Testing Code Coverage

Wenn es mal etwas weniger sein darf

Sebastian Götzinger, Verifysoft Technology GmbH

Roland Bär, Verifysoft Technology GmbH

Abstract

Dieser Beitrag beschäftigt sich mit den Fragen, was Code Coverage ist, wofür sie benötigt wird, welchen Herausforderungen sich moderne Tester und Testmanager gegenübersehen und wie diese Probleme angegangen werden können. Der Beitrag richtet sich vor allem an Entwickler aus dem Bereich der eingebetteten Systeme, welche den fehlenden Speicherplatz als Hürde für Nachweis der Code Coverage durch Tests nehmen müssen.

1 Code Coverage

"Zerlegt man die zu testende Software in Einheiten (zum Beispiel Anweisungen, Zweige, Pfade), so definiert Testabdeckung den Anteil an Einheiten, die durch Tests bereits ausgeführt wurden. Die Testabdeckung wird dabei meist in Prozent ausgedrückt."[1]

Besonders in sicherheitskritischen Systemen muss für eine gewisse Qualität gesorgt werden, um Schäden an Material, Menschen oder dem Firmenimage zu vermeiden. Hierbei gilt der Grundsatz: „Was nicht nachweislich getestet wurde, wurde gar nicht getestet.“ Oder eben etwas informeller „Jede Codezeile ist schuldig, bis das Gegenteil bewiesen wurde.“ Vorhandenen Codezeilen wird also prinzipiell misstraut. Da dies jedoch nicht ausreicht, gibt es neben der Zeilenüberdeckung die Zweigüberdeckung, Mehrfachbedingungsüberdeckung („Multicondition Coverage “) und die modifizierte Bedingungs- /Entscheidungsüberdeckung (MC/DC) sowie weitere, auf die in diesem Beitrag nicht weiter eingegangen wird.

1.1 Anweisungsüberdeckung

Statement- oder Line-Coverage ist die schwächste Form der Testüberdeckung. Hier werden nur einzelne Anweisungen oder Zeilen betrachtet. Wichtig ist dabei nur, ob die Zeile durchlaufen wird, und nicht, unter welchen Bedingungen dies geschieht. In der Theorie und den Normen wird diese Art der Testüberdeckung auch C_0 genannt.

Bei diesem Beispiel wurden 6 von 7 Anweisungen, also 85 % abgedeckt.

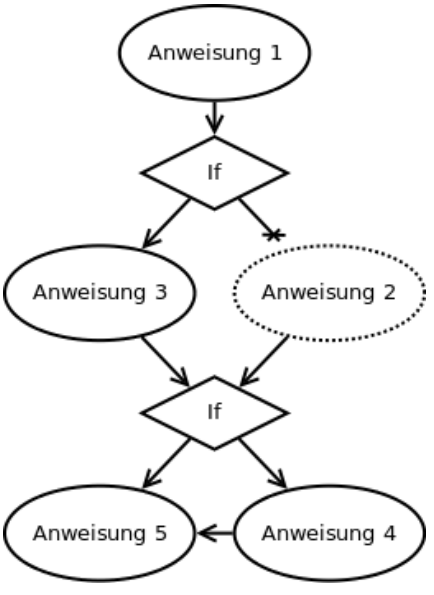

Abbildung 1: Anweisungsüberdeckung

1.2 Zweigüberdeckung

Die Zweigüberdeckung ist stärker als die einfache Anweisungsüberdeckung. Sie prüft, ob alle möglichen Programmzweige verwendet wurden. In der Theorie und den Normen wird diese Art der Testüberdeckung auch C_1 genannt. Es gilt also: Wenn C_1 erreicht wird, wird auch C_0 erreicht.

Bei diesem Beispiel wurden 5 von 7 Zweigen, also 71 %, abgedeckt.

