Minimalinvasiv testen

Die ISO-Norm 26262 stellt hohe Anforderungen an die Software-Qualität von Automotive-Steuergeräten. Die nötigen Tests sind oft nur möglich, wenn man den Code verändert. Bei Mikrocontrollern, die einen vollständigen Code-Trace unterstützen, gibt es jetzt aber ein Verfahren, das die nötigen Branch-Coverage-Tests ohne jegliche Eingriffe in den Code möglich macht.

Von Heiko Rießland

Bedingt durch die in den letzten Jahren explodierte Komplexität der Steuerungs-Software in kritischen und sicherheitsrelevanten Systemen, sind Entwickler heute mehr denn je auf verlässliche und reproduzierbare Methoden zur Überprüfung und Beurteilung der funktionalen Zuverlässigkeit des manuell erstellten Quellcodes angewiesen. Besonders systematische Messungen der Codeabdeckung (Code Coverage) werden zur Qualitätssicherung empfohlen oder sogar gefordert (IEC 61508, ISO 26262). Beim Code Coverage handelt es sich um ein kontrollflussorientiertes Testverfahren. Die Grundidee dabei ist, zu messen, ob der zu einer Applikation gehörende Programmcode während des Tests in allen seinen Teilen mindestens einmal durchlaufen wurde. Das kann mit speziell erzeugten Stimuli oder auch parallel zu funktionalen Tests erfolgen. Ein großer Vorteil des Verfahrens ist, dass die Anwendung und Auswertung weitgehend automatisierbar ist. Es handelt sich um eine rein formale Prüfung, die nichts über die Software-Qualität bezüglich Funktionalität, Robustheit usw. aussagt. Allerdings ist sie ein sehr gutes Maß für die erreichte Testtiefe.

Stufen der Codeabdeckung

Man unterscheidet verschiedene Stufen der Codeabdeckung, deren Bezeichnungen und insbesondere Abkürzungen in der Literatur allerdings nicht einheitlich verwendet werden. Die von uns gewählten orientieren sich an der Luftfahrtnorm D0-178B. Die Hauptgruppen sind nach aufsteigender Testgüte:

- ➜ Anweisungsüberdeckung bzw. Statement Coverage (Abkürzung C0),
- ➜ Zweigüberdeckung bzw. Branch Coverage (Abkürzung C1),
- ➜ Pfadüberdeckung bzw. Path Coverage (Abkürzung C2) und
- ➜ Bedingungsüberdeckung bzw. Condition Coverage (Abkürzung C3).

Dabei deckt jede höhere Gruppe die Anforderungen aller vorhergehenden mit ab. In der untersten Stufe, dem Statement Coverage, wird die Ausführung jedes Maschinenbefehls ohne Berücksichtigung von Programmverzweigungen gemessen. Mit diesem Verfahren kann nicht ausgeführter Programmcode (dead code) aufgespürt werden. Es erfüllt aber meistens nicht die aktuellen Anforderungen an die Testqualität. Branch Coverage verlangt die Ausführung jeder möglichen Programmverzweigung (**Bild 1**).

Demgegenüber müssen beim Path Coverage alle möglichen Pfade durch ein Modul durchlaufen werden (**Bild 2**).

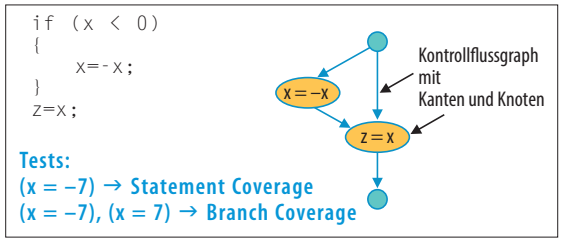

Bild 1. Branch Coverage: Der Zweigüberdeckungstest spürt nicht ausführbare Programmzweige auf. Wenn x größer Null ist, wird der (linke) IF-Zweig nicht ausgeführt.

