Code-Coverage on Embedded Systems

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

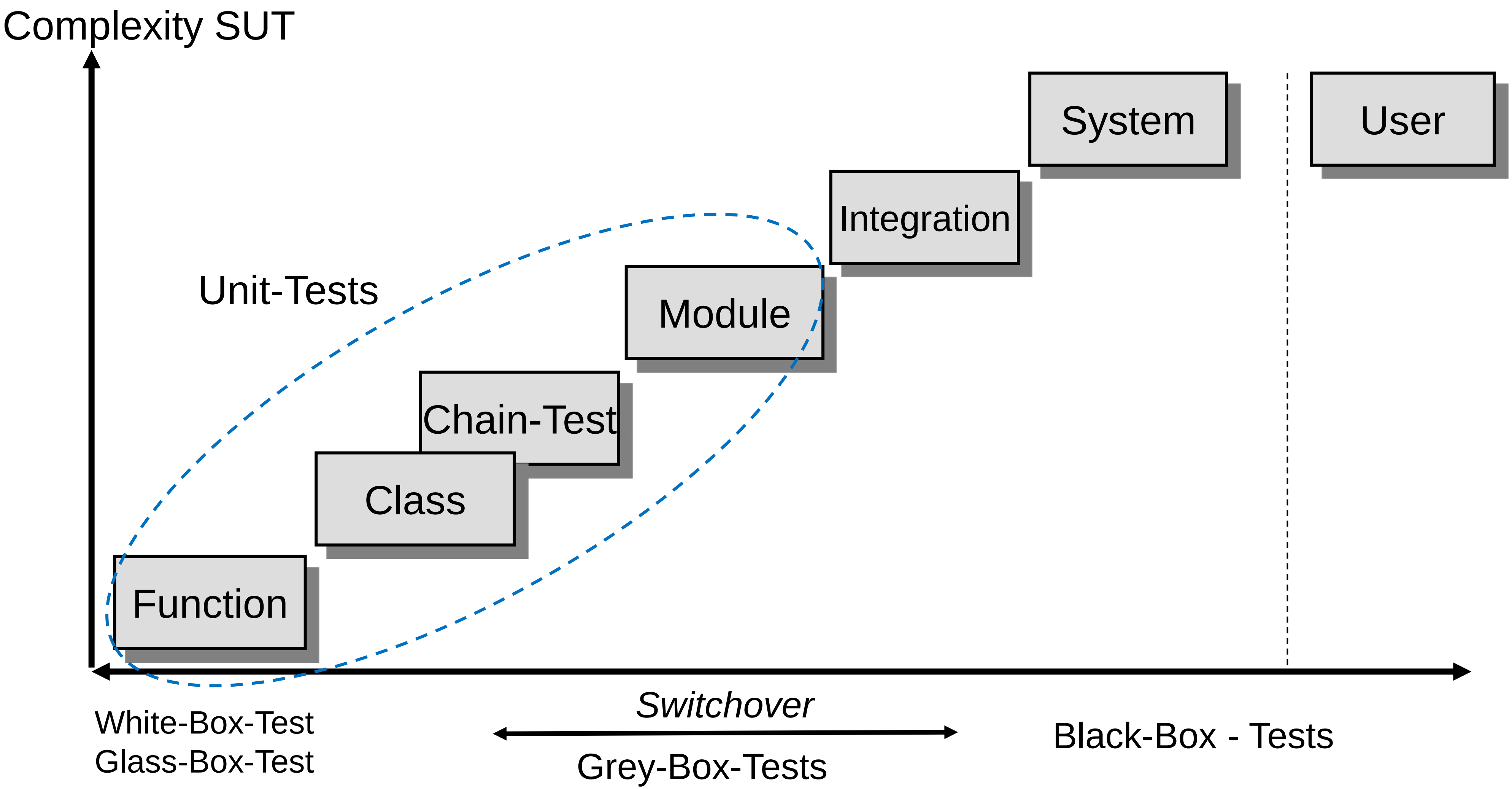
Andreas Behr
Verifysoft Technology GmbH
In der Spöck 10-12
77656 Offenburg
behr@verifysoft.com
Tel.:0781-127-81189

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



Developer
Code-Coverage

Testcoverage

Tester
Requirement-Coverage

Basics – Techniques / Principle

Cause-Reason-Graph

Static Testing

Back-to-Back Testing

Classification Tree Method (CTM)

CRUD

Rare Event Testing

Realtime Testing

Mutation Testing

Equivalent Classes
Multidimensional Equivalent Classes
Boundary Value Analysis
Critical Value Analysis
Informal Tests
Smoke Tests
Basis

Zufallsgesteuerter Test

Rare Event Testing

Last Tests

Monkeytest

Recovery Tests

Fuzzing (Fuzz Testing)

Stress Tests

Evolutionary Testing

Advanced

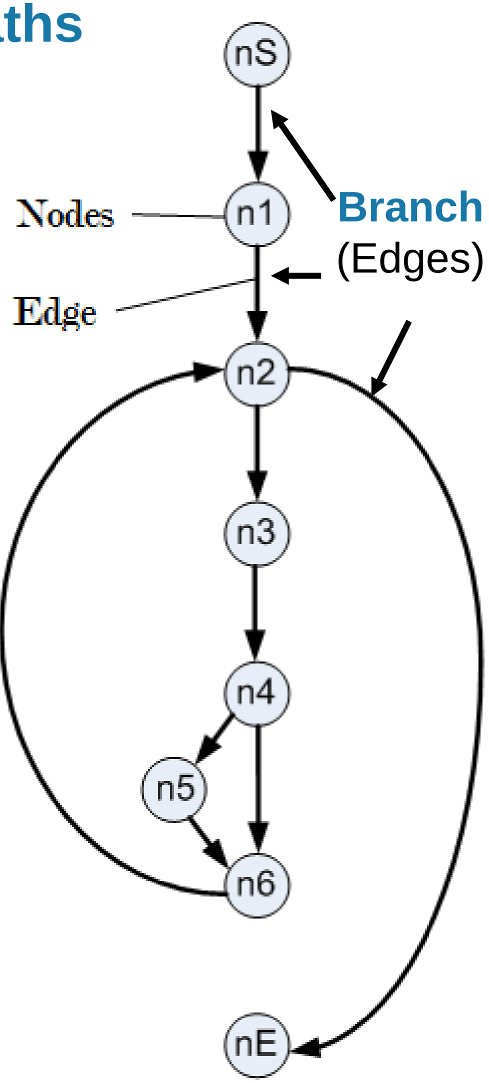
Control Flow Oriented Testing

Pairwise Testing

Established test technique for critical Embedded Systems
Test-End criterion (White-Box-Tests)
Necessary for gratification of several standards