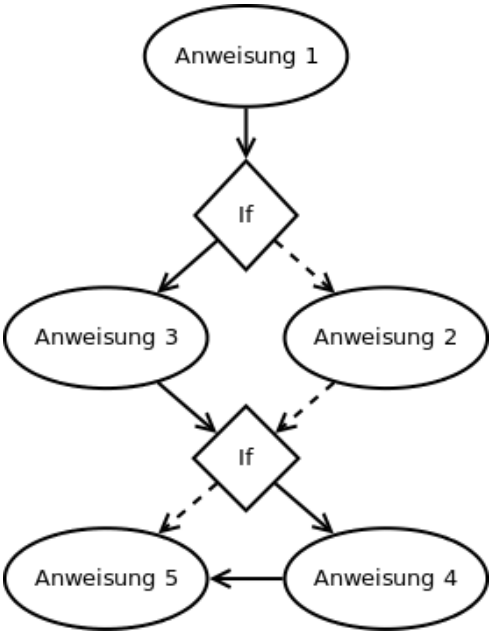

Abbildung 2: Zweigüberdeckung

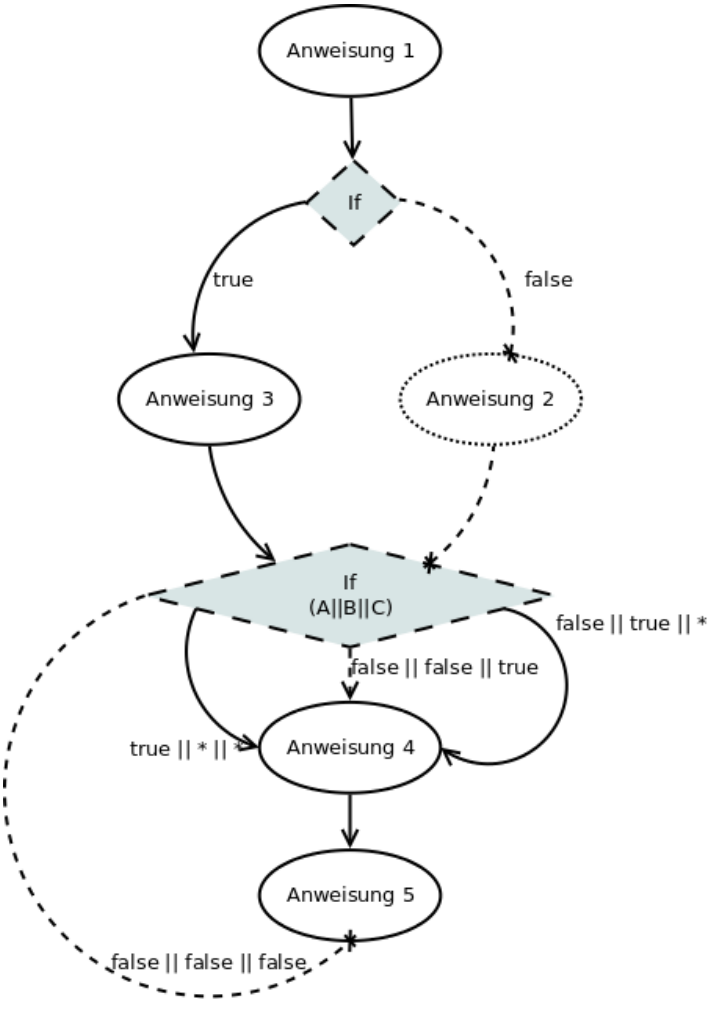

Abbildung 3: Entscheidungsüberdeckung

1.3 Entscheidungsüberdeckung

Die Entscheidungsüberdeckung ist in unserer Betrachtung das stärkste Kriterium. Es ist dann erreicht, wenn alle möglichen Entscheidungen durch den Test abgedeckt wurden. Im Zuge des Beitrags wird die echte Entscheidungsüberdeckung hier auch mehrfache Entscheidungsüberdeckung oder auch Multicondition Coverage genannt.

Die echte Entscheidungsüberdeckung wird in den Normen als C_3 bezeichnet. Werden alle drei Stufen der Coverage erreicht, ist auch die Entscheidungsüberdeckung gegeben. MC/DC kann in diesem Fall als Zwischenstufe zwischen C_1 und C_3 angesehen werden.

Hier wurden 3 von 6, also 50% der Bedingungen abgedeckt.

2 Embedded Systems

Eingebettete Systeme können in vielen Erscheinungsformen auftreten. Allgemein sind es elektrische Bauteile, die eine (Teil-) Funktionalität mit möglichst geringem Aufwand realisieren und fest in das Gesamtsystem verbaut sind. Effizienz wird hierbei gegenüber Modularität präferiert. Sie bestehen hierbei aus der Hardware und einem Betriebssystem, welches entweder bereits die Anwendung integriert oder aber ihren reibungslosen Ablauf ermöglicht. Allen eingebetteten System ist gemein, dass sie im Normalfall sehr klein sind.

