Code-Coverage auf Embedded-Systemen

Roland Bär, Verifysoft Technology, baer@verifysoft.com
Andreas Behr, Verifysoft Technology, behr@verifysoft.com
Daniel Fischer, Hochschule Offenburg, daniel.fischer@hs-offenburg.de

Abstract
Die Testabdeckung sowie der Testfortschritt lassen sich mittels Code-Coverage quantitativ bestimmen. Je nach Kritikalität des Embedded Systems sind dabei verschiedenartige Coveragestufen (Anweisungs- und Zweigüberdeckung, Modified Condition/Decision Coverage) anzuwenden. Zur Messung dieser Coveragestufen ist der Code zu instrumentieren und auf dem Host oder Target möglichst mittels automatisierten Tests auszuführen. Bei der Instrumentierung des Codes entstehen bei kleinen eingebetteten Systemen verschiedenartige Herausforderungen. Neben der Begrenzung durch RAM und ROM sollte beispielsweise das Zeitverhalten kaum oder nur gering beeinflusst werden.
Im Rahmen des Papers wird auf die verschiedenen Coveragestufen und die Instrumentierung eingegangen. Die Probleme die dabei bei eingebetteten Systemen entstehen werden aufgezeigt und Lösungsmöglichkeiten dargestellt.

1. Einleitung

Mit steigenden Sicherheitsanforderungen an das eingebettete System muss auch die entsprechende Testtiefe erhöht werden. Es muss also wesentlich genauer und detaillierter geprüft werden. In diesem Fall werden die Testtechniken dann intensiver angewandt als es bei der klassischen Desktop-Softwareentwicklung der Fall ist.

Häufiger verlangte und eingesetzte Methoden zum Testen eingebetteter Systeme sind Unit-Tests in Kombination mit einem Nachweis der getesteten Programmabschnitte in Form von Code-Coverage. Mit Code-Coverage hat man die Möglichkeit kontrollflussorientierte Tests nachzuweisen. Der Nachweis zeigt auf, welche Programmabschnitte ausgeführt wurden und welche nicht. Das Testen eingebetteter Systeme mit Code-Coverage und Unit-Tests zählt somit zu der Kategorie der kontrollflussorientierten Tests, die auch zu den strukturorientierten Tests gezählt werden. Durch Eingabe spezifischer Werte zum Test einzelner Funktionen (sogenannter Testvektoren) werden möglichst viele Programmabschnitte durchlaufen, was dann zu einer entsprechend hohen Code-Coverage führt.

Abhängig von Umfang und Genauigkeit der Tests gibt es unterschiedliche Stufen von Code-Coverage. Je höher eine Stufe desto mehr Aufwand wird benötigt, um eine akzeptable Code-Coverage zu erreichen. Zumeist wird Code-Coverage nur in Kombination mit White-Box- Testverfahren eingesetzt, kann aber ebenso effizient eingesetzt werden, um die Überdeckung von Black-Box-Tests zu ermitteln. Mit diesen Ergebnissen lassen sich anschließend Abschnitte im Code lokalisieren, bei denen weitere detaillierte White-Box-Tests notwendig sind.

Kontrollflussorientierte Testtechniken können ebenso dazu verwendet werden, um zu bestimmen, wann das Ende der Testphase erreicht ist (Testendekriterium). Bei nicht sicherheitskritischen Systemen könnte dies z.B. beim Erreichen eines bestimmten Schwellwertes sein. Bei sicherheitskritischen Systemen wird oftmals das

Durchführen einer vollständigen Code-Coverage auf einer bestimmten Stufe gefordert.

Mit den gewonnenen Daten und einem guten Testmanagement lassen sich auch mögliche anstehende Testaufwände abschätzen und somit das ungefähre Ende der Testphase prognostizieren.

2. Teststufen von Code-Coverage

Im Wesentlichen werden die einzelnen Stufen von Code-Coverage wie folgt zugeordnet:

- Funktionsebene
- Anweisungsebene
- Zweigebene
- Pfadebene
- Bedingungsebene

In Tabelle 1 sind die verschiedenen Coveragestufen aufgeführt, welche sich zur Funktions-, Anweisungs-, Zweig-, Pfad- und Bedingungsebene zuordnen lassen.

