Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 34

Erweiterung von CodeCover für die Programmiersprache C

Steffen Hanikel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Jochen Ludewig
Betreuer:	Dipl.-Ing. Rainer Schmidberger
begonnen am:	12. November 2012
beendet am:	14. Mai 2013
CR-Klassifikation:	D.2.5, D.3.4

Zusammenfassung

Ziel der Arbeit ist es, das Glass-Box-Test-Werkzeug *CodeCover* um eine Unterstützung für die Programmiersprache *C* zu erweitern. Der Umgang mit Präprozessor und Makros im Glass-Box-Test stellt eine besondere Herausforderung dar. In dieser Arbeit wird ein Verfahren vorgestellt, wie Makros im Testbericht dargestellt werden können, um den Umgang mit ihnen im Glass-Box-Test zu erleichtern. Dazu wird die Expansion der Makros interaktiv dargestellt.

Das Frontend wurde im Rahmen der Arbeit vollständig umgesetzt. Die spezielle Visualisierung der Makros im Testbericht wurde nicht vollständig umgesetzt.

Abstract

The aim of this work is to add support for the *C* programming language to the glass box testing tool *CodeCover*. Special consideration must be given to the preprocessor and macros in the code under test. This bachelor thesis presents a way to visualize macros in the test report in order to make it easier to reason about them in a glass box test. To achieve this, the expansion of macros can be explored interactively.

The frontend has been implemented completely in the course of this work. The visualization of macros in the report has not been implemented completely.

Inhaltsverzeichnis

1. Einführung

1.1. Hintergrund

Das Open-Source Glass-Box-Test-Werkzeug CodeCover nutzt zur internen Darstellung der Programme und zur Berechnung der Überdeckungsmetriken ein von Programmiersprachen unabhängiges Modell. Konkrete Adapter zur Transformation von Programmen in dieses Modell gibt es für die Programmiersprachen Java und COBOL. Das Anbinden weiterer Programmiersprachen ist vom Entwurf her prinzipiell vorgesehen. Wegen der starken Verbreitung der Programmiersprache C soll CodeCover um einen solchen Adapter für C erweitert werden. CodeCover nutzt als Parser-Generator javacc, eine Grammatik für C ist unter einer Open- Source-Lizenz verfügbar.

1.2. Aufgabenstellung

CodeCover soll um einen Adapter für die Programmiersprache C erweitert werden. Dieser Adapter soll vergleichbar den bereits bestehenden Adaptern für Java oder COBOL aufgebaut sein. Allerdings hat C mit dem sogenannten Präprozessor ein gegenüber Java abweichendes Konzept. Wie mit der Erhebung und Visualisierung von Überdeckung bei Präprozessor-Direktiven sinnvoll umgegangen wird, gehört zum analytischen Teil der Arbeit. Die Instrumentierung wird nur für den CodeCover Batch-Modus gefordert, d. h. die Integration in den eclipse-Build-Prozess, vergleichbar der Java-Integration, wird nicht gefordert. Auch wird eine Visualisierung der Überdeckung im eclipse-Framework nicht verlangt. Stattdessen werden die bereits bestehenden Berichte genutzt. Dort ist aber ggf. eine Anpassung wegen der Präprozessor-Besonderheit erforderlich. Zur Aufwandsreduktion soll ausdrücklich nur C und nicht C++ unterstützt werden.

2. Grundlagen

Sowohl bei der Entwicklung, als auch bei der Wartung eines Softwaresystems hat die Qualitätssicherung einen hohen Stellenwert. Sie ist notwendig, um ein hohes Vertrauen in das System zu gewährleisten ([LL07], 13.1 Software-Qualitätssicherung, S. 252). Eines der analytischen Qualitätssicherungsverfahren ist der dynamische Test:

> **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.
>
> (IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology)

Dynamische Tests lassen sich nach unterschiedlichen Kriterien klassifizieren. Eine Testart ist der Glass-Box-Test (GBT):

> „Berücksichtigt man bei der Wahl der Testfälle die innere Struktur des Prüflings und die (durch spezielle Werkzeuge erstellten) Aufzeichnungen früherer Programmläufe, dann handelt es sich um einen *Glass-Box-Test* […]."
>
> ([LL07], 19.2.1 Klassifikation nach den Grundlagen des Tests, S. 455)

Ziel eines Glass-Box-Tests ist es immer, verschiedene Metriken zu erreichen:

- Die *Anweisungsüberdeckung* ist erreicht, wenn alle Anweisungen des Programms ausgeführt werden.
- Die *Zweigüberdeckung* ist erreicht, wenn bei allen Verzweigungen des Programms alle möglichen Wege (Zweige) durchlaufen werden.
- Die *Termüberdeckung* ist erreicht, wenn jeder logische Term, der den Ablauf in einer Verzweigung steuert, mit beiden möglichen Werten (true und false) wirksam geworden ist.

([LL07], 19.6.1 Überdeckungskriterien für den Glass-Box-Test, S. 481)

2.1. CodeCover

CodeCover ist ein Glass-Box-Test-Werkzeug, das im Jahr 2007 als Studienprojekt der Abteilung Software Engineering des Fachbereichs Informatik an der Universität Stuttgart entstand. Es ist in Java implementiert und verfügt über Frontends für die Sprachen Java und Cobol. CodeCover kann sowohl über die Kommandozeile, als auch über eine in Eclipse integrierte Oberfläche bedient werden. Die erfassten Daten können über einen HTML-Report oder direkt in Eclipse visualisiert werden. Eine einfache Erweiterung für neue Frontends, Überdeckungsmetriken und Visualisierungen der Ergebnisse ist vorgesehen.

CodeCover erfasst alle gängigen Metriken (Anweisungs-, Zweig- und Termüberdeckung). Bei der Zweigüberdeckung werden allerdings keine Schleifen, sondern nur **if**- und **switch**-Anweisungen betrachtet. Schleifen werden gesondert durch einen Boundary-Interior-Pfadtest erfasst. Dieser gibt an, ob eine Schleife keinmal, einmal oder öfter als einmal ausgeführt wird.

Das Frontend einer Sprache besteht aus einem Parser, Modell-Ersteller und Instrumentierer. Diese drei Komponenten sind eng miteinander verknüpft und werden oft zusammen als Instrumentierer bezeichnet. In diesem Bericht steht der Instrumentierer allerdings nur für das Einfügen von Anweisungen in den Quellcode.

Die bestehenden Frontends nutzen zur Parsergenerierung JavaCC, einen Java-Parsergenerator. Um CodeCover um einen Adapter für C zu erweitern, muss ein neues Frontend angelegt werden und gegebenenfalls die Berichterstellung und Visualisierung angepasst werden. Die GBT-Analyse bleibt unverändert.

2.2. Die Sprache C

C ist eine allgemeine, imperative und maschinennahe Sprache. Sie wurde ursprünglich in den Jahren 1969–1973 für das UNIX-Betriebsystem der DEC PDP-11 von Dennis Ritchie entworfen. Das Buch *The C programming language* [KR78] von Brian Kernighan und D. Ritchie aus dem Jahr 1978 galt bis zum Jahr 1983 als Referenz für C. 1983 wurde C dann von einer Kommission des American National Standards Institute standardisiert. Es folgten weitere Überarbeitungen des Standards durch Kommissionen der ISO in den Jahren 1989, 1999 und 2011.

