Rainer Schmidberger

Wohldefinierte Überdeckungsmetriken für den Glass-Box-Test

Cuvillier Verlag Göttingen
Internationaler wissenschaftlicher Fachverlag

Wohldefinierte Überdeckungsmetriken für den Glass-Box-Test

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

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2014
Zugl.: Stuttgart, Univ., Diss., 2014

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2014
Gedruckt auf umweltfreundlichem, säurefreiem Papier aus nachhaltiger Forstwirtschaft.

ISBN 978-3-95404-794-9
eISBN 978-3-7369-4794-8

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.