Code Coverage mit gcov

Seminararbeit zur Veranstaltung
Verteilte und parallele Systeme II
bei Prof. Dr. Rudolf Berrendorf

Von Jan Seidel, Stefan Winarzki

Fachbereich Informatik
Fachhochschule Bonn-Rhein-Sieg, St. Augustin

Inhaltsverzeichnis:

1 Einleitung

Diese Seminararbeit wurde im Rahmen der Lehrveranstaltung „Verteilte und parallele Systeme II“ des Wintersemesters 2003 / 2004 erstellt. Sie befasst sich mit Code Coverage im Allgemeinen und mit dem Thema, die zahlreichen Aspekte, die der Begriff Code Coverage umfasst, mit dem Test Coverage Programm **gcov** zu realisieren.

1.1 Begriffsdefinitionen

Code Coverage - Nachdem man ein Programm geschrieben hat, möchte man in der Regel dessen korrekten Ablauf mit Hilfe von Testprogrammen und darin enthaltenen Testfällen überprüfen. Dabei lässt man das Programm ausführen und vergleicht die daraus resultierenden Daten mit erwarteten Werten. Allerdings nützt es dem Programmierer nicht viel, wenn sämtliche Tests zwar erfolgreich beendet wurden, jedoch, nur 25% des Testprogrammcodes ausgeführt wurde. Der Begriff 'Code Coverage' bedeutet, zu protokollieren, wie oft welche Zeile eines Programms ausgeführt wurde und so lässt sich ein Programm, falls es nicht die gewünschten Befehle ausführt, wesentlich leichter modifizieren, als wenn man gar keine Kontrollmöglichkeit hätte. Code Coverage ist eine von mehreren Möglichkeiten, um ein Testprogramm so effektiv und umfassend wie möglich zu gestalten. Dass keine Fehler in einem Programm existieren, kann man zwar nie hundertprozentig ausschließen, jedoch kann man die Wahrscheinlichkeit für deren Auftreten mittels Code Coverage minimieren.

Es gibt verschiedene Möglichkeiten, um die Qualität bzw. Funktionalität und Effektivität eines Testes zu analysieren, bspw. das sog. 'Statement Coverage', das lediglich angibt, ob eine Anweisung ausgeführt wurde oder nicht. Weitere Kriterien werden in Kapitel 2.2 eingehender behandelt.

gcov - gcov ist ein Programm, das die durch den Begriff Code Coverage bestimmten Eigenschaften und benötigten Funktionen umsetzt, also ein sog. Test Coverage Programm. Sein Ziel ist es, die Wahrscheinlichkeit, dass ein Code Fehler enthält, zu minimieren und Testprogramme effizienter zu gestalten. Auf die Handhabung und Funktionen von gcov wird in Kapitel 3 ausführlich eingegangen.

1.2 Ziele von Code Coverage

Wie im vorhergehenden Kapitel schon kurz angesprochen, soll Code Coverage die Funktionalität und Effizienz von Testprogrammen verbessern. Dieses Ziel erreicht es, indem Bereiche eines Programms, die nicht durch Testfälle des Testprogramms abgedeckt werden, ermittelt werden. Nun kann der Programmierer diese Ergebnisse auswerten und neue Testfälle entwerfen, die diese Lücken beheben sollen. Code Coverage allein kann dies natürlich nicht bewirken, jedoch hilft es dem Programmierer, sich diese Arbeit zu erleichtern. Durch die quantitative Erhöhung des getesteten Programmcodes wird folglich die Qualität des Programms erhöht.

2 Code Coverage

2.1 Beschreibung / Abgrenzung

Mit Code Coverage lässt sich die Überdeckung, also wie oft welche Zeile bzw. Anweisung eines Programms ausgeführt wurde, feststellen. Ein möglichst hoher Überdeckungsgrad ist zwar wünschenswert um die in Kapitel 1.2 genannten Ziele zu erreichen, jedoch bedeutet auch ein hundertprozentiger Überdeckungsgrad nicht automatisch, dass das Programm vollständig getestet oder gar fehlerfrei ist. Anhand von einem kurzen Stück Beispielcode wird diese Aussage verdeutlicht:

```
10: int x = testMethod();  /* testMethod() wird ausgeführt, x wird der
                              Rückgabewert (hier ein Fehlercode) zugewiesen */
11: if(x == FATAL_ERROR)
12:   exit(5); // Programmende mit Fehlercode 5
13: else
14:   ... // Kein Fehler, Programm fährt fort
```

