Code-Coverage on Embedded Systems

Daniel Fischer
Hochschule Offenburg
Badstraße 24
77652 Offenburg
daniel.fischer@hs-offenburg.de
Tel.: 0781-205-148

Andreas Behr
Verifysoft Technology GmbH
In der Spöck 10-12
77656 Offenburg
behr@verifysoft.com
Tel.: 0781-127-81189

Roland Bär
Verifysoft Technology GmbH
In der Spöck 10-12
77656 Offenburg
baer@verifysoft.com
Tel.: 0781-127-81189
Agenda

1. Basics
2. Coverage Level
3. Standards
4. Instrumentation
5. Small Targets
6. Example
Basics – Levels Of Testing

Complexity SUT

- Unit-Tests
- Module
- Integration
- System
- User

Switchover

White-Box-Test
Glass-Box-Test

Chain-Test
Class

Complexity SUT

Developer

Code-Coverage

Tester

Requirement-Coverage

Testcoverage

Black-Box - Tests

Grey-Box-Tests
Established test technique for critical Embedded Systems
Test-End criterion (White-Box-Tests)
Necessary for gratification of several standards
void ZaeheZchn (int& VokalAnzahl, int& GesamtAnzahl) {
    unsigned char Zchn;
    Zchn = getch();

    while (((Zchn>=’A’) && (Zchn<=’Z’) && (GesamtAnzahl<INT_MAX))) {
        GesamtAnzahl = GesamtAnzahl + 1;

        if (((Zchn==’A’)|| (Zchn==’E’)|| (Zchn==’I’)||(Zchn==’O’)||(Zchn==’U’)) {
            VokalAnzahl = VokalAnzahl + 1;
        }
        Zchn = getch();
    }
}
Coverage Levels – Overview

Path Level
- Complete Path Coverage C8a
  - Structured Path Coverage Test C8c
  - Boundary Interior Path Coverage Test C8b

Condition Level
- Multi Condition Coverage (C3b or MCC)
  - Modified Condition Decision Coverage (MCDC)
  - Minimal Multi-Condition Coverage (C3c)

Branch Level
- Branch Coverage C1 and C1-primitive
  - Decision Coverage

Statement Level
- Statement Coverage C0
  - Line Coverage

Function Level
- Call Coverage

100% Coverage A  100% Coverage B
Coverage Levels – Important Levels

* Defined by standards

- Condition Level
  - Multi Condition Coverage (C3b or MCC)
  - Modified Condition / Decision Coverage (MC/DC)

- Branch Level
  - Branch Coverage C1 and C1-primitive

- Statement Level
  - Statement Coverage C0

- Function Level
  - Call Coverage

100% Coverage A 100% Coverage B

Prof. Dr. Daniel Fischer - Hochschule Offenburg
Andreas Behr – Verifysoft Technology GmbH
Roland Bär - Verifysoft Technology GmbH

Code-Coverage on Embedded Systems
Embedded-System-Development 2013
int goo( int a, int b, int c) 
{
    int x;

    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }

    return x;
}
**Coverage Levels – Function Coverage**

**TER**
Test Effectiveness Ratio

TER depending on selected coverage level

There's a 100% Coverage for function goo()

---

### CTC++ Coverage Report - Functions Summary

**Directory Summary**

**Files Summary**

**Functions Summary**

**Unverified Code**

**Execution Profile**

To directories: First | Previous | Next | Last | Index | No Index

**Directory:** .
**TER:** 56% (5/9) structural, N.A. statement

**File:** `\main.c`
**Instrumentation mode:** function
**TER:** 100% (3/3) structural, N.A. statement

To files: Previous | Next

<table>
<thead>
<tr>
<th>TER % - function</th>
<th>TER % statement Calls Line Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% (1/1)</td>
<td>N.A.</td>
</tr>
<tr>
<td>100% (1/1)</td>
<td>N.A.</td>
</tr>
<tr>
<td>100% (1/1)</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

100% (3/3) structural, N.A. statement

main.c
 Coverage Levels – Statement Coverage C0

```c
int goo( int a, int b, int c)
{
    int x;

    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }

    return x;
}
```

UCUNIT__TESTCASE_BEGIN("Statement Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1,goo(1,0,1));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("Statement Coverage");

```
---

TESTCASE:Statement Coverage:BEGIN
CHECK: Line 22:IsEqual<1,goo<1,0,1>>:PASSED
CHECK: Line 23:Checklist<>:PASSED

TESTCASE:Statement Coverage:PASSED

Testcases: failed: 0
          passed: 1
Checks:   failed: 0
          passed: 1
---
```
Coverage Levels – Decision Coverage

TER
Test Effectiveness Ratio

TER depending on selected coverage level

There's only 75% Coverage for function goo() using decision coverage
int goo(int a, int b, int c)
{
    int x;

    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }

    return x;
}

---

UCUNIT__TESTCASE_BEGIN("Branch Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1, goo(1,0,1));
UCUNIT__CHECK_IS_EQUAL(0, goo(1,0,0));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("Branch Coverage");
Coverage Levels – Branch Coverage $C_1$

