Systeme hoher Qualität und Sicherheit
Universität Bremen WS 2015/2016

Lecture 08 (30-11-2015)

Testing

Christoph Lüth   Jan Peleska   Dieter Hutter

Universität Bremen

Where are we?

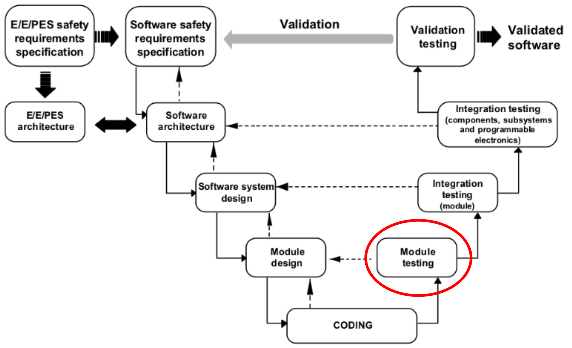
- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10 and 11: Software Verification (Hoare-Calculus)
- 12: Model-Checking
- 13: Concurrency
- 14: Conclusions

Your Daily Menu

What is testing?

- Different **kinds** of tests.
- Different test methods: **black-box** vs. **white-box**.
- The basic problem: cannot test **all** possible inputs.
- Hence, coverage criteria: how to test **enough**.

Testing in the Development Cycle



What is Testing?

> Testing is the process of executing a program or system with the intent of finding errors.
>
> *Myers, 1979*

- In our sense, testing is selected, controlled program execution.
- The **aim** of testing is to detect bugs, such as
  - derivation of occurring characteristics of quality properties compared to the specified ones;
  - inconsistency between specification and implementation;
  - or structural features of a program that cause a faulty behavior of a program.

> Program testing can be used to show the presence of bugs, but never to show their absence.
>
> *E.W. Dijkstra, 1972*

The Testing Process

- Test cases, test plan, etc.
- System-under-test (s.u.t.)
- Warning -- test literature is quite expansive:

> Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.
>
> *Hetzel, 1983*

Test Levels

- **Component** tests and **unit** tests: test at the interface level of single components (modules, classes)
- **Integration test**: testing interfaces of components fit together
- **System test**: functional and non-functional test of the complete system from the user's perspective
- **Acceptance test**: testing if system implements contract details

Test Methods

- Static vs. dynamic:
  - With **static** tests, the code is **analyzed** without being run. We cover these methods as static program analysis later.
  - With **dynamic** tests, we **run** the code under controlled conditions, and check the results against a given specification.
- The central question: where do the **test cases** come from?
  - **Black-box**: the inner structure of the s.u.t. is opaque, test cases are derived from specification **only**;
  - **Grey-box:** some inner structure of the s.u.t. is known, eg. Module architecture;
  - **White-box**: the inner structure of the s.u.t. is known, and tests cases are derived from the source code;

Black-Box Tests

- Limit analysis:
  - If the specification limits input parameters, then values **close** to these limits should be chosen.
  - Idea is that programs behave **continuously**, and errors occur at these limits.
- Equivalence classes:
  - If the input parameter values can be decomposed into **classes** which are treated equivalently, test cases have to cover all classes.
- Smoke test:
  - “Run it, and check it does not go up in smoke.”

Example: Black-Box Testing

Example: A Company Bonus System
The loyalty bonus shall be computed depending on the time of employment. For employes of more than three years, it shall be 50% of the monthly salary, for employees of more than five years, 75%, and for employees of more than eight years, it shall be 100%.

- Equivalence classes or limits?

Example: Air Bag
The air bag shall be released if the vertical acceleration a_v equals or exceeds $15\ {}^{m}/_{s^2}$. The vertical acceleration will never be less than zero, or more than $40\ {}^{m}/_{s^2}$.

- Equivalence classes or limits?

Black-Box Tests

- Quite typical for **GUI tests**, or **functional testing**.
- Testing **invalid input**: depends on programming language – the stronger the typing, the less testing for invalid input is required.
  - Example: consider lists in C, Java, Haskell.
  - Example: consider ORM in Python, Java.

Other approaches: Monte-Carlo Testing

- In Monte-Carlo testing (or random testing), we generate **random** input values, and check the results against a given spec.
- This requires **executable** specifications.
- Attention needs to be paid to the **distribution** values.
- Works better with **high-level languages** (Java, Scala, Haskell) where the datatypes represent more information on an abstract level.
  - ScalaCheck, QuickCheck for Haskell
- Example: consider list reversal in C, Java, Haskell
  - Executable spec:
    - Reversal is idempotent.
    - Reversal distributes over concatenation.
  - Question: how to generate random lists?