Die Programmiersprache C wurde früher sehr häufig verwendet. Ihre Popularität hat im Laufe der Jahre aber immer weiter abgenommen, da sie von höheren Sprachen verdrängt wurde. Sie

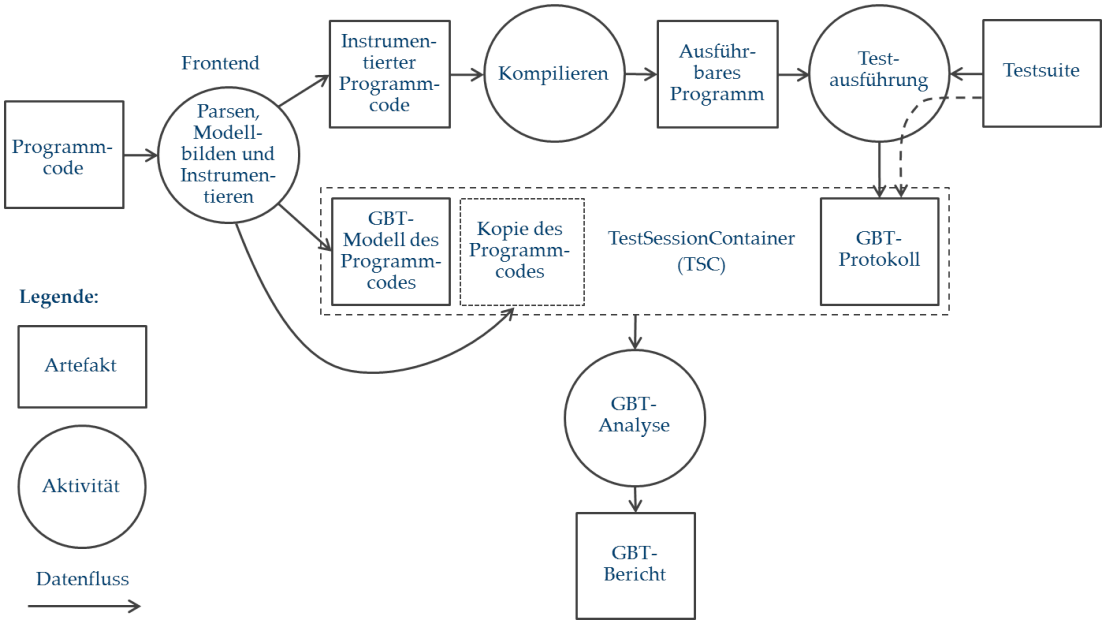


Abbildung 2.1.: CodeCover Datenfluss

wird aber auch heute noch für die Entwicklung von Betriebssystemen, eingebetteten Systemen und performance-kritischen Systemen verwendet.

Eine Besonderheit von C ist, dass zur Übersetzung eines Programms der Präprozessor gehört. „[Dieser] ersetzt Makros im Programmtext, fügt andere Quelldateien ein und ermöglicht die bedingte Übersetzung.“([KR90], Einführung, S.1). Außerdem hat C den Ruf, den Programmierer nur wenig einzuschränken und ihm volle Flexibilität bei der Nutzung der Sprache zu lassen.

2.2.1. Duff's Device

Folgendes Beispiel aus dem Jahr 1983 soll die ungewöhnliche Flexibilität von C verdeutlichen:

Listing 2.1: Duff's Device

```
void send(int *to, int *from, int count)
{
  int n = (count + 7) / 8;
  switch (count % 8)
  {
    case 0: do { *to = *from++;
    case 7: *to = *from++;
    case 6: *to = *from++;
    case 5: *to = *from++;
    case 4: *to = *from++;
    case 3: *to = *from++;
    case 2: *to = *from++;
    case 1: *to = *from++;
      } while(--n > 0);
  }
}
```

Diese Funktion schreibt den Inhalt eines Vektors (*from*) mit der Länge *count* in ein IO-Register (*to*). Dazu wird das Feld Integer für Integer in das Register kopiert. Zu der Zeit, in der dieser Code entstand, war die Überprüfung der Schleifenbedingung im Verhältnis zum Schleifeninhalt häufig sehr teuer. Deswegen hat der Autor (Tim Duffy) den Schleifeninhalt (**to = *from++;*) von Hand acht Mal dupliziert. Daraus ergibt sich ein Problem, wenn man ein Feld übergibt, dessen Länge kein Vielfaches von Acht ist. Deswegen springt der Autor direkt mit einer Switch-Anweisung in die Schleife, um die korrekte Anzahl an Ausführungen zu erhalten. Der Autor selbst hat dabei eine zwiespältige Meinung zu der Qualität seines Codes [1].

[1] http://www.lysator.liu.se/c/duffs-device.html

2.3. Der Präprozessor

Der Präprozessor ersetzt Zeichenfolgen im Quelltext, die genauen Ersetzungsregeln sind dabei sehr umfangreich. Deswegen werden nur die wichtigsten Regeln vorgestellt. Die vollständigen Regeln lassen sich im Standard im Kapitel 6.10 Preprocessing Directives ([ISO11]) nachlesen.

2.3.1. Einfügen von Dateien

```
#include <stdio.h>
```

Mit diesem Befehl wird die Datei *stdio.h* eingebunden. D.h. der Präprozessor sucht nach der Datei und fügt diese als exakte Kopie an die Stelle des Befehls ein. Das Einbinden von Dateien wird für die Modularisierung eingesetzt. Funktionsdefinitionen und Konstanten werden so normalerweise in Header-Dateien ausgelagert.

2.3.2. Einfache Makros

```
#define M_PI 3.14159265358979323846
```

Dieser Befehl definiert das Symbol *M_PI* und veranlasst den Präprozessor alle folgenden Vorkommen von *M_PI* durch 3.14… zu ersetzen. Mit diesen einfachen, parameterlosen Makros werden häufig Konstanten definiert.

2.3.3. Bedingte Kompilierung

```
#ifdef DEBUG
printf("Debug Modus aktiv");
#endif
```

Dieser Befehl veranlasst den Präprozessor die Zeile *printf("...");* nur dann in den Quelltext einzufügen, wenn das Symbol *DEBUG* vorher definiert wurde. Die bedingte Kompilierung wird für plattformübergreifenden Code und zur Trennung von Code zu Debuggingzwecken und normalem Code eingesetzt.

2.3.4. Makros

```
#define GREATER (x,y) x > y
```

Dieser Befehl veranlasst den Präprozessor ein später folgendes *GREATER (M_PI, M_E)* durch *M_PI > M_E* zu ersetzen. Mit dieser Funktion des Präprozessors werden funktionsähnliche Textersetzungen definiert.

