die Autoren Programmcode, der häufiger Änderungen unterworfen war, und entwickeln daraus eine Reihe an *„Change-Based Coverage Criteria“*. Neben der Änderungshistorie lassen sich aber aus der Literatur auch einige weitere Indikatoren entnehmen, die die Bestimmung einer „Erfolgsprognose“ für einen Testfall-Hinweis ermöglichen. Testfall-Hinweise mit hoher Erfolgsprognose sollen dann priorisiert zur Entwicklung neuer Testfälle verwendet werden. Diese Indikatoren zur Erfolgsprognose werden im Folgenden vorgestellt. Einen Ausgangspunkt für die Überlegungen zur Priorisierung bildet Abbildung 66 auf Seite 187. Zwei zentrale Elemente sind damit an einem Testfall-Hinweis beteiligt:

- das Testziel (das nicht ausgeführte Ausführungselement) und
- die Testfälle, die das Testziel tangieren

Dabei bildet das Testziel den Bezug zum Programmcode, die tangierenden Testfälle bilden den Bezug zu den Anforderungen und damit zu den Programmfunktionen. Auch Aufwandswerte zur Testausführung können aus den tangierenden Testfällen herangezogen werden.

Ein Testfall-Hinweis wird priorisiert, d. h. bevorzugt zum Entwurf eines neuen Testfalls verwendet,

- wenn diese Testfallherleitung gute Erfolgsaussichten hat, d. h. wenn aus dem Testziel und den tangierenden Testfällen mit möglichst geringem Aufwand Eingabedaten ermittelt werden können, die dazu führen, dass das Testziel überdeckt wird,
- wenn im Programmcode des Testziels mehr Fehler als bei anderen Testzielen vermutet werden,
- wenn die dort verborgenen Fehler vermutlich schwerwiegend sind und
- wenn die Ausführung des neuen Testfalls möglichst geringen Aufwand verursacht.

Ausgehend von diesen prinzipiellen qualitativen Beschreibungen werden im Folgenden konkrete qualitative Kriterien zur Priorisierung der Testfall-Hinweise [Schm11, Ebe11] beschrieben.

8.5.1 Erfolgsaussichten der Testfallherleitung

Der Tester versucht, aus dem Testfall-Hinweis einen neuen Testfall zu herzuleiten, indem er die tangierenden Testfälle als Ausgangspunkt nutzt und die Eingabedaten so variiert, dass das Prädikat der Verzweigungsstelle den gewünschten Wert annimmt und so das Testziel zur Ausführung kommt. Auf die Erfolgsaussichten, dass ihm dies in möglichst kurzer Zeit gelingt, haben nach [Ebe11] die folgenden Faktoren Einfluss:

- Spezifische tangierende Testfälle: Hinweise, die auf spezifischen tangierenden Testfällen basieren, bieten dem Tester einen konkreteren Anhaltspunkt für den neuen Testfall als Hinweise mit unspezifischen Testfällen.
- Interpretierbarer Verzweigungsausdruck: Die Ausdrücke der Prädikate an den Verzweigungsstellen lassen sich für den Tester unterschiedlich schwer interpretieren. So waren in der Untersuchung von Ebert [Ebe11] viele Ausdrücke der Form „x == null“ oder „x != null“. Rückschlüsse auf zu verändernde Eingabedaten sind bei diesen Ausdrücken nach Ebert überdurchschnittlich schwierig.

8.5.2 Fehlerprognose

Der Einfluss des Testziels auf die Priorisierung eines Testfall-Hinweises entsteht im Wesentlichen aus der Fehlerprognose des betreffenden Programmcodes. In der Literatur finden sich auch viele Untersuchungen zu Korrelationen zwischen Fehlerdichte und Programmcode-Metriken oder anderen Eigenschaften des untersuchten Programms [RAF04, HP04, OWB05, NBZ06, ZPZ07, Hol09]. Eine ausführliche Zusammenstellung hierzu findet

sich auch bei Ebert [Ebe11]. Die Fehlerprognose des Testziel-Programmcodes kann demnach durch folgende Faktoren bewertet werden:

- Programmcode-Fehlerprognose: Es sollen solche Testziele bevorzugt werden, die in Modulen mit hoher Fehlerdichte liegen. Als sehr ergiebig erweisen sich hier auch Arbeiten von Hovemeyer und Pugh, die in [HP04] den Begriff des *bug pattern* prägen und damit Muster benennen, die empirisch gesichert eine höhere Fehlerwahrscheinlichkeit haben.
- Vorausgehende Prüfungen des Testziels: Testziele des Systemtests, die im Unit-Test ebenfalls nicht überdeckt sind, enthalten nach [MND09] statistisch mehr Fehler als solche, die im Unit-Test ausgeführt und geprüft werden. Auch Gittens et al. empfehlen in [Gi06], solchen Programmcode im GBT mit geringerer Priorität zu berücksichtigen, der entweder bereits im Unit-Test oder in einem Review geprüft wurde.
- Die Größe (z. B. Anzahl Anweisungen) des Testziels: Ein Anweisungsblock mit zwei Anweisungen wird bei gleicher Fehlerdichte weniger Fehler enthalten als ein Anweisungsblock mit 50 Anweisungen. In [Li05] wird hierzu jedem Anweisungsblock ein Gewicht zugewiesen, das neben der Anzahl an Anweisungen auch Funktionsaufrufe und die in den Funktionen enthaltene Anzahl an Anweisungen bewertet. Der Autor weist auch in einem Experiment nach, dass durch diese Priorisierung die Überdeckung im GBT deutlich schneller ansteigt, als wenn keine Priorisierung erfolgt.
- Aus der Änderungshäufigkeit von Programmcode, die aus dem Versionskontrollsystem ermittelt werden kann, kann nach [Fi11] auf die Fehlerdichte geschlossen werden. Die Autoren legen hierbei die Annahme zugrunde, dass viele Programmänderungen mit einer höheren Fehlerdichte korrelieren.

Allerdings finden sich in der Literatur auch Arbeiten, die die Präzision dieser Fehlerprognosen anzweifeln. So stellen Nagappan, Ball und Zeller bei ihren Untersuchungen zur Fehlerprognose auf Grundlage der Analyse von Programmcode in [NBZ06] zusammenfassend fest: *„Predictors are accurate only when obtained from the same or similar projects“*. Einen pragmatischen Ansatz zur Bestimmung einer Fehlerprognose für Programmcode, um daraus Priorisierung beim GBT abzuleiten, beschreiben Gittens et al. in [Gi06]. Die Entwickler vergeben ohne strenge Systematik ein „High“ und „Low“ für hoch- und geringpriorisierten Code. „High“ bedeutet dabei eine hohe, d. h. überdurchschnittliche Fehlerprognose, „Low“ eine geringe. Die Autoren nennen im Wesentlichen die oben genannten Kriterien, die die Entwickler als Anhaltspunkt zur Angabe dieser Priorisierung verwenden. Im Werkzeug CodeCover wird dieser Ansatz auch aufgegriffen, und vom Tester können auch unabhängig von den genannten automatisiert erhebbaren Kriterien für einzelne Programmcode-Dateien die Fehlerprognose „manuell“ vergleichbar der Empfehlung von [Gi06] angegeben werden.

