Softwaretechnik / Software-Engineering

Lecture 16: Testing

2017-07-09

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Code Quality Assurance: Content

- Introduction and Vocabulary
  - Test case, test suite, test execution.
  - Positive and negative outcomes.

- Limits of Software Testing

- Glass-Box Testing
  - Statement-, branch-, term-coverage.

- Other Approaches
  - Model-based testing,
  - Runtime verification.

- Program Verification
  - Partial and total correctness,
  - Proof System PD.

- Review
**Test Case**

**Definition.** A test case is a pair \((s, r)\) consisting of:
- a description \(s\) of a set of finite input sequences
- a description \(r\) of expected outcomes

\[ s = \{ s_0, \ldots, s_n \} \]

\[ r = \{ r_0, \ldots, r_n \} \]

**Example:**
- Test case for procedure `strlen`: `s0` is a sequence of characters, `r0` is expected length.
- `strlen`('abc') = 3

**Observation: Software Usually Has Many Inputs**

- Example: Simple Pocket Calculator.
- In other words, 9999999999 possible inputs remain uncovered.
- In other words, 1 \(\times\) 9999999999 = 9999999999000000 possible inputs remain uncovered.
- In other words, 10^16 possible inputs remain uncovered.

### Executing Test Cases

- A computation path \(p\) is a derived sequence of test cases \((s, r)\) such that:
  \[ (r_i) \Rightarrow (s_{i+1}, r_{i+1}) \]

- A path is a derived sequence of test cases \((s, r)\) such that:
  \[ (s_i) \Rightarrow (s_{i+1}, r_{i+1}) \]

- A computation path is a sequence of test cases \((s, r)\) such that:
  \[ (s_i) \Rightarrow (r_i) \Rightarrow (s_{i+1}, r_{i+1}) \]

### When To Stop Testing?

- **Statistical testing**
- **Habitat-based**
- **Glass-Box Testing**
- **Statement / Branch / term coverage**
- **Conclusions from coverage measures**

### Formal Program Verification

- **Deterministic Programs**
  - **Syntax, Semantics, Termination, Divergence**
  - **Correctness** of deterministic programs
    - partial correctness, total correctness.
Specific Testing Notions

- How are the test cases chosen?
  - Considering only the specification (black-box or function test).
  - Considering the structure of the test item (glass-box or structure test).

- How much effort is put into testing?
  - execution trial — does the program run at all?
  - throw-away-test — invent input and judge output on-the-fly (→ “rumprobieren”),
  - systematic test — somebody (not author!) derives test cases, defines input/soll, documents test execution.

Experience: In the long run, systematic tests are more economic.

- Complexity of the test item:
  - unit test — a single program unit is tested (function, sub-routine, method, class, etc.)
  - module test — a component is tested,
  - integration test — the interplay between components is tested.
  - system test — tests a whole system.
Specific Testing Notions Cont’d

• Which property is tested?
  
  - function test — functionality as specified by the requirements documents,
  
  - installation test — is it possible to install the software with the provided documentation and tools?
  
  - recommissioning test — is it possible to bring the system back to operation after operation was stopped?
  
  - availability test — does the system run for the required amount of time without issues,
  
  - load and stress test — does the system behave as required under high or highest load? … under overload?
    
    “Hey, let’s try how many game objects can be handled!” — that’s an experiment, not a test.
  
  - resource tests — response time, minimal hardware (software) requirements, etc.
  
  - regression test — does the new version of the software behave like the old one on inputs where no behaviour change is expected?

Specific Testing Notions Cont’d

• Which roles are involved in testing?
  
  - inhouse test — only developers (meaning: quality assurance roles),
  
  - alpha and beta test — selected (potential) customers,
  
  - acceptance test — the customer tests whether the system (or parts of it, at milestones) test whether the system is acceptable.
Some more vocabulary

Choosing Test Cases

- Generic requirements on good test cases

Approaches:

- Statistical testing
- Expected outcomes: Test Oracle : - /
- Habitat-based
- Glass-Box Testing
  - Statement / Branch / term coverage
  - Conclusions from coverage measures

When To Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification

- Deterministic Programs
  - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.

Choosing Test Cases
How to Choose Test Cases?

- A first rule-of-thumb:

   “Everything, which is required, must be examined/checked. Otherwise it is uncertain whether the requirements have been understood and realised.”

   (Ludewig and Lichter, 2013)

In other words:

- Not having
  - at least one (systematic) test case
  - for each (required) feature
  - is (grossly?) negligent. (Dt.: (grob?) fahrlässig).