2.4. Verwendung von Makros in realem Code

Makros werden meistens zur Definition einfacher Einzeiler verwendet. So sind die wahrscheinlich am häufigsten eingesetzten Makros die Makros zur Bestimmung des Minimums und des Maximums:

```
// Liefert das Minimum zweier Ausdruecke
#define MIN(a,b) (((a)<(b))?(a):(b))
// Liefert das Maximum zweier Ausdruecke
#define MAX(a,b) (((a)>(b))?(a):(b))
```

Fortgeschrittene Anwendungen sind das Definieren ganzer Datenstrukturen als Makros. Häufig verwendet werden beispielsweise die Implementierung von verketteten Listen im Linux Kernel (include/linux/list.h) oder die Implementierung von Red-Black-Trees als Makros in diversen *BSD Projekten (tree.h). Sie lassen sich leicht im Quelltext der Projekte finden.

Das folgende Beispiel zeigt die Verwendung eines Makros zur Iterierung über verkette Listen aus dem Linux Kernel. Die Deklaration der Datenstrukturen wurde aufgrund der Übersichtlichkeit weggelassen:

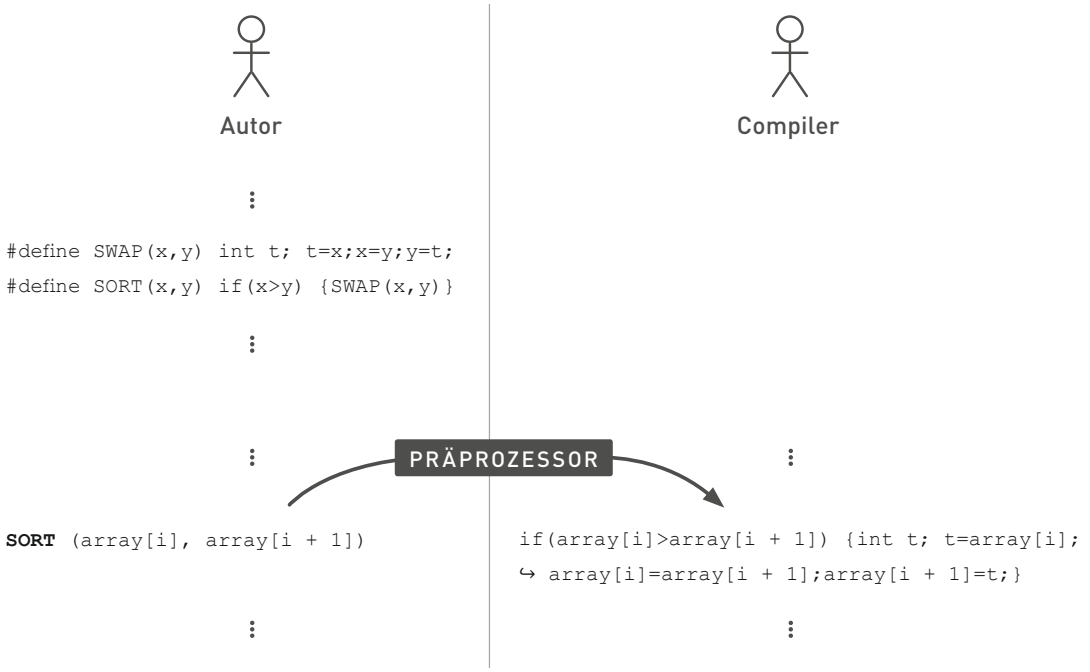
```
/**
 * list_for_each - iterate over a list
 * @pos: the &struct list_head to use as a loop cursor.
 * @head: the head for your list.
 */
#define list_for_each(pos, head) \
       for (pos = (head)->next; pos != (head); pos = pos->next)

// Iteriert ueber alle Elemente von list und gibt diese nacheinander aus:
list_for_each(iterator, list)
{
  // list_entry liefert fuer einen Iterator das eigentliche Element
  printf(list_entry(iterator, struct list, list));
}
```

Der Programmierer verwendet in diesem Beispiel das Makro **list_for_each** wie eine Kontrollstruktur der Sprache selbst.

2.5. Verwendung von Makros und der Glass-Box-Test

Die Beispiele aus der Praxis zeigen, dass Makros sich nicht an die syntaktischen Konventionen von C halten. Ein Element wie der **for_each** Schleifenkopf existiert in C nicht und kein C Parser würde den Code verstehen können. Außerdem gibt es nur sehr wenige Konventionen wie Makros verwendet werden. In der Praxis werden alle Varianten von Makros, seien es Konstanten, einfache Funktionen, komplette Code-Blöcke, neue Kontrollstrukturen und neue Datentypen verwendet.

Dabei lässt sich weder bei der Deklaration, noch bei der Verwendung eines Makros die syntaktische Funktion oder die Semantik erkennen. Zum Parsen, zur GBT-Modellbildung und zur Instrumentierung ist also der nicht-expandierte Code unbrauchbar. Hieraus ergibt sich eine Diskrepanz, denn für einen Glass-Box-Test muss der Entwickler den Code zwar nicht vollständig lesen und verstehen, aber er muss „den *Zweck* des betreffenden Code-Abschnitts [erkennen].“ ([LL07], 19.6 Der Glass-Box-Test, S.480) Den Zweck eines durch den Präprozessors ersetzten Code-Abschnitts kann er aber nicht im expandierten Code erkennen. Das Schaubild 2.2 soll das Problem verdeutlichen:



- Zweck ersichtlich und damit zur Visualisierung der Überdeckung geeignet
- Syntaktisch *nicht* korrekt und damit nicht für die Instrumentierung geeignet.

- Zweck *nicht* ersichtlich und damit *nicht* zur Visualisierung der Überdeckung geeignet
- Syntaktisch korrekt und damit für die Instrumentierung geeignet.

Abbildung 2.2.: Makros im GBT aus Sicht des Autors und des Compiler

3. Umsetzungskonzept

3.1. Vergleich von Java und C

Augenscheinlich ist die Instrumentierung von C Code einfacher als die Instrumentierung von Java Code. Java ist objektorientiert und es gibt komplexe Elemente wie Exceptions, Generics und SynchronizedStatements. Diese beeinflussen die Komplexität der Instrumentierung aber kaum. Klassen, Generics und SynchronizedStatements benötigen keine Sonderbehandlung, da diese den Kontrollfluss nicht ändern können. Anweisungen können aber jederzeit Exceptions werfen und dadurch den Kontrollfluss ändern. Dieser Umstand muss beim Instrumentieren beachtet werden.

In C fehlen viele dieser fortgeschrittenen Elemente. Allerdings ist die Syntax von C weniger streng und erlaubt dadurch auch ungewöhnliche und gefährliche Konstruktionen. Konstellationen wie Duff's Device (siehe Abschnitt 2.2.1), die einen klassischen Glass-Box-Test unmöglich machen, sind durch die Syntax Javas ausgeschlossen.

Ein viel größeres Problem ergibt sich jedoch durch den Präprozessor. Keine andere allgemeine Programmiersprache verwendet einen Präprozessor in so einem Umfang wie C und C++. Durch den Präprozessor wird die Übersetzung in zwei logisch getrennte Vorgänge geteilt. Dies erschwert einen Glass-Box-Test, da viele Informationen zwischen diesen beiden Schritten verloren gehen.