Abk.	Coveragestufe (dt.)	Coveragestufe (engl.)	Hinweis
Funktionsebene			
-	Aufrufüberdeckung	Function Coverage	Synonym: Function entry
-		Call Coverage	Synonyme: Call Pair Coverage oder Function-Call Coverage
Anweisungsebene			
C_0	Anweisungsüberdeckung	Statement Coverage	
-	Zeilenüberdeckung	Line Coverage	Eine Zeile kann mehrere Statements enthalten
Zweigebene			
C_1	Zweigüberdeckung	Branch Coverage	Je nach Autor wird dies als äquivalent angesehen
-	Entscheidungsüberdeckung	Decision Coverage	
$C_{(1)primitiv}$	Primitive Zweigüberdeckung	Primitive Branch Coverage	Meist durch die Tools schon als C_1 realisiert
Pfadebene			
C_{2a}	Vollständiger Pfadüberdeckungstest	Path Coverage	Bis auf wenige Ausnahmen nicht praktikabel
C_{2b}	Boundary-Interior Pfadüberdeckungstest	Boundary Interior	
C_{2c}	Strukturierter Pfadüberdeckungstest	Boundary Interior Coverage	
	Modifizierter Boundary-Interior Pfad-überdeckungstest		Siehe [Lig09]
LCSAJ		Linear Code Sequence And Jumps	Betrachtung von Codesequenzen, die mit einem Sprungbefehl enden

Bedingungsebene			
C_{3a}	Einfachbedingungsüber-deckungstest	(Simple) Condition Coverage	
-	Bedingungs-/Entscheidungsüberdeckung	Condition/Decision Coverage	
C_{3c}	Minimaler Mehrfach-bedingungsüberdeckungstest	Minimal Multiple Condition Coverage	
MC/DC	Modifizierter Bedingungs-/Entscheidungsüber-deckungstest	Modified Condition Decision Coverage	
C_{3b} MCC	Mehrfachbedingungsüber-deckungstest	Multiple Condition Coverage	

Tabelle 1: *Coveragestufen*

Abbildung 1 zeigt die Relation der einzelnen möglichen Stufen zueinander auf. Die Pfeile stellen von A nach B eine Subsumption in folgender Form dar: Wurde beispielsweise eine Coverage von 100% auf Stufe A erreicht, bedeutet dies, dass auch eine Coverage von 100% auf Stufe B erzielt wurde. Es ist allerdings nicht möglich diese Folgerung zu invertieren, diese lässt also keine Rückschlüsse von Stufe B nach Stufe A zu.