8.5.3 Fehlerschwere

Testfall-Hinweise werden dann bevorzugt, wenn sie zu einem neuen Testfall führen, der besonders kritische Funktionen des Programms prüft. Kriterien, die zur Bewertung dieser Kritikalität führen, liefert Amland in [Am00]. Da die tangierenden Testfälle beim Ent-

wurf des neuen Testfalls als Vorlage dienen, können Eigenschaften dieser tangierenden Testfälle auf den neuen Testfall übertragen werden. Unter der Annahme, dass der neue Testfall inhaltlich eine Variante des tangierenden Testfalls bildet, sollen tangierende Testfälle mit hoher Priorität bevorzugt zur Entwicklung neuer Testfälle herangezogen werden. Man nimmt dabei an, dass die Variante eines wichtigen Testfalls ebenfalls auch wichtig, die Variante eines unwichtigen Testfalls auch eher unwichtig wäre.

8.5.4 Ausführungsaufwand

Die Ausführungsaufwände der tangierenden Testfälle können als Anhaltspunkt für den Aufwand des neu zu entwickelnden Testfalls genommen werden. Unter der Annahme, dass zwei Testfall-Hinweise zu zwei gleich effektiven neuen Testfällen führen, wird der tangierende Testfall bevorzugt als Ausgangspunkt verwendet werden, der geringeren Ausführungsaufwand verursacht.

8.5.5 Zusammenfassung der Priorisierung

Da alle genannten Kriterien zur Priorisierung ganz verschiedene Aspekte betrachten, deren Zusammenwirken empirisch nicht geprüft ist, sind generelle Empfehlungen zur Priorisierung der Testfall-Hinweise nicht möglich. Solange also keine gesicherten empirischen Erkenntnisse zum Zusammenwirken der genannten Einflussfaktoren auf die Priorität eines Testfall-Hinweises vorliegen, dient die qualitative Auflistung dem Tester als Anhaltspunkt zur Priorisierung. Dementsprechend ist im GBT-Werkzeug CodeCover zur Priorisierung der Testfall-Hinweise auch eine eigene Konfigurierungs- und Gewichtungsmöglichkeit aus den genannten Faktoren für den Tester vorgesehen.

Ebert fasst in [Ebe11] aus seinen Untersuchungen an einem Industrieprojekt die Priorisierung der Testfall-Hinweise wie folgt zusammen: Ein Testfall-Hinweis wird dann bevorzugt, wenn

1. die tangierenden Testfälle spezifisch sind,
2. der Bedingungsausdruck nicht „!= null“ oder „== null“ enthält,
   das Testziel
3. ein größerer Code-Block ist,
4. bei dem eine hohe Fehlerdichte vermutet wird,
   die tangierenden Testfälle
5. als besonders fehlersensitiv gelten,
6. besonders kritische Funktionen prüfen
7. und einen geringen Ausführungsaufwand verursachen.

In den bereits untersuchten Projekten [Schm11, Ebe11] waren auch bei Weitem nicht alle der genannten Kriterien zur Priorisierung verfügbar. Auch für viele andere Projekte wäre es unrealistisch, alle Kriterien einzufordern, um sie für die Priorisierung zu nutzen. Zu-

mindest aber die Punkte 1. - 3. sind leicht zu erheben und werden z. B. vom Werkzeug CodeCover geliefert.

8.6 Implementierung in CodeCover

Die beschriebenen Funktionen zur Generierung der Testfall-Hinweise sind in CodeCover prototypisch im Rahmen der Diplomarbeit von Ebert [Ebe11] implementiert worden. CodeCover liefert in der Sicht „RecommendationsView“ eine Liste mit Testfall-Hinweisen, die im Wesentlichen der von Tabelle 12 auf Seite 188 entspricht. Abbildung 67 zeigt diese Liste für ein kleines Beispielprogramm, das auch den bereits exemplarisch betrachteten Programmausschnitt von Abbildung 65 auf Seite 185 enthält. Jede Zeile dieser Liste ist ein Testfall-Hinweis. Die einzelnen Spalten der Liste sind in Tabelle 13 beschrieben.

Sobald für einen Prüfling ein GBT mit dem Werkzeug CodeCover durchgeführt wurde, kann diese Generierung der Testfall-Hinweise automatisch erfolgen; für den Tester entsteht kein zusätzlicher Aufwand. Der Tester kann nun die Testfall-Hinweise in CodeCover bearbeiten oder aber über die Funktion „CSV-Export“ in ein externes Tabellen-System übertragen, in denen z. T. auch sehr mächtige Filter- und Sortierfunktionen zur Verfügung stehen.

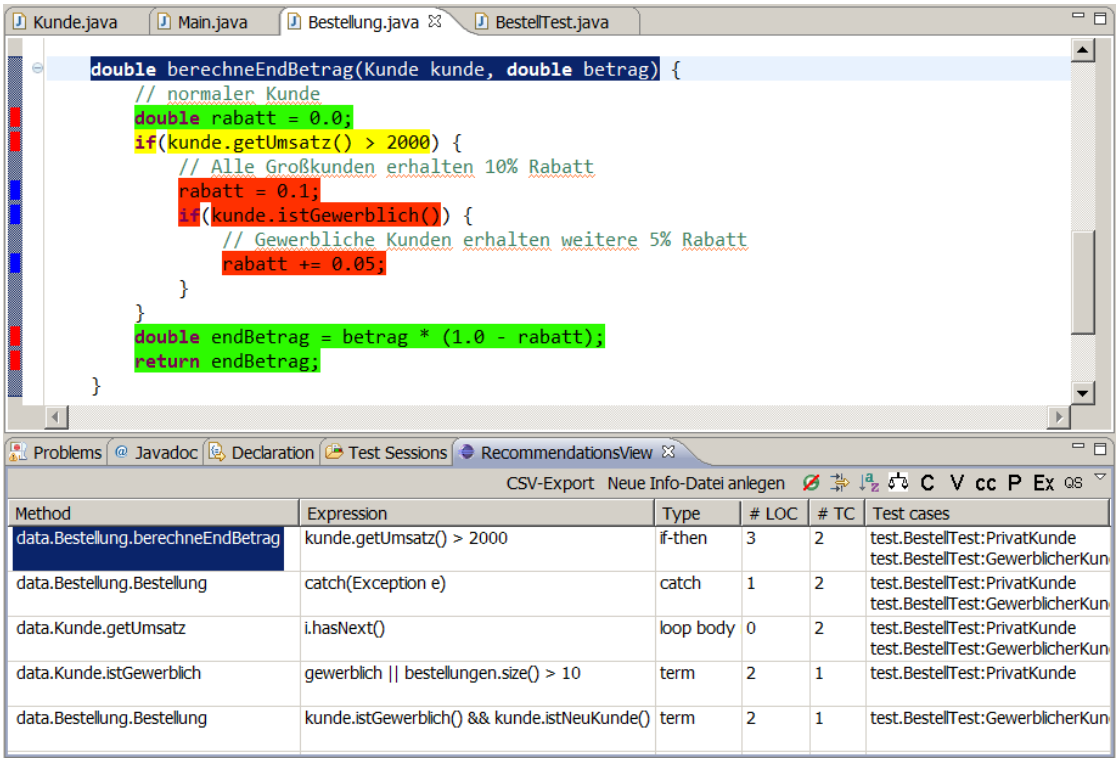


Abbildung 67: Liste der Testfall-Hinweise in CodeCover

Über eine Filterfunktion können in CodeCover die Testfall-Hinweise anhand des Typs des Testziels gefiltert werden. In einer ersten Betrachtung können dann z. B. nur die Testfall-Hinwiese mit einem then-Block als Testziel betrachtet werden. Insbesondere bei grö-

ßeren Projekten ist dies eine notwendige Funktion, um zunächst weniger „attraktive“ Testfall-Hinweise herauszufiltern wie z. B. solche, die auf unwirksame Terme hinweisen.