3.2. Behandlung von Makros im Glass-Box-Test

Wie in Abschnitt 2.5 auf Seite 10 festgestellt, muss der Glass-Box-Test auf dem expandierten Code erfolgen. Die Visualisierung der Ergebnisse muss aber im Originalcode erfolgen.

```
#define SWAP(x,y) int t; t=x;x=y;y=t;
#define SORT(x,y) if(x>y) {SWAP(x,y)}

void sort (int *array, int length)
{
  int n, i, getauscht;
  for (n = length - 1; n > 0; n--) {
    // Annahme: nicht getauscht
    getauscht = 0;
    for (i = 0; i < n; i++) {
      if (array[i] > array[i + 1]) {
        // es wurde getauscht
        getauscht = 1;
      }
      SORT (array[i], array[i + 1])
    }
    if (!getauscht) {
      // es wurde nicht getauscht:
      // das array ist sortiert
      break;
    }
  }
}
```

Abbildung 3.1.: Visualisierung im Bericht (1)

3.3. Visualisierung der Ergebnisse

Im Bericht wird die Überdeckungsstatistik angezeigt und die überdeckten Programmteile im Originalcode farbig markiert. Überdeckte Teile haben die Farbe grün, zum Teil ausgeführte (etwa bei Schleifen und Bedingungen) gelb und nicht ausgeführte rot.

Makros im Code werden fett gesetzt (Abbildung 3.1). Wenn der Benutzer auf ein Makro klickt, öffnet sich ein Fenster, in dem der expandierte Code des Makros zu sehen ist (Abbildung 3.2). Es wird allerdings nur ein einziges Mal expandiert und nicht rekursiv. Falls sich dort wieder ein Makro befindet, lässt sich ein weiteres Fenster öffnen (Abbildung 3.3). Der Code wird wie sonst auch farbig markiert.

```
#define SWAP(x,y) int t; t=x;x=y;y=t;
#define SORT(x,y) if(x>y) {SWAP(x,y)}

void sort (int *array, int length)
{
  int n, i, getauscht;
  for (n = length - 1; n > 0; n--) {
    // Annahme: nicht getauscht
    getauscht = 0;
    for (i = 0; i < n; i++) {
      if (array[i] > array[i + 1]) {
        // es wurde getauscht
        getauscht = 1;
      }
      SORT (array[i], array[i + 1])
    }
      if(array[i]>array[i + 1]){SWAP(array[i],array[i + 1])}
      break;
    }
  }
}
```

Der Benutzer hat das Makro **SORT** angeklickt und sieht den expandierten Code.

Abbildung 3.2.: Visualisierung im Bericht (2)

```
#define SWAP(x,y) int t; t=x;x=y;y=t;
#define SORT(x,y) if(x>y) {SWAP(x,y)}

void sort (int *array, int length)
{
  int n, i, getauscht;
  for (n = length - 1; n > 0; n--) {
    // Annahme: nicht getauscht
    getauscht = 0;
    for (i = 0; i < n; i++) {
      if (array[i] > array[i + 1]) {
        // es wurde getauscht
        getauscht = 1;
      }
      SORT (array[i], array[i + 1])
    }
    break;
    }
  }
}
```

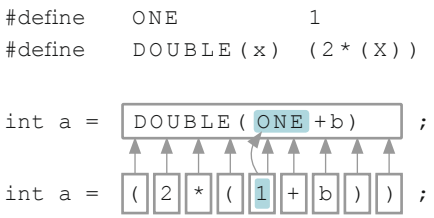
```
if(array[i]>array[i + 1]){SWAP(array[i],array[i + 1])}
```

```
int t;
t=array[i];
array[i]=array[i + 1];
array[i + 1]=t;
```

Der Benutzer hat das Makro **SWAP** angeklickt und sieht den expandierten Code.

Abbildung 3.3.: Visualisierung im Bericht (3)

3.4. Vergleich mit anderen Lösungen



Die Positionen der expandieren Tokens (untere Zeilen) werden der Position des Makros im ursprünglichen Code zugeordnet.

Abbildung 3.4.: β-Mapping nach Platoff et al.

ATAC (Automatic Test Analysis for C programs [HL92]) ist ein Glass-Box-Test-Werkzeug für C und C++. Saul Londong und Bob Horgan entwickelten es in den Jahren 1992-1994 bei BellCore. 1995 wurde daraus ein kommerzielles Produkt. ATAC instrumentiert auch expandierten Code aus Makros. Die Überdeckung wird im Originalcode visualisiert. Überdeckungsinformationen für expandierten Code werden dabei im Originalcode an der Stelle des Makros angezeigt.

Diese Zuordnung zwischen dem expandierten Code und dem Originalcode entspricht dem β-Mapping von Platoff et al. [LS94] (siehe Abbildung 3.4). Platoff et al. stellen auch vier weitere Mappings vor. Alle Mappings haben gemeinsam, dass sie eine direkte Zuordnung von expandierten Tokens auf Tokens im Quellcode vorsehen. Dies wurde allerdings für die Visualisierung der Überdeckungsinformationen von Makros als unzureichend angesehen.

4. Umsetzung

4.1. Vergleich von JavaCC und Clang als Parser

Neben dem Parsergenerator JavaCC, der auch für das Java und Cobol Frontend CodeCovers eingesetzt wird, existiert auch ein alternativer Parser (Clang). Clang ist ein C/Objective-C/C++ Frontend für das Compiler Framework LLVM. Diese beiden Ansätze werden hier verglichen.

	JavaCC	Clang
Integration	Sehr einfach, da das Framework bereits existiert.	Es besteht bisher keine Integration.
Präprozessor	Es muss ein separater Präprozessor verwendet werden.	Der Präprozessor ist integriert.
Kompatibilität	Erweiterungen des C Standards anderer Hersteller müssen übernommen werden.	Clang ist mit allen gängigen Erweiterungen anderer Hersteller kompatibel.
Wartbarkeit	Das Frontend ist in das bestehende Framework integriert und erhöht damit den Wartungsaufwand nicht mehr als nötig. Allerdings müssen Änderungen am C Standard selber eingepflegt werden.	Das Frontend muss in C++ geschrieben werden. Damit erhöht sich der Wartungsaufwand für CodeCover erheblich, da eine zweite Programmiersprache eingeführt wird. Änderungen des C Standards werden von den Clang-Entwicklern eingepflegt.
Erweiterbarkeit	Der Instrumentierer ist nur mit hohem Aufwand auf C++ erweiterbar.	Da Clang auch eine C++ Unterstützung mitbringt, ist eine Erweiterung für C++ leicht möglich.
Portabilität	Das Framework ist vollständig in Java implementiert und läuft damit auf allen Plattformen mit einer Java-Umgebung.	Clang hat gute Unterstützung für Unix-ähnliche Plattformen. Die Unterstützung für Windows ist allerdings nicht ausgereift.