Nehmen wir an, in diesem Beispiel wird Zeile 12 nie ausgeführt. Das könnte entweder daran liegen, dass wirklich niemals ein Fehler auftritt und das Programm korrekt läuft, aber es könnte auch sein, dass ein anderer Fehlercode vorliegt, den der Programmierer nicht bedacht hat. Im zweiten Fall hätte man also trotz fehlerhafter Ausführung des Programms einen sehr hohen Überdeckungsgrad von ca. 90%. Dieser Fehler kann folgendermaßen behoben werden:

```
10: int x = testMethod();  /* testMethod() wird ausgeführt, x wird der
                              Rückgabewert (hier ein Fehlercode) zugewiesen */
11: if(x == FATAL_ERROR)
12:   exit(5); // Programmende mit Fehlercode 5
13: else if(x == RECOVERABLE_ERROR) {
14:   ... // Fehler beheben
15: }
16: else
17:   ... // Kein Fehler, Programm fährt fort
```

Nun hätte man die fehlende Fallunterscheidung korrigiert, gleichzeitig den Grad der Codeüberdeckung vermindert und dennoch einen Fehler des Programms behoben. Das liegt vor allem daran, dass Code Coverage Tools nur den vorhandenen Code analysieren können - wie wir im Beispiel gesehen haben, werden fehlende Codestücke, wie die hier nicht vorhandene Fallunterscheidung, folglich nicht berücksichtigt, so dass sich der Programmierer nicht nur auf das Code Coverage Programm und den Überdeckungsgrad allein verlassen kann. Das soll jedoch nicht heißen, dass der Überdeckungsgrad unwichtig ist. Anhand einer fehlenden Überdeckung einer Codekomponente lässt sich zwar nicht sagen, dass dieser Fehler enthält, jedoch kann man daraus schließen, dass das Stück Code nicht ausreichend betrachtet bzw. getestet wurde. Es sei hier nochmals gesagt, dass ein Test Coverage Programm nicht das Denken des Menschen ersetzen, sondern nur eine Hilfestellung darstellen soll.

Der Fehler, der im obigen Beispiel auftrat, gehört zu den Unterlassungsfehlern oder auch Faults of Omission. Unter diese Fehlerklasse fallen u. a. unvollständige Fallunterscheidungen (fehlende Bedingungen in IF-Abfragen, fehlender ELSE-Zweig,

fehlender case in switch, …), fehlende catch-Anweisungen für Exceptions und andere wesentlich komplexere Fälle.

Die komplette Abwesenheit von Fehlern in einem Programm kann man ohnehin nie mit absoluter Sicherheit festlegen, da es einfach nicht gelingen kann, für **alle** auftretenden Fälle Testfälle in einem Testprogramm zu schreiben. Auch die Vollständigkeit des Codes ist mit Code Coverage allein nicht zu zeigen. Allerdings kann der Programmierer aus den Analysedaten, die das Code Coverage Programm ihm liefert, schließen, welche Segmente eines Programms noch nicht ausreichend getestet wurden und somit eine entsprechende Test-Methode schreiben oder vorhandene Fehler korrigieren. Wie wir im obigem Beispiel gesehen haben, ist die Maßnahme, eine neue Test-Methode zu schreiben, nicht immer die beste Lösung. Schließlich kann es auch vorkommen, dass ein Fehler in der Programmlogik vorliegt, daher sollte der Programmierer bei nicht ausreichender Überdeckung einer Code-Region, diese eingehender betrachten und nach Fehlern suchen, anstatt einfach die Codeüberdeckung und damit scheinbar auch die Codekorrektheit und -vollständigkeit zu erhöhen, indem er eine neue Test-Methode entwirft.

Natürlich kann man das Testprogramm mit Hilfe einer Code Coverage Analyse nicht nur erweitern, sondern auch reduzieren. Wenn man feststellt, dass bestimmte Bereiche des zu testenden Programmes, hervorgerufen durch redundante Testmethoden, viel zu oft mit den selben Daten ausgeführt werden, kann man dies relativ leicht erkennen und die entsprechenden Methoden modifizieren oder löschen.

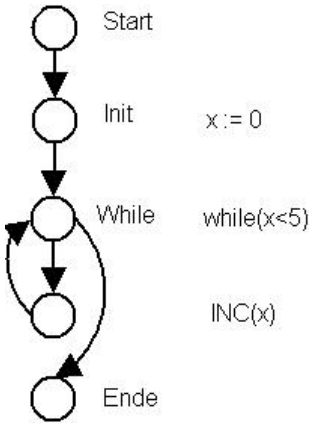
2.2 Kontrollflussgraphen

Um den Ablauf eines Programmes und die Kriterien der verschiedenen Überdeckungsmaße anschaulicher darstellen zu können, verwenden wir in den folgenden Kapiteln sog. Kontrollflussgraphen. Der Kontrollflussgraph ist ein gerichteter Graph mit festem Anfangs- und Endpunkt. Jeder Knoten stellt eine ausführbare Instruktion dar, während die Kanten anzeigen, wo das Programm mit der Ausführung fortfährt. Die Verzweigung der Kanten ist abhängig von Anweisungsfolgen (einfache Ausführung einer Instruktion), Selektionen (z. B. IF- oder switch-Anweisungen) und Iterationen (Schleifen). Das folgende Beispiel soll dies verdeutlichen:

Quellcode:

```
10: ...
11: int x = 0;
12: while(x<5)
13:   x++;
14: ...
```

Kontrollflussgraph:



2.3 Verschiedene Möglichkeiten

Es gibt eine große Anzahl von Codeüberdeckungsmaßen. Wir wollen hier nur die wichtigsten betrachten und deren Stärken und Schwächen verdeutlichen.