Tabellenspalte	Funktion
Method	Gibt das Java-Paket, die Klasse und die Methode des Testziels an. Mit dem Anklicken dieser Zelle wird der betreffende Programmcode im Editor angezeigt. In Abbildung 67 wurde der oberste Testfallhinweis angeklickt (berechneEndBetrag), und entsprechend wurde der Quelltext dieser Methode geöffnet. Die Anzeige des Quelltexts hat praktische Vorteile, weil dort die Überdeckung visualisiert wird.
Expression	Der Ausdruck, der variiert werden muss, um zur Ausführung des Testziels zu kommen.
Type	Gibt den Ausführungselement-Typ des Testziel an. Also beispielsweise then- oder else-Block, Schleifenkörper oder Term.
LOC	Die Anzahl der Anweisungen des Testziels. Damit erhält der Tester eine ungefähre Größenangabe für das Testziel. Dem Ziel der Erhöhung der Überdeckung folgend, sind große Testziele attraktiver.
TC	Die Anzahl an tangierenden Testfällen.
Test cases	Die tangierenden Testfälle.

Tabelle 13: Tabellenspalten der „RecommandationsView“

Abbildung 68 zeigt diesen Filter-Dialog, dessen Einstellung die Liste der Testfall-Hinweise auf solche Testziele filtert, die Anweisungsblöcke betreffen. Nicht vollständige Schleifenüberdeckung oder Termüberdeckung spielt dann in dieser Betrachtung keine Rolle. Wenn die Testfall-Hinweise zu den „attraktiveren“ Testzielen abgearbeitet sind, kann der Filter wieder entfernt werden, und Testziele, die sich durch unvollständige Schleifen- oder Termüberdeckung ergeben, können bearbeitet werden.

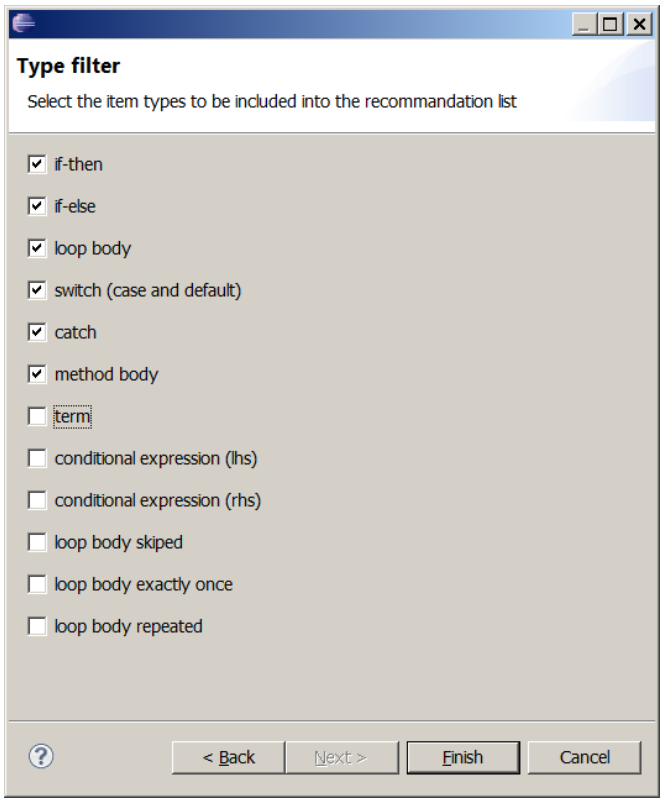


Abbildung 68: Filterfunktion auf die Typen der Ausführungselemente

Neben dieser Filterung lassen sich in CodeCover die Testfall-Hinweise auch nach den Kriterien sortieren, die in Abschnitt 8.5 angegeben sind. Falls beispielsweise das Versionskontrollsystem Apache-Subversion für den Programmcode des Prüflings genutzt wird, kann automatisch von CodeCover für jede Datei die Änderungshäufigkeit ausgewertet werden. In CodeCover werden dann für diese Eigenschaft „Punktwerte“ vergeben, die mit den anderen Eigenschaften wie Codegröße des Testziels oder Anzahl der tangierenden Testfällen zu einer „Gesamtpunkzahl“ kombiniert werden. Auch kann eine Defektprognose der Testziele mit einbezogen werden, die über das Werkzeug Findbugs [HP04] ermittelt wird. Ebert hat diese Auswertungs- und Sortierfunktionen in einem Projekt der Industrie eingesetzt und evaluiert. Die Resultate sind ausführlich in [Ebe11] beschrieben. Weitere empirische Untersuchungen zu dieser Priorisierung der Testfall-Hinweise sind aber wünschenswert.

9 Zusammenfassung und Ausblick

In diesem Kapitel wird zunächst die Arbeit mit ihren wichtigsten Ergebnissen zusammengefasst. Darauf folgt ein Ausblick, der mögliche zukünftige Weiterentwicklungen des Modells sowie des Werkzeugs vorstellt. Am Ende folgen Schlussbemerkungen.

9.1 Zusammenfassung

Den Kern dieser Arbeit bildet das GBT-Modell mit der GBT-Modellsprache RPR, die die GBT-relevanten Programmkonstrukte abbildet. Der besondere Vorteil einer GBT-Modellsprache lässt sich wie folgt zusammenfassen:

- Es ist klar ersichtlich, welche Programmkonstrukte im GBT-Modell abgebildet werden und welche nicht.
- Für die Transformation der realen Programme in das GBT-Modell bilden Gegenüberstellungen einzelner Produktionsregeln der Grammatiken eine sehr einfache und anschauliche Transformationsvorschrift. Exemplarisch wird diese Gegenüberstellung für die Transformation der Programmiersprache Java nach RPR in Abschnitt 5.13.4 ab Seite 120 ausführlich dargestellt.

Neben den Kontrollstrukturen wie Entscheidung oder Schleife modelliert die RPR auch die GBT-relevanten Ausdrücke. Die Modellsprache definiert auch die Abstraktion der realen Programmkonstrukte zu den sogenannten Ausführungselementen. Die Ausführungselemente Anweisungen, Anweisungsblöcke und boolesche Ausdrücke bilden im Folgenden die Grundlage für die Definitionen der Überdeckungsmetriken. Zu den Merkmalen der realen Programme werden in der GBT-Modellsprache zudem die wichtigen abundanten Bezeichner der Ausführungselemente ergänzt. Dieses für die GBT-Auswertungen erforderliche Attribut des Modells wird zwar im CFG in Form der Knotenbezeichner ebenfalls genutzt, eine systematische Herleitung und Verankerung fehlt dort aber.