Brauchbarkeit	Der Parser ist eine Neuimplementierung und damit nicht ausgereift.	Clang ist ein über viele Jahre ausgereiftes und erprobtes Compilerfrontend. Es wird unter anderem von Apple entwickelt und eingesetzt.
Lizenz	–	Clang verwendet eine BSD-ähnliche Lizenz, die „University of Illinois/NCSA Open Source License“.[1]

4.2. Ansatz mit Clang

Nach der Vorstellung beider Ansätze vor Mitarbeitern der Abteilungen Software Engineering und Programmiersprachen und Übersetzerbau des Fachbereichs Informatik an der Universität Stuttgart ist die Entscheidung für Clang gefallen.

Clang ist ein vollständiges C/C++ Frontend und damit in seiner Komplexität nicht zu unterschätzen. Es existieren mehrere APIs, mit denen Clang angesprochen werden kann. Speziell für Refactoring-Tools wird die sogenannte LibTooling API bereitgestellt. Zum Zeitpunkt der Arbeit ist diese API noch nicht ausgereift.

Es ist auch möglich, den Parser Clangs direkt aufzurufen und sich einen Abstract Synatx Tree (AST) des Quellcodes geben zu lassen. Unter anderem enthält er Informationen wie die Originalposition, an der das Token stand, bevor der Präprozessor den Code expandiert hat, und auch Informationen darüber, aus welchen Makros Teile des AST expandiert wurden. Diese Informationen werden für das GBT-Modell CodeCovers gebraucht, um die Überdeckungsinformationen an der richtigen Stelle anzuzeigen.

Um Code zu manipulieren, existiert der sogenannte Rewriter. Mit Hilfe dieser Klasse lässt sich zusätzlicher Code in den Originalcode einfügen. Code, der aus einem Makro expandiert wurde, lässt sich allerdings nicht manipulieren. Dies ist aber eine harte Anforderung des Instrumentierers.

Deswegen wurden verschiedene Ansätze verfolgt, diesen Umstand zu beheben. Schließlich stellte sich dieser Ansatz als nicht durchführbar heraus. Nachdem abzusehen war, dass Clang die Anforderungen entgegen der ursprünglichen Erwartung nicht erfüllt, wurde der Fokus auf die Implementierunge mittels JavaCC gelegt.

[1] `http://opensource.org/licenses/UoI-NCSA.php`

4.3. Implementierung mit JavaCC

Der zweite Ansatz umfasst das Verwenden eines bereits vorhandenen Präprozessors in Kombination mit einem von JavaCC generierten Parser.

Eine geeignete Lösung dafür ist JCPP (Java C Preprocessor)[2], eine Implementierung des C-Präprozessors in Java. JCPP kennt jedoch nur Angaben zur Zeile und Spalte, um Tokens zurückzuverfolgen. Deshalb musste JCPP um die Unterstützung von Offsets, wie sie in CodeCover verwendet werden, erweitert werden. Durch eine Adaptionsschicht können anschließend die Tokens, die JCPP ausgibt, als Eingabe für JavaCC verwendet werden.

Es existiert auch bereits eine C Grammatik aus dem Jahr 1997 für JavaCC. Diese Grammatik hält sich nahe an den Grammatikregeln des C89 Standard und wurde überarbeitet, damit sie dem C11 Standard entspricht. Dabei wurde darauf geachtet, dass die Grammatikregeln möglichst nah dem Standard folgen, damit eine spätere Überarbeitung möglichst einfach ist.

4.4. Ablauf des Parsens, GBT-Modellerstellung und Instrumentierung

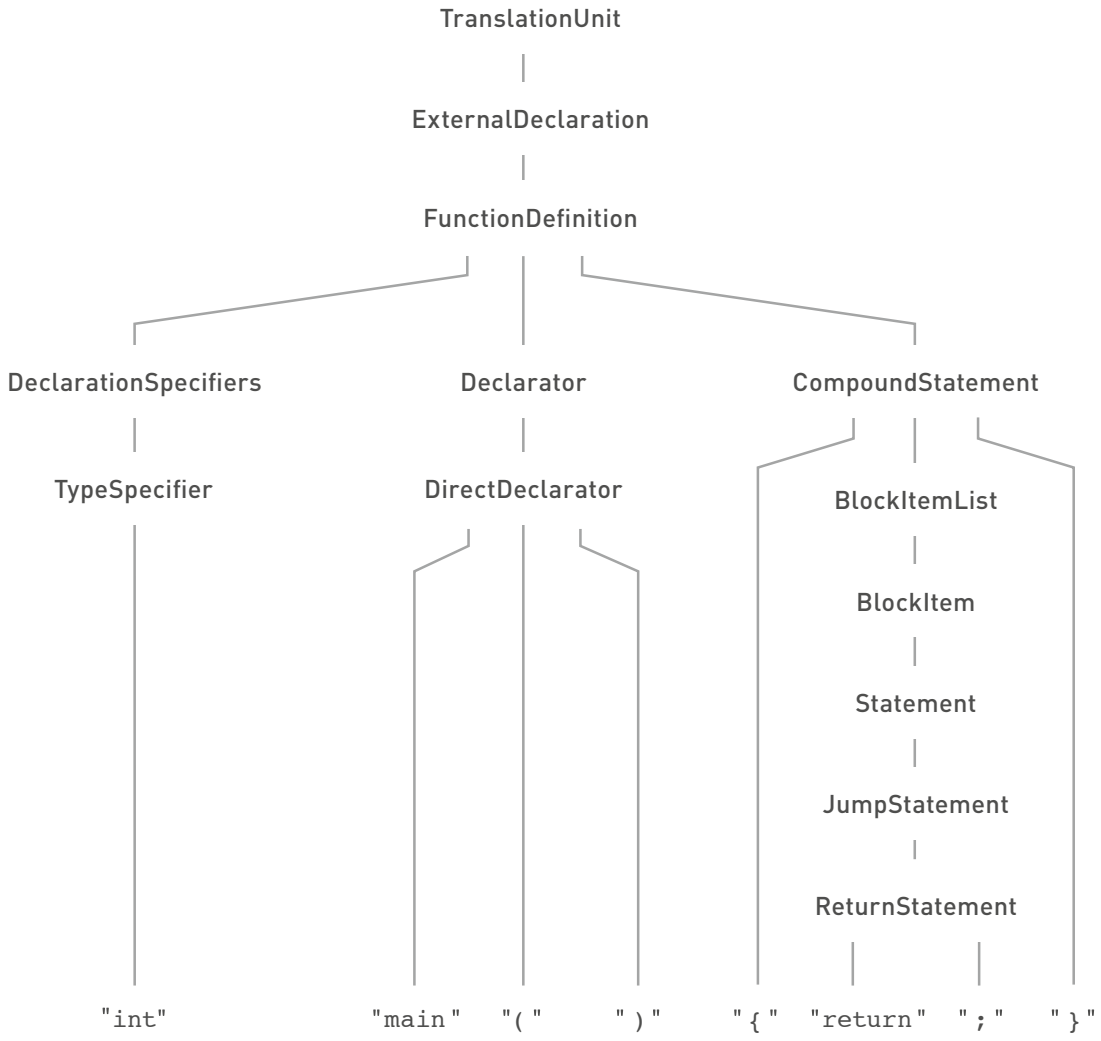
Nachdem der Präprozessor den Code expandiert hat und der Parser den Code in AST-Form gebracht hat (siehe Abbildung 4.1), übernehmen zwei Visitor die GBT-Modellerstellung und Instrumentierung. Die Implementierung folgt in großen Teilen der Visitor-Klasse des Java Frontends.

