#### Model-Based Testing (DIT848 / DAT261) Spring 2017

#### Lecture 3 White Box Testing - Coverage

Gerardo Schneider Dept. of Computer Science and Engineering Chalmers | University of Gothenburg

Some slides based on material by Magnus Björk, Thomas Arts and Ian Somerville

## What have we seen so far?

- V&V: Validation & Verification
- The V model
  - Test levels
- Black box testing
- (Extended) Finite State Machines

#### Any question?



# White box testing

#### Do I need more test cases?

- I think I have test cases for all aspects of the specification,
- I've added test cases for boundary values,
- ...guessed error values,
- ...and performed 10.000 random test cases.

#### Is that enough?

#### Do I need more test cases?

#### • The bad:

• There is no way to know for certain

#### • The good:

- There are techniques that can help us
  - Identify some aspects that may otherwise go unnoticed
  - Give some criteria for "enough"

Coverage techniques

#### Black box and White box testing

Glass box testing

software

Black box testing: Test tactic in which the test object is addressed as a box one **cannot** open.

A test is performed by sending a sequence of input values and observing the output without using any knowled the test object internals. Also called Structural testing or

White box testing: Test tactic in which the test object is addressed as a box one can open

A test is performed by sending a **sequence of input values** and observing the output and internals while explicitly using knowledge about the test object internals

#### What white box testing is not

- White box testing is (typically) NOT:
  - Black box test cases that refer to internal constructs

Id: calc.h/pressPlus/1 Purpose: verifying that the correct operation is stored Precondition: state is a CalcStatePtr pointing to a valid calculator state Action: call pressPlus(state) Expected outcome: state->op = Plus Refers to internal representation,

- Drawbacks of test cases like this:
  - Test properties not in specification
  - Fail if internal representation is changed
  - And when they fail, it may be hard to understand why/where
- ...but sometimes they may be necessary
  - Unit testing of internal functions

not interface

# What white box testing is

- 'Normal' white box testing is:
  - Black box testing, combined with tools that analyze implementation of specific properties

- White box techniques covered in this lecture
  - Code coverage analysis
    - Are there parts of the code that are not executed by any test cases?
    - Used to find inadequacies in the test suite
  - In this lecture: Some examples in C (GCov) and functional programming

# Coverage checking

# Coverage checking

The structure of the software is used to determine whether a set of tests is a sufficient/adequate one

Thus:

- 1) Decide which structure to consider
- 2) Decide upon coverage criteria

3) Find a set of tests such that this structure is covered according to the decided criteria

#### Common structures

- Function coverage
  - All functions have been executed

#### Entry/exit coverage

- All entry and exit points of all functions have been executed
  - Entry points: all calls to a function
  - Exit points: each return statement
- Statement coverage (lines of code)
  - All lines of code have been executed
- **Branch coverage** (condition coverage)
  - If: both "if" and "else" part, even if no else part
  - While loop: both with true and false conditions
  - Lazy logical ops (&& and ||): first arguments both true and false

#### Path coverage

- All possible routes through the code (combination of branches)
- Infinitely many if there are while loops (only feasible for small functions)



- A function from (simple) ATM system
- Representation of amount of cash in machine:
  - [(100,23),(500,11)] means that machine contains:
    - 23 bills of 100kr
    - 11 bills of 500kr
  - We call it "pair-notes"

- Function to look at: subtract
  - subtract a number of notes notes remaining in the ATM
  - subtract(<list\_of\_pair-notes\_to\_withdraw>, <list\_of\_pair-notes\_in\_Bank>)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

```
Expected output: [{100,100}, {500,1}]
```

```
subtract([],Notes) ->
```

Notes;

```
subtract([{Value,Nr}|Rest],Notes) ->
```

subtract(Rest, subtract2(Value, Nr, Notes)).

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

subtract([{500,2}],[{100,100},{500,3}]).

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

```
subtract([], subtract2(500, 2, [{100, 100}, {500, 3}])).
```

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

```
subtract([], subtract2(500, 2, [{100, 100}, {500, 3}])).
```

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

subtract([], subtract2(500, 2, [{100, 100}, {500, 3}])).

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

```
Test case: subtract([{500,2}],[{100,100},{500,3}]).
```

```
subtract([],[{100,100}|subtract2(500,2,[{500,3}])]).
```

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

```
Test case: subtract([{500,2}],[{100,100},{500,3}]).
```

```
subtract([],[{100,100}|subtract2(500,2,[{500,3}])]).
```

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

subtract([],[{100,100}|subtract2(500,2,[{500,3}])]).

```
subtract([],Notes) ->
```

Notes;

```
subtract([{Value,Nr}|Rest],Notes) ->
```

subtract(Rest, subtract2(Value, Nr, Notes)).

subtract2(Value, N, [{Value, M} | Notes]) when M>=N ->
 [{Value, M-N}];
subtract2(Value, N, [{V, M} | Notes]) ->
 [{M, V} | subtract2(Value, N, Notes)].