**TER**
Test Effectiveness Ratio

**TER depending on selected coverage Level**

100% coverage for function goo() using decision coverage

### CTC++ Coverage Report - Functions Summary

**Directory Summary | Files Summary | Functions Summary | Untested Code | Execution Profile**

To directories: **First | Previous | Next | Last | Index | No Index**

**Directory:** .
**TER:** 51% (24/47) structural, 63% (77/122) statement

**File:** `main.c`
**Instrumentation mode:** decision
**TER:** 57% (20/35) structural, 66% (75/113) statement

To files: Previous | Next

<table>
<thead>
<tr>
<th>TER % - decision</th>
<th>TER % statement</th>
<th>Calls</th>
<th>Line</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% (2/2)</td>
<td>100% (3/3)</td>
<td>2</td>
<td>10 main()</td>
<td></td>
</tr>
<tr>
<td>48% (14/29)</td>
<td>64% (57/105)</td>
<td>2</td>
<td>18 test suite goo()</td>
<td></td>
</tr>
<tr>
<td><strong>100% (4/4)</strong></td>
<td><strong>100% (5/5)</strong></td>
<td><strong>4</strong></td>
<td><strong>38 goo()</strong></td>
<td></td>
</tr>
</tbody>
</table>

**57% (20/35)**  
**66% (75/113)**  
**main.c**
**DO-178B Definition:**

"Every decision has taken all possible outcomes at least once, and every condition in a decision is shown to independently affect that decision's outcome."

<table>
<thead>
<tr>
<th>( (a&gt;0) )</th>
<th>( (b&gt;0) )</th>
<th>( (c&gt;0) )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>-</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>true</td>
<td>-</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Incomplete Evaluation in C/C++
- is not to be evaluated, can be true or false

\( n+1 \) Tests necessary, \( n := \) amount of atomic conditions
Coverage Levels – Condition Level MC/DC

```c
int goo(int a, int b, int c)
{
    int x;

    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }

    return x;
}
```

```c
UCUNIT__TESTCASE_BEGIN("MC/DC Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1, goo(1,0,1));
UCUNIT__CHECK_IS_EQUAL(1, goo(0,1,1));
UCUNIT__CHECK_IS_EQUAL(0, goo(0,0,0));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("MC/DC Coverage");
```

```plaintext
---
TESTCASE:MC/DC Coverage:BEGIN
CHECK: Line 22:isEqual{1,goo(1,0,1)}:PASSED
CHECK: Line 23:isEqual{1,goo(0,1,1)}:PASSED
CHECK: Line 24:isEqual{0,goo(0,0,0)}:PASSED
CHECK: Line 25:checklist<()>:PASSED
---
TESTCASE:MC/DC Coverage:PASSED
---
```

Coverage Levels – Condition Level MC/DC

```c
int goo(int a, int b, int c)
{
    int x;

    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }

    return x;
}
```

```c
UCUNIT__TESTCASE_BEGIN("MC/DC Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1, goo(1,0,1));
UCUNIT__CHECK_IS_EQUAL(1, goo(0,1,1));
UCUNIT__CHECK_IS_EQUAL(0, goo(0,0,0));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("MC/DC Coverage");
```

```plaintext
---
TESTCASE:MC/DC Coverage:BEGIN
CHECK: Line 22:isEqual{1,goo(1,0,1)}:PASSED
CHECK: Line 23:isEqual{1,goo(0,1,1)}:PASSED
CHECK: Line 24:isEqual{0,goo(0,0,0)}:PASSED
CHECK: Line 25:checklist<()>:PASSED
---
TESTCASE:MC/DC Coverage:PASSED
---
```
Coverage Levels – Condition Level MC/DC

Last atomic condition (cond 3) is missing a test for MC/DC. Missing pairs of tests are shown in red.

3 atomic conditions → 4 test cases
int goo( int a, int b, int c)
{
    int x;
    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }
    return x;
}

<table>
<thead>
<tr>
<th>(a&gt;0)</th>
<th>(b&gt;0)</th>
<th>(c&gt;0)</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>-</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>true</td>
<td>-</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

All 5 test cases are shown. Standards do not claim for MCC. But 100% Multi Condition Coverage means 100% MC/DC (subsumption).