> Das Visitor-Pattern ist ein Entwurfsmuster und wurde von Gamma et al. [Gam95] geprägt. Das Vistor-Pattern erlaubt es, neue Operationen zu Klassenhierarchien hinzuzufügen, ohne diese selbst ändern zu müssen. Die Idee dahinter ist, eine sogenannte **accept**-Methode zu jeder Klasse hinzuzufügen und die eigentliche Operation in sogenannte Visitor-Klassen auszulagern. Dadurch können leicht neue Visitor-Klassen geschrieben werden, ohne dabei die Klassenhierarchie anpassen zu müssen.

Bei der ersten Traversierung (Listing 4.1) wird jeder Anweisung, Bedingung und jedem Zweig eine eindeutige ID zugewiesen und die Struktur des AST in das GBT-Modell CodeCovers (MAST) transformiert.

Bei der zweiten Traversierung (Listing 4.2) wird der im AST gespeicherte Code instrumentiert. Das Instrumentieren übernehmen dabei für jede Überdeckungsart eigene Klassen (Listing 4.3). Dabei

[2] `http://www.anarres.org/projects/jcpp/`



Der AST, erstellt durch den von JavaCC generierten Parser. In der untersten Zeile stehen die Tokens aus dem Quellcode. Darüber sind die Knoten, die durch die Produktionsregeln der Grammatik vorgegebenen werden.

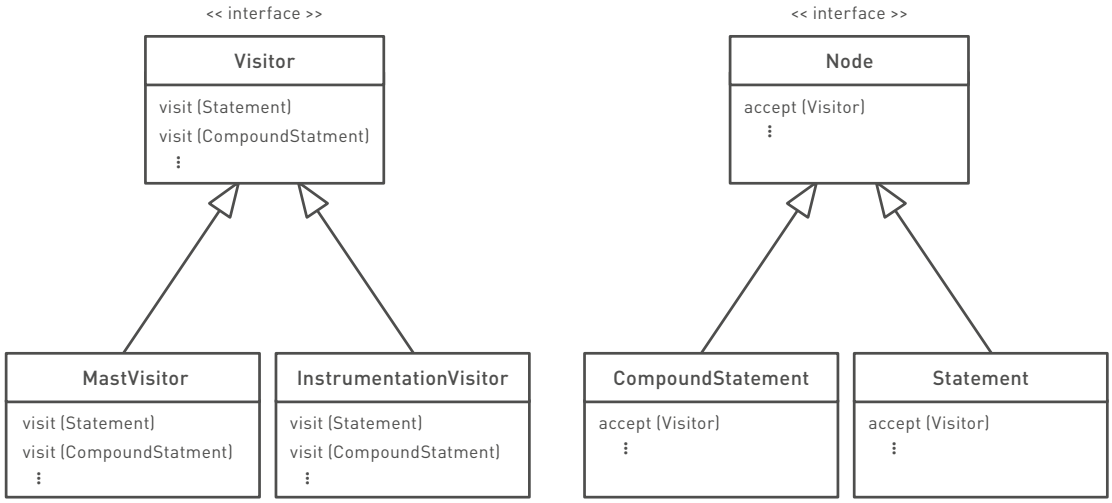
Abbildung 4.1.: Abstract Syntax Tree (AST)

werden vor jeder Anweisung, Zweig oder Bedingung Zähler eingefügt, die bei jedem Durchlauf erhöht werden. Diese Zähler werden pro Quelldatei in einem Array gebündelt.

Das Aufteilen in zwei Visitor-Klassen hat den Vorteil, dass die Komplexität der Visitor-Klassen deutlich sinkt. Beide Klassen enthalten jeweils 200 und 400 Zeilen Code. Außerdem sinkt auch die Komplexität der einzelnen Funktionen, so dass die meisten Funktionen weniger als 15 Zeilen und maximal 25 Zeilen enthalten. Im Vergleich dazu umfasst der Visitor des Java-Frontends 1300

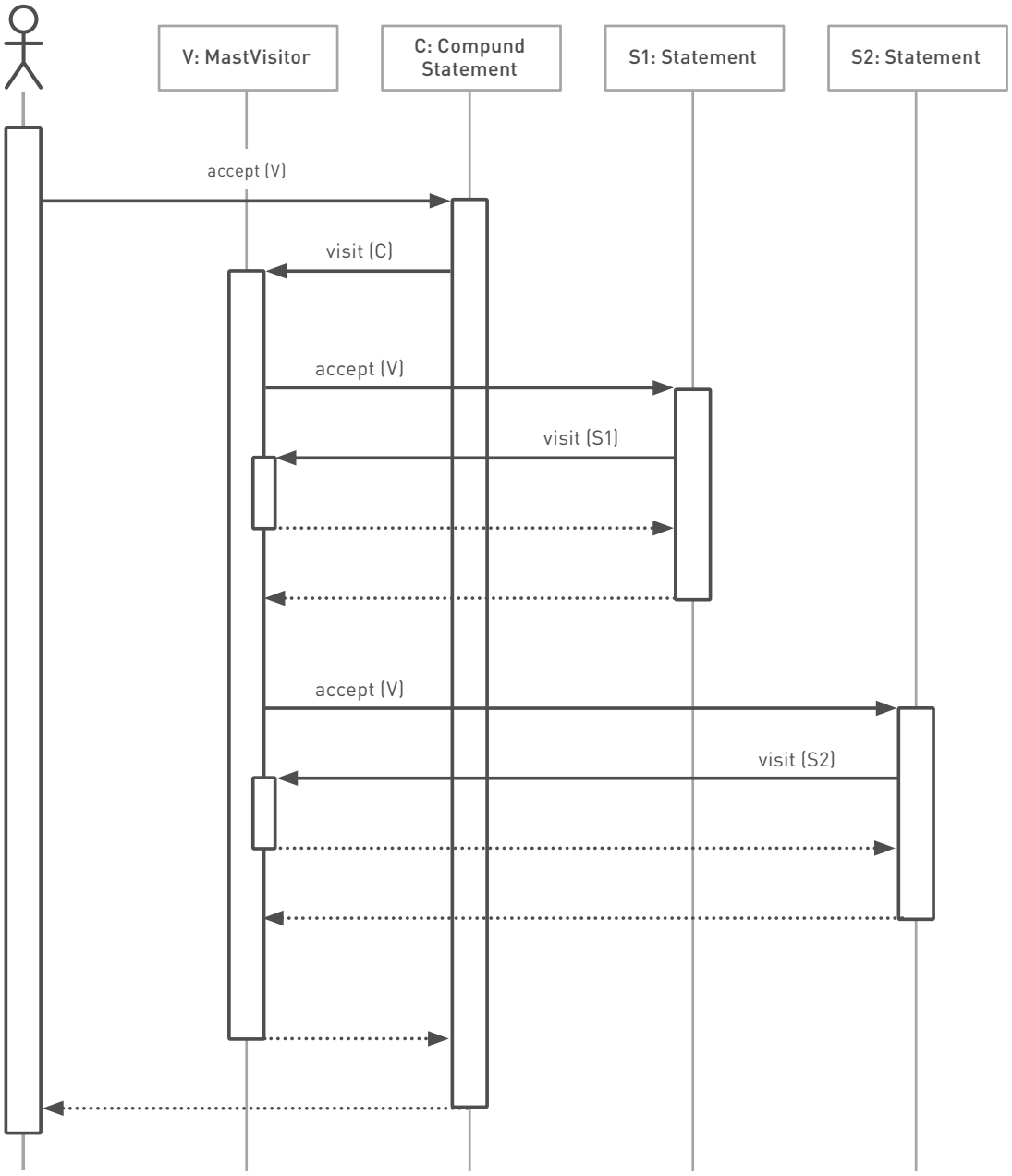
Zeilen Code und die längste Funktion 150 Zeilen.