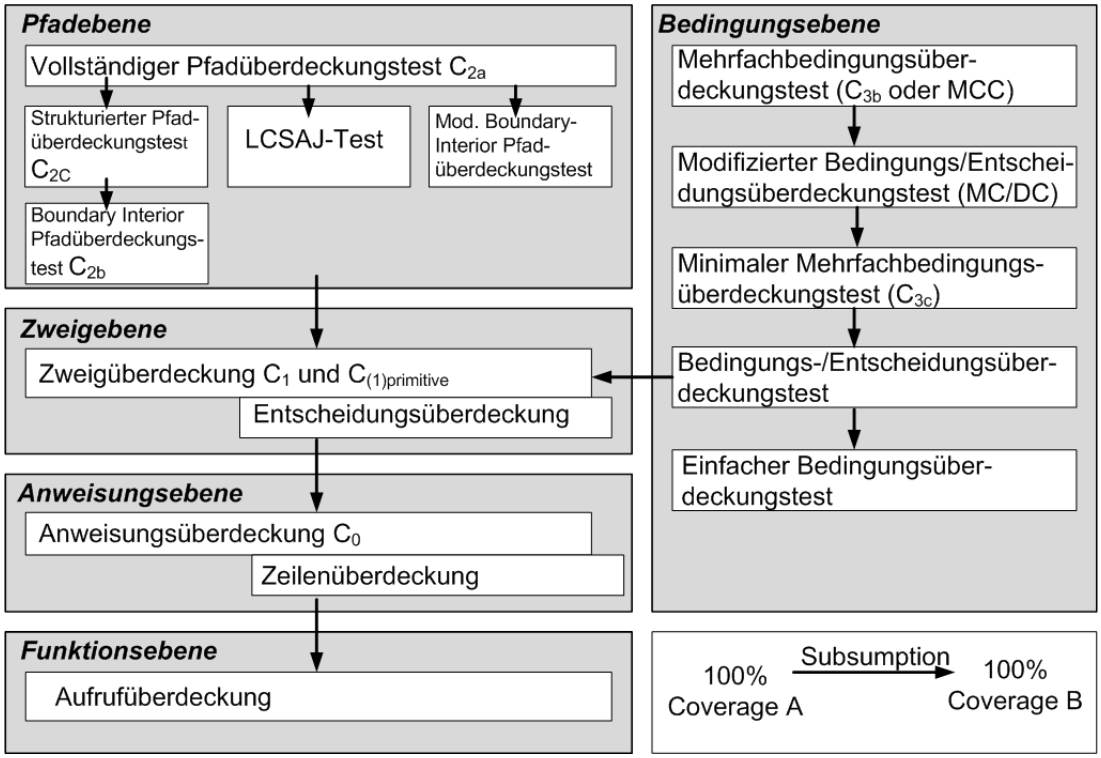

Abbildung 1: *Coveragestufen und deren Subsumption*

Wichtige Normen und Richtlinien verweisen insbesondere auf Funktions-, Anweisungs- und Zweigüberdeckungstest sowie auf den modifizierten Bedingungs-

und Entscheidungs-überdeckungstest (MC/DC). Code-Coverage auf Pfadebene ist hingegen nicht (mehr) von großer praktischer Bedeutung.

Bei der Aufrufüberdeckung wird die Anzahl aufgerufener Funktionen im Verhältnis zu den insgesamt vorhandenen Funktionen berechnet. Der Nutzen dieses Tests ist sehr gering, da innere Abläufe dabei vollständig ignoriert werden.

$$Aufrufüberdeckung = \frac{Anzahl\ aufgerufener\ Funktionen}{Gesamtanzahl\ Funktionen} \cdot 100\% \qquad (1)$$

Die Anweisungsüberdeckung gibt an, wie viele Anweisungen durch die Tests erreicht und somit ausgeführt wurden. Auf dieser Stufe ist es somit bereits möglich, sogenannten toten Code ausfindig zu machen oder Anweisungen zu finden für die es noch keine Tests gibt.

$$Anweisungsüberdeckung\ C_0 = \frac{Anzahl\ ausgeführter\ Anweisungen}{Gesamtanzahl\ Anweisungen} \cdot 100\% \quad (2)$$

Zur Berechnung der Zweigüberdeckung wird meist eine Berechnung der primitiven Zweige verwendet ohne dies gesondert auszuweisen. Die Zusammenhänge zwischen C_1 und $C_{(1)primitive}$ sind in [Lig09] beschrieben und der Vorteil der Verwendung von $C_{(1)primitive}$ zur besseren Abschätzung des weiteren Testaufwandes ist dort ebenso dargestellt.

$$Zweigüberdeckung\ C_1 = \frac{Anzahl\ ausgeführter\ Zweige}{Gesamtanzahl\ Zweige} \cdot 100\% \quad (3)$$

Für die Berechnung von MC/DC werden alle atomaren Bedingungen einer zusammengesetzten Bedingung herangezogen. Für jede dieser atomaren Bedingungen ist ein Testfallpaar zu implementieren, welches zur Veränderung des Gesamtergebnisses der zusammengesetzten Bedingung führt, wobei sich jedoch nur der Wahrheitswert der betrachteten atomaren Bedingung ändert. Der Wahrheitswert der anderen atomaren Bedingungen muss dabei konstant bleiben. Ist es nicht möglich einen solchen Testfall zu generieren, so ist möglicherweise eine unnötige oder funktional nicht aussagekräftige Bedingung aufgestellt worden.