Im zweiten zentralen Aspekt des GBT-Modells wird die Ausführungssemantik der Ausführungselemente als Petri-Netze, sogenannte Modellnetze, definiert. Primitive Ausführungselemente lassen sich direkt als Modellnetze beschreiben, die Verbund-Ausführungselemente wie z. B. die Entscheidungsanweisung oder der zusammengesetzte

boolesche Ausdruck werden durch Komposition dieser Teilnetze beschrieben. Die Modellsprache RPR liefert hierzu die Kompositionsregeln. Die Modellnetze nutzen bei dieser Komposition und Dekomposition die Netztransformation der Vergröberung und der Verfeinerung. Die Teilnetze der Ausführungselemente sind hierzu alle stellenberandet und markentreu; so lassen sich auch auf vergröberte Netze die bekannten Netzanalysen wie z. B. die Erreichbarkeitsanalyse anwenden. Da die Modellnetze fast immer S-Netze sind, ist der Nachweis der Markentreue einfach. In den anderen Fällen erfolgt dieser Nachweis durch die Erreichbarkeitsanalyse. Der Vorteil der Modellnetze gegenüber dem CFG lässt sich wie folgt zusammenfassen:

- Für die Netzanalyse können die vorhandene Theorie sowie die Werkzeuge genutzt werden. Für die komplizierteren Teilnetze wurde in dieser Arbeit ein Werkzeug für die Erreichbarkeitsanalyse genutzt.
- Die Modellnetze sind für alle primitiven Ausführungselemente eindeutig definiert und folgen für die Verbundausführungselemente einer über die Modellsprache präzise definierten Kompositionsstruktur. Unklarheiten wie beim CFG, ob z. B. am Ende einer Entscheidung ein zusammenführender Knoten ergänzt wird oder nicht, gibt es bei den Modellnetzen nicht.
- Auf Grundlage der Modellnetze lassen sich die zur Metrikenbestimmung wichtigen Ausführungszähler präzise platzieren.
- In den Modellnetzen ist die Ausführungssemantik für abruptes Beenden in einer Form berücksichtigt, die der Ausführungssemantik der gängigen Programmiersprachen entspricht.
- Das Modell umfasst sowohl den Kontrollfluss als auch die Ausführungssemantik der Ausdrücke.
- Die Netze sind auch für nebenläufige Ausführung nutzbar.

Durch die Platzierung der Zählerstellen im Eintritts- und Austrittsbereich eines Ausführungselements A (einer Anweisung oder eines Anweisungsblocks) lässt sich die Ausführung für einen Testfall bestimmen:

$exe(A, t) \Leftrightarrow A$ wird von t mindestens einmal ausgeführt

Und für einen booleschen Ausdruck C lässt sich die Ausführung mit den Resultaten false und true bestimmen:

$exeF(C, t) \Leftrightarrow C$ wird bei der Ausführung mit t mindestens einmal zu false
$exeT(C, t) \Leftrightarrow C$ wird bei der Ausführung mit t mindestens einmal zu true

Hiervon ausgehend sind alle in dieser Arbeit beschriebenen Überdeckungsmetriken definiert. So wird z. B. die Anweisungsüberdeckung als der Anteil der ausgeführten Anweisungen definiert:

$$A \in exeStmts\ (P, T) \Leftrightarrow A \in stmts(P) \wedge \exists t \in T : exe(A, t)$$

$$\text{stmtCov}(P, T) = \frac{| \text{exeStmts}(P, T) |}{| \text{stmts}(P) |}$$

Die Definition der Anweisungen wurde bereits in der Modellsprache vorgenommen und ist ebenso einfach und präzise.

9.1.1 Theoretische Validierung der Überdeckungsmetriken

Mit den in dieser Arbeit aufgestellten Plausibilitätsregeln für GBT-Überdeckungsmetriken wird das Ziel einer theoretischen Validierung der Metriken hinsichtlich Plausibilität, Differenziertheit und Vergleichbarkeit verfolgt. Die folgenden fünf Regeln sollen für GBT-Überdeckungsmetriken gelten:

- **Plausibilitätsregel 1:** Null-Überdeckung bei leerer Testsuite. Die leere Testsuite führt zu keiner Überdeckung.
- **Plausibilitätsregel 2:** Normierung. Die Überdeckungswerte sollen immer zwischen null und eins liegen.
- **Plausibilitätsregel 3:** Nicht fallende Monotonie. Eine Teilmenge einer Testsuite kann nie eine höhere Überdeckung erzielen als die gesamte Testsuite.
- **Plausibilitätsregel 4:** Erfüllbarkeit. Es muss ein Programm mit einer Testsuite geben, sodass eine vollständige Überdeckung erzielt wird.
- **Plausibilitätsregel 5:** Wachsende Monotonie. Jede Teilmenge einer redundanzfreien Testsuite soll wieder redundanzfrei sein.

Die in dieser Arbeit definierten Überdeckungsmetriken werden alle anhand dieser Plausibilitätsregeln überprüft. Es gibt aber auch Überdeckungsmetriken in der Literatur, für die nicht alle Regeln gelten. So ist z. B. die sogenannte Pfadüberdeckung bei praxistypischen Programmen nicht erfüllbar, d. h. Plausibilitätsregel 4 wird dort nicht erfüllt.

9.1.2 Anforderungen an ein Glass-Box-Test-Werkzeug

Aus den Modell- und Metrikendefinitionen ergeben sich drei zentrale Anforderungen an ein GBT-Werkzeug:

- Programmcode-Abstraktion: Das Werkzeug bildet aus dem Originalprogramm eine Abstraktion, die entsprechend der RPR-Definition die Ausführungselemente Anweisung, Anweisungsblock und boolescher Ausdruck enthält.
- Testfallgenaues Ausführungsprotokoll: Für ein Ausführungselement A liefert das Werkzeug für eine Programmausführung mit einem Testfall t die Ausführung exe(A, t).
- Metrikenbestimmung: Das GBT-Werkzeug bestimmt die Überdeckungsmetriken entsprechend den Definitionen von Kapitel 6.

Mit dem Open-Source-Werkzeug CodeCover wurde eine Implementierung des in dieser Arbeit entwickelten GBT-Referenzmodells präsentiert.

Ergänzend zur präzisen Metrikenbestimmung, die auf Grundlage der Definitionen dieser Arbeit erfolgt, sind aber noch weitere Auswertungen durch das testfallgenaue

GBT-Protokoll möglich. Für das Werkzeug CodeCover wurde die Generierung der sogenannten Testfall-Hinweise beschrieben. Diese Auswertung leistet dem Tester eine methodische Hilfestellung beim Entwurf neuer Testfälle, die zu einer systematischen Erhöhung der Überdeckung führen.

9.2 Ausblick

Die in dieser Arbeit präsentierte Modellsprache ist keinesfalls abgeschlossen, sondern kann um weitere GBT-relevante Aspekte erweitert werden. Eine Erweiterungsmöglichkeit wurde bereits in Abschnitt 4.2.7 für die in objektorientierten Programmen genutzte dynamische Bindung aufgezeigt. Solche Methodenaufrufe weisen ähnlich dem bedingten Ausdruck die Charakteristik einer Fallunterscheidung auf. Die bereits skizzierten Probleme, die bei der Nutzung einer Bibliothek auftreten, müssen dabei natürlich berücksichtigt werden.