Test case: subtract([{500,2}],[{100,100},{500,3}]).

```
subtract([],[{100,100}|[{500,3-2}]]).
```

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

Test case: subtract([{500,2}],[{100,100},{500,3}]).

Evaluates to

```
subtract([],[{100,100}|[{500,1}]]).
```

```
subtract([],Notes) ->
```

#### Notes;

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest, subtract2(Value, Nr, Notes)).
```

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
  [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
  [{M, V}|subtract2(Value, N, Notes)].
```

```
Test case: subtract([{500,2}],[{100,100},{500,3}]).
```

```
output: [{100,100}, {500,1}]
```

All statements and all branches have been executed. Expected output. 24

Are we happy? Is the program correct?

```
What happen with the following?
```

```
Test case: subtract([{500,2}],[{100,5}, {500,3}]).
```

It will not work!! The case [100,100] was a particular case; we inverted values!

```
subtract([],Notes) ->
```

Notes;

```
subtract([{Value,Nr}|Rest],Notes) ->
```

subtract(Rest, subtract2(Value, Nr, Notes)).

```
subtract2(Value, N, [{Value, M}|Notes]) when M>=N ->
 [{Value, M-N}];
subtract2(Value, N, [{V, M}|Notes]) ->
 [{V, M}|subtract2(Value, N, Notes)].
```

Are we happy now? Is the program correct?

```
What happen with the following?
```

```
Test case: subtract([{100,2}],[{100,100},{500,3}]).
```

It will **not** work!! We are "loosing" the suffix of the list!

```
subtract([],Notes) ->
```

Notes;

```
subtract([{Value,Nr}|Rest],Notes) ->
```

subtract(Rest, subtract2(Value, Nr, Notes)).

subtract2(Value, N, [{Value, M} | Notes]) when M>=N ->

```
[{Value,M-N}|Notes];
```

```
subtract2(Value, N, [{V, M} | Notes]) ->
[{M, V} | subtract2(Value, N, Notes)].
```

# Coverage (example in C)

Actually:

else { }

void printPos(int n) {
 Should be: <=
 printf("This is ");</pre>

if (n < 0)
 printf("not ");</pre>

printf("a positive integer.\n");

return; Code originally from Wikipedia

#### Test case 3

Action: call printPos(0)

Expected outcome:

"This is not a positive integer" (printed on stdout)



Boundary value

Action: call printPos(-1)

Test case 1

Expected outcome:

"This is not a positive integer" (printed on stdout)

Are we

happy with

coverage?

*Coverage*: 100% statement, 50% branch, 50% path

Test case 2

Action: call printPos(1)

Expected outcome:

"This is a positive integer" (printed on stdout)

**Coverage:** 100% statement, branch & path (including previous)<sup>27</sup>

Group exercise

 Come up with pieces of code (in any language) and a few test cases such that following conditions are met, or motivate why it is impossible:

- 1. 100% branch coverage, less than 100% path coverage
- 2. 100% path coverage, less than 100% statement coverage
- 3. 100% function coverage, less than 100% exit point coverage

#### Suggestions

1: 100% branch coverage, less than 100% path coverage

void foo(int n) {

if(n>0)

```
printf("Positive\n");
```

else

```
printf("Not positive\n");
```

if(n % 2)

```
printf("Odd\n");
```

else

printf("Even\n");

Id: Test case 1: pos/odd Action: call foo(1) Expected outcome: "Positive" and "Odd"

Id: Test case 2: neg/even
Action: call foo(-2)
Expected outcome: "Not positive" and "Even"

> Path positive/even not covered!

## Suggestions

# 2: 100% path coverage, less than 100% statement coverage

int main(void) {

printf("Hello world\n");

return 0;

printf("Unreachable code\n"); }

Id: Test case 1 Action: run main Expected outcome: "Hello world" printed

This statement is **not** covered!

## Suggestions

# 3: 100% function coverage, less than 100% exit point coverage

int abs(int n) {

if(n < 0)

Id: abs/1 Action: call abs(-17) Expected outcome: returns 17

return –n;

return n;

Didn' t cover this exit point

## White box test design

**Strategy** for using **coverage measure**:

- 1. Design test cases using *black box test design* techniques
- 2. Measure code coverage
- Design test cases by inspecting the code to cover unexecuted code

100% coverage does **not** mean there are no errors left!