Basics– Control Flow Oriented Testing

Paths



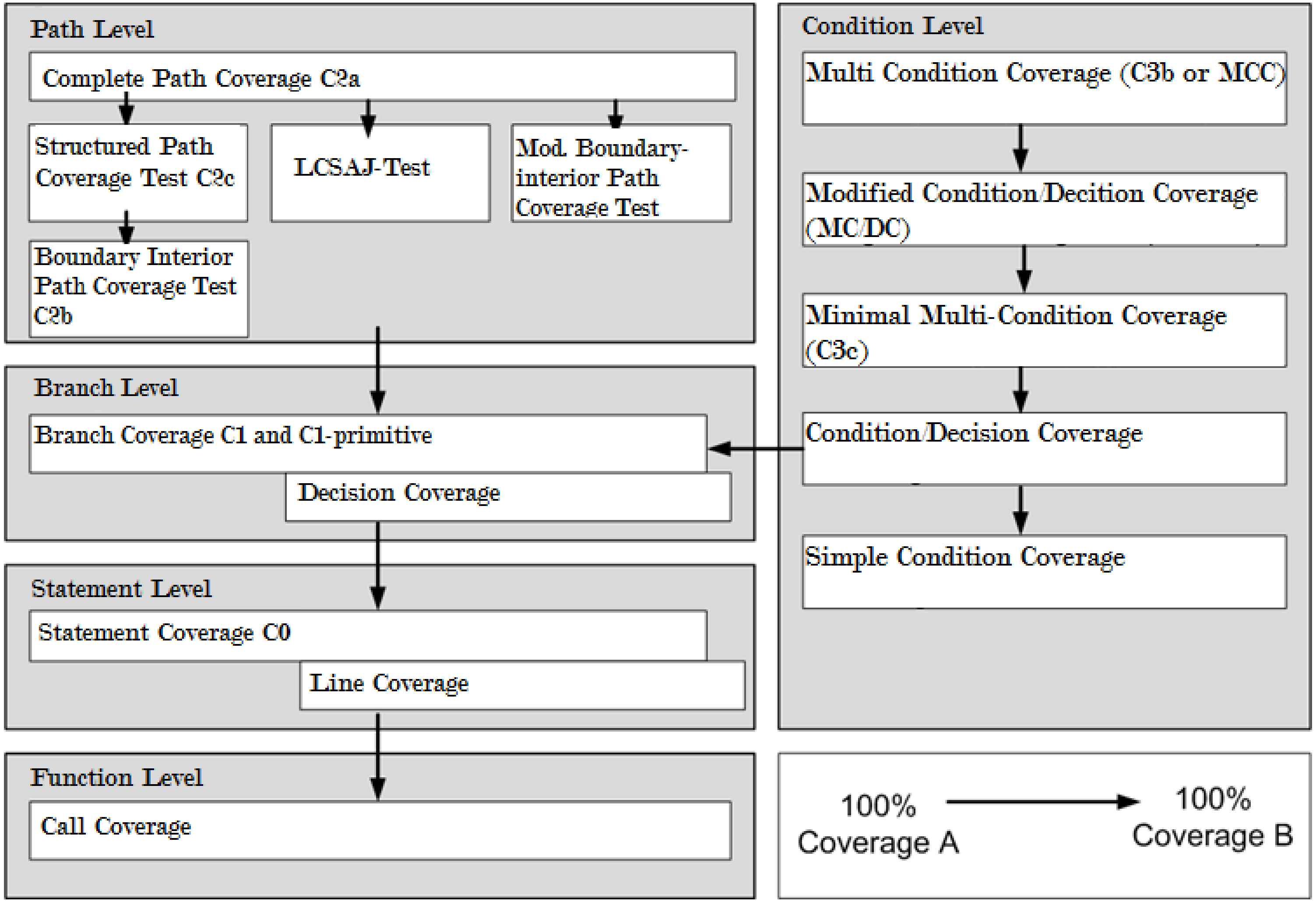

```
void ZaehleZchn (int& VokalAnzahl, int& GesamtAnzahl)
{
unsigned char Zchn;

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();
}

}
```

Functions

Statements
(Nodes)