2.1 Sicherheitsbegriff

In diesen Beitrag meint Sicherheit den gefahrlosen Betrieb einer Sache, und damit verbundene Unversehrtheit von Mensch und Material. Dies entspricht dem englischen Wort „Safety“.

2.2 Coverage Measurement

Um die Abdeckung der Tests zu ermitteln sind Proben zur Laufzeit nötig. Diese müssen, wenn irgendwie möglich, auf dem Zielsystem durchlaufen und gemessen werden. Nur wenn etwas nachweislich nicht auf dem Zielsystem getestet werden kann, darf es auf einem Wirtsystem getestet werden.

3 Probleme und mögliche Lösungen

Das Hauptproblem mit dem sich dieser Beitrag beschäftigt ist die Speicherung der Ergebnisse. Insbesondere ist es der Fall, dass entwedas RAM odas ROM der Flaschenhals ist. Dieser Abschnitt beschäftigt sich mit der Lösung des Problems.

3.1 Testsplitting

Eine mögliche Lösung, Speicher zu sparen ist es, die Tests aufzuteilen. Dies reduziert die nötige Speichermenge in RAM und ROM, führt aber dazu, dass die Planung und Durchführung je nach Zielsystem aufwendiger werden kann. Sind alle Tests abgeschlossen, müssen die Ergebnisse aller Durchläufe zusammengeführt werden.

Pro: geringerer Speicherverbrauch pro Testablauf

Contra: umständlich

Primäres Anwendungsgebiet: Eingebettete Systeme, die eine große Menge an Tests tätigen müssen, sodass die Laufzeitbibliothek nicht ins Gewicht fällt.

3.2 Arbeitsteilung

Bei genauerer Betrachtung stellt man fest, dass das Zielsystem eigentlich gar nicht dazu genötigt sein sollte, die Messdaten auszuwerten. Ein einfacher Datendump, der mit einer passenden Speicherreferenzen-Datei kombiniert wird, sollte einem anderen Rechner alle für eine Analyse notwendigen Daten bereitstellen können. Wie nun genau der Datendump erhalten wird, ist hierbei nicht wichtig. Wichtig ist, dass man ihn fehlerfrei erhält und die Speicherreferenzen stimmen.

Pro: in Kombination mit Testsplitting auf nahezu jedem System durchführbar

Contra: alle Messdatenspeicher müssen zunächst genullt werden

Primäres Anwendungsgebiet: Eingebettete Systeme, die zu klein für andere Lösungen sind.

3.3 Realisierung in Testwell CTC++

Der Testcoverage Analyzer Testwell CTC++ nutzt normalerweise die komplette Datensammlung durch das Zielsystem, es sei denn, es werden die Addons Bitcov, Bytecov oder Bytecov++ genutzt.

Hierbei nutzt Bitcov/Bytecov ein in den Code integriertes char-Array, oder man gibt an, unter welcher absoluten Speicheradresse Messwerte abgelegt werden können.

Bitcov

Die steigende Nachfrage nach Code-Coverage für Kleinstsysteme fordert eine Lösung für die Messung der Testabdeckung auf kleinen Zielsystemen. Testsplitting ist vor allem durch die große Laufzeitbibliothek selten praktikabel. Mittels der Bitcov-Strategie wird dieser Ballast nun ausgelagert und nur noch ein Flag im Speicher gesetzt, der angibt ob der vorher ermittelte Messpunkt erreicht wurde.

Mit diesem Vorgehen kann der Speicherverbrauch hinsichtlich des RAMs minimiert werden, da zum einen die Laufzeitbibliothek auf dem Zielsystem wegfällt, zum anderen der Speicherverbrauch pro Messpunkt um den Faktor 8 oder mehr reduziert wird.