2.3.1 Statement Coverage (Anweisungsüberdeckung)

Statement Coverage überprüft und protokolliert, welche Instruktionen eines Programms ausgeführt wurden, und welche nicht. Vollständige Anweisungsüberdeckung wird erreicht, falls jede Anweisung eines Programms mindest einmal ausgeführt wurde, d.h. alle Knoten des Kontrollflussgraphen mindestens einmal besucht wurden. Dazu braucht man, falls mindestens eine IF- oder switch-Anweisung im Programm vorkommt, mehrere Testfälle, da es mit nur einem Testfall nicht möglich wäre, sämtliche Anweisungen auszuführen. Nach folgender Formel hat der Anweisungsüberdeckungsgrad sein Maximum bei 1 und Minimum bei 0:

Anweisungsüberdeckungsgrad := Anzahl der besuchten Knoten / Anzahl aller Knoten

Falls eine Sequenz von Instruktionen ohne Verzweigungen vorkommt, könnte man theoretisch diese als eine Instruktion werten bzw. mehrere Knoten in einer Reihe zu einem Knoten zusammenfassen, da ohnehin entweder keine oder alle Anweisungen ausgeführt werden. I.d.R. wird aber dennoch jede Zeile bzw. Instruktion für sich gezählt, da es die Auswertung von statistischen Zusammenfassungen eines Programmablaufs erleichtert.

Diese Art der Überdeckung ist eher schwach, da trotz hundertprozentiger Anweisungsüberdeckung immer noch schwerwiegende Fehler auftreten können. Einige durch Statement Coverage nicht zu erkennende Fehler können durch andere Überdeckungsmaße erkannt werden, jedoch ist es dazu allein oftmals nicht in der Lage. Dennoch wird es eingesetzt, um bspw. beim ersten Testlauf umfangreicher Programme die gröbsten Fehler zu erkennen und zu beseitigen.

2.3.2 Decision Coverage (Zweigüberdeckung)

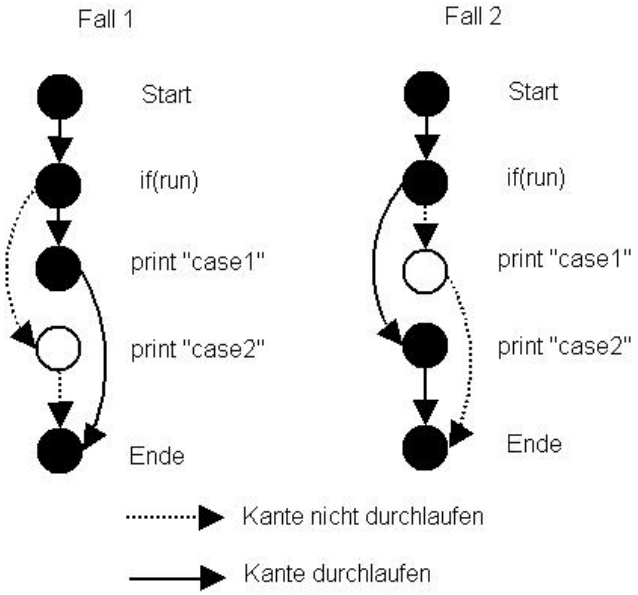
Decision Coverage, auch Branch oder Arc Coverage genannt, wird erreicht, indem alle möglichen Entscheidungen durch IF- oder switch-Anweisungen oder Schleifen mit bestimmten Bedingungen durchlaufen werden, d.h. dass alle Kanten des Kontrollflussgraphen mindestens einmal besucht wurden. Auch bei der hier geltenden Formel gilt wieder das selbe wie für den Anweisungsüberdeckungsgrad. Das Maximum liegt bei 1, das Minimum bei 0:

Zweigüberdeckungsgrad := Anzahl der besuchten Zweige / Anzahl aller Zweige

Im folgenden Beispiel ...

```
10: if (run) {
11:     print "case1";
12: }
13: else {
14:     print "case2";
15: }
```

... wird Decision Coverage nur dann erreicht, falls tatsächlich 2 Fälle durchlaufen werden, in denen ‚run' einmal TRUE und einmal FALSE ist. Diesen Ablauf verdeutlicht der folgende Kontrollflussgraph:



Vollständige Decision Coverage impliziert logischerweise vollständige Statement Coverage. Decision Coverage erhöht die Effektivität der Suche nach Fehlern geringfügig, jedoch bleiben auch bei diesem Maß noch viele Fehler verborgen.