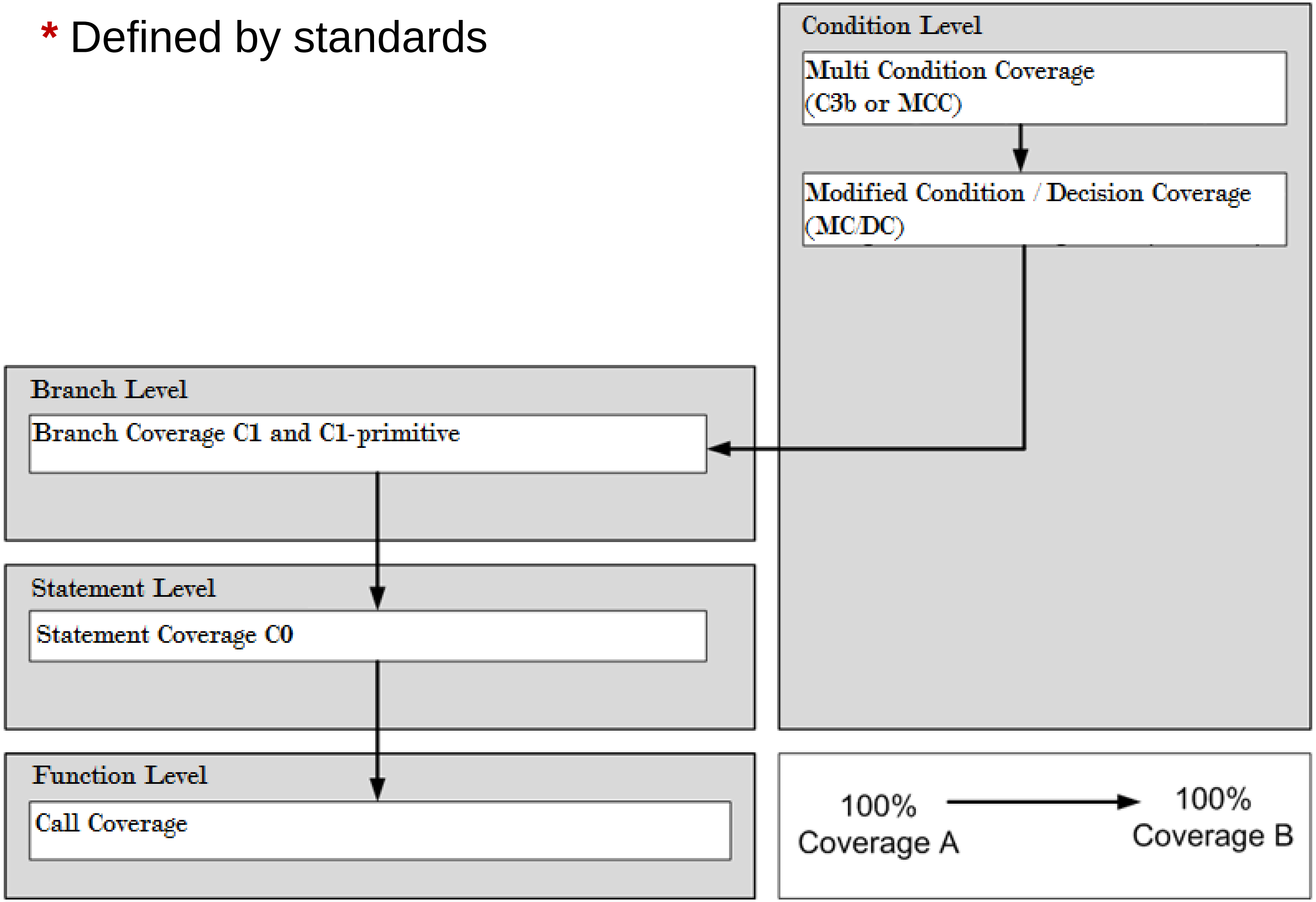
Conditions

Coverage Levels – Overview



Coverage Levels – Important Levels

* Defined by standards



Coverage Levels – Function Level

```
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("Function Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1,goo(1,0,1));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("Function Coverage");
```

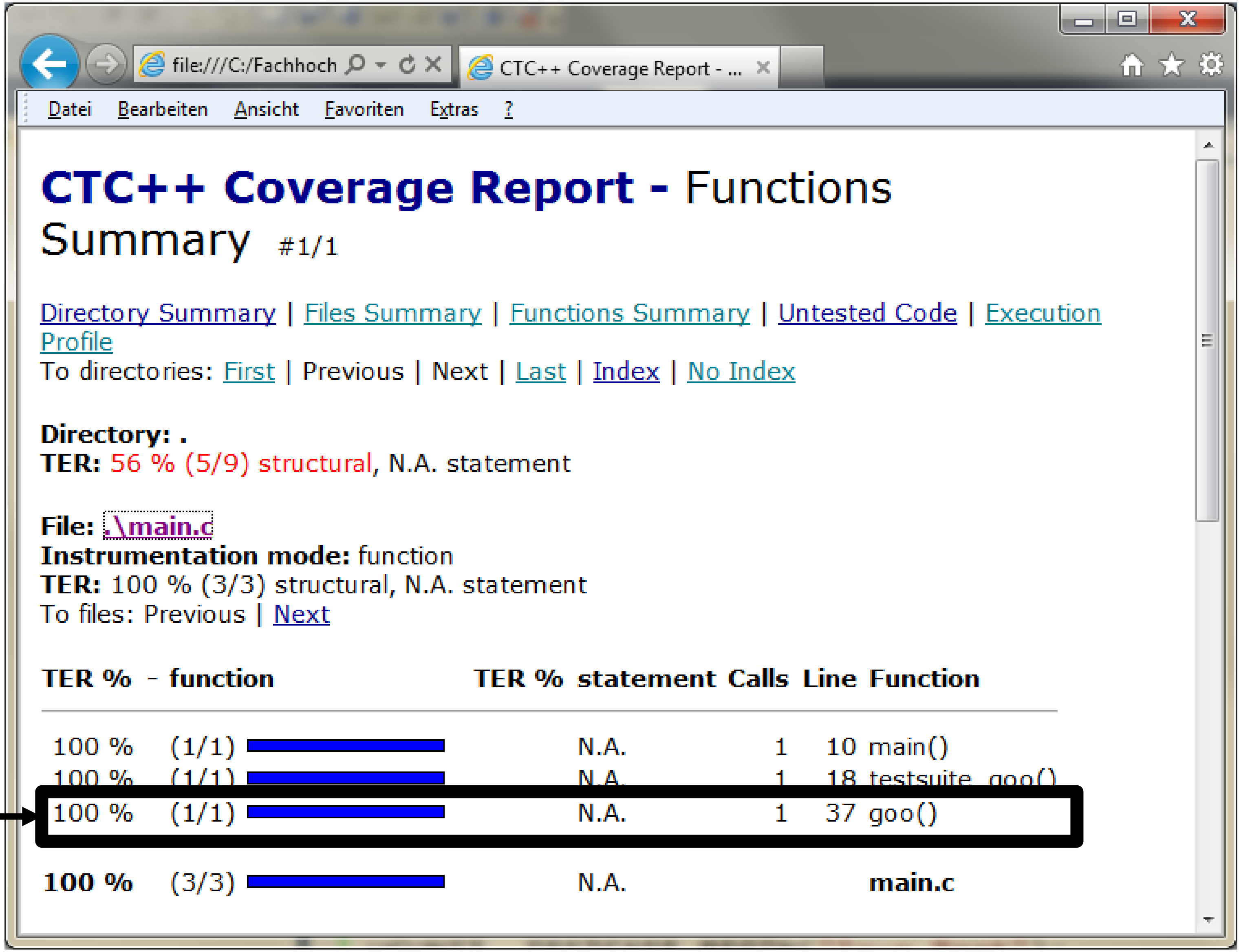
```
==============================================
TESTCASE:Function Coverage:BEGIN
----------------------------------------------
CHECK: Line 22:IsEqual(1,goo(1,0,1)):PASSED
CHECK: Line 23:Checklist():PASSED
----------------------------------------------
TESTCASE:Function Coverage:PASSED
==============================================

**************************************
Testcases: failed: 0
           passed: 1
Checks:    failed: 0
           passed: 1
**************************************
```