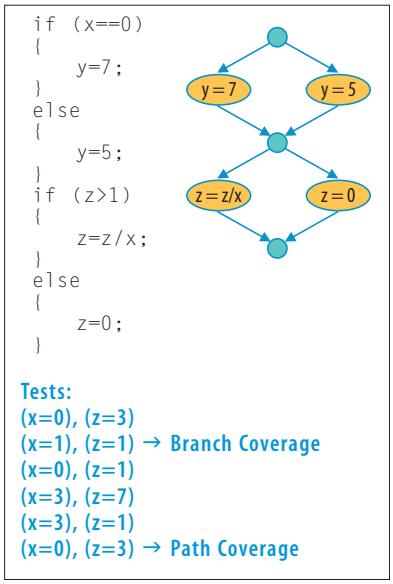

Bild 2. Path Coverage: Beim Test der Pfadüberdeckung müssen alle Pfade ausgeführt werden. Das steigert den Testaufwand exponentiell.

Dies führt bei mehreren nacheinander folgenden Entscheidungen, insbesondere aber bei Schleifen, die Entscheidungslogik enthalten, zu extrem vielen Pfaden. Vollständiges Condition Coverage wiederum verlangt, dass jede atomar zu testende Bedingung einer Entscheidung in allen Kombinationen einmal mit wahr (true) und einmal mit falsch (false) durchlaufen werden muss. Selbst bei einem einfachen Entscheidungsausdruck wie if (a || b) ergibt das bereits vier zu testende Möglichkeiten.

Branch Coverage meistens gute Wahl

Eine höhere Stufe der Codeabdeckung lässt sich also bei konventioneller Vorgehensweise in jedem Fall nur mit einem höheren Testaufwand erzielen. In der Praxis ist deshalb immer eine Kosten-Nutzen-Abwägung zu treffen. ISO 26262 gibt beispielsweise zum Erreichen bestimmter Automotive Safety Integrity Levels (ASIL) die in der **Tabelle** aufgeführten Empfehlungen. Diesen Empfehlungen zufolge ist Branch Coverage über alle ASIL-Levels als solides Verfahren zur Codeabdeckungsanalyse eingestuft.

	ASIL A	ASIL B	ASIL C	ASIL D
Statement Coverage (C0)	++	++	+	+
Branch Coverage (C1)	+	++	++	++
MC/DC (Modified Condition/ Decision Coverage) (C3)	+	+	+	++

++ steht für „sehr zu empfehlen (highly recommended)“,
+ steht für „zu empfehlen (recommended)“

Empfehlungen zu Coverage-Verfahren nach ISO 26262.

Aufgabe einer Testanordnung für Branch Coverage ist neben der Stimulierung des Systems mit speziellen oder funktionalen Tests die Erkennung der durchlaufenen Kanten des Kontrollflussgraphen. Dies erfolgt häufig immer noch durch eine Instrumentierung des Programmcodes, der Speicherung der Ergebnisse im Datenspeicher des Systems und/oder der Aussendung der Daten. Für die hier betrachtete Klasse von Steuergeräten im Automotive- und insbesondere Powertrain-Bereich ist ein solches Vorgehen aber nur sehr bedingt anwendbar. Zum einen hat die Instrumentierung einen signifikanten Einfluss auf das Laufzeitverhalten des Systems, zum anderen wird durch den zusätzlichen Programmcode die Gesamtgröße und damit gegebenenfalls auch das Speicherlayout der Applikation verändert. Außerdem lässt sich das Verfahren nur an nicht optimiertem Code sinnvoll durchführen, weil die Code-Optimierung die Struktur des Maschinencodes derart verändern kann, dass die eineindeutige Abbildung von Basisblöcken der Quellcode-Ebene auf Basisblöcke der Maschinencode-Ebene nicht mehr möglich ist. Unter anderem werden Basisblöcke bei der Optimierung vervielfältigt, zusammengefasst oder eliminiert. Solche Programmtransformationen treten z.B. bei Optimierungen wie Schleifentransformationen (Schleifenspaltung, -vereinigung, oder -expansion), *if-then-else*-Transformationen (z.B. Eliminierung eines Anweisungszweiges) und Umwandlung von Basisblöcken in Tabellenzugriffe auf. Die zuletzt genannte Optimierung soll am Beispiel einer *switch*-Anweisung demonstriert werden.

```
enum color{red, blue, orange,
           violet, yellow};
int ColorToOffset(enum color c) {
       int offset;
       switch(c) {
       case(red):
              offset=10;
              break;
       case(blue):
              offset=12;
              break;
       ...
       case(yellow):
              offset=0;
              break;
       default:
              offset=-1;
              break;
       }
       return offset;
}
```

Listing 1. Quellcodebeispiel für switch-Anweisung.