Im Anschluss an die Instrumentierung kann das Programm ausgeführt werden und das Ausführungsprotokoll mit CodeCover ausgewertet werden. Die spezielle Visualisierung der Makros im Überdeckungsbericht wurde aus zeitlichen Gründen nicht umgesetzt.



Dargestellt ist die für das Vistor-Pattern notwendige Struktur.

Abbildung 4.2.: Auszug aus der Klassenhierachie des Frontends



Die eigentlich Erstellung des GBT-Modells findet im MastVisitor statt. Die einzelnen **accept**-Methoden der anderen Klassen steuern den Ablauf.

Abbildung 4.3.: Schematischer Ablauf der GBT-Modellerstellung

Listing 4.1: Auszug aus MastVisitor.java

```
public void visit(IfStatement n) {
  List<Branch> branchList = new ArrayList<Branch>(2);
  n.condID = cm.newCondID();
  n.branchID = cm.newBranchID();
  // another ID for the else part
  cm.newBranchID();

  n.nodeToken.accept(this);
  n.nodeToken1.accept(this);
  n.terms = expressionParser.visit(n.expression);
  RootTerm rootTerm = builder.createRootTerm(
         n.terms.toBooleanTerm(builder, sourceFile),
         createCoverableItem(cm.condID(n.condID)));
  n.expression.accept(this);
  n.nodeToken2.accept(this);

  pushStatementLevel();
  n.statement.accept(this);
  branchList.add(createBranch(cm.branchID(n.branchID),
         BeginOffset.getStartOffset(n.statement),
         lastEndOffset, -1, -1, false));

  pushStatementLevel();

  if(n.nodeOptional.present()) {
    n.nodeOptional.accept(this);
    NodeToken elseNode = (NodeToken)
          ((NodeSequence)n.nodeOptional.node).elementAt(0);
    branchList.add(createBranch(cm.branchID(n.branchID +1),
           BeginOffset.getStartOffset(n.nodeOptional), lastEndOffset,
           elseNode.beginOffset, elseNode.endOffset,
           false));
  } else {
    branchList.add(createBranch(cm.branchID(n.branchID +1),
           -1,-1, -1,-1, true));
  }

  createConditionalStatement(
         cm.stmtID(((Statement)n.getParent().getParent()).stmtID),
         n.nodeToken.beginOffset, lastEndOffset,
         rootTerm, branchList,
         n.nodeToken.beginOffset, n.nodeToken.endOffset
  );
}
```

Listing 4.2: Auszug aus InstrumentationVisitor.java

```
@Override
public void visit(IfStatement n) {
  // We need another block because the
  // condition manipulator adds a tmp variable
  out.println("{");
  InstrBooleanTerm term = conditionManipulator.visit(out, n);

  n.nodeToken.accept(this);
  n.nodeToken1.accept(this);
  try {
    term.writeToTarget(out);
  } catch (IOException e) {
    e.printStackTrace();
  }
  n.nodeToken2.accept(this);
  out.println("{");
  branchManipulator.visit(out, n);
  n.statement.accept(this);
  out.println("} else {");
  branchManipulator.visitElse(out, n);
  if ( n.nodeOptional.present() ) {
    // skip the "else" and visit the else body directly
    ((NodeSequence)n.nodeOptional.node).elementAt(1).accept(this);
  }
  out.println("}");
  out.println("}");
}
```

Listing 4.3: Auszug aus DefaultBranchManipulator.java

```
@Override
public void visit(PrintWriter out, CCNode n) {
  out.format("%s[%d]++;\n", cm.branchVarName(), n.branchID);
}

@Override
public void visitElse(PrintWriter out, CCNode n) {
  out.format("%s[%d]++;\n", cm.branchVarName(), n.branchID + 1);
}
```

5. Erprobung

Zur Erprobung der Implementierung werden Beispiele aus den Übungen von R. Schmidberger herangezogen. Das erste Beispiel zeigt einen Überdeckungsbericht mit Bedingungsüberdeckung und das zweite Beispiel einen Überdeckungsbericht mit Anweisungs-, Zweig- und Schleifenüberdeckung.

	Term Coverage		
program	3 / 6	50 %	
getriebe.c	3 / 6	50 %	
aktiviereAbstandsanzeige	3 / 6	50 %	

```
1   int aktiviereAbstandsanzeige(
2           int getriebeStellung,
3           int geschwindigkeit,
4           int abstand)
5   {
6     if(getriebeStellung == 0 ||
7        geschwindigkeit < 2 ||
8        abstand < 120) {
9       return 1;
10    }
11    return 0;
12  }
```

Testfall	Belegung
1	(0, 1, 200)
2	(0, 50, 200)
3	(1, 50, 200)
4	(1, 1, 200)

Abbildung 5.1.: Bedingungsüberdeckung mit 4 Testfällen

	Term Coverage		
program	4 / 6	66 %	
getriebe.c	4 / 6	66 %	
aktiviereAbstandsanzeige	4 / 6	66 %	

```
1   int aktiviereAbstandsanzeige(
2           int getriebeStellung,
3           int geschwindigkeit,
4           int abstand)
5   {
6     if(getriebeStellung == 0 ||
7        geschwindigkeit < 2 ||
8        abstand < 120) {
9       return 1;
10    }
11    return 0;
12  }
```

Testfall	Belegung
1	(0, 1, 200)
2	(0, 50, 200)
3	(1, 50, 200)
4	(1, 1, 200)
5	(1, 50, 1)

Abbildung 5.2.: Bedingungsüberdeckung mit 5 Testfällen

	Statement Coverage		Branch Coverage		Loop Coverage	
program	2 / 6	33 %	3 / 6	50 %	2 / 6	33 %
sort.c	2 / 6	33 %	3 / 6	50 %	2 / 6	33 %
sort	2 / 6	33 %	3 / 6	50 %	2 / 6	33 %

```
1  #define SWAP(x,y) int t; t=x;x=y;y=t;
2  #define SORT(x,y) if(x>y) {SWAP(x,y)}
3
4  void sort (int *array, int length)
5  {
6    int n, i, getauscht;
7    for (n = length - 1; n > 0; n--) {
8      // Annahme: nicht getauscht
9      getauscht = 0;
10     for (i = 0; i < n; i++) {
11       if (array[i] > array[i + 1]) {
12         // es wurde getauscht
13         getauscht = 1;
14       }
15       SORT (array[i], array[i + 1])
16     }
17     if (!getauscht) {
18       // es wurde nicht getauscht:
19       // das array ist sortiert
20       break;
21     }
22   }
23 }
```