2.3.3 Condition Coverage (Bedingungsüberdeckung)

Bei Condition Coverage wird noch ein weiterer Aspekt berücksichtigt, der bei Decision Coverage ignoriert wurde. Bei Decision Coverage wurde jede Bedingung als genau ein Prädikat gesehen, egal ob es tatsächlich nur eine Bedingung oder aus mehreren atomaren Prädikaten zusammengesetzt und durch logische AND's und OR's getrennt war. Bei Condition Coverage wird dieser Mangel nun beseitigt und jeder einzelne Teil eines Prädikats betrachtet, wobei man zwischen mehreren verschiedenen Arten von Condition Coverage unterscheiden muss.

Minimale Bedingungsüberdeckung wird erreicht, wenn alle atomaren Prädikate einmal zu TRUE und einmal zu FALSE ausgewertet werden.
Bei **Mehrfach-Bedingungsüberdeckung** dagegen müssen alle atomaren Prädikate zu allen möglichen Wertekombinationen ausgewertet werden.
Einen Kompromiss zwischen diesen beiden Überdeckungsmaßen stellt die **Minimale Mehrfach-Bedingungsüberdeckung** dar. Bei diesem Maß muss jedes Prädikat, unabhängig davon ob es atomar oder zusammengesetzt ist, einmal zu TRUE und einmal zu FALSE ausgewertet werden.

2.3.4 Path Coverage (Pfadüberdeckung)

Path Coverage wird erreicht, wenn alle möglichen Wege, beginnend beim Startknoten und endend beim Endknoten, des Kontrollflussgraphen eines Programms durchlaufen werden, d.h. wenn alle möglichen Pfade aller Funktionen durchlaufen werden. Solch ein Pfad ist eine einzigartige Folge von Instruktionen und Zweigen, der vom Programmanfang zum -ende führt.

Dieses Maß zu bestimmen kann ggf. natürlich sehr aufwendig werden, da die Anzahl der möglichen Pfade mit jedem hinzukommenden IF-Statement exponentiell ansteigt. Hat man bspw. 10 IF-Anweisungen, müssten 2^{10} (1024) verschiedene Pfade getestet werden. Fügt man eine weitere IF-Anweisung hinzu, wären es 2^{11} (2048) Pfade.
Ein anderer Nachteil ist, dass bestimmte zu testende Pfade unmöglich zu erreichen sind. Beispiel:

```
10: if (success)
11:    statement1;
12: statement2;
13: if (success)
14:    statement3;
```

Path Coverage würde hier 2^2 (4) Pfade erwarten, was in diesem Fall unmöglich ist. Es gibt lediglich 2 Möglichkeiten, nämlich, dass ‚success' TRUE oder FALSE ist.

3 gcov

3.1 Beschreibung (Realisierung von Code Coverage)

Gcov ist ein Akronym für GNU Coverage und ist ein Teil von gcc, der GNU Compiler Collection. Als solches arbeitet es nur auf Code, der vom GNU CC Compiler erzeugt wurde. Es dient dazu, Code Coverage für C-Code zu ermöglichen.

Gcov ermittelt, welche Zeilen eines Programms ausgeführt wurden und wie oft diese ausgeführt wurden. Da diese Ermittlung zeilenweise auf dem Code arbeitet, sollte das Programm ohne Compiler-Optimierungen kompiliert werden, da mit Optimierungseinstellungen der Compiler eventuell mehrere Zeilen zu einer zusammenfasst. Auf einigen Maschinen ist es beispielsweise möglich, Code wie folgenden

```
if (a != b)
  c = 1;
else
  c = 0;
```

zu einer Zeile zusammenzufassen. Gcov würde dann nur die Häufigkeit der Ausführung des kompletten Blockes berechnen. Dies ist zwar in der Hinsicht korrekt, dass auf dieser Maschine der komplette Block als eine Anweisung ausgeführt wurde, es lässt sich aber nicht feststellen, wie oft die if-Abfrage zu „true“ evaluiert wurde und wie oft zu „false“. Aufgrund der zeilenweisen Abarbeitung der Anweisungen ist es weiterhin empfehlenswert, pro Zeile nur eine Anweisung zu schreiben, da diese anderenfalls auch nicht in ihrer Ausführungshäufigkeit unterschieden werden können.

Um zu ermitteln, welche Teile eines Programms die längste Rechenzeit brauchen, lässt sich gcov in Verbindung mit gprof anwenden (siehe Seminararbeit zu gprof).