- In even other words:
  Without at least one test case for each feature, we can hardly speak of software engineering.

- Good project management: document for each test case which feature(s) it tests.

What Else Makes a Test Case a Good Test Case?

A test case is a good test case if it discovers — with high probability — an unknown error.

An ideal test case (In, Soll) would be

- of low redundancy, i.e. it does not test what other test cases also test.

- error sensitive, i.e. has high probability to detect an error,
  (Probability should at least be greater than 0.)

- representative, i.e. represent a whole class of inputs,
  (i.e., software S passes (In, Soll) if and only S behaves well for all In’ from the class)

The idea of representative:

- If (12345678, 27; 12345705) was representative for (0, 27; 27), (1, 27; 28), etc.

- then from a negative execution of test case (12345678, 27; 12345705)

- we could conclude that (0, 27; 27), etc.
  will be negative as well.

- Is it / can we?
Thus: The wish for representative test cases is problematic:

- In general, we do not know which inputs lie in an equivalence class wrt. a certain error.
- Yet there is a large body of literature on how to construct representative test cases, assuming we know the equivalence classes.

Of course: *If* we *know* equivalence classes, we should exploit that knowledge to optimise the number of test cases.

But it is perfectly reasonable to test representatives of equivalence classes induced by the specification, e.g.

- valid and invalid inputs (to check whether input validation works at all),
- different classes of inputs considered in the requirements, like “C50”, “E1” coins in the vending machine → have at least one test case with each.

Recall: one should have at least one test case per feature.

---

**Content**

- Some more vocabulary
- Choosing Test Cases
  - Generic requirements on good test cases
  - Approaches:
    - Statistical testing
    - Expected outcomes: Test Oracle
    - Habitat-based
    - Glass-Box Testing
      - Statement / Branch / term coverage
      - Conclusions from coverage measures
- When To Stop Testing?
- Model-Based Testing
- Testing in the Development Process
- Formal Program Verification
  - Deterministic Programs
    - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
One Approach: Statistical Tests

Classical statistical testing is one approach to deal with
• in practice not exhaustively testable huge input space,
• tester bias.
(People tend to choose “good-will” inputs and disregard (tacit?) corner-cases; recall: the developer is not a good tester.)

Procedure:
• Randomly (!) choose test cases $T_1, \ldots, T_n$ for test suite $\mathcal{T}$.
• Execute test suite $\mathcal{T}$.
• If an error is found:
  • good, we certainly know there is an error,
• if no error is found:
  • refuse hypothesis “program is not correct” with a certain significance niveau.
  (Significance niveau may be unsatisfactory with small test suites.)

Note: Approach needs stochastical assumptions on error distribution and truly random test cases.
(Ludewig and Lichter, 2013) name the following objections against statistical testing:

- In particular for interactive software, the primary requirement is often no failures are experienced by the “typical user”.
  Statistical testing (in general) may also cover a lot of “untypical user behaviours” unless (sophisticated) user-models are used.

- Statistical testing needs a method to compute “soll”-values for the randomly chosen inputs.
  That is easy for requirement “does not crash”, but can be difficult in general.

- There is a high risk for not finding point or small-range errors.
  If they live in their “natural habitat”, carefully crafted test cases would probably uncover them.

Findings in the literature can at best be called inconclusive.

Getting Soll-Values
Where Do We Get The “Soll”-Values From?

Recall: A test case is a pair \((In, Soll)\) with proper expected (or “soll”) values.

- In an ideal world, all “soll”-values are defined by the (formal) requirements specification and effectively pre-computable.

- In this world,
  - the formal requirements specification may only reflectively describe acceptable results without giving a procedure to compute the results.
  - there may not be a formal requirements specification, e.g.
    - "the game objects should be rendered properly",
    - "the compiler must translate the program correctly",
    - "the notification message should appear on a proper screen position",
    - "the data must be available for at least 10 days",
    - etc.
  - Then: need another instance to decide whether the observation is acceptable.

- The testing community prefers to call any instance which decides whether results are acceptable a (test) oracle.

I’d prefer not to call automatic derivation of “soll”-values from a formal specification an “oracle”… :) (‘person or agency considered to provide wise and insightful […] prophetic predictions or precognition of the future, inspired by the gods.” says Wikipedia)

Content

- Some more vocabulary

Choosing Test Cases

- Generic requirements on good test cases
- Approaches:
  - Statistical testing
  - Expected outcomes: Test Oracle : - /
- Habitat-based
- Glass-Box Testing
  - Statement / Branch / term coverage
  - Conclusions from coverage measures

- When To Stop Testing?