Instead of taking four test cases (n+1), all five tests are executed. This leads to 100% MC/DC without building matching test pairs.
## SIL: Safety Integrity Level

<table>
<thead>
<tr>
<th>Method</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7a Function Coverage</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>7b Statement Coverage</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>7c Branch Coverage</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>7d MC/DC</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

Table B.2 from DIN EN 61508-3

++ Very recommended method, must be reasonable if not used
+ Recommended method
ASIL: Automotive Safety Integrity Level

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Statement coverage</td>
<td>++</td>
</tr>
<tr>
<td>1b Branch coverage</td>
<td>+</td>
</tr>
<tr>
<td>1c MC/DC (Modified Condition/Decision Coverage)</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 12 (Software Unit Level), ISO 26262-6

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Function coverage</td>
<td>+</td>
</tr>
<tr>
<td>1b Call coverage</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 15 (Software Architectural Level), ISO 26262-6

+++ Very recommended method, must be reasonable if not used
+  Recommended method
**DO-178B/C**

<table>
<thead>
<tr>
<th>Level</th>
<th>Impact</th>
<th>Coverage Level</th>
<th>% of Systems</th>
<th>% of Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>MC/DC, C1, C0</td>
<td>20-30%</td>
<td>40%</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous/Severe</td>
<td>C1, C0</td>
<td>20%</td>
<td>30%</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>C0</td>
<td>25%</td>
<td>20%</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
<td>-</td>
<td>20%</td>
<td>10%</td>
</tr>
<tr>
<td>E</td>
<td>No Effect</td>
<td>-</td>
<td>10%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Statement Coverage $C_0$, Branch Coverage $C_1$, Modified Condition/Decision Coverage MC/DC

**IEC 62304**

„... it might be **desirable** to use white box methods to more efficiently accomplish certain tests, initiate stress conditions or faults, or increase code coverage of the qualification tests.“ (IEC 62304, Chapter B.5.7 Software System testing)
• Integrated counter variables (array) for code coverage
• Matching of counter variable to source code
• Increment counter when executed
• Save counter values
• Use counter to generate coverage report

Without comma operator

if ( (a<0) )
    { /* … */}
else
    { /* … */}

Using comma operator

if ( (a<0) ? (counter1++,1) : (counter2++,0) )
    { /* … */}
else
    { /* … */}

if ( (a<0) ? (counter1++,1) : (counter2++,0) )
    { /* … */}
else
    { /* … */}
Instrumentation – Tool-Chain

Memory usage of target
without instrumentation

ROM
RAM

Memory usage of target
with instrumentation

ROM
RAM
Small Targets – Limited Resources

- RAM
- ROM

Reason for lack of memory: 80% RAM, 20% ROM (pract. experience)

- Mostly no file system (counter have to be stored in Memory)

- Limited amount of interfaces on the target device (transfer of datafile)

  Respect for additional interfaces for testing in the hardware design (design for test)
Automating white and black box tests is recommended.

Approach: **Partial instrumentation**
Reduces RAM- and ROM-requirements

In case of insufficient memory

Memory lack:
20% ROM
80% RAM
Reduction of RAM usage

• 32-, 16- or 8-Bit counter?
  Economize RAM… but probably overflow of counter variables

• Using single bits as flag (Bit-Coverage):
  Used to cover whether code was executed or not. But no information about frequency.

Reduction of ROM usage

• Choose minimal required instrumentation(Function-, Branch- and Condition Level)

• Use hardware support to set bits when using Bit-Coverage
  -No HW Support-
    MOV 0x200, %reg1
    OR  2, %reg1
    MOV %reg1, 0x200
  -HW Support-
    ORL 0x200, 2
    SETB 0x1602
    C51
    Melexis

Small Targets – Limited Memory (2)
Small Targets – Limited CPU Time

Uninstrumented Execution Profile

Instrumented Execution Profile

Cycle 1
Some Code

Cycle 2
Interrupt

Cycle 3
Send Data

Cycle 4
Interrupt

Cycle 5
Send Data

Cycle 1
Some Code

Cycle 2
CTC++ coverage

Cycle 3
Interrupt

Cycle 4
CTC++ coverage

Cycle 5
Send Data

Cycle 6
CTC++ coverage

Cycle 7
Interrupt

Cycle 8
CTC++ coverage

Cycle 9
Send Data
int goo(int a, int b, int c)
{
    int x;
    if (((a>0) || (b>0)) && (c>0))
    {
        x = 1;
    }
    else
    {
        x = 0;
    }
    return x;
}

**ROM- Usage**
Without instrumentation: 60 Byte
Function Coverage: 67 Byte
Branch Coverage: 118 Byte
Condition Coverage: 285 Byte

*Simple example with small code and big instrumentation overhead (mean 30% of code size).*

**Additional RAM-Usage without Bit-Coverage**
Function Coverage: 1 Integer
Branch Coverage: 4 Integer
Condition Coverage: 7 Integer

Integer: 32 Bit (unsigned long) as default

**Additional RAM-Usage using Bit-Coverage**
Function Coverage: 1 Bit
Branch Coverage: 4 Bit
Condition Coverage: 7 Bit
Abstract

Code Coverage gets more important in future projects (standards and test end criterion)

Different coverage levels with different time and effort for implementation

Approaches to solve the basic problems with Code Coverage on small embedded systems are shown. (Limited Memory and CPU, Interfaces)

Most Code Coverage tools distinguish in that. Default tools are usually only practical on desktop systems with less limited resources.

Recommendation : Evaluate different Code Coverage tools for their embedded systems capabilities!
Questions

Roland Bär
baer@verifysoft.com

Andreas Behr
behr@verifysoft.com

Daniel Fischer
daniel.fischer@hs-offenburg.de