Gcov lässt sich zum Einen einsetzen, um Regionen im Code zu finden, die besonders oft ausgeführt werden und somit mögliche sogenannte Bottlenecks zu finden. Werden solche Regionen gefunden, kann versucht werden, den Code an diesen Stellen zu optimieren. Die Bottlenecks im Code befinden sich aber nicht notwendigerweise bei den Anweisungen, die am häufigsten ausgeführt werden. Können beispielsweise Festplattenzugriffe durch Code-Optimierung minimiert werden, kann dies einen sehr großen Effekt für die Ausführgeschwindigkeit des Programms haben, auch wenn die Festplattenzugriffe vorher im Vergleich zu anderen Programmanweisungen nur selten ausgeführt wurden. Aus diesem Grund sollte man sich bei der Optimierungen von Code nicht darauf beschränken, nur die mit gcov ermittelten am häufigsten ausgeführten Zeilen zu optimieren.

Zum Anderen lässt sich gcov in Verbindung mit Testsuites einsetzen, um sicherzustellen, dass der Code ausreichend und korrekt getestet wird. Die Testsuite dient dazu, sicherzustellen, dass der Code fehlerfrei und wie erwartet abläuft, gcov wiederum überwacht sozusagen die Testsuite, indem es feststellt, welche Anweisungen des Programms getestet wurden. Stellt gcov fest, dass bestimmte Bereiche des Programms noch nicht getestet wurden, lassen sich in der Testsuite neue Tests hinzufügen, die den Test möglichst aller Anweisungen des Programms sicherstellen. Eine Testsuite sollte nach Möglichkeit jede Anweisung eines

Programms mindestens ein Mal ausführen, also vollständige Anweisungsüberdeckung garantieren. Dies kann gcov überwachen. Dass damit noch nicht alle durch Code Coverage erkennbaren Fehler entdeckt werden können, wurde bereits in der Beschreibung der unterschiedlichen Coverage-Arten erwähnt, aus diesem Grund bietet gcov weitere Protokollierungsmöglichkeiten an, die weiter unten genauer beschrieben werden.

3.2 Anwendung

3.2.1 Übersetzen des Quellcodes

Um ein Programm mit gcov analysieren zu können, muss es als erstes mit dem GNU CC Compiler und den zusätzlichen Einstellungen –fprofile-arcs und –ftest-coverage kompiliert werden. Dies sieht beispielsweise wie folgt aus:

```
$ gcc -fprofile-arcs –ftest-coverage –o sample sample.c
```

Mit der Option –fprofile-arcs misst der Compiler, wie oft eine Sprunganweisung erreicht wird und wie oft der Sprung vorgenommen wurde. Dazu erstellt GCC einen Programmfluss-Graphen und für diesen Graphen einen Spanning Tree (ein Gerüst). Anweisungen, die nicht in diesem Spanning Tree enthalten sind müssen gezählt werden, da diese Anweisungen nicht auf jeden Fall ausgeführt werden. Für jede Datei, die mit der –fprofile-arcs Option kompiliert wurde, wird nach dem erstmaligen Ausführen eine dateiname.da Datei erstellt, in der der Graph und der Spanning Tree gespeichert sind. Jetzt ist bekannt, welche Sprünge durchgeführt wurden und welche Anweisung wie oft ausgeführt wurde. Mit diesen Informationen kann in der dateiname.gcov Datei die statistische Auswertung erstellt werden.

Die Compiler-Option –ftest-coverage bewirkt, dass zwei weitere für gcov nötige Dateien zu jeder kompilierten Datei erstellt werden, eine mit der Endung .bb und die andere mit .bbg. Die .bb Datei ordnet den sogenannten „basic blocks“, das sind Folgen von Anweisungen ohne Sprünge und Sprungziele, Zeilennummern in der Quellcode-Datei zu, um die Anzahl der Ausführungen für jede Zeile angeben zu können. Die .bbg Datei erstellt und verwaltet eine Liste aller „arcs“ im Programmfluss-Graphen, also Sprungbefehle von einem Block in einen anderen. Damit wird gcov ermöglicht, den Graphen zur Analyse zu rekonstruieren und alle vorgenommenen Sprünge zu berechnen.

Die Option –g des GNU C Compilers ist nicht gcov-spezifisch, wird aber bei Code Coverage mit gcov trotzdem häufig benötigt. Mit ihr erzeugt GCC Debugging-Informationen für den Debugger gdb. Mit diesem lassen sich unter anderem im Code Stopps („breaks“) setzen und so beispielsweise fehlgeschlagene Speicher-Alloziierungen simulieren. Springt man nach solch einer fehlgeschlagenen Alloziierung zu einer Anweisung, die auf diese Speicherstellen zugreift, lässt sich der korrekte Umgang mit Fehlern testen. So lassen sich also auch Zeilen des Programms testen, die der Ausnahmebehandlung dienen und im normalen Abarbeiten eines Programms nicht erreicht werden. Um eine vollständige Anweisungsüberdeckung zu erhalten, kann es also nötig sein, den gdb-Debugger zu verwenden. Die genauere Benutzung des Debuggers wird hier nicht weiter beschrieben, da sie nicht Thema dieser Ausarbeitung ist.