Code-Optimierung: von leicht geändert bis total umgewandelt

Für eine normale *switch*-Anweisung in C/C++ (**Listing 1**) erzeugt der Compiler einen Maschinencode (**Listing 2**), der den Basisblöcken der *switch*-Anweisung noch zugeordnet werden kann. Zu Beginn jedes Blockes wird überprüft, ob der Wert im Kopf der *switch*-Anweisung mit der *case*-Konstante übereinstimmt. Falls nicht, wird das nächste Label angesprungen, wo dann die Prüfung mit der nächsten *case*-Konstante stattfindet. Existiert ein *default*-Block, wird dieser ausgeführt, wenn alle anderen Vergleiche fehlschlugen.

In der ersten Optimierungsstufe versucht der Compiler zu erkennen, ob der Wertebereich der *case*-Konstanten klein genug ist, um den Vergleichswert im Kopf der *switch*-Anweisung als Index einer Tabelle verwenden zu können. Ist dem so, erzeugt der Compiler eine Sprungtabelle, indem er die Adressen der Labels in einer Tabelle speichert. Die *case*-Konstanten dienen in der Tabelle als Index. Ohne Einschränkung der Allgemeingültigkeit nehmen wir in

```
ColorToOffset: @ c is stored in r0
.L1:
       CMP r0, #red      @ if(c != red)
       BNE .L2           @   goto L2
       MOV r0, 10        @ retval =10
       B .Lend           @ break
.L2:
       CMP r0, #blue     @ if(c != blue) goto L3
       BNE .L3
       MOV r0, 12
       B .Lend
...

.Ln:
       CMP r0, #yellow   @ if(c != yellow) goto Ldefault
       BNE .Ldefault
       MOV r0, 0
       B .Lend
.Ldefault:
       MOV r0,0
       SUB r0,r0,#1
.Lend:
       MOV PC,LR @return value is stored in r0
```

Listing 2. Assembler-Notation für die nicht optimierte Switch-Anweisung

```
ColorToOffset: @ c is stored in r0
       CMP r0, #red          @ if(expression < const_1) goto default
       BLT .Ldefault
       CMP r1, #yellow       @ if(const_n < expression) goto default
       BGT .Ldefault
       ADR r2, jump_table    @ goto *jump_table[expression]
       ADD pc, r2, r1
.Lred:
       MOV r0, #10
       B .Lend
.Lblue:
       MOV r0, #12
       B .Lend
...
.Lyellow:
       MOV r0, #0
       B .Lend

.Ldefault:
       MOV r0, #0
       SUB r0, r0, #1
.Lend
       MOV PC,LR
jump_table:
       .word .Lred
       .word .Lblue
       ...
       .word .Lyellow
```

Listing 3. Assembler-Notation für Sprungtabellen-Optimierung

Listing 3 an, dass die kleinste Konstante gleich Null ist. Ist dem nicht so, generiert der Compiler mit Hilfe eines konstanten Offsets eine Index-Umrechnung. Fehlen für Daten im Wertebereich der Tabelle entsprechende *case*-Statements, werden diese vom Compiler mit Referenzen auf den Default-Block ergänzt. Die Blöcke des Maschinencodes können im Falle der Sprungtabelle noch den Quellcode-Blöcken zugeordnet werden, jedoch erfordert es einigen Aufwand, die drei Basisblöcke der *if-then-else-if-else*-Anweisungen dem gesamten Kopf der *switch*-Anweisung richtig zuzuweisen.

In unserem Beispiel kann der Compiler aber noch eine weitere Optimierung durchführen. Dafür versucht er zu erkennen, ob die Ausdrücke der Blöcke konstant sind bzw. ob sich ihr Wert bereits während der Compilierung ermitteln lässt. Im Beispiel können alle Zuweisungen durch Konstanten ersetzt werden. Ist dies für alle *case*-Blöcke der Fall, kann der Compiler die sonst nötigen Sprünge komplett einsparen und den Wert durch einen Wertetabellen-Zugriff (**Listing 4**) ermitteln. Dadurch werden alle *case*-Basisblöcke in einem einzigen Maschinencode-Basisblock vereint. Auch hier erscheint der Kopfblock der switch-Anweisung als *if-then-else*-Anweisung und kann dementsprechend wie zuvor erkannt und zugeordnet werden.