Weitere Verbesserungsmöglichkeiten liegen im Zusammenspiel des GBT-Werkzeugs mit einem Werkzeug zur Testsuite-Verwaltung. Als „motivierendes Moment“ könnte in einem solchen Black-Box-Testwerkzeug ein Fortschrittsbalken integriert werden, der die aktuell erzielte GBT-Überdeckung anzeigt. Verbessert werden könnte auch die Rückmeldung an den Tester, ob eine angestrebte Überdeckung mit den gewählten Eingabedaten tatsächlich erzielt wurde. Es müsste also beim Testfall im Testsuite-Werkzeug das Überdeckungsziel hinterlegt werden, und dem Tester müsste bei der Testausführung unmittelbar signalisiert werden, ob dieses Ziel erreicht wurde. Mit den heutigen Werkzeugen – auch mit CodeCover – muss der Tester recht umständlich über den GBT-Bericht diese Zielerreichung überprüfen.

Um im Testsuite-Werkzeug Testfälle besser priorisieren zu können, wäre auch die Auskunft für den Tester hilfreich, welchen Anteil an der gesamten GBT-Überdeckung ein Testfall erzielt. Mit der in CodeCover eingesetzten Technik zum testfallgenauen GBT-Protokoll könnte diese Information automatisch bereitgestellt werden.

Weitere Arbeiten im Bereich der Testfall-Hinweise von Kapitel 8 sind ebenfalls wünschenswert. Zwar wurden einzelne empirische Arbeiten bereits durchgeführt, eine umfangreiche Evaluierung fehlt jedoch. Gerade beim praktischen Nutzen der Priorisierungen fehlen Erkenntnisse aus empirischen Untersuchungen. Da diese Priorisierungen vorwiegend bei großen Projekten und bei geringer GBT-Überdeckung sinnvoll eingesetzt werden können, ist eine solche Evaluierung auf Grundlage studentischer Projekte kaum möglich.

Das Werkzeug CodeCover wird zwar schon verbreitet in Industrie und Lehre genutzt und hat einen weit höheren Reifegrad, als dass man es als „prototypische“ Implementierung bezeichnen müsste. Es gibt aber dennoch laufend Anfragen der Nutzer nach Verbesserungen oder neuen Funktionen. Es geht hier überwiegend um praktische Verbesserungen wie z. B. die verbesserte Integrationsmöglichkeit in sogenannte Continuous Integration Server oder die Optimierung der Instrumentierung, um den „Application blowup“ zu begrenzen. Obwohl CodeCover Open-Source ist und die Nutzer oft selbst die Möglichkeit (und die Expertise) hätten, die Änderungen vorzunehmen, wird von dieser Möglichkeit bislang kaum Gebrauch gemacht. Ein professioneller Support lässt sich aber in dem Rah-

men, wie CodeCover bislang entwickelt wurde, nicht leisten. Um CodeCover längerfristig am Markt eine Chance zu geben, muss dieses Problem gelöst werden.

9.3 Schlussbemerkung

Testen entwickelt sich in den letzten Jahren von einer „Randerscheinung“ zu einem zentralen und als erfolgskritisch wahrgenommenen Projektbestandteil. Dieser gestiegenen Bedeutung folgend müssen auch die vormals „intuitiv“ und „hemdsärmelig“ durchgeführten Tests systematisiert werden. Für den GBT wurde mit dieser Arbeit die Systematisierung der zugrundeliegenden Modelle, der Metrikendefinitionen und des Testfall-Entwurfs vorangetrieben. Gerade bei der Metrikendefinition wäre es wünschenswert, wenn die anderen Werkzeughersteller den Vorschlägen dieser Arbeit in Zukunft folgten.

Diese Arbeit hat sich praktisch ohne Ausnahme mit dem Glass-Box-Test beschäftigt und dabei alle anderen Test- und Prüftechniken weitgehend außer Acht gelassen. Um hier keinen falschen Eindruck entstehen zu lassen, sei nochmals darauf hingewiesen, dass Software-Prüfung und insbesondere der Programmtest eine Kombination aus vielen Techniken erfordert. Es lohnt sich aber, den GBT hierbei angemessen zu berücksichtigen.

Literaturverzeichnis

[Ag99] Agrawal. H., 1999, Efficient coverage testing using global dominator graphs. In Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '99). ACM, New York, NY, USA, 11-20

[Am00] Amland, S., 2000, Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study. J. Syst. Softw. 53, 3 (September 2000), 287-295

[An06] Andrews, J. H., et al., 2006, Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Trans. Softw. Eng. 32, 8 (August 2006), 608-624

[ASU99] Aho, A., Senthi, R., Ullmann, J., D., 1999, Compilerbau, Teil 1 und Teil 2, zweite Auflage, Oldenbourg, München, Wien

[AOH03] Ammann, P., Offutt, A.J., Hong, H.S., 2003, Coverage Criteria for Logical Expressions. In Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03). IEEE Computer Society, Washington, DC, USA, 99

[Ba90] Baumgarten, B., 1990, Petri-Netze Grundlagen und Anwendung, BI-Wissenschaftsverlag

[Be03] Beck, K., 2003, Test Driven Development: By Example. Addison-Wesley

[Bei90] Beizer, B., 1990, Software Testing Techniques, New York, Van Nostrand Reinhold

[BWK07] Berner, S., Weber, R., Keller, R. K., 2007, Enhancing Software Testing by Judicious Use of Code Coverage Information. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 612-620

[Bin99] Binder, R. V., 1999, Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[BL94] Ball, T., Larus, J. R., 1994, Optimally profiling and tracing programs. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92). ACM, New York, NY, USA, 59-70

[BL96] Ball, T., Larus, J. R., 1996, Efficient path profiling. In Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO 29). IEEE Computer Society, Washington, DC, USA, 46-57

[Boe79] Boehm, B.W., 1979, Guidelines for verifying and validating software requirements and design specifications. In Proceedings Euro IFIP'79, ed. P. A. Samet. Amsterdam: North-Holland. 711-719

[CC08a] Hanussek, R. et al., 2008, CodeCover - Glass Box Testing Tool, Design, Student Project "OST-WeST", University of Stuttgart, Version 1.2, http://www.codecover.org/development/Design.pdf (letzter Zugriff am 30.09.2013)

[CC08b] Starzmann, M. et al., 2008, CodeCover - Glass Box Testing Tool, Specification, Student Project "OST-WeST", University of Stuttgart, Version 1.1, http://www.codecover.org/development/Specification.pdf (letzter Zugriff am 30.09.2013)

[CC13] CodeCover Webseite, http://www.codecover.org (letzter Zugriff am 26.07.2014)

[CF13] Conference Information System Stuttgart (ConfISS), http://sourceforge.net/projects/confiss (letzter Zugriff am 26.07.2014)

[CK11] Carver, J.C., Kraft, N.A., 2011, Evaluating the testing ability of senior-level computer science students. In Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET '11). IEEE Computer Society, Washington, DC, USA, 169-178

[CM94] Chilenski, J., Miller, S., 1994, Applicability of modified condition/decision coverage to software testing, Software Engineering Journal. Software Engineering Journal, 9, 5, (1994) 191-200

[CL05] Cai, X. Lyu, M.R., 2005, The effect of code coverage on fault detection under different testing profiles. SIGSOFT Softw. Eng. Notes 30, 4 (Mai 2005), 1-7