Model-Based Testing

Testing in the Development Process

- Formal Program Verification
  - Deterministic Programs
    - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
Choosing Test Cases Habitat-based

Some traditional popular belief on software error habitat:

- Software errors (seem to) enjoy
  - range boundaries, e.g.
    - 0, 1, 27 if software works on inputs from [0, 27],
    - -1, 28 for error handling,
    - \(-2^{31} - 1, 2^{31}\) on 32-bit architectures,
  - boundaries of arrays (first, last element),
  - boundaries of loops (first, last iteration),
  - etc.

- special cases of the problem (empty list, use-case without actor, ...),
- special cases of the programming language semantics,
- complex implementations.

→ Good idea: for each test case, note down why it has been chosen.
For example, “demonstrate that corner-case handling is not completely broken”.
Some more vocabulary

Choosing Test Cases
• Generic requirements on good test cases
  • Approaches:
    • Statistical testing
    • Expected outcomes: Test Oracle
      • Habitat-based
  • Glass-Box Testing
    • Statement/Branch/term coverage
    • Conclusions from coverage measures

When To Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification
• Deterministic Programs
  • Syntax, Semantics, Termination, Divergence
  • Correctness of deterministic programs
    • partial correctness, total correctness.

Glass-Box Testing: Coverage
In the following, we assume that

- $S$ has a control flow graph $(V, E, S)$, and statements $Stm_S \subseteq V$ and branches $Cnd_S \subseteq E$.
- each computation path prefix $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \ldots \xrightarrow{\alpha_n} \sigma_n$ gives information on statements and control flow graph branch edges which were executed right before obtaining $\sigma_n$.

```
1: int f(int x, int y, int z)
2: {
3:   if (x > 100 && y > 10)
4:     z = z + 2;
5:     else
6:     z = z + 2;
7:   if (x > 500 || y > 50)
8:     z = z + 5;
9:   return z;
10: }
```

\(Stm_f = \{s_1, s_2, s_3, s_4\}\)

\(Cnd_f = \{e_1, e_2, e_3, e_4\}\)
Coverage is a property of test cases and test suites.

- Execution $\pi = \sigma_0 \stackrel{\alpha_1}{\Rightarrow} \cdots$ of test case $T$ achieves $p\%$ statement coverage if and only if
  
  $$p = \text{cov}_\text{stm}(\pi) := \frac{1}{|\text{Stm}_S|} \sum_{i \in \mathbb{N}_0} \text{stm}(\sigma_0 \cdots \sigma_i),$$

  Test case $T$ achieves $p\%$ statement coverage if and only if $p = \min_{\pi} \text{cov}_\text{stm}(\pi)$.

- Execution $\pi$ of $T$ achieves $p\%$ branch coverage if and only if
  
  $$p = \text{cov}_\text{cnd}(\pi) := \frac{1}{|\text{Cnd}_S|} \sum_{i \in \mathbb{N}_0} \text{cnd}(\sigma_0 \cdots \sigma_i),$$

  Test case $T$ achieves $p\%$ branch coverage if and only if $p = \min_{\pi} \text{cov}_\text{cnd}(\pi)$.

- Define: $p = 100$ for empty program. (More precisely: $\text{Stm}_S = \emptyset$ and $\text{Cnd}_S = \emptyset$, respectively.)

- Statement/branch coverage canonically extends to test suite $T = \{T_1, \ldots, T_n\}$.

  For example, given $\pi_1 = \sigma_0^{i_1} \cdots, \pi_n = \sigma_0^{i_n}$, then $T$ achieves
  
  $$p = \frac{1}{|\text{Stm}_S|} \sum_{1 \leq j \leq n} \sum_{i \in \mathbb{N}_0} \text{stm}(\sigma_0 \cdots \sigma_i),$$

  statement coverage.

---

**Coverage Example**

```c
int f(int x, int y, int z) {
  i1: if (x > 100 \&\& y > 10)    
  s1: z = z * 2;
  else
    s2: z = z / 2;
  i2: if (x > 500 \lor y > 50)   
  s3: z = z + 5;
  i3: return z;
}
```

- **Requirement**: $\{\text{true}\} f \{\text{true}\}$ (no abnormal termination), i.e. $\text{Soll} = \Sigma^* \cup \Sigma^\omega$.