White-Box Tests

- In white-box tests, we derive test cases based on the structure of the program (**structural testing**)
  - To abstract from the source code (which is a purely **syntactic** artefact), we consider the **control flow graph** of the program.

Def: Control Flow Graph (cfg)
- Nodes are elementary statements (e.g. assignments, **return**, **break**, . . .), and control expressions (eg. in conditionals and loops), and
- there is a vertex from *n* to *m* if the control flow can reach node *m* coming from *n*.

- Hence, **paths** in the cfg correspond to runs of the program.

A Very Simple Programming Language

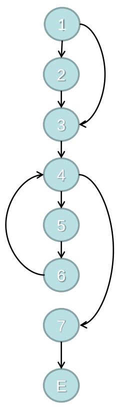
- In the following, we use a very simple language with a C-like syntax.
- **Arithmetic** operators given by
  $$a\ ::= x \mid n \mid a_1\ op_a\ a_2$$
  with x a variable, n a numeral, op_a arith. op. (e.g. +, -, *)
- **Boolean** operators given by
  $$b\ \coloneqq \mathrm{true} \mid \mathrm{false} \mid \mathrm{not}\ b \mid b_1 op_b\ b_2 \mid a_1 op_r\ a_2$$
  with op_b boolean operator (e.g. and, or) and op_r a relational operator (e.g. =, <)
- **Statements** given by
  S ::=
  $$[x \coloneqq a]^l \mid [skip]^l \mid S_1; S_2 \mid if\ [b]^l\ \{S_1\}\ else\ \{S_2\} \mid while\ [b]^l\ \{S\}$$

We may write the labels als comments

```
x:= a+ 10; /* 1 */ if (y < 3) /* 2 */ { x:= x+1; /* 3 */  } else { y:= y+1; /* 4 */ }
```


Example: Control-Flow Graph

```
if (x < 0) /* 1 */ {
  x := – x; /* 2 */
  }
z := 1; /* 3 */
while (x > 0) /*4*/ {
  z := z * y; /* 5 */
  x := x – 1; /* 6 */
}
return z /* 7 */
```



An execution path is a path though the cfg.

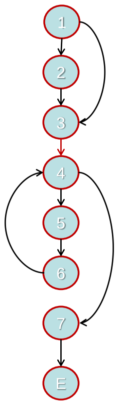
Examples:
- [1,3,4,7, E]
- [1,2,3,4,7, E]
- [1,2,3,4,5,6,4,7, E]
- [1,3,4,5,6,4,5,6,4,7, E]
- …

Coverage

- **Statement coverage**: Each **node** in the cfg is visited at least once.
- **Branch coverage**: Each **vertex** in the cfg is traversed at least once.
- **Decision coverage**: Like branch coverage, but specifies how often **conditions** (branching points) must be evaluated.
- **Path coverage**: Each **path** in the cfg is executed at least once.

Example: Statement Coverage

```
if (x < 0) /* 1 */ {
  x := – x /* 2 */
  };
z := 1; /* 3 */
while (x > 0) /*4*/ {
  z := z * y; /* 5 */
  x := x – 1 /* 6 */
};
return z /* 7 */
```



- Which (minimal) path covers all statements?

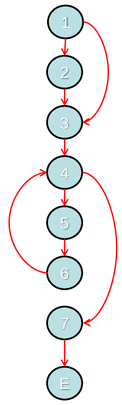
  p = [1,2,3,4,5,6,4,7,E]

- Which state generates p?

  x = -1
  y any
  z any

Example: Branch Coverage

```
if (x < 0) /* 1 */ {
  x := – x /* 2 */
  };
z := 1; /* 3 */
while (x > 0) /*4*/ {
  z := z * y; /* 5 */
  x := x – 1 /* 6 */
};
return z /* 7 */
```



- Which (minimal) path covers all vertices?

  $p_1 = [1,2,3,4,5,6,4,7,E]$
  $p_2 = [1,3,4,7,E]$

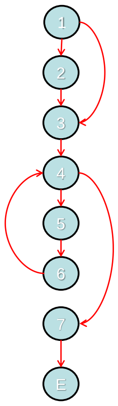
- Which states generate p_1, p_2?

	p_1	p_2
x	−1	0
y	any	any
z	any	any

- Note p_3 (x= 1) does not add coverage.