[Clover] Java and Groovy code coverage , https://www.atlassian.com/software/clover (letzter Zugriff am 26.07.2014)

[Co11] Cornett, S., 2011, Code Coverage Analysis, Bullseye Testing Technology, http://www.bullseye.com/coverage.html (letzter Zugriff am 30.09.2013)

[CodePro] CodePro, Google Java Developer Tools, Code Coverage, https://developers.google.com/java-dev-tools/codepro/doc/features/codecoverage/code_coverage, (letzter Zugriff am 26.07.2014)

[DDH72] Dahl, O.-J., Dijkstra, E. W., Hoare, C. A. R., 1972, Structured Programming, Academic Press

[DE95] Desel, J., Esparza, J., 1995, Free Choice Petri Nets, volume 40 of Cambridge Tracts in TheoreticalComputer Science. Cambridge University Press

[DT09] Dern, C., Tan, R., 2009, Synchronization Coverage: Code Coverage for Concurrency, MSDN Magazine (September 2009)

[DL00] Dupuy, A., Leveson, N., 2000, An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In Proceedings of the Digital Aviations Systems Conference DASC (Oktober 2000)

[Ebe11] Ebert, R., 2011, Priorisierung von Testfall-Vorschlägen. Diplomarbeit, Institut für Softwaretechnologie, Universität Stuttgart, Online-Version: ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3147/DIP-3147.pdf

[EBI06] Ellims, M., Bridges, J., Ince, D., C., 2006, The Economics of Unit Testing. Empirical Softw. Eng. 11, 1 (März 2006), 5-31

[EclEmma] Java Code Coverage for Eclipse, www.eclemma.org (letzter Zugriff am 26.07.2014)

[eCob] Eclipse Plugin for Cobertura , http://ecobertura.johoop.de (letzter Zugriff am 26.07.2014)

[Emma] EMMA: a free Java code coverage tool , http://emma.sourceforge.net (letzter Zugriff am 26.07.2014)

[FAA01] Federal Aviation Administration, 2001, An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion, DOT/FAA/AR-01/18, http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf (letzter Zugriff am 30.09.2013)

[FAA07] Federal Aviation Administration, 2007, Software Verification Tools Assessment Study, DOT/FAA/AR-06/54, http://www.tc.faa.gov/its/worldpac/techrpt/ar0654.pdf (letzter Zugriff am 30.09.2013)

[Fi11] Fisher, M., II, Wloka, J., Tip, F., Ryder, B. G., Luchansky, A., 2011, An evaluation of change-based coverage criteria. In Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools (PASTE '11). ACM, New York, NY, USA, 21-28

[FW93] Frankl, P.G., Weiss, S.N., 1993, An Experimental Comparison of the effectiveness of Branch Testing and Data Flow Testing, IEEE Trans. Softw. Eng. 19, 8 (August 1993), 774-787

[FL04] Fettke, P., Loos, P., 2004, Referenzmodellierungsforschung. Wirtschaftsinformatik (2004), Nr. 5, S. 331-340

[FLS07] Frühauf, K., Ludewig, J., Sandmayr, H., 2007, Software-Prüfung – Eine Anleitung zum Test und zur Inspektion, 6. Auflage, vdf Hochschulverlag AG an der ETH Zürich

[FP97] Fenton, N., Pfleeger, S. L., 1997, Software Metrics (2nd Ed.): A Rigorous and Practical Approach. PWS Pub. Co., Boston, MA, USA

[Ga95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam

[Ga10] Garousi, V., 2010, An Open Modern Software Testing Laboratory Courseware - An Experience Report. In Proceedings of the 2010 23rd IEEE Conference on Software Engineering Education and Training (CSEET '10). IEEE Computer Society, Washington, DC, USA, 177-184

[Gi06] Gittens, M., et al., 2006, All code coverage is not created equal: a case study in prioritized code coverage. In Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON '06), Hakan Erdogmus, Eleni Stroulia, and Darlene Stewart (Eds.). IBM Corp., Riverton, NJ, USA, Article 11

[GMS00] Gupta, N., Mathur, A.P., Soffa, M. L., 2000, Generating Test Data for Branch Coverage. In Proceedings of the 15th IEEE international conference on Automated software engineering (ASE '00). IEEE Computer Society, Washington, DC, USA, 219-

[Go05] Gosling, J., Joy, B., Steele, G., Bracha., G., 2005, Java(Tm) Language Specification, the 3rd Edition, Addison-Wesley. Addison-Wesley Professional

[Gri95] Grimm, K., 1995, Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie. GMD-Bericht 251. R. Oldenburg Verlag, München/Wien

[Ham10] Hampp, T., 2010, Ein Kosten-Nutzen-Modell für die Softwareprüfung. Shaker Verlag, Aachen

[He84] Hetzel, C.W., 1984, The Complete Guide to Software Testing. John Wiley & Sons, Inc., New York

[HLL94] Horgan, J. R., London, S., Lyu, M. R., 1994. Achieving software quality with testing coverage measures. Computer 27, 9 (September 1994)

[Ho75] Howden, W.E., 1975, Methodology for the generation of program test data. IEEE Trans. Comput. 24, 5 (Mai 1975), 554–560

[Hol09] Holschuh, T., et al., 2009, Predicting defects in SAP Java code: An experience report, In Proceedings of the 31th international conference on Software Engineering (ICSE), (Mai 2009), 172 - 181

[HP04] Hovemeyer, D., Pugh, W., 2004, Finding bugs is easy. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04). ACM, New York, NY, USA, 132-136

[Hu75] Huang, J.C., 1975, An Approach to Program Testing. ACM Comput. Surv. 7, 3 (Sep. 1975), 113-128

[Hut94] Hutchins, M. et al., 1994, Experiments on the Effectiveness of Dataflow- and Control-flow-Based Test Adequacy Criteria, In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 191-200

[IEC61508] IEC 61508. Functional safety of electrical/electronic/programmable electronic (E/E/PE) safety related systems. Part 1–7, Edition 1.0

[IEEE610] IEEE 1990 Standard for Software Test Documentation IEEE Std 829-1990

[IEEE829] IEEE 1998 Standard for Software Test Documentation IEEE Std 829-1998

[ISO14977] EBNF Syntax Specification Standard, EBNF: ISO/IEC 14977 : 1996(E)

[ISO26262] ISO/DIS 26262-8:2009. Draft International Standard Road vehicles — Functional safety - Part 8: Supporting processes. 2009

[JCC] JavaCC parser/scanner generator for Java, http://java.net/projects/javacc (letzter Zugriff am 26.07.2014)

[JG05] Jeffrey, D., Gupta, N., 2005, Test Suite Reduction with Selective Redundancy, In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05). IEEE Computer Society, Washington, DC, USA, 549-558

[JH03] Jones, J.A., Harrold, M. J., 2003, Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Trans. Softw. Eng. 29, 3 (März 2003), 195-209

[JSE96] Jones, B. F., Sthamer H.-H., Eyres D. E., 1996, Automatic structural testing using genetic algorithms, Software Engineering Journal, 11(5), (September 1996), 299-306

[JUnit] JUnit Webseite, http://www.junit.org (letzter Zugriff am 26.07.2014)

[Justus] Justus: Test case organization tool for software developers. http://justus.tigris.org (letzter Zugriff am 26.07.2014)