3. Normen und Richtlinien

Die industrieübergreifende Norm DIN EN 61508 – häufig auch die „Mutter aller Normen“ genannt – enthält in Teil 3 ([DIN61508-3]) auch Empfehlungen für Code-Coverage. Der Safety Integrity Level (SIL) bestimmt welche Stufe von Code-Coverage eingesetzt werden soll. Ein Auszug der Tabelle B.2 aus der DIN EN 61508 ist in Tabelle 2 aufgezeigt.

Verfahren/Maßnahme		SIL 1	SIL 2	SIL 3	SIL 4
…	…	…	…	…	…
7a	Strukturabhängige Tests mit einer Testabdeckung (Eingangspunkte) 100%	++	++	++	++
7b	Strukturabhängige Tests mit einer Testabdeckung (Anweisungen) 100%	+	++	++	++
7c	Strukturabhängige Tests mit einer Testabdeckung (Verzweigungen) 100%	+	+	++	++
7d	Strukturabhängige Tests mit einer Testabdeckung (Bedingungen) 100%	+	+	+	++

Tabelle 2: *Auszug einer Tabelle B.2 der DIN EN 61508-3 ([DIN61508-3])*

Die mit „++“ gekennzeichneten Felder zeigen an, dass das angegebene Verfahren oder die Maßnahme besonders empfohlen wird [DIN61508-3]. Entscheidet man aus einem bestimmten Grund diese Vorgabe nicht einzuhalten und auf die beschriebene Maßnahme zu verzichten, so ist ausführlich zu erläutern, weshalb diese Entscheidung getroffen wurde und welche Auswirkungen diese hat. Empfohlene Maßnahmen werden mit einem einfachen „+“ gekennzeichnet und zeigen an, dass die entsprechende Maßnahme eine Verbesserung der Qualität bzw. Sicherheit zur Folge hat.

Für sicherheitsrelevante elektrische/elektronische Systeme bei Kraftfahrzeugen ist die ISO 26262 ([ISO26262-6]) anzuwenden. Anstelle des SIL wird hier der Automotive Safety Integrity Level (ASIL) eingeführt. Auf diesem basierend werden auch hier Empfehlungen für die zu verwendende Code-Coverage auf Funktionsebene (Tabelle 3) und Architekturebene (Tabelle 4) gegeben. Ein Vergleich von SIL und dem in ISO 26262 verwendeten ASIL findet sich unter [Lin12].

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

Tabelle 3: *Structural coverage metrics at the software unit level, [ISO26262], Table 12*

Methods		ASIL			
		A	B	C	D
1a	Function coverage	+	+	++	++
1b	Call coverage	+	+	++	++

Tabelle 4: *Structural coverage metrics at the software architectural level, [ISO26262], Table 15*

Entsprechend zur DIN EN 61508 ist hierbei „++“ als „highly recommended“ und „+“ als „recommended“ zu verstehen. Als Ergänzung zur DIN EN 61508 wird an dieser Stelle noch das zusätzliche Call coverage (Tabelle 4) gefordert.

Die DO-178C ([RTCA11], [HB08]) Norm, die in der Luft- und Raumfahrt Anwendung findet, stellt ebenfalls Richtlinien für Softwaretests auf. Tabelle 5 zeigt die Verteilung der geforderten Coveragestufen auf die fünf Level A bis E.

Level	**Auswirkungen**	**Coveragestufen**	**Anteile Systeme**	**Anteile Software**
A	„Catastrophic“	MC/DC, C_1, C_0	20-30%	40%
B	„Hazardous/Severe“	C_1, C_0	20%	30%
C	„Major“	C_0	25%	20%
D	„Minor“	-	20%	10%
E	„No Effect“	-	10%	5%

Tabelle 5: *Coveragestufen, Anteile Systeme und Software in der Luft- und Raumfahrt [HB08]*

4. Instrumentierung

Prinzipiell gibt es mehrere Möglichkeiten eine Messung der Code-Coverage durchzuführen. Dabei werden software- und hardwareseitige Lösungen unterschieden. In diesem Beitrag wird die universell verwendbare softwareseitige Lösung betrachtet.