3.2.2 Ausführung des Programms

Um festzustellen, welche Anweisungen des Programms wie oft ausgeführt wurden, muss dieses nun mindestens ein Mal gestartet werden, im Falle des oben angeführten Beispiels also wie folgt:

```
$ ./sample 100
```

Nun wird für jede Quelldatei, die mit der Option –fprofile-arcs kompiliert wurde, die Datei dateiname.da erstellt, die den Programmflussgraphen enthält und somit speichert, welche Anweisungen ausgeführt und welche Sprünge durchgeführt wurden. Startet man das Programm mehrmals, beispielsweise im Rahmen einer Testsuite, mit unterschiedlichen Eingabeparametern, werden die Ergebnisse in der dateiname.da-Datei aufsummiert. Das Format dieser Datei ist sehr simpel. Zu den einzelnen Anweisungen und Sprüngen wird einfach die Anzahl ihrer Ausführungen als 8-byte-Wert gespeichert.

3.2.3 Starten von gcov / Auswerten der Ergebnisse

Nun sind alle benötigten Daten für gcov vorhanden, um die Auswertungsstatistiken erstellen können. Dafür wird gcov auf der Kommandozeile aufgerufen, mit der Quelldatei, für die die Auswertung erstellt werden soll als Parameter.

```
$ gcov sample.c
 63.64% of 22 source lines executed in file sample.c
Creating sample.c.gcov.
```

In diesem Beispiel wurden also 63.64 % der 22 Zeilen ausgeführt, dies sind 14 von 22 Zeilen. Weiterhin wurde die Datei test.c.gcov erstellt, die im Grunde wie die Original-Quelltext-Datei aussieht, mit der Ausnahme, dass vor jeder Zeile die Ausführungshäufigkeit notiert wurde. Einige Konstrukte, wie „includes“ und geschlossene Klammern sind hier ausgenommen, da an diesen Stellen keine Maschinenanweisungen ausgeführt werden. Eine genauere Erläuterung solch einer .gcov-Datei folgt in Kapitel 3.3. Folgende Optionen existieren für gcov:

-b (branch-probabilities):
Diese Option veranlasst gcov, zusätzliche Statistiken darüber zu erstellen, welche Verzweigungen im Programmcode wie oft durchlaufen wurden. Dies wird als Prozentzahl angegeben. Die Ausgabe sieht dann wie folgt aus:

```
$ gcov –b sample.c
 63.64% of 22 source lines executed in file sample.c
100.00% of 7 branches executed in file sample.c
100.00% of 7 branches taken at least once in file sample.c
 33.33% of 12 calls executed in file sample.c
Creating sample.c.gcov.
```

In diesem Beispiel wurden alle 7 Verzweigungen („branches“) im Programmcode erreicht und jeweils mindestens ein Mal ausgeführt. Von den 12 Funktionsaufrufe wurden nur 33.33 % ausgeführt, also 4 der 12 Aufrufe. In der Datei sample.c.gcov lässt sich nun genauer sehen, welche Verzweigungen und Aufrufe ausgeführt wurden. Dies kann beispielsweise wie folgt aussehen:

```
           1     if(argc != 2) {
   branch 0 taken = 100%
        #####       printf("Usage: %s Enter arraysize value \n",argv[0]);
   call 0 never executed
        #####       exit(-1);
   call 0 never executed
                  }
```

Ausschnitt sample.c.gcov

Bei diesem Beispiel wurde also in 100 % aller Fälle hinter den Block nach der if-Abfrage gesprungen und demzufolge die beiden Funktionsaufrufe in dem Block nie aufgerufen. So lässt sich das in Kapitel 2.3.2. beschriebene Decision Coverage überwachen, indem man feststellen kann, ob und wie oft jeder Sprung durchgeführt wurde. Für Funktionsaufrufe wird die Prozentzahl ihrer Rückkehren in das Programm gezählt. Bei Aufruf der exit()-Funktion erfolgt beispielsweise keine Rückkehr in den Programmablauf, bei bestimmten Arten von Sprüngen ist dies ebenfalls der Fall.

-c (branch-counts):
Diese Option ist nahezu identisch mit der –b Option, mit der Ausnahme, dass hier für jede Verzweigung die Anzahl ihrer Ausführungen in der .gcov Datei protokolliert wird und nicht ihre Prozentzahl.

-f (function-summaries):
Diese Option gibt zusätzlich zu der globalen Ausgabe des kompletten Programms Auswertungen für die einzelnen Funktionen an. Hier ein Beispiel:

```
$ gcov –f sample2.c
 80.00% of 5 source lines executed in function read
 66.66% of 18 source lines executed in function main
 71.64% of 23 source lines executed in file sample2.c
```

An der .gcov-Datei ändert sich nichts, da dort bereits die Auswertungen für jede einzelne Anweisung und Verzweigung aufgeführt sind. Diese Funktions-Auswertungen lassen sich auch mit der –b oder –c Option kombinieren, in diesem Fall wird auch für jede Funktion die Auswertung der Verzweigungen und Aufrufe erstellt.