[JVMPI] Java Virtual Machine Profiling Interface, http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html (letzter Zugriff am 26.07.2014)

[Kan96] Kaner, C., 1996, Software Negligence and Testing Coverage, Proceedings of STAR 96, 5th International Conference on Software Testing, Analysis, and Review, Software Quality Engineering, Orlando FL

[KGM09] Koochakzadeh, N., Garousi, V., Maurer, F., 2009, Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned. In Proceedings of the 2009 International Conference on Software Testing Verification and Validation (ICST '09). IEEE Computer Society, Washington, DC, USA, 220-229

[Ki03] Kim, Y.W., 2003, Efficient use of code coverage in large-scale software development, Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, October 06 - 09, 2003). IBM Press, 145-155

[Kle09] Kleuker, S., 2009, Werkzeuge zur Qualitätssicherung in der Software-Engineering Ausbildung. In: U. Jaeger, K. Schneider (Hrsg.): Software Engineering im Unterricht der Hochschulen (SEUH 11 – Hannover 2009). dpunkt Verlag, Heidelberg

[KL95] Kamsties, E., Lott, C.M., 1995, An empirical evaluation of three defect-detection techniques. In Proceedings of the 5th European Software Engineering Conference, Wilhelm Schäfer and Pere Botella (Eds.). Springer-Verlag, London, UK, 362-383

[KMT12] Kajo-Mece, E., Tartari, M., 2012, An Evaluation of Java Code Coverage Testing Tools. In Local Proceedings of the Fifth Balkan Conference in Informatics (BCI 2012), ISBN 978-86-7031-200-5, Novi Sad, Serbia, (September 2012)

[KS73] Knuth, D.E., Stevenson F.R., 1973, Optimal measurement points for program frequency counts. BIT Numerical Mathematics, 13, 3 (September 1973), Verlag Springer Netherlands, 313-322

[Ko90] Korel, B., 1990, Automated software test data generation. IEEE Trans. Softw. Eng. 16, 8 (August 1990), 870-879

[La05] Lawrance, J. et al., 2005, How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study. In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '05). IEEE Computer Society, Washington, DC, USA, 53-60

[Li02] Liggesmeyer, P., 2002, Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, Heidelberg, Berlin

[Li05] Li, J.J., 2005, Prioritize Code for Testing to Improve Code Coverage of Complex Software. In Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE '05). IEEE Computer Society, Washington, DC, USA, 75-84

[LL10] Ludewig, J., Lichter, H., 2010, Software Engineering, dpunkt Verlag, 2. Auflage

[LGJ07] Lingampally, R., Gupta, A., Jalote, P., 2007, A Multipurpose Code Coverage Tool for Java. In Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07). IEEE Computer Society, Washington, DC, USA, 261b-

[LHL93] Lyu, M.R., Horgan, J.R., London, S., 1994, A Coverage Analysis Tool for the Effectiveness of Software Testing, IEEE transactions on reliability, 43, 4 (1994), 527-535

[Lö05] Lösch, F., 2005, Instrumentation of Java Program Code for Control Flow Analysis, Stuttgart, Univ., Studiengang Softwaretechnik, Diplomarbeit Nr. 2258 http://elib.uni-stuttgart.de/opus/volltexte//2005/2256/pdf/DIP_2258.pdf (letzter Zugriff am 30.09.2013)

[Mar99] Marick, B., 1999, How to misuse code coverage. In Proc. 16th International Conference on Testing Computer Software (Juni 1999), 16-18

[MB03] Marré, M., Bertolino, A., 2003, Using Spanning Sets for Coverage Testing. IEEE Trans. Softw. Eng. 29, 11 (November 2003), 974-984

[McM04] McMinn, P., 2004, Search-based software test data generation: A survey. Research Articles. Softw. Test. Verif. Reliab. 14, 2 (Juni 2004), 105-156

[MISRA] MISRA, 2004, Guidelines for the Use of the C Language in Critical Systems, Motor Industry Software Reliability Association, ISBN 0 9524156 2 3 (Oktober 2004)

[MM63] Miller, J. C., Maloney, C. J., 1963. Systematic mistake analysis of digital computer programs. Commun. ACM 6, 2 (Februar 1963), 58-63

[MND09] Mockus, M., Nagappan, N., Dinh-Trong, T.T., 2009, Test coverage and post-verification defects: A multiple case study. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09). IEEE Computer Society, Washington, DC, USA, 291-301

[Mü98] Müller, U., Wiegmann, T., 1998, State of the practice der Prüf- und Testprozesse in der Softwareentwicklung — Ergebnisse einer empirischen Untersuchung bei Softwareunternehmen in Deutschland, Technischer Bericht, Universität Köln, März 1998

[My79] Myers, G.J., 1979, Art of Software Testing, John Wiley & Sons, Inc., New York

[Na88] Ntafos, S.C., 1988, A Comparison of Some Structural Testing Strategies. IEEE Trans. Softw. Eng. 14, 6 (Juni 1988), 868-874

[NBZ06] Nagappan, N., Ball, T., Zeller, A., 2006, Mining metrics to predict component failures. In Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 452-461

[NHH99] Nielson, F., Nielson, H.R., Hankin, C., 1999, Principles of program analysis, Heidelberg [u.a.], Springer, 1999

[NS73] Nassi, I., Shneiderman, B., 1973, Flowchart Techniques for Structured Programming. SIGPLAN Not. 8, 8 (August 1973), 12-26

[Ost07] Oster, N., 2007, Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken, Dissertation, in Arbeitsberichte des Instituts für Informatik, Vol. 40, Nr. 2, Universität Erlangen-Nürnberg

[OW91] Ostrand, T. J., Weyuker, E. J., 1991, Data flow-based test adequacy analysis for languages with pointers. In Proceedings of the symposium on Testing, analysis, and verification (TAV4). ACM, New York, NY, USA, 74-86

[OWB05] Ostrand, T. J., Weyuker, E. J., Bell, R. M., 2005, Predicting the Location and Number of Faults in Large Software Systems. IEEE Trans. Softw. Eng. 31, 4 (April 2005), 340-355

[RAD] IBM Rational Application Developer for WebSphere Software http://www.ibm.com/developerworks/downloads/r/rad (letzter Zugriff am 26.07.20914)

[PAR09] Priya, S., Askarunisa, A., Ramaraj, N., 2009, Measuring the Effectiveness of Open Coverage based Testing Tools. Journal of Theoretical and Applied Information Technology, 5, 5 (2009), 499-514

[PC90] Podgurski, A., Clarke, L.A., 1990, A formal model of program dependences and its implications for software testing, debugging, and maintenance, IEEE Trans. Softw. Eng. 16, 9 (September 1990), 965-979

[POC93] Piwowarski, P., Ohba, M., Caruso, J., 1993, Coverage Measurement Experience During Function Test. In Proceedings of the 15th International conference on Software engineering, (ICSE'93), ACM, New York, NY, USA, 287-301

[Pr05] Pretschner, A. et al., 2005, One evaluation of model-based testing and its automation. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 392-401

[RAF04] Rutar, N., Almazan, C. B., Foster, J. S., 2004, A Comparison of Bug Finding Tools for Java. In Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04). IEEE Computer Society, Washington, DC, USA, 245-256