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 #1/1

Directory Summary | Files Summary | Functions Summary | Untested 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

TER % - function			TER % statement	Calls	Line	Function
100 %	(1/1)		N.A.	1	10	main()
100 %	(1/1)		N.A.	1	18	testsuite_goo()
100 %	(1/1)		N.A.	1	37	goo()
100 %	(3/3)		N.A.			**main.c**

Coverage Levels – Statement Coverage C0

```
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");
```

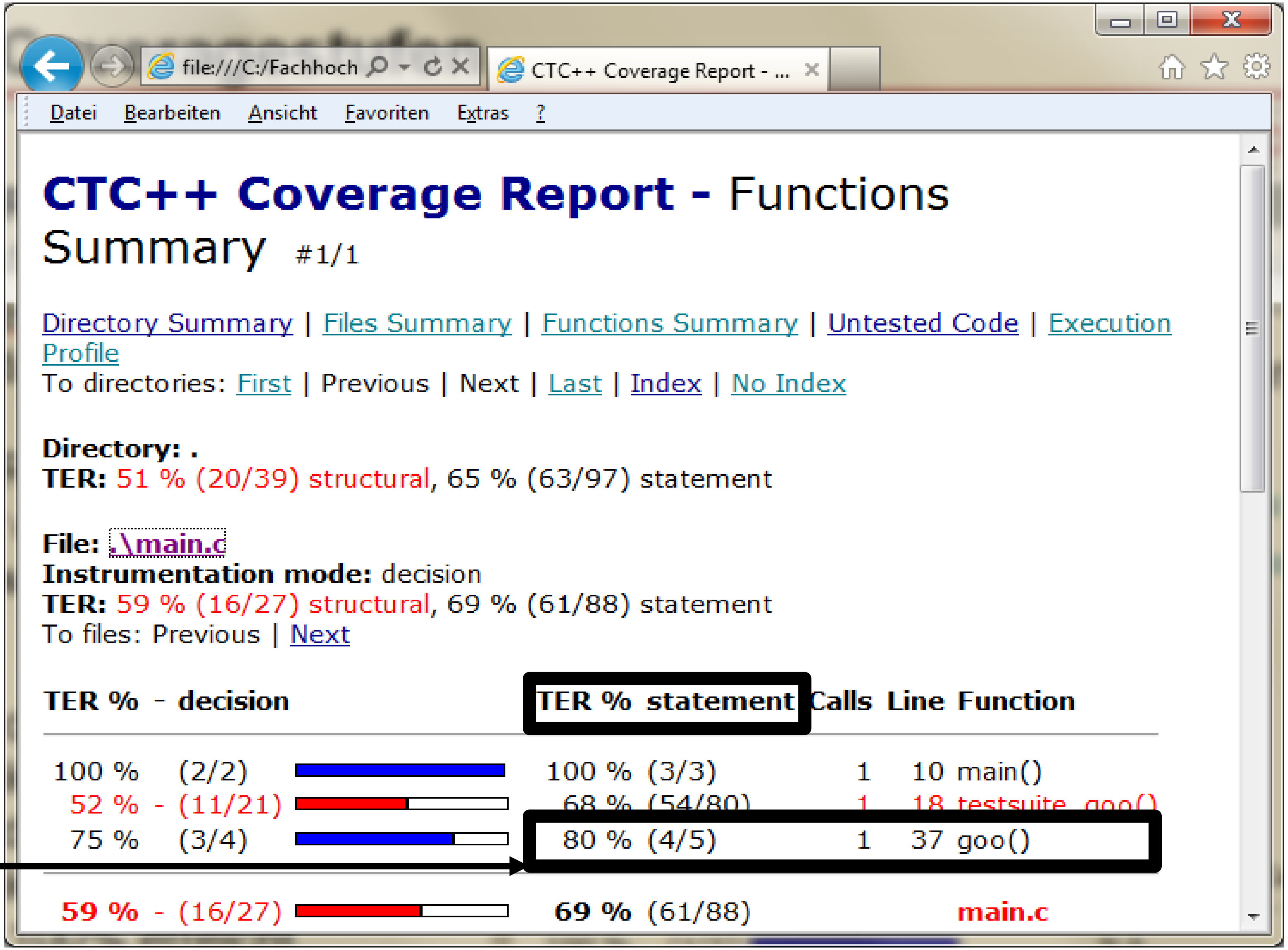
```
==============================================
TESTCASE:Statement Coverage:BEGIN
----------------------------------------------
CHECK: Line 22:IsEqual(1,goo(1,0,1)):PASSED
CHECK: Line 23:Checklist():PASSED
----------------------------------------------
TESTCASE:Statement Coverage:PASSED
==============================================

**************************************
Testcases: failed: 0
           passed: 1
Checks:    failed: 0
           passed: 1
**************************************
```

Coverage Levels – Decision Coverage

TER
Test
Effectiveness
Ratio

TER depending on selected coverage Level

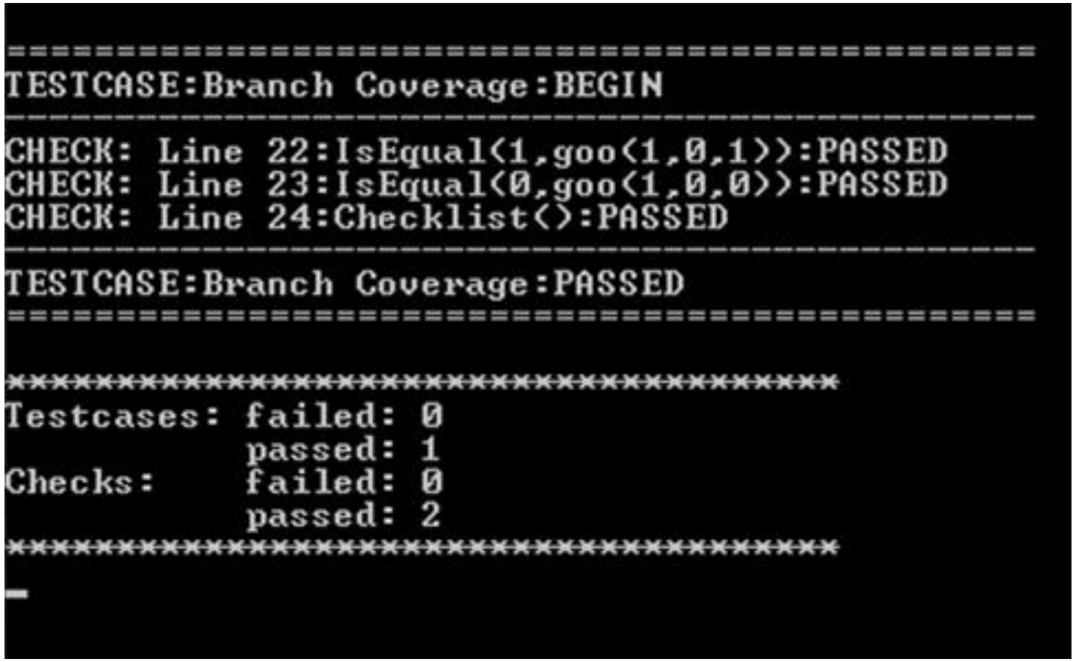
There‘s only 75% Coverage for function goo() using decision coverage