<table>
<thead>
<tr>
<th>In</th>
<th>$x, y, z$</th>
<th>$i_1/f$</th>
<th>$i_1/f$</th>
<th>$i_1$</th>
<th>$i_2/f$</th>
<th>$i_2/f$</th>
<th>$i_3/f$</th>
<th>$i_3/f$</th>
<th>$i_3/f$</th>
<th>$% $</th>
<th>$% $</th>
<th>$i_2/%$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x, y, z$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
</tbody>
</table>
**Coverage Example**

```c
int f(int x, int y, int z)
{
    s1: if (x > 100 ∧ y > 10)
        s = z * 2;
    else
        s2: z = z / 2;
    s3: if (x > 500 ∨ y > 50)
        s = z * 5;
    s4: return z;
}
```

- **Requirement:** \{true\} f \{true\} (no abnormal termination), i.e. \(S_{oll} = \Sigma^* \cup \Sigma^ω\).

<table>
<thead>
<tr>
<th>In (x, y, z)</th>
<th>(s_1/t)</th>
<th>(s_1/f)</th>
<th>(s_2)</th>
<th>(s_3/f)</th>
<th>(s_3/t)</th>
<th>(s_4)</th>
<th>%stm</th>
<th>%cnd</th>
<th>%term</th>
</tr>
</thead>
<tbody>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>501, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>75</td>
<td>25</td>
</tr>
</tbody>
</table>

**Coverage Example**

```c
int f(int x, int y, int z)
{
    s1: if (x > 100 ∧ y > 10)
        s = z * 2;
    else
        s2: z = z / 2;
    s3: if (x > 500 ∨ y > 50)
        s = z * 5;
    s4: return z;
}
```

- **Requirement:** \{true\} f \{true\} (no abnormal termination), i.e. \(S_{oll} = \Sigma^* \cup \Sigma^ω\).

<table>
<thead>
<tr>
<th>In (x, y, z)</th>
<th>(s_1/t)</th>
<th>(s_1/f)</th>
<th>(s_2)</th>
<th>(s_3/f)</th>
<th>(s_3/t)</th>
<th>(s_4)</th>
<th>%stm</th>
<th>%cnd</th>
<th>%term</th>
</tr>
</thead>
<tbody>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>501, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>
Coverage Example

int f(int x, int y, int z)
{
    if (x > 100 ∧ y > 10)
    { s1: z = z * 2;
    
    } else
    { s2: z = z / 2;
    
    } s3: if (x > 500 ∨ y > 50)
    { return z;
    
    } s4: return z;
}

- Requirement: \{true\} f \{true\} (no abnormal termination), i.e. Soll = Σ* ∪ Σω.

<table>
<thead>
<tr>
<th>In</th>
<th>t1/t</th>
<th>t1/f</th>
<th>s1</th>
<th>s2</th>
<th>t2/t</th>
<th>t2/f</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>% stm</th>
<th>% cnd</th>
<th>% t2/term</th>
</tr>
</thead>
<tbody>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>501, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>0, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>100</td>
<td>75</td>
</tr>
<tr>
<td>0.51, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Term Coverage

- Consider the statement

  if \(A ∧ (B ∨ (C ∧ D)) ∨ E\) then ...;

  where \(A, . . . , E\) are minimal boolean terms, e.g. \(x > 0\), but not \(a ∨ b\).

Branch coverage is easy in this case:

Use \(I_{n1}\) such that \((A = 0, . . . , E = 0)\), and \(I_{n2}\) such that \((A = 0, . . . , E = 1)\).

- Additional goal: check whether there are useless terms, or terms causing abnormal program termination.

- Term Coverage (for an expression \(expr\)):

  - Let \(β : \{A_1, . . . , A_n\} → B\) be a valuation of the terms.

  - Term \(A_i\) is \(b\)-effective in \(β\) for \(expr\) if and only if

    \[β(A_i) = b \text{ and } [expr](β[A_i/true]) ≠ [expr](β[A_i/false]).\]

  - \(\Xi \subseteq \{A_1, . . . , A_n\} → B\) achieves \(p\%\) term coverage if and only if

    \[ p = \frac{|\{A_i \mid β ∈ \Xi \text{, } A_i \text{ is } b\text{-effective in } β\}|}{2^n}.\]
```c
int f(int x, int y, int z)
{
    i1: if (x \neq x)
    s1:  z = y/0;
    i2: if (x = x \lor z/0 = 27)
    s2:  z = z * 2;
    s3: return z;
}
```

- Statement \( s_1 \) is **never executed** (because \( x \neq x \iff \text{false} \)), thus 100\% statement-/branch-/term-coverage is **not achievable**.