[RB85] Ramsey, J., Basili, V. R., 1985, Analyzing the test process using structural coverage. In Proceedings of the 8th international Conference on Software Engineering (London, England, August 28 - 30, 1985). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 306-312

[Re85] Reisig, W., 1985, Systementwurf mit Netzen, Springer Verlag

[Re86] Reisig, W., 1986, Petri-Netze, Eine Einführung, zweite Auflage, Springer Verlag

[Rie97] Riedemann, E. H., 1997, Testmethoden für sequentielle und nebenläufige Software-Systeme, B. G. Teubner, Stuttgart

[RJB04] Rumbaugh, J., Jacobson, I., Booch, G., 2004, Unified Modeling Language Reference Manual, the (2nd Edition). Pearson Higher Education

[RTCA92] RTCA, 1992, Software considerations in airborne systems and equipment certification, Document RTCA/DO-178B. RTCA, Inc., (Dezember 1992)

[RW85] Rapps, S., Weyuker, E.J., 1985, Selecting Software Test Data Using Data Flow Information. IEEE Trans. Softw. Eng. 11, 4, (1985), 367-375

[Schm04] Schmidberger, R.: Use-Case-bezogenes Reverse-Engineering von Entscheidungstabellen. EMISA 2004 Proceedings, GI-Edition, ISBN 3-88579-385-7 (September 2004)

[Schm06] Schmidberger, R., 2006, Nachdokumentation von Geschäftsregeln aus Quelltext. Softwaretechnik-Trends, Band 26 Heft 2, Gesellschaft für Informatik, ISSN 07208928 (Mai 2006), 61-62

[Schm07a] Schmidberger, R., 2007, Analyse von Testprozessen in der Industrie. Software Engineering 2007 - Beiträge zu den Workshops, GI-Edition, ISBN 978-3-88579-200-0 (März 2007)

[Schm07b] Schmidberger, R., Biermann, S., 2007, Testresultatsvergleich mit UML-Analysemodellen und OCL-Ausdrücken. Informatik 2007 - Band 2, GI-Edition, ISBN 978-3-88579-204-8 (September 2007)

[Schm08] Schmidberger, R., 2008, Glass-Box-Test zur Testsuite-Optimierung. Software Engineering 2008 - Beiträge zu den Workshops, GI-Edition, ISBN 978-3-88579-216-1 (März 2008)

[Schm09] Schmidberger, R., 2009, Glass-Box-Test zur Äquivalenzklassenbildung von Produktionsdaten. Softwaretechnik- Trends Band 29 Heft 2, Gesellschaft für Informatik, ISSN 0720-8928 (Mai 2009)

[Schm11] Schmidberger, R., 2011, Ein kombinierter Black-Box- und Glass-Box-Test. Software Engineering 2011, GI-Edition, ISBN 978-3-88579-277-2, (Februar 2011)

[Schn98] Schneider, H.-J., 1998, Lexikon Informatik und Datenverarbeitung, Oldenbourg; 4. Auflage

[Schu09] Schumm, S., 2009, Praxistaugliche Unterstützung beim selektiven Regressionstest, Stuttgart, Univ., Studiengang Softwaretechnik, Diplomarbeit Nr. 2923 http://elib.uni-stuttgart.de/opus/volltexte/2009/4762/pdf/DIP_2923.pdf (letzter Zugriff am 30.09.2013)

[SI11] Shahid, M., Ibrahimn, S., 2011, Evaluation of Test Coverage Tools in Software Testing, International Conference on Telecommunication Technology and Applications, Proc. of CSIT 5 (2011), IACSIT Press, Singapore

[SL04] Spillner, A., Linz, T., 2004, Basiswissen Softwaretest, 2. Auflage, dpunkt Verlag

[Sn04] Sneed, H. M., 2004, Reverse Engineering of Test Cases for Selective Regression Testing. In Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04) (CSMR '04). IEEE Computer Society, Washington, DC, USA, 69-

[Sp95] Spillner, A., 1995, Test criteria and coverage measures for software integration testing. Software Quality Journal, 4,4 (1995), 275-286

[Sta12] Staats, M., Gay, G., Whalen, M. W., Heimdahl, M., 2012, On the danger of coverage directed test case generation. In Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering (FASE'12), Juan Lara and Andrea Zisman (Eds.). Springer-Verlag, Berlin, Heidelberg, 409-424

[SVW11] Spillner, A., Vosseberg, K., Winter , M., 2011, Umfrage 2011: Softwaretest in der Praxis, dpunkt Verlag, Heidelberg

[Ta89] Tai, K.C., 1989. What to do beyond branch testing. SIGSOFT Softw. Eng. Notes 14, 2 (April 1989), 58-61

[Ta96] Tai, K.-C., 1996, Theory of Fault-Based Predicate Testing for Computer Programs. IEEE Trans. Softw. Eng. 22, 8 (August 1996), 552-562

[UZ98] Ur, S., Ziv, A., 1998, Off-The-Shelf vs. Custom Made Coverage Models, Which is the One for You? In proceedings of the 7th international conference on software testing analysis and review (STAR98)

[VA00] Verbeek, W., van der Aalst, W., 2000, Woflan 2.0: a Petri-net-based workflow diagnosis tool. In Proceedings of the 21st international conference on Application and theory of petri nets (ICATPN'00), Mogens Nielsen and Dan Simpson (Eds.). Springer-Verlag, Berlin, Heidelberg, 475-484

[VMXT04] Das V-Modell XT, 2004, Version 1.2.0, Bundesrepublik Deutschland, Webseite, www.vmodell-xt.de (letzter Zugriff am 27.07.2014)

[WBS01] Wegener, J., Baresel, A., Sthamer, H., 2001, Evolutionary test environment for automatic structural testing. Information and Software Technology 43,14 (2001), 841-854.

[We88] Weyuker, E., 1988, Evaluating Software Complexity Measures. IEEE Trans. Softw. Eng. 14, 9 (September 1988), 1357-1365

[WH11] Wasilewski, M., Hasselbring, W., 2011, A Formal and Pragmatic Approach to Engineering Safety-critical Rail Vehicle Control Software, Software Engineering 2011, GI-Edition, ISBN 978-3-88579-277-2, 99-110 (Februar 2011)

[Wi86] Wirth, N., 1986, Compilerbau, 4. Auflage, Teubner Verlag, Stuttgart

[WoPeD] Workflow Petri Net Designer, http://woped.dhbw-karlsruhe.de/woped/ (letzter Zugriff am 26.07.2014)

[YLW09] Yang, Q., Li, J.J., Weiss, D., 2009, A Survey of Coverage-Based Testing Tools. Comput. J. 52, 5 (August 2009), 589-597

[YCP09] Yu, Y.T., Chan, E.Y., Poon, P., 2009, On the Coverage of Program Code by Specification-Based Tests. In Proceedings of the 9th International Conference on Quality Software (QSIC '09). IEEE Computer Society, Washington, DC, USA, 41-50

[ZHM97] Zhu, H., Hall, P.A.V., May J.H.R., 1997, Software unit test coverage and adequacy. ACM Comput. Surv. 29, 4 (December 1997), 366-427

[ZPZ07] Zimmermann, T., Premraj, R., Zeller, A., 2007, Predicting Defects for Eclipse. In Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07). IEEE Computer Society, Washington, DC, USA, 9-