Coverage Levels – Branch Coverage C_1

```
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");
```



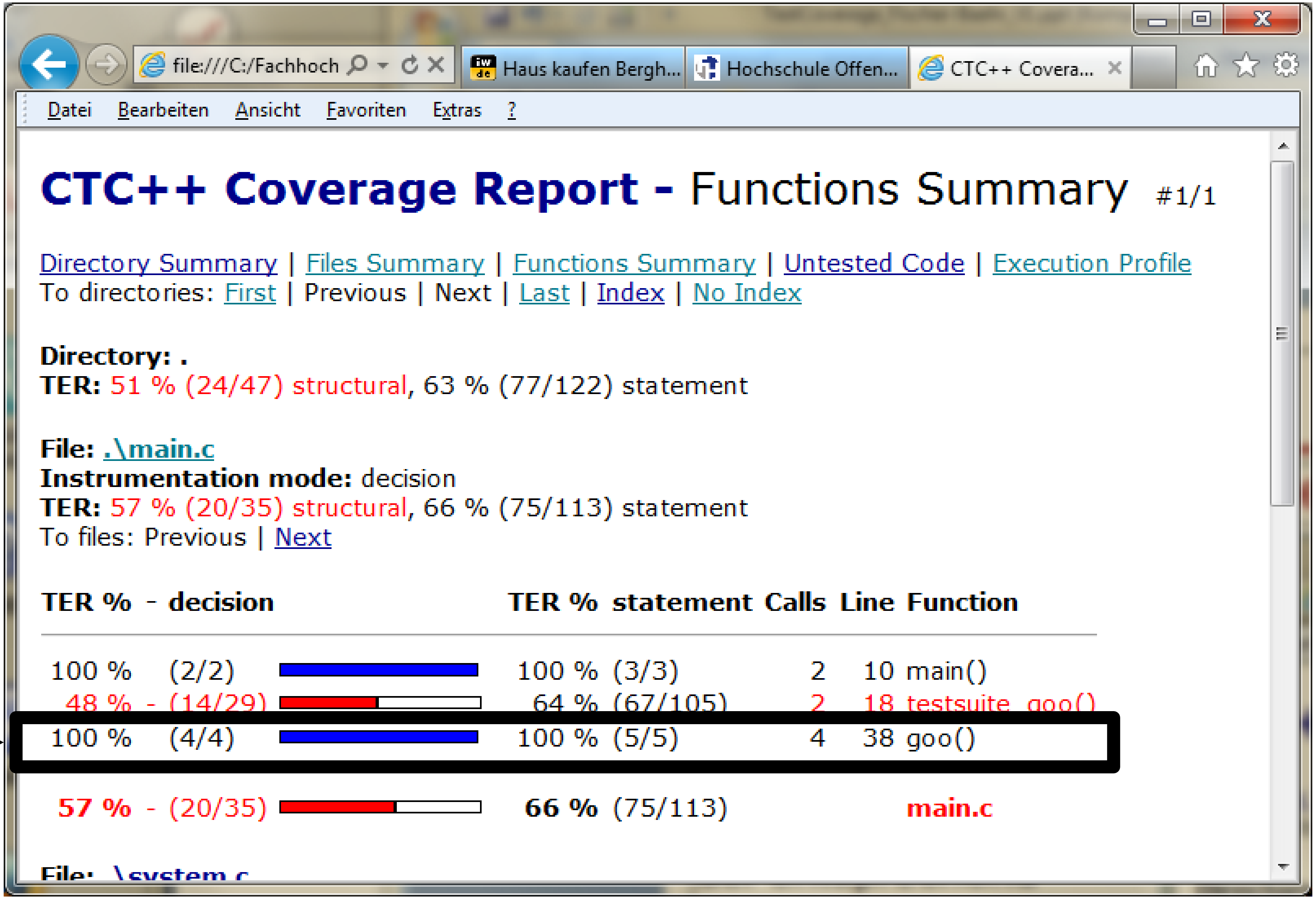
```
==============================================
TESTCASE:Branch Coverage:BEGIN
----------------------------------------------
CHECK: Line 22:IsEqual(1,goo(1,0,1)):PASSED
CHECK: Line 23:IsEqual(0,goo(1,0,0)):PASSED
CHECK: Line 24:Checklist():PASSED
----------------------------------------------
TESTCASE:Branch Coverage:PASSED
==============================================

**************************************
Testcases: failed: 0
           passed: 1
Checks:    failed: 0
           passed: 2
**************************************
_
```

Coverage Levels – Branch Coverage C_1

TER
Test
Effectiveness
Ratio

TER depending on selected coverage Level

100% coverage for function goo() using decision coverage



Coverage Levels – Condition Level MC/DC

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

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

   return x;
}
```

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."

(a>0)	(b>0)	(c>0)	x
true	-	true	1
true	-	false	0
false	true	true	1
false	true	false	0
false	false	-	0

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

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

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

   return x;
}
```