Dies sind die wichtigsten Optionen von gcov, die weiteren Optionen finden sich in der zugehörigen manpage. Gcov ermöglicht also das Statement Coverage und das Decision Coverage, weitere Möglichkeiten existieren nicht, da das Path Coverage beispielsweise viel zu aufwendig wäre. Hier steigt die Anzahl der zu untersuchenden Pfade exponentiell mit der Anzahl der Verzweigungsmöglichkeiten, weswegen es von gcov nicht als Option angeboten wird.

3.3 Gcov anhand eines Beispiels

Anhand eines kurzen Beispiels soll die Funktionsweise von gcov jetzt noch einmal verdeutlicht werden. Folgender Quellcode dient dabei als Beispielfall:

```
int main(int argc, char **argv) {
      int i;
      int j = 0;
      int size;
```

```
      if (argc < 2) {
      printf("Usage: %s Arraysize value1 value2 ... \n",argv[0]);
      exit(-1);
      }

      size = atoi(argv[1]);
      int values[size];

      for(i = 0; i < size; i++) {
            values[i] = 0;
      }

      if(argc > size) {
            printf("Too many values entered.\n");
            exit(-1);
      }

      for(i = 0; i < argc-2; i++) {
            values[i] = atoi(argv[i+2]);
            j++;
      }

      printf(„Entered %d values\n“,j);
}
```

mysample.c

Das Programm erwartet mindestens einen Übergabeparameter, ansonsten gibt es eine Fehlermeldung aus und bricht ab. Wird dieser Parameter angegeben, erstellt es ein Array des Typs int mit der Größe des Parameters. Daraufhin wird das Array in einer for-Schleife mit dem Wert „0“ gefüllt. Hat der Benutzer weitere Parameter angegeben, werden diese der Reihe nach in das Array gefüllt. Ist die Anzahl der Parameter jedoch größer als das angelegte Array, wird eine Fehlermeldung ausgegeben. Am Ende wird noch eine Bestätigung ausgegeben, wie viele Werte gespeichert wurden.

Das Programm wird jetzt mit dem gcc Compiler und den genannten Optionen kompiliert:

```
$ gcc –fprofile-arcs –ftest-coverage –o mysample mysample.c
```

Daraufhin lässt man das Programm ein Mal starten und führt danach gcov aus, beispielsweise wie folgt:

```
$ ./mysample 5
Entered 0 values
$ gcov mysample.c
 66.67% of 18 source lines executed in file mysample.c
Creating mysample.c.gcov.
```

12 von 18 Zeilen wurden bis jetzt ausgeführt, noch nicht ausgeführt wurden die beiden if-Blöcke mit jeweils 2 Zeilen und auch die letzte Schleife wurde noch nicht durchlaufen, da dies nur bei mindestens 2 Übergabeparametern geschieht. Um ein vollständiges Statement Coverage zu erreichen, benötigt man also noch drei weitere Testfälle, einen, bei dem kein Argument angegeben wird, einen mit mehr Argumenten als Array-Größe und einen, bei dem zusätzliche Werte in das Array geschrieben werden:

```
$ ./mysample // zu wenig Parameter
Usage: ./mysample Arraysize value1 value2 ...
$ ./mysample 2 5 5 5 // Arraygröße 2, aber 3 Werte angegeben
Too many values entered
$ ./mysample 5 4 3
Entered 2 values
$ gcov mysample.c
100% of 18 source lines executed in file mysample.c
Creating mysample.c.gcov.
```

Nach Hinzufügen dieser drei Testfälle wurden nun also alle Anweisungen mindestens ein Mal ausgeführt. Nun kann man noch die –b Option von gcov hinzufügen, um das Decision Coverage zu verwirklichen.

```
$ gcov –b mysample.c
100.00% of 18 source lines executed in file mysample.c
100.00% of 8 branches executed in file mysample.c
100.00% of 8 branches taken at least once in file mysample.c
100.00% of 7 calls executed in file mysample.c
```