Dieser Code kann noch weiter optimiert werden, wenn architekturbedingte Anweisungen eingeführt werden. **Listing 5** zeigt eine Variante der Optimierung, in der alle Sprünge eliminiert wurden. Voraussetzung dafür ist, bedingte Anweisungen so einzusetzen, dass der Kontrollfluss linear fortgesetzt werden kann und Prozessor-Flags über die Ausführung einer Anweisung entscheiden. In diesem Fall stehen dann *n* Basisblöcke des Quellcodes einem einzigen Basisblock in Maschinencode gegenüber. Die Erkennung der im Maschinencode noch enthaltenen *if-then-else*-Anweisung erfordert allerdings einen enormen Aufwand durch das Debug Tool. In der Regel sind bei einer solchen Optimierung auch die ursprünglichen Debug-Zeileninformationen so verändert, dass alle Zeilennummern auf den gesamten Anweisungsblock verweisen.

Das nachfolgend beschriebene Verfahren beruht auf der Übergabe der dem Compiler bekannten Informationen zur Änderung der Programmstruktur an den Debugger. Das macht eine nachträgliche Erkennung überflüssig, die aufwendig, fehlerträchtig und ggf. sogar nicht eindeutig ist.

Erkennung der Programmstruktur

Die Programmstruktur, d.h. die Basisblöcke, ihre Programmadresse und -größe sowie ihre Verknüpfung, lässt sich sowohl statisch als auch dynamisch ermitteln. Dabei können auch im DWARF-Format vorliegende Debug-Informationen aus der Programmdatei genutzt werden, welche u.a. die Zuordnungen von Zeilennummern zu Programmadressen und die Programmadressen der Basisblöcke enthalten.

Allerdings sind im regulären DWARF-Format bislang weder die Größe der Basisblöcke noch ihre Verknüpfung enthalten. Letztere lassen sich nur durch Disassemblierung der Maschinencode-Anweisungen und anschließender Interpretation, Emulation oder Simulation ermitteln. Diese statische Code-Analyse ist sehr aufwendig und zeitintensiv und wird daher auch nur selten durchgeführt.

Die dynamischen Trace-Informationen enthalten die Adressen der tatsächlich ausgeführten Anweisungen und die Reihenfolge der Ausführung. Aus diesen Informationen lässt sich zwar eine Sicht auf mögliche Basisblöcke erzeugen, aber der Nachteil hier ist, dass man leider nur die tatsäch-

```
ColorToOffset: @ c is stored in r0
       CMP r0, #red          @ if(expression < red) goto default
       BLT .Ldefault
       CMP r1, #yellow       @ if(yellow < expression) goto default
       BGT .Ldefault
       LDR r0, value_table [r0, LSL #2]      @ retval = value_table[c]
       B .Lend
.Ldefault:
       MOV r0, #0
       SUB r0, r0, #1
.Lend
       MOV PC,LR
value_table:
       .word 10
       .word 12
       ...
       .word 0
```

Listing 4. Assembler-Notation für Wertetabellen-Optimierung

```
ColorToOffset: @ c is stored in r0
       MOV r1,r0                  @ save c in temp register
       LDR r0, &value_default     @   retval=-1;
       CMP r1, #red               @ if(expression < red) {
       MOVLT PC,LR                @   return retval; (in r0) }
       CMP r1, yellow             @ if(yellow >= expression) {
       ADRLE r2, &value_table     @
       LDRLE r0, [r2, r1, LSL #2] @   retval = value_table[c] }
       MOV PC,LR                  @ return retval; (in r0) value_default:
       .word 0xffffffff
value_table:
       .word 10
       .word 12
       ...
       .word 0
```

Listing 5. Assembler-Notation für Wertetabellen-Optimierung und Sprungeliminierung

lich ausgeführten Basisblöcke sieht. Ohne zu wissen, welche Basisblöcke nicht ausgeführt werden, kann jedoch keine verlässliche Überdeckungsanalyse durchgeführt werden. Was also tun in so einem Fall?