Um Code-Coverage messen zu können, muss der bestehende Programmcode modifiziert und ergänzt werden. Dieser Vorgang wird Instrumentierung genannt und geschieht noch bevor der produktive Programmcode an den Compiler übergeben wird (Präprozessor). Dabei werden üblicherweise Zähler in Form von globalen Arrays hinterlegt. Wann diese Zähler verändert werden und wie viele es gibt, hängt von der gewählten Stufe des Code-Coverage ab. Die Arrays befinden sich im Datensegment des Speichers und werden mit dem Wert null initialisiert. Ein Zählerstand beschreibt nach Ablauf des Programms wie oft ein bestimmter Programmabschnitt im Code durchlaufen wurde. Aus der Übersicht der Zählerstände lässt sich anschließend folgern, welche Programmabschnitte noch weiterer Tests bedürfen. Zur Laufzeit wird also nicht nur der eigene produktive Code ausgeführt, sondern auch die für das Coverage notwendigen Anweisungen.

Das wichtigste Kriterium dabei ist, dass die vom Entwickler vorgenommene Programmierung (der Produktivcode) keine logischen Änderungen im Programmablauf nach der Instrumentierung aufweist und somit die logische Funktion des Programms nicht verändert.

Im folgenden Beispiel wird die Instrumentierung einer einfachen logischen Bedingung wie beispielsweise *if (a<0)* ohne Kommaoperator aufgezeigt. Diese Art der Umsetzung setzt die unvollständige Evaluation der Bedingungen voraus.

```
if ( (a<0&&(zaehler1++||1)) || (zaehler2++&&0) )
```

In Sprachen in denen der Komma- und der ternäre Operator zur Verfügung steht, lässt sich die Anweisung entsprechend vereinfachen:

```
if ( (a<0) ? (zaehler1++,1) : (zaehler2++,0) )
```

Das folgende Beispiel (Listing 1) zeigt eine in C programmierte Funktion, welche eine Prüfung einer Zahl auf eine Primzahl vornimmt:

```
int is_prime(unsigned val)
{
  unsigned divisor;

  if (val == 1 || val == 2 || val == 3)
    return 1;

  if (val % 2 == 0)
   return 0;

  for (divisor = 3; divisor < val / 2; divisor += 2)
  {
    if (val % divisor == 0)
      return 0;
  }
 return 1;
}
```

Listing 1: *Nicht instrumentierter Beispielcode einer Funktion*

Listing 2 zeigt die Veränderungen nach der automatischen Instrumentierung des Quellcodes auf, nachdem dieser von einem automatisierten Code-Coverage-Tool instrumentiert wurde. In diesem Beispiel wurde das Werkzeug Testwell CTC++ zur Instrumentierung verwendet. Anhand dieses einfachen Beispiels lässt sich bereits leicht erkennen, dass sowohl für den Programmcode als auch für die Zählerarrays zusätzlicher Speicher benötigt wird. Durch die Instrumentierung steigt sowohl der RAM- als auch ROM-Bedarf, was bei kleinen eingebetteten Systemen schnell zu Problemen führen kann. Ein weiteres Augenmerk sollte auf die Ausführungszeit der Anwendung gelegt werden. Durch die zusätzlichen Anweisungen für die Code-Coverage ergibt sich auch eine längere Ausführungszeit der instrumentierten Programmabschnitte.

```
int is_prime ( unsigned val )
{
unsigned int ctc_m;
if (!ctc_s)
  ctc_module_init(&ctc_m);
ctc_s[0]++;

{
unsigned divisor ;

if ( ctc_m = 0, ( ((val == 1 ) && (ctc_m += 3)) || ((val == 2 ) && (ctc_m += 2)) || ((val ==
3 )
              && (ctc_m += 1)) ) ? (ctc_c[0 + ctc_m]++, 1) : (ctc_c[0 + ctc_m]++, 0))
  {
   ctc_j[0]++;
   return 1 ;
   }

if ( (( val % 2 == 0 ) ? (ctc_t[0]++, 1):(ctc_f[0]++, 0)) )
  {
  ctc_j[1]++;
  return 0 ;
  }

for ( divisor = 3 ; (( divisor < val / 2 ) ? (ctc_t[1]++, 1):(ctc_f[1]++, 0)) ; divisor += 2 )
{
  if ( (( val % divisor == 0 ) ? (ctc_t[2]++, 1):(ctc_f[2]++, 0)) )
     {
     ctc_j[2]++;
     return 0 ;
     }
}

{
ctc_j[3]++;
return 1 ;
}
}
}
```