Example: Path Coverage

```
if (x < 0) /* 1 */ {
  x := – x /* 2 */
  };
z := 1; /* 3 */
while (x > 0) /*4*/ {
  z := z * y; /* 5 */
  x := x – 1 /* 6 */
};
return z /* 7 */
```



- How many paths are there?
- Let $q_1 = [1,2,3]$
  $q_2 = [1,3]$
  $p = [4,5,6]$
  $r = [4,7,E]$

  then all paths are
  $P = (q_1|q_2)\ p^*\ r$
- Number of possible paths:
  $|P| = 2 \cdot MaxInt - 1$

Statement, Branch and Path Coverage

- **Statement Coverage**:
  - Necessary but not sufficient, not suitable as only test approach.
  - Detects dead code (code which is never executed).
  - About 18% of all defects are identified.
- **Branch coverage**:
  - Least possible single approach.
  - Detects dead code, but also frequently executed program parts.
  - About 34% of all defects are identified.
- **Path Coverage:**
  - Most powerful structural approach;
  - Highest defect identification rate (100%);
  - But no **practical** relevance.

Decision Coverage

- Decision coverage is **more** then branch coverage, but less then full **path** coverage.
- Decision coverage requires that for all decisions in the program, each possible outcome is considered once.
- **Problem**: cannot sufficiently distinguish boolean expressions.
  - For A || B, the following are sufficient:

A	B	Result
false	false	false
true	false	true

  - But this does not distinguish A || B from A; B is effectively not tested.

Decomposing Boolean Expressions

- The binary boolean operators include conjunction $x \wedge y$, disjunction $x \vee y$, or anything expressible by these (e.g. exclusive disjunction, implication).

Elementary Boolean Terms
An elementary boolean term does not contain binary boolean operators, and cannot be further decomposed.

- An elementary term is a variable, a boolean-valued function, a relation (equality =, orders $<, \leq, >, \geq$, etc), or a negation of these.
- This is a fairly syntactic view, e.g. $x \leq y$ is elementary, but $x < y \vee x = y$ is not, even though they are equivalent.
- In formal logic, these are called **literals**.

Simple Condition Coverage

- **In simple condition coverage**, for each condition in the program, each elementary boolean term evaluates to *True* and *False* at least once.
- Note that this does not say much about the possible value of the condition.
- Examples and possible solutions:

if (temperature > 90 && pressure > 120) {...

C1	*C2*	*Result*
True	True	True
True	False	False
False	True	False
False	False	False

Modified Condition Coverage

- It is not always possible to generate all possible combinations of elementary terms, e.g. 3 <= x && x < 5.
- In **modified** (or minimal) **condition coverage**, all possible combinations of those elementary terms the value of which determines the value of the whole condition need to be considered.
- Example:

3 <= x && x < 5

False	False	False	← not needed
False	True	False	
True	False	False	
True	True	True	

- Another example: (x > 1 && ! p) || q

Modified Condition/Decision Coverage

- Modified Condition/Decision Coverage (MC/DC) is required by **DO-178B** for Level A software.
- It is a **combination** of the previous coverage criteria defined as follows:
  - Every point of entry and exit in the program has been invoked at least once;
  - Every decision in the program has taken all possible outcomes at least once;
  - Every condition in a decision in the program has taken all possible outcomes at least once;
  - Every condition in a decision has been shown to independently affect that decision's outcome.

How to achieve MC/DC

- **Not**: Here is the source code, what is the minimal set of test cases?
- **Rather**: From requirements we get test cases, do they achieve MC/DC?
- Example:
  - Test cases:

Test case	1	2	3	4	5
Input A	F	F	T	F	T
Input B	F	T	F	T	F
Input C	T	F	F	T	T
Input D	F	T	F	F	F
Result Z	F	T	F	T	T

Source Code:
Z := (A || B) && (C || D)

Question: do test cases achieve MC/DC?

Source: Hayhurst *et al*, A Practical Tutorial on MC/DC. NASA/TM2001-210876

Summary

- (Dynamic) Testing is the controlled execution of code, and comparing the result against an expected outcome.
- Testing is (traditionally) the main way for **verification**
- Depending on how the test cases are derived, we distinguish **white-box** and **black-box** tests.
- In black-box tests, we can consider **limits** and **equivalence classes** for input values to obtain test cases.
- In white-box tests, we have different notions of **coverage**: statement coverage, path coverage, condition coverage, etc.
- Next week: **Static testing** aka. static **program analysis**.