Systeme Hoher Qualität und Sicherheit
Vorlesung 7 vom 02.12.2013: Testing

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

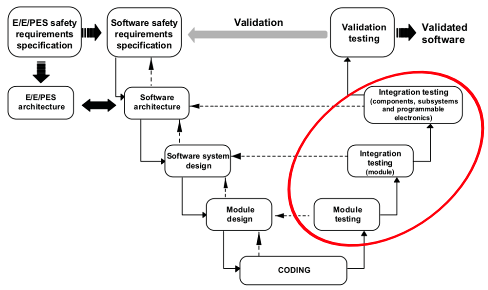
Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- **Lecture 7: Testing**
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Your Daily Menu

- What is testing?
- Different **kinds** of tests.
- Different test methods: **black-box** vs. **white-box**.
- Problem: cannot test **all** possible inputs.
- Hence, coverage criteria: how to test **enough**.

Testing in the Development Process



- **Tests** are one way of **verifying** that the system is built according to the specifications.
- Note we can test on **all** levels of the 'verification arm'.

What is testing?

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

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

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

Testing Process

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

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

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.

Basic Kinds of Test

- Functional test
- Non-functional test
- Structural test
- Regression test

Test Methods

- Static vs. dynamic:
  - With **static** tests, the code is **analyzed** without being run. We cover these methods separately 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 continously, 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 employess 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 $15m/s^2$. The vertical acceleration will never be less than zero, or more than $40m/s^2$.

- Equivalence classes or limits?

Black-Box Tests

- Quite typical for GUI tests.
- 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.
- Example: consider lists in C, Java, Haskell, and list reversal.
- 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.
- To abstract from the source code (which is a purely **syntactic** artefact), we consider the **control flow graph** of the program.

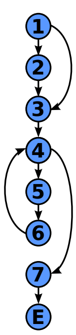
Control Flow Graph (cfg)

- 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.

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 */;
```



- A **path** through the program is a **path** through the cfg.
- Possible paths include:

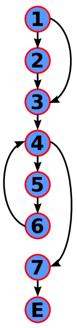
  $[1, 3, 4, 7, E]$
  $[1, 2, 3, 4, 7, E]$
  $[1, 2, 3, 4, 5, 6, 4, 7, E]$
  $[1, 3, 4, 5, 6, 4, 5, 6, 4, 7, E]$
  ...

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 p covers all statements?

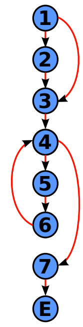
  $p = [1, 2, 3, 4, 5, 6, 4, 7, E]$

- 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) paths cover all vertices?

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

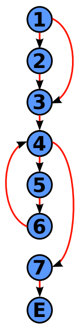
- Which states generate p_1, p_2?

p_1	p_2
$x = -1$	$x = 0$
y *any*	y *any*
y *any*	z *any*

- Note p_3 (corresponding to $x = 1$) does not add to 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 $q_1 \stackrel{def}{=} [1, 2, 3]$
  $q_2 \stackrel{def}{=} [1, 3]$
  $p \stackrel{def}{=} [4, 5, 6]$
  $r \stackrel{def}{=} [4, 7, E]$

  then all paths are given by

  $$P = (q_1 \mid q_2)\ p^*\ r$$

- Number of possible paths:

  $$|P| = 2n_{MaxInt} - 1$$

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 because of restricted practicability.

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 operational view, e.g. $x \leq y$ is elementary, but $x < y \vee x = y$ is not, even though they are equivalent.
- In 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) { ...
```

T1 T2

T_1	T_2	Result
true	**false**	**false**
false	**true**	**false**

T_1	T_2	Result
true	**true**	**true**
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	Result	
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.

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.