Es wurden also auch alle möglichen Verzweigungen mindestens einmal durchgeführt. Damit sind die Möglichkeiten des Code Coverage mit gcov ausgeschöpft, im Statement Coverage und im Decision Coverage wurden 100 % erreicht.
Die zugehörige mysample.c.gcov Datei sieht nun wie folgt aus:

```
4    int main(int argc, char **argv) {
           4      int i;
           4      int j = 0;
           4      int size;

           4      if (argc < 2) {
branch 0 taken = 75%
           1            printf("Usage:  %s  Arraysize  value1  value2  ...
\n",argv[0]);
call 0 returns = 100%
           1            exit(-1);
call 0 returns = 0%
                 }

           3      size = atoi(argv[1]);
call 0 returns = 100%
           3      int values[size];

          15      for(i = 0; i < size; i++) {
branch 0 taken = 80%
branch 1 taken = 100%
branch 2 taken = 100%
          12            values[i] = 0;
                 }

           3      if(argc-2 > size) {
branch 0 taken = 67%
           1            printf("Too many values entered.\n");
call 0 returns = 100%
           1            exit(-1);
call 0 returns = 0%
                 }

           4      for(i = 0; i < argc-2; i++) {
branch 0 taken = 50%
```

```
branch 1 taken = 100%
branch 2 taken = 100%
           2            values[i] = atoi(argv[i+2]);
call 0 returns = 100%
           2            j++;
                  }

           2      printf("Entered %d values \n", j);
call 0 returns = 100%
            }
```

mysample.c.gcov

Links neben jeder Anweisung steht, wie oft sie ausgeführt wurde bzw. für eine Verzweigung, wie oft gesprungen wurde und für Funktionsaufrufe, wie oft sie zurück in das Programm gekehrt sind. Die beiden exit() Aufrufe sind logischerweise nicht in das Programm zurück gekehrt, alle anderen Aufrufe hingegen schon. Alle Verzweigungen (branches) wurden mindestens einmal durchgeführt, branch 0 steht in einer if-Abfrage für das negative Testergebnis und damit für das Springen hinter den if-Block. Bei der ersten if-Abfrage wurde beispielsweise nur in einem von vier Fällen in den Block hinein gesprungen.

Somit wurde für dieses einfache Beispiel ein Testdurchlauf durchgeführt, der sowohl vollständiges Statement Coverage als auch vollständiges Decision Coverage verwirklicht. In diesem Fall waren dafür nur vier unterschiedliche Programmaufrufe nötig, was an der relativ einfachen Struktur des Programms liegt, aber auch hier sieht man bereits, dass es relativ aufwendig ist, Quellcode komplett zu testen. Dass in diesem Programm durch falsche Eingaben keine Fehler mehr passieren können ist dadurch noch nicht bewiesen, dies zu verhindern bleibt auch mit gcov weiterhin Sache des Programmierers.

4 Bewertung / Zusammenfassung

4.1 Code Coverage

Code Coverage legt verschiedene Maße der Codeüberdeckung fest, die auf unterschiedliche Arten erreicht werden. Wenn man die Daten, die man aus einer Code Coverage Analyse gewinnt, richtig betrachtet und auswertet, kann man mit großer Wahrscheinlichkeit eine Vielzahl von Fehlern ausfindig machen und korrigieren.

4.2 gcov

Gcov dient der Umsetzung von Code Coverage für C und C++ Code. Wie beschrieben werden die Möglichkeiten der Statement und der Decision Coverage umgesetzt. Die Nutzung ist sehr komfortabel, es müssen lediglich zwei Optionen beim Kompilieren angegeben werden und schon lassen sich Programme mit gcov auswerten. Mit den Einschränkungen, die bereits bei Code Coverage hinsichtlich der Fehlersuche in einem Programm gemacht wurden, eignet sich gcov gut, zu überwachen, welche Teile des Codes ausgeführt wurden. So lassen sich Testsuites gut anpassen und die Wahrscheinlichkeit für unentdeckte Fehler sinkt.

5 Quellenverzeichnis

Steve Cornett - Code Coverage Analysis, Stand: 03.10.2002, Aufgerufen: 17.11.2003
http://www.bullseye.com/webCoverage.html#release

GCC Administrator - gcov: a Test Coverage Program, Stand: 17.06.2001, Aufgerufen: 17.11.2003
http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

Cem Kaner - Software Negligence and Testing Coverage, Stand: 1996, Aufgerufen: 17.11.2003
http://www.kaner.com/coverage.htm

Analytische Qualitätssicherung
http://www-sst.informatik.tu-cottbus.de/~db/doc/SoftwareEngineering/Testen_six_page1.pdf

M. Kaul – SE2 Datenfluss
http://ux-2d00.inf.fh-brs.de/home/script/se2/folien/04_Datenfluss.pdf

M. Kaul – SE2 Code Coverage
http://ux-2d00.inf.fh-brs.de/home/script/se2/folien/1page/05.pdf

Linux Magazine – Analyzing Code Coverage with gcov, Stand: Juli 2003, Aufgerufen: 14.11.2003
http://www.linux-mag.com/2003-07/compile_01.html

gcov: a Test Coverage Program, Aufgerufen: 14.11.2003
http://www.fnal.gov/docs/products/gcc/v3_1/gcc.info,.Gcov.html

Fröhlich, Peter H.: The GNU Coverage Tool – A Brief Tutorial. 13.10.2003

GCC Homepage – GNU Project, Stand: 1.11.2003, Aufgerufen: 10.11.2003-11-23
http://gcc.gnu.org/