Listing 2: *Instrumentierter Beispielcode einer Funktion*

5. Herausforderungen bei kleinen Targets

Wie bereits in Kapitel 4 erwähnt, führt das Instrumentieren zu zusätzlichem RAM- und ROM-Bedarf und erhöht außerdem die Ausführungszeit des Programms. Üblicherweise sind diese Einschränkungen auf Desktop-Systemen zu vernachlässigen, da in diesen Systemen in fast allen Fällen genügend Ressourcen zur Verfügung stehen. Bei eingebetteten Systemen liegen meist strikte Begrenzungen der Ressourcen zugrunde. Im einfachsten Fall ist nicht genügend RAM-Speicher vorhanden, um die große Menge der Zähler im Datensegment unterzubringen. Die meist ohnehin nicht großen Speicher im Bereich der eingebetteten Systeme werden zudem kleinstmöglich ausgelegt, um Kosten in der Produktion einzusparen. Diese

Probleme fallen in aller Regel frühzeitig auf und können dann auf unterschiedliche Art und Weise behandelt werden. Die trivialste Möglichkeit besteht darin den Speicher zu vergrößern. Eine alternative Behandlung des Problems ist es, eine partielle Instrumentierung durchzuführen. Hierbei werden nur kleine Teile des gesamten Programms instrumentiert und diese partiell getestet. Nachdem alle Teile des Programms separat getestet wurden, lassen sich die gewonnenen Daten anschließend zu einem großen Ganzen zusammenfügen. Ebenso hängt der benötigte RAM-Bedarf von der verwendeten Coveragestufe ab.

Um den Bedarf an RAM zu reduzieren gibt es außerdem die Möglichkeit die Größe dieser Zähler anzupassen. Die üblichen Zähler haben eine Größe von 32 Bit (uint32), können jedoch in ihrer Größe reduziert werden. So ist es ohne Weiteres möglich die Zähler auf eine Größe von 16 oder 8 Bit zu reduzieren. Die in diesem Fall eventuell auftretenden Überläufe der Zählerstände erfordern jedoch weitere Überlegungen und eine sorgfältige Interpretation der gewonnenen Daten.

Die Größe des ROMs spielt in eingebetteten Systemen eine ebenso große Rolle. Zu den bei der Instrumentierung eingefügten Anweisungen wird auch eine kleine Bibliothek benötigt, welche die wichtigsten grundlegenden Funktionen bereitstellt, wie beispielsweise das Übertragen der Zählerstände auf den Host-Rechner. Tabelle 6 zeigt den Speicherbedarf eines kleinen Programms mit und ohne Instrumentierung. Die Wahl der Coveragestufe ist hierbei von entscheidender Bedeutung für die Größe des Programms. In der Praxis ist im Mittel mit einem Zuwachs von ca. 35% zu rechnen.

ROM-Bedarf	
Ohne Instrumentierung	60 Bytes
Coveragestufe Funktionsüberdeckung	67 Bytes
Coveragestufe Zweigüberdeckung	118 Bytes
Coveragestufe Bedingungsüberdeckung	285 Bytes
Zusätzlicher RAM-Bedarf ohne Bit-Coverage (32-Bit-Zähler)	
Coveragestufe Funktionsüberdeckung	4 Bytes
Coveragestufe Zweigüberdeckung	16 Bytes
Coveragestufe Bedingungsüberdeckung	28 Bytes
Zusätzlicher RAM-Bedarf mit Bit-Coverage	
Coveragestufe Funktionsüberdeckung	1 Bit
Coveragestufe Zweigüberdeckung	4 Bit
Coveragestufe Bedingungsüberdeckung	7 Bit