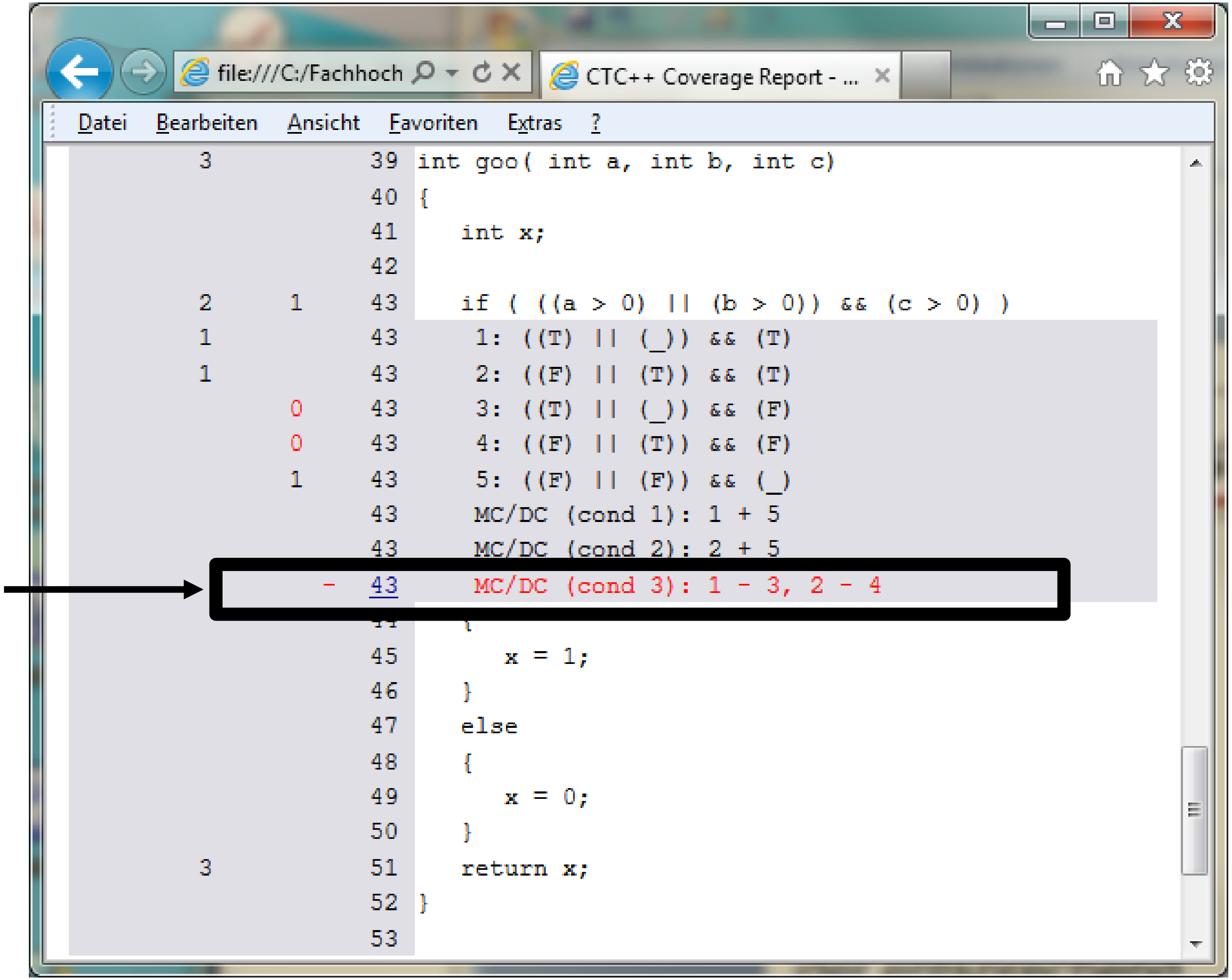
```
UCUNIT__TESTCASE_BEGIN("MC/DC Coverage");
UCUNIT__CHECKLIST_BEGIN(UCUNIT__ACTION_WARNING);
UCUNIT__CHECK_IS_EQUAL(1,goo(1,0,1));
UCUNIT__CHECK_IS_EQUAL(1,goo(0,1,1));
UCUNIT__CHECK_IS_EQUAL(0,goo(0,0,0));
UCUNIT__CHECKLIST_END();
UCUNIT__TESTCASE_END("MC/DC Coverage");
```

```
==================================================
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
==================================================

****************************************
Testcases: failed: 0
           passed: 1
Checks:    failed: 0
           passed: 3
****************************************
```

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



Coverage Levels – Multicondition Coverage MCC

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

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

   return x;
}
```

(a>0)	(b>0)	(c>0)	x
true	-	true	1
true	-	false	0
false	true	true	1
false	true	false	0
false	false	-	0

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

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

Standards / Norm – DIN EN 61508

General Industry

SIL: Safety Integrity Level

Method		SIL 1	SIL 2	SIL 3	SIL 4
…	…	…	…	…	…
7a	Function Coverage	++	++	++	++
7b	Statement Coverage	+	++	++	++
7c	Branch Coverage	+	+	++	++
7d	MC/DC	+	+	+	++

Table B.2 from DIN EN 61508-3

++ Very recommended method, must be reasonable if not used
+ Recommended method

Standards / Norm – ISO 26262

Automotive

ASIL: Automotive Safety Integrity Level

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

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

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

Table15 (Software Architectural Level), ISO 26262-6

++ Very recommended method, must be reasonable if not used

\+ Recommended method

Standards / Norm – DO 178B/C and IEC 62304

Aerospace

DO-178B/C

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

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

Medical Systems

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)

Instrumentation – Basics

- 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

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

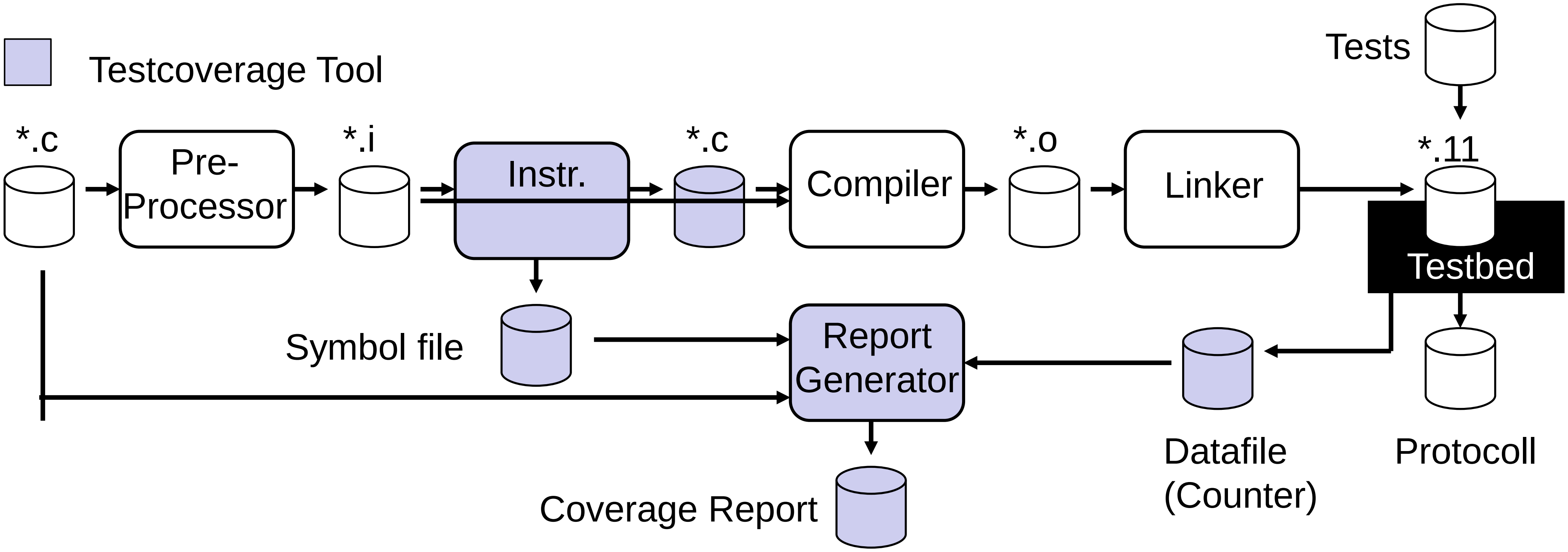
Without comma operator