Wege zur sicheren Erkennung der Programmüberdeckung

Normalerweise reichen die Standard-Debug-Informationen aus, um die Überdeckung der Quellcode-Zeilen (C0-Coverage) zu ermitteln. Um jedoch die Zweigüberdeckung (C1-Coverage) aus Trace-Daten ermitteln zu können, werden zusätzlich die Verknüpfungen zwischen den Basisblöcken benötigt. Diese Verknüpfungen sind, wie schon angesprochen, in den DWARF-Informationen nicht vorhanden. Aus dem Anweisungs-Trace kann ermittelt werden, welche Programmverzweigungen genommen wurden, bei direkten Sprüngen – mit einigem Aufwand – zudem auch, welche Verzweigungen nicht genommen wurden. Anders sieht es bei indirekten Sprüngen aus. Da diese nahezu beliebig viele Sprungziele haben können, ist in der Regel nicht mehr ermittelbar, wie viele Verzweigungen tatsächlich existieren und wohin diese Verzweigungen führen.

Für das vorliegende Beispiel mit der Sprungtabelle bedeutet dies, dass in diesem Fall nicht mehr ermittelbar ist, welche möglichen Verzweigungen ausgeführt wurden und welche nicht. Voraussetzung dafür wäre entweder eine aufwendige statische Code-Analyse, welche die Sprungtabellen-Optimierung erkennt, oder eine Debug-Information über die Basisblock-Verknüpfungen.

Im Beispiel mit der Wertetabelle wird die statische Code-Analyse noch aufwendiger, da hier anhand der Speicherzugriffe ermittelt werden müsste, um welchen ausgeführten *case*-Zweig es sich handelt. Anweisungs-Trace allein reicht dafür nicht mehr aus; für die Auswertung ist zusätzlich zwingend auch noch Daten-Trace erforderlich.

Anreicherung der Debug-Informationen

Um eine aufwendige und damit auch fehleranfällige statische Code-Analyse zu vermeiden, bietet es sich an, die Basisblock-Informationen zu nutzen, die dem Compiler nach allen Programmtransformationen ohnehin zur Verfügung stehen. Dies sind unter anderem die Identifikationsnummern, Adressen und Größen der verschiedenen nach Funktion aufgeteilten Basisblöcke sowie Anzahl und Identifikationsnummern der Nachfolger eines Basisblockes. Diese Informationen reichen aus, um einen Graphen von Basisblöcken zu ermitteln und aus den Verknüpfungen zwischen den Basisblöcken und dem Anweisungs-Trace eine Zweigüberdeckung (Branch Coverage) zu ermitteln.

Grundvoraussetzung für diese Vorgehensweise ist, dass die eingesetzten Mikrocontroller vollständigen internen oder externen Programm-Trace unterstützen, der direkt zur Statement-Coverage-Analyse verwendet werden kann. Um damit auch Branch Coverage realisieren zu können, bedarf es zusätzlich nur noch eines effizienten Ersatzes für die Instrumentierung des Programmcodes.

Da die benötigten Informationen über die Basisblöcke, also Anfangsadresse, Länge und ihre Verbindungen, dem in den Beispielen verwendeten Compiler intern bei der Codegenerierung schon zur Verfügung stehen, bestand die Aufgabe von PLS vor allem darin, diese Informationen auch von außen zugänglich zu machen. Gelöst werden konnte die Aufgabe mittels Ergänzung des DWARF-Debug-Formats und Erweiterung des verwendeten Compilers. Durch geschickte Verknüpfung mit den Trace-Daten innerhalb des Debug- und Testwerkzeuges können jetzt erstmals Branch-Coverage-Analysen auch nichtinvasiv, ohne Instrumentierung des Codes durchgeführt werden. Darüber hinaus ist dieses von PLS in Zusammenarbeit mit der Fa. Hightec entwickelte und im Feldeinsatz bereits auf seine Praxistauglichkeit hin überprüfte Verfahren auch mit optimiertem Code anwendbar. *jk*

Dipl.-Ing. Heiko Rießland
war nach erfolgreichem Abschluss des Infomationstechnik-Studiums an der Technischen Universität Dresden zwölf Jahre in der Entwicklung und dem Vertrieb von Software-Entwicklungswerkzeugen und Emulatoren für 16- und 32-bit-Mikrocontroller tätig. Seit acht Jahren leitet Rießland das Produktmarketing der PLS Programmierbare Logik & Systeme GmbH.
heiko.riessland@pls-mc.com