Bytecov/Bytecov++

Der Speicherverbrauch in Bezug auf den RAM wird mit der Bitcov-Strategie reduziert, wenn nicht gar optimiert. Der Engpass ist jedoch nicht immer das RAM. Manche Systeme haben das Problem, dass das ROM und nicht das RAM unzureichend vorhanden ist, da Bitcov erst den Speicherplatz ermitteln muss, ehe es das Bit setzen kann. Auf Kosten das RAM-Optimierung (der Speicherverbrauch erhöht sich um Faktor 8) ist es nun möglich, die Tests hinsichtlich des ROMs zu optimieren.

Bytecov++ ist eine weitere, theoretische Möglichkeit den ROM-Verbrauch weiter zu reduzieren. Anstelle des Setzens eines Bytes wird dieses inkrementiert, wodurch ein geringer Zyklenverbrauch auf niederster Ebene erreicht wird.

4. Messergebnisse der Experimente

Um die Effektivität der Verfahren zu untersuchen, wird ein einfaches Programm zur Ermittlung von Primzahlen instrumentiert. Als Referenz für die normale Speicherung dient hierbei ein durch CTC++ Instrumentierter Code, der weder Bitcov noch Bytecov enthält. Da mit dem Bitcov-Addon auch die Möglichkeit besteht, an eine absolute Adresse zu speichern, wird auch der ROM-Verbrauch hinsichtlich dieser Variante dargestellt.

4.1 ARMCC, GGC –M32

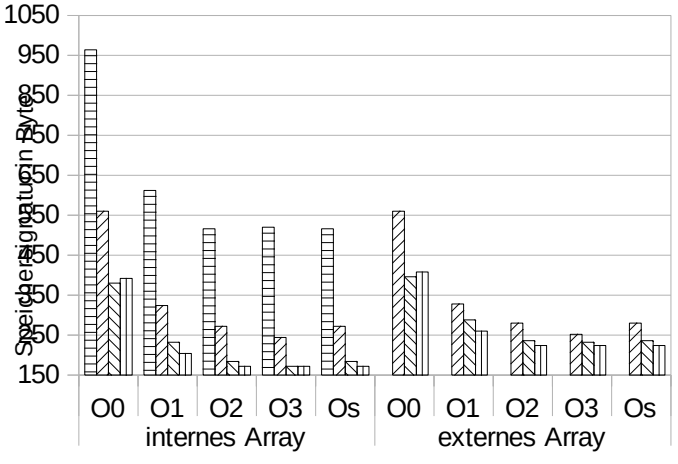

Abbildung 4: ARMCC Speichersignatur

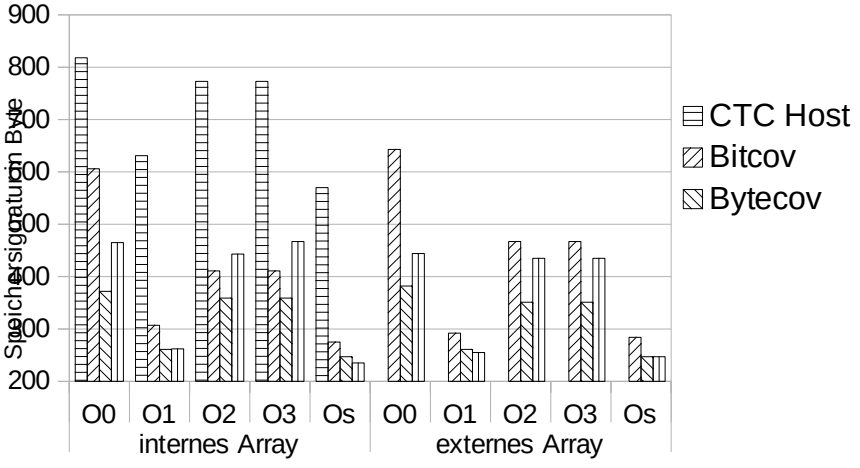

Abbildung 5: GGC-M32 Speichersignatur

Die Grafik zeigt die Auswirkung des Bytecov auf die Speichersignatur. Auch wenn das 8-fache an RAM-Speicher für die Persistierung verwendet werden muss, schafft es die Bytecov-Strategie die Menge des zu verwendenden ROMs um mehr als ein Drittel zu senken. Dies ist vor allem bei Projekten wichtig, in denen genügend RAM aber zu wenig ROM vorhanden ist. Auch sieht man, dass Bytecov++ je nach Compiler zu besseren oder schlechteren Ergebnissen als Bytecov führen kann. Für beide Architekturen gilt, dass das eingebettete Array die bessre Lösung darstellt, da der Overhead den gesparten Speicherplatz übersteigt.