```
if ( (a<0&&(counter1++||1)) || (counter2++&&0) )
  { /* … */}
else
  { /* … */}
```

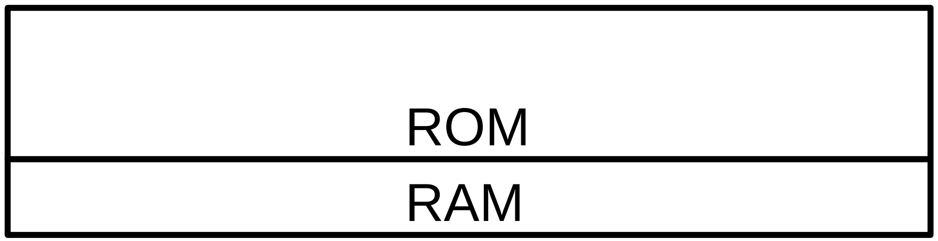
Using comma operator

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

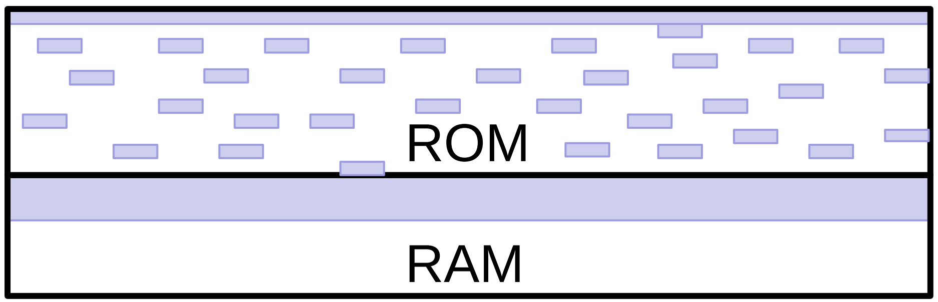
Instrumentation – Tool-Chain



Memory usage of target without instrumentation



Memory usage of target with instrumentation

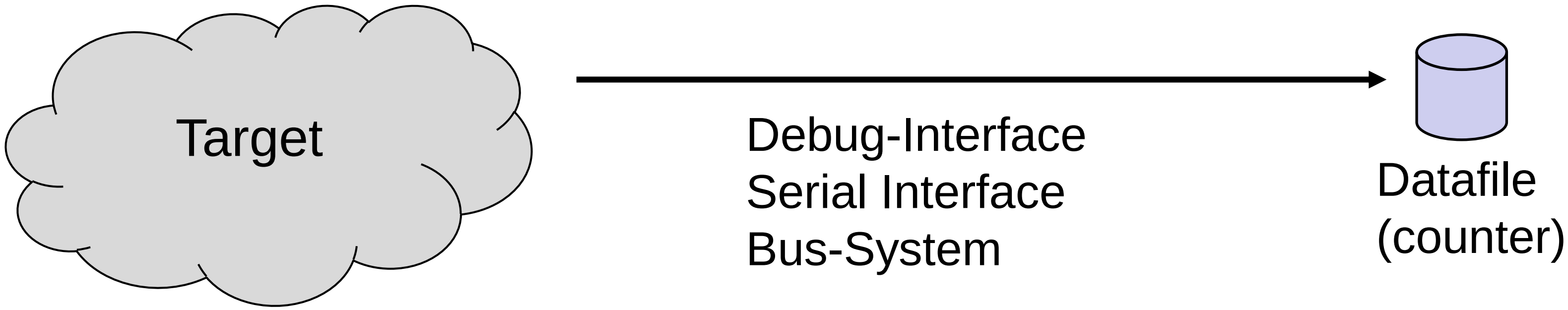


Small Targets – Limited Resources

- **RAM**
- **ROM**

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

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

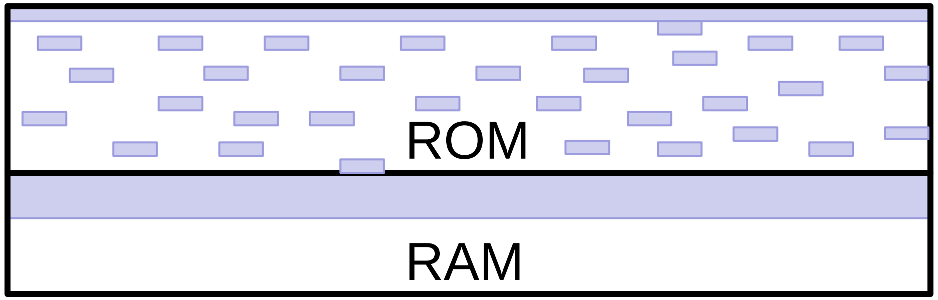


- Limited amount of interfaces on the target device (transfer of datafile)

Respect for additional interfaces for testing in the hardware design (design for test)

Small Targets – Limited Memory (1)