Tabelle 6: *Zusätzlicher Speicherverbrauch nach Instrumentierung*

Die kritischsten Probleme in Verbindung mit einer Instrumentierung sind die zeitlichen Verzögerungen bei der Ausführung durch eine Mehrbeanspruchung der CPU. Durch diese Beanspruchung kann es möglicherweise dazu kommen, dass ein vorbestimmtes Timing nicht mehr eingehalten werden kann. Insbesondere wenn die CPU nahe ihrer maximalen Leistungsfähigkeit betrieben wird, können so leicht unvorhersehbare Fehlabläufe entstehen. Dies könnte beispielsweise bei einer

Buskommunikation oder ähnlichen Anwendungen der Fall sein, bei denen eine strikte Vorgabe des Timings eingehalten werden muss.

Ist man lediglich daran interessiert ob ein Programmabschnitt ausgeführt wurde, aber nicht wie oft, so bietet es sich an die sonst großen Zählerinformationen in einzelne Bits zu komprimieren. Damit kann man den Bedarf an RAM bis um den Faktor 32 verringern. Die Verwendung von einzelnen Bits statt Zählervariablen wird als Bit-Coverage bezeichnet.

6. Zusammenfassung

Kontrollflussorientiertes Testen und die damit verbundene Bestimmung der Code-Coverage wird auch in Zukunft an Bedeutung gewinnen. Code-Coverage in Verbindung mit Unit-Tests kann erheblich dazu beitragen, die Qualität der Software zu verbessern. Code-Coverage ist auch eine Metrik, mit der die geschätzte Zeit bis zur Fertigstellung der Tests prognostiziert werden kann. In kleinen eingebetteten Systemen wird es immer eine Herausforderung sein, mit den verschiedenartigen Problemen der Ressourcenknappheit umzugehen. Hierzu bieten Coverage-Tools unterschiedliche Lösungsansätze und Unterstützungen an, welche im Rahmen von Pilotprojekten evaluiert werden sollten.

Literatur

[DIN61508-3] DIN/VDE: *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme – Teil 3: Anforderungen an Software*, Februar 2011

[HB08] Hilderman, Vance; Baghai, Tony: *Avionics Certification – A Complete Guide to DO-178 (Software) DO-254 (Hardware)*; Avionics Communication Inc., 2008

[ISO26262-6] ISO: *International Standard ISO/FDIS 26262 –Road vehicles – Functional safety – Part 6: Product development at the software level*, 2011

[Lig09] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*; 2. Auflage, Spektrum Verlag, 2009

[Lin12] Lindenberg, Uwe: *Vergleich IEC EN 61508 SIL mit ISO CD 26262 ASIL*, http://www.uwe-lindenberg.de/files/sil_asil-tabelle.pdf, Stand 31.05.2012

[RTCA11] Radio Technical Commission for Aeronautics: *DO-178C – Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., 2011

Roland Bär leitet die Entwicklung und den Support bei Verifysoft Technology. Schwerpunkte seiner Tätigkeit sind die statische Codeanalyse sowie die Weiterentwicklung und Erweiterung des Coveragewerkzeuges Testwell CTC++ speziell für kleine Targets. Compilerspezifische Anpassungen der Werkzeuge gehören ebenso zu seinem Fachgebiet.

Andreas Behr ist bei Verifysoft Technology u.a. für die Integration von Testwell CTC++ in verschiedene Entwicklungsumgebungen zuständig. Ebenso ist er für die nationalen und internationalen Produktschulungen von Testwell CTC++ verantwortlich.

Daniel Fischer ist seit 2001 Professor für Informationstechnologie an der Hochschule Offenburg. Er leitet dort als Studiendekan die Studiengänge Angewandte Informatik und Wirtschaftsinformatik ^{plus}. In Zusammenarbeit mit der Firma Verifysoft Technology bietet er seit geraumer Zeit die Seminare „Testen von Embedded Systems“ und „Effizientes Testmanagement“ an.