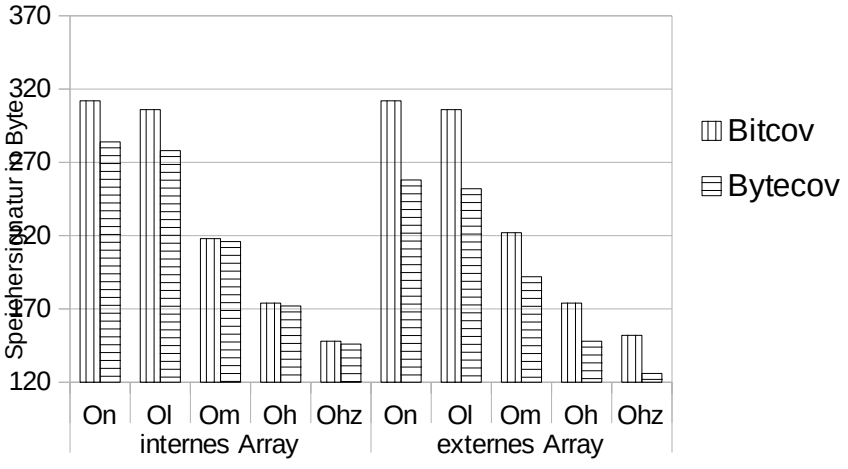

Abbildung 6: ICCV850 Speichersignatur

4.2 ICCV850, der Ausreisser

Im Vergleich zu den anderen in diesen Versuchen verwendeten Compilern zeigt der von IAR bereitgestellte ICCV850-Compiler ein anderes Verhalten hinsichtlich des externen Speicherns. Es ist zu beobachten, dass das Auslagern des Speicherarrays in einen externen Speicherbereich weniger Overhead lieferte, als durch das einbinden des Arrays belegt geworden ist. Eine Auslagerung kann hier also Sinn machen.

4.3 Fazit

Allgemein kann gesagt werden, dass manuelle Optimierung aufgrund guter Compiler nicht mehr sinnvoll ist. Viele Umstellungen beherrschen sie bereits selbst. Für weitere Optimierung müssen andere Strategien genutzt werden.

Bytecov++

Das theoretisch vielversprechende Bytecov++ erbringt in Bezug auf die ROM-Optimierung nicht die erwarteten Ergebnisse. Wie man jedoch am Compiler ARMCC sehen kann, kann auch Bytecov++ zu einer weiteren Verbesserung führen.

Bitcov

Bitcov ist ein sehr guter Ansatz, sofern man über genügend ROM verfügt. Die optimierte RAM-Nutzung wird, mangels expliziter Bitaddressierung, über Anweisungen und Rechnungen realisiert. Aus Erfahrung hat sich gezeigt, dass dies meist ausreicht.

Bytecov

Wo Bitcov versagt, hilft oft Bytecov. Mithilfe dieses Ansatzes werden 10 % bis 25 % eingespart, was jedoch mit 8-fachem RAM-Verbrauch ermöglicht wird. Byte- und Bitcov sind kein Allheilmittel. Aber es ist mit ihnen oft möglich, das sehr aufwendige Testsplitting zu vermindern wenn nicht gar komplett zu vermeiden.

Literatur

[1] Grünfelder, Stephan: *Software-Test für Embedded Systems*, dpunkt.verlag, 2013

Die Autoren

Sebastian Götzinger

Seit 2012 studiert Sebastian Götzinger angewandte Informatik an der Hochschule Offenburg, wo er auch seine Zertifzierung ”ISTQB - Certified Tester - Foundation Level” erlangte. Seit 2013 ist er für die Verifysoft Technology GmbH tätig.

Roland Bär

Seit 1994 hat Roland Bär verschiedene Positionen, unter anderem als Entwickler von Softwaretesttools, bekleidet. Er war maßgeblich an der Entwickling von Testtools der Firmen Testwell und Parasoft beteiligt. Seit 2003 ist Roland CTO bei der Verifysoft Technology GmbH.

Kontakt: quality@verifysoft.com
Weitere Informationen: ese2014.verifysoft.com