Memory on target
with instrumentation

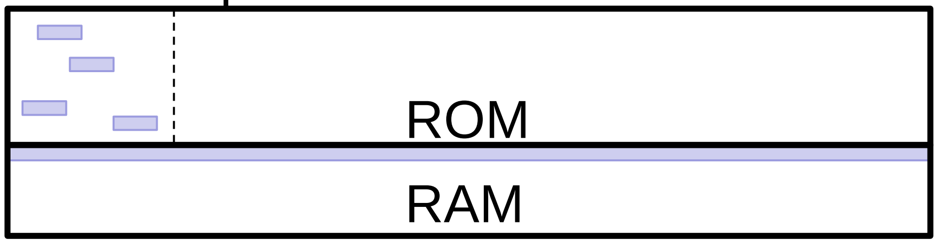


In case of
insufficient memory

Memory lack:
20% ROM
80% RAM

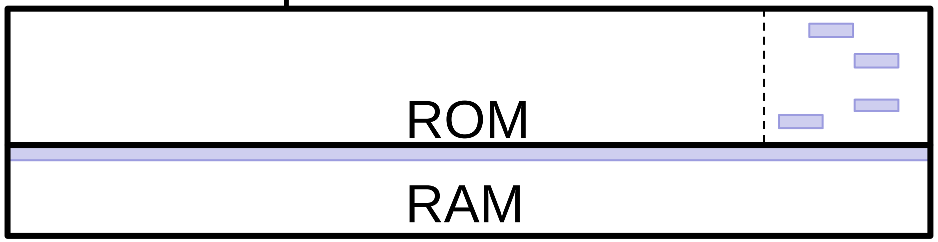
Approach: **Partial** **instrumentation**
Reduces RAM- and ROM-requirements

Memory on target
with partial instrumentation



…

Memory on target
with partial instrumentation



Automating white and black box tests is recommended

Small Targets – Limited Memory (2)

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-

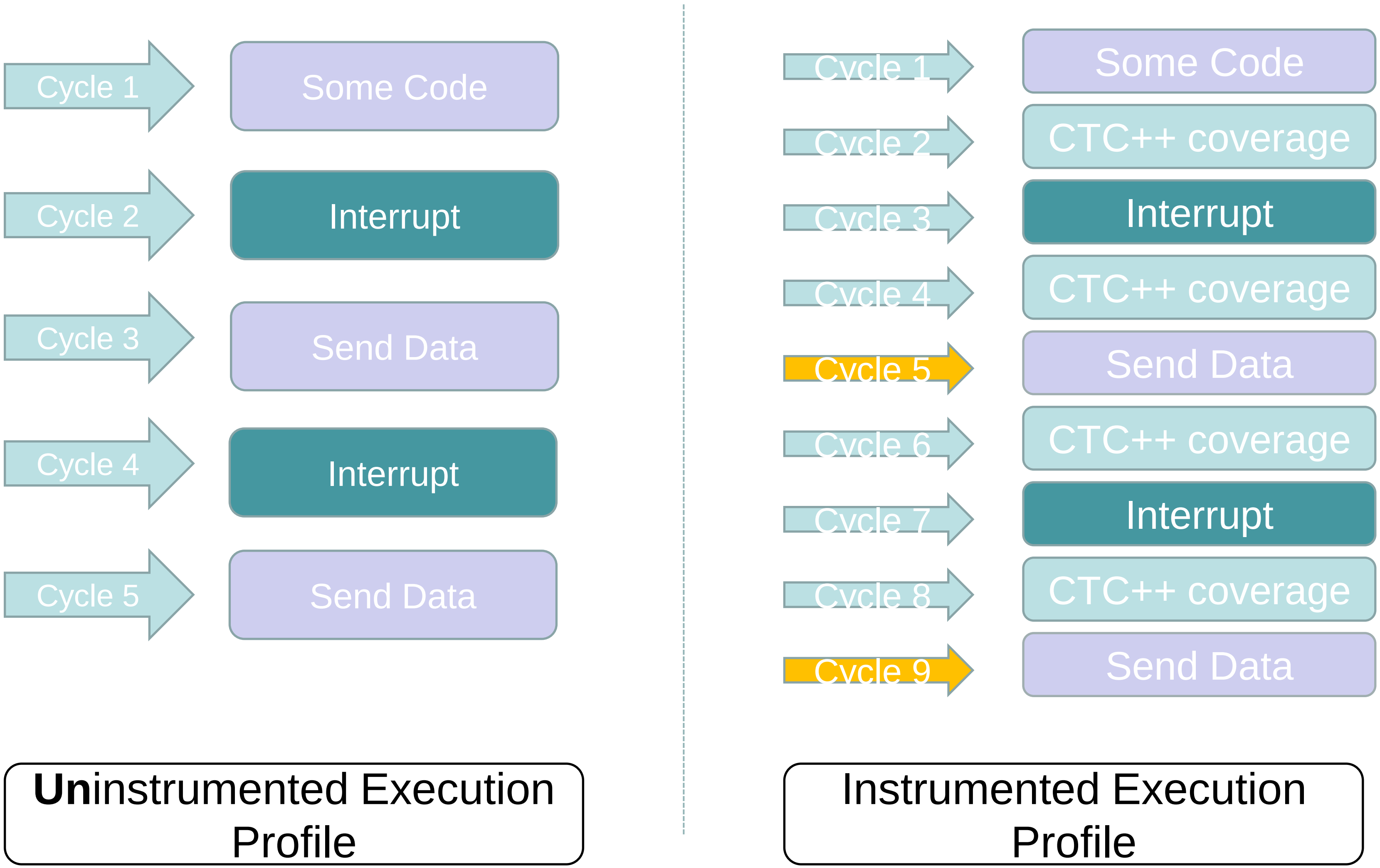
```
ORL 0x200, 2
```

C51

```
SETB 0x1602
```

Melexis

Small Targets – Limited CPU Time



Example – Z80 with sdcc Compiler and Testwell CTC++

```
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 i 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