So, code coverage should be seen as complementary method – It cannot do the thinking for you

However, coverage analysis catches aspects that are otherwise easily forgotten

#### Adding test cases after coverage analysis

 The new test cases should still be black box test cases, not referring to the code

Good test case:

*Id:* abs/2 *Purpose:* Execute abs on negative integer *Action:* call abs(-17) *Expected outcome: Call returns 17* 

Bad test case:

Id: abs/2 Purpose: Cover line 3 of abs Action: call abs(-17) Expected outcome: Line 3 executed Refers to code

## Practical coverage analysis

In order to measure coverage, most languages require a compile flag to enable keeping track of line numbers during execution

Consequences:

- Performance changes, hence timing related faults may be undiscoverable
- Memory requirements change, hence one may experience problems running in embedded devices

There are a lot of tools available for many languages

# Coverage vs Profiling

Both methods count executions of entities, but purpose is different

• Coverage tool: find out which entities have been executed, to establish confidence in verification

 Profiler: identify bottlenecks and help programmer improve performance of software

## Example: Gcov (C)

The program **avg** (short for "average") reads a text file, whose name is given as a command line argument, containing a number of integers, and reports the average value of all the integers. The program has been implemented in C (see below and next page), and the following small test suite has been developed by a programmer to start testing the system:

Test case avg.1: Normal integers Prerequisites: The file avgtest1.txt contains "10 15 35" Action: Run ./avg avgtest1.txt Expected outcome: The program prints "The average is 20"

Test case avg.2: Negative numbers Prerequisites: The file avgtest2.txt contains "-2 2 -6" Action: Run ./avg avgtest2.txt Expected outcome: The program prints "The average is -2"

Executing this test suite together with gcov reveals that there is untested code, the tool giving the message "Lines executed: 63.33% of 30". The actual output from gcov can be seen in next slide.

NOTE: The uncovered statements are those lines preceded with #### 36

#### Example: GCov

	<ul> <li>-: 41: const char* filename;</li> </ul>
-: 1:#include <stdio.h></stdio.h>	-: 42:
-: 2:#include <stdlib.h></stdlib.h>	2: 43: if(argc < 2) {
-: 3:	#####: 44: printf("Error: missing argument\n"):
-: 4:// readInts: read a file containing integers, and return their	######: 45:
-: 5:// sum and the number of integers read.	$\frac{1}{1}$
	40. $\}$
-: /:#define READINTS_SUCCESS 0 // Indicates success	2: $47$ : filename = argv[1];
-: 0:#define READINTS_FILEERR 1 // the file could not be read	-: 48:
- 10 <sup>.</sup>	2: 49: switch(readInts(filename, ∑, &length)) {
2: 11:int readInts(const char* filename, int* sumRslt, int* lengthRslt){	-: 50: case READINTS_FILEERR:
2: 12: FILE* file = fopen(filename, "r");	#####: 51: printf("Error reading file %s\n", filename);
2: 13: if(!file)	#####: 52: exit(EXIT_FAILURE);
#####: 14: return READINTS_FILEERR;	-: 53:
-: 15:	-: 54: case READINTS SYNTAXERR:
2: 16: *sumRsIt=0;	##### 55: printf("Syntax error in file %s\n" filename);
2: 17:  "length Rsit=0;	
-10. where $(1)$	$\frac{1}{1}$
8: 20: if(fscanf(file, "%d", &theInt) == 1) {	
-: 21: // Successfully read integer	-: 58: case READINTS_SUCCESS:
6: 22: (*sumRslt) += theInt;	-: 59: default:
6: 23: (*lengthRslt)++;	-: 60: break;
-: 24: } else {	-: 61: }
-: 25: // Could not read integer. End of file or syntax error?	-: 62:
2: 26: if(feof(file)) {	2: 63: if(length==0) {
-: 27: // End of file	#####: 64: printf("Error: no integers found in file %s\n",
2: 20: return READINTS SUCCESS:	filename);
-: 30: } else {	#####: 65: exit(EXIT_FAILURE);
-: 31: // Syntax error	-: 66: }
######: 32: fclose(file);	-: 67:
#####: 33: return READINTS_SYNTAXERR;	2. 68. printf("The average is %d\n" sum / length).
-: 34: }	-: 60·
-: 35: }	
0. 30. } _· 37·\	2. 70. letuin EXIT_0000E00,
-: 38:	/1.}

2: 39:int main(int argc, char\*\*argv) {

-: 40: int sum, length;

Group exercise

 Provide additional test cases so that all cases together yield 100% statement coverage

 Write complete test cases as shown in the test cases above, and indicate which lines each test case cover

#### Exercise: Proposed solution

-To cover I.64-65 (avgtest3.txt is an empty file - Test case avg3:

Prerequisites: The file avgtest3.txt exists but is empty

Action: ./avg avgtest3.txt

Expected outcome: An error is reported, stating that the input is empty

- To cover I.32-33 and 55-56 - Test case avg4:

Prerequisites: avgtest4.txt contains a list of nonintegers

Action: ./avg avgtest4.txt

Expected outcome: An error message is given that there is a syntax error

- To cover I.14 and 51-52 - Test case avg5:

Prerequisites: Call the function with an argument, not a file

Action: -./avg "asdfdf" (or ./avg non\_existing\_file.txt)

Expected outcome: An error reading file could be given

- To cover I.44-45 Test case avg5:

Prerequisites: None

Action: ./avg

given

Expected outcome: Error missing argument is

Any problem understanding the solution? Try it yourself with GCoV!





#### **Testing: The Bigger Picture**

Monday March 27

• NOTE: Lecture starts at 8:30!