Abbildung 5.3.: Anweisungs-, Zweig-, und Schleifenüberdeckung, Testfall mit **array** = {1,2,3}

	Statement Coverage		Branch Coverage		Loop Coverage	
program	5 / 6	83 %	3 / 6	50 %	3 / 6	50 %
sort.c	5 / 6	83 %	3 / 6	50 %	3 / 6	50 %
sort	5 / 6	83 %	3 / 6	50 %	3 / 6	50 %

```
1  #define SWAP(x,y) int t; t=x;x=y;y=t;
2  #define SORT(x,y) if(x>y) {SWAP(x,y)}
3
4  void sort (int *array, int length)
5  {
6    int n, i, getauscht;
7    for (n = length - 1; n > 0; n--) {
8      // Annahme: nicht getauscht
9      getauscht = 0;
10     for (i = 0; i < n; i++) {
11       if (array[i] > array[i + 1]) {
12         // es wurde getauscht
13         getauscht = 1;
14       }
15       SORT (array[i], array[i + 1])
16     }
17     if (!getauscht) {
18       // es wurde nicht getauscht:
19       // das array ist sortiert
20       break;
21     }
22   }
23 }
```

Abbildung 5.4.: Anweisungs-, Zweig-, und Schleifenüberdeckung, Testfall mit **array** = {3,2,1}

	Statement Coverage		Branch Coverage		Loop Coverage	
program	6 / 6	100 %	6 / 6	100 %	3 / 6	50 %
sort.c	6 / 6	100 %	6 / 6	100 %	3 / 6	50 %
sort	6 / 6	100 %	6 / 6	100 %	3 / 6	50 %

```
1  #define SWAP(x,y) int t; t=x;x=y;y=t;
2  #define SORT(x,y) if(x>y) {SWAP(x,y)}
3
4  void sort (int *array, int length)
5  {
6    int n, i, getauscht;
7    for (n = length - 1; n > 0; n--) {
8      // Annahme: nicht getauscht
9      getauscht = 0;
10     for (i = 0; i < n; i++) {
11       if (array[i] > array[i + 1]) {
12         // es wurde getauscht
13         getauscht = 1;
14       }
15       SORT (array[i], array[i + 1])
16     }
17     if (!getauscht) {
18       // es wurde nicht getauscht:
19       // das array ist sortiert
20       break;
21     }
22   }
23 }
```

Abbildung 5.5.: Anweisungs-, Zweig-, und Schleifenüberdeckung, Testfall mit **array** = {2,1,3}

6. Rückblick

Die Unterstützung für die Programmiersprache C als Frontend für CodeCover konnte erfolgreich implementiert und anhand von Beispielfällen validiert werden. Dabei hat sich der Einsatz des Parsers von JavaCC gegenüber dem von Clang als geeigneter erwiesen. Die Erweiterung CodeCovers ist aufgrund der guten Dokumentation mit vergleichsweise geringem Aufwand möglich. Weiterhin wünschenswert wäre die besondere Visualisierung der Makros im Testbericht, die nicht vollständig umgesetzt wurde.

Außerdem deutet das erfolgreiche Testen des Frontends mit verschiedenen Compilern auf unterschiedlichen Plattformen auf eine gute Praxistauglichkeit hin, eine endgültige Bewertung lässt sich durch eine Erprobung mit Industriepartnern herstellen.

A. Anhang

A.1. Projektplan und Verlauf

Eine Bachelorarbeit umfasst 12 ECTS Credits. Dies entspricht einem Arbeitsaufwand von 360 Stunden. Jeweils ein Drittel der Zeit sind für die Recherche, die Implementierung und die Erstellung des Berichts vorgesehen. Die Implementierung erfolgt im Time-Boxing-Verfahren.

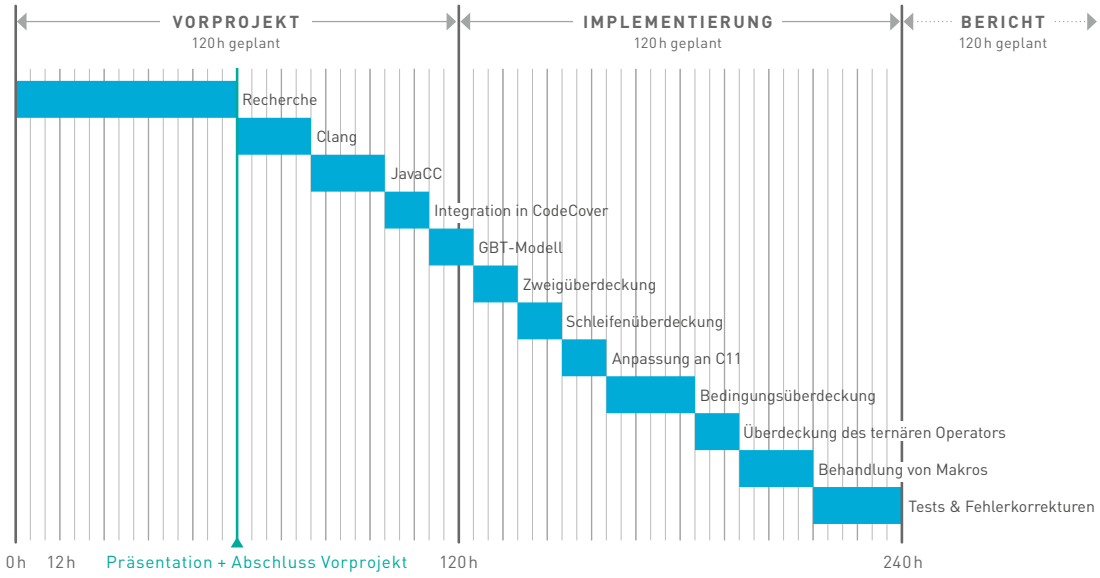


Abbildung A.1.: Verlauf des Projekts

Literaturverzeichnis

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass. [u.a.], 2. print. edition, 1995.

[HL92] J.R. Horgan and S. London. A data flow coverage testing tool for c. In *Assessment of Quality Software Development Tools, 1992., Proceedings of the Second Symposium on*, pages 2–10, 1992.

[ISO11] ISO. Programming languages – C. ISO 9899:2011, International Organization for Standardization, Geneva, Switzerland, 12 2011.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[KR90] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C: mit dem C-Reference Manual in deutscher Sprache*. Hanser, Prentice-Hall Internat., München, 2. ausg., ansi c edition, 1990.

[LL07] Jochen Ludewig and Horst Lichter. *Software-Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verl., Heidelberg, 1. aufl. edition, 2007.

[LS94] P.E. Livadas and D.T. Small. Understanding code containing preprocessor constructs. In *Program Comprehension, 1994. Proceedings., IEEE Third Workshop on*, pages 89–97, 1994.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Steffen Hanikel)