- Assume, evaluating \( n/0 \) causes (undesired) **abnormal program termination**. Is statement \( s_1 \) an **error** in the program...?

- Term \( z/0 \) in \( i_2 \) also looks critical...
  (In programming languages with short-circuit evaluation, it is never evaluated.)

---

**Conclusions from Coverage Measures**

- **Assume**, test suite \( T \) tests software \( S \) for the following property \( \varphi \):
  - **pre-condition**: \( p \), **post-condition**: \( q \),
  and \( S \) passes (!) \( T \), and the execution achieves 100\% statement / branch / term coverage.

**What does this tell us about \( S \)?** Or: what can we conclude from coverage measures?

- **100\% statement** coverage:
  - “there is no statement, which **necessarily** violates \( \varphi \)”
    (Still, there may be many, many computation paths which violate \( \varphi \), and which just have not been touched by \( T \).)
  - “there is no unreachable statement”

- **100\% branch (term)** coverage:
  - “there is no single branch (term) which **necessarily causes** violations of \( \varphi \)”
    In other words: “for each condition (term), there is one computation path satisfying \( \varphi \) where the condition (term) evaluates to true, and one for false.”
  - “there is no unused condition (term)”

**Not more** (\( \rightarrow \) exercises)!
That’s definitely **something**, but not as much as “100\%” may sound like...
(Seems that) DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", (which deals with the safety of software used in certain airborne systems) requires that certain coverage measures are reached, in particular something similar to term coverage (MC/DC coverage). (Next to development process requirements, reviews, unit testing, etc.)

If not required, ask: what is the effort / gain ratio? (Average effort to detect an error; term coverage needs high effort.)

Currently, the standard moves towards accepting certain verification or static analysis tools to support (or even replace?) some testing obligations.

Content

• Some more vocabulary
• Choosing Test Cases
  • Generic requirements on good test cases
  • Approaches:
    • Statistical testing
    • Expected outcomes: Test Oracle : - /
    • Habitat-based
    • Glass-Box Testing
    • Statement / Branch / term coverage
    • Conclusions from coverage measures
• When To Stop Testing?
• Model-Based Testing
• Testing in the Development Process
  • Formal Program Verification
    • Deterministic Programs
      • Syntax, Semantics, Termination, Divergence
      • Correctness of deterministic programs
        • partial correctness, total correctness.
When To Stop Testing?

- There need to be defined criteria for when to stop testing; project planning should consider these criteria (and previous experience).

- Possible "testing completed" criteria:
  - all (previously) specified test cases have been executed with negative result,
    (Special case: All test cases resulting from a certain strategy, like maximal statement coverage have been executed.)
  - testing effort time sums up to $x$ (hours, days, weeks),
  - testing effort sums up to $y$ (any other useful unit),
  - $n$ errors have been discovered,
  - no error has been discovered during the last $z$ hours (days, weeks) of testing,

Values for $x$, $y$, $n$, $z$ are fixed based on experience, estimation, budget, etc.

- Of course: not all criteria are equally reasonable or compatible with each testing approach.
Another Criterion

- Another possible “testing completed” criterion:
  - The average cost per error discovery exceeds a defined threshold \( c \).

\[
\begin{align*}
\text{Number of Discovered Errors} & \quad \text{Cost Per Discovered Error} \\
\text{Errors} & \quad \text{Cost Threshold} \\
\text{End of Tests} & \quad t
\end{align*}
\]

Value for \( c \) is again fixed based on experience, estimation, budget, etc..

Content

- Some more vocabulary
- Choosing Test Cases
  - Generic requirements on good test cases
  - Approaches:
    - Statistical testing
    - Expected outcomes: Test Oracle : - /
    - Habitat-based
    - Glass-Box Testing
      - Statement / Branch / term coverage
      - Conclusions from coverage measures
- When To Stop Testing?
- Model-Based Testing
- Testing in the Development Process
- Formal Program Verification
  - Deterministic Programs
    - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
Does some software implement the given CFA model of the CoinValidator?

- **One approach: Location Coverage.**
  - Check whether for each location of the model there is a corresponding configuration reachable in the software (needs to be observable somehow).
  - Input sequences can automatically be generated from the model, e.g., using Uppaal’s “drive-to” feature.
  - Check ‘can we reach ‘idle’, ‘have_c50’, ‘have_c100’, ‘have_c150’?’ by
    \[ T_1 = (C50, C50, C50; \{ \pi | \exists i < j < k < \ell \cdot \pi_i \sim \text{idle}, \pi_j \sim h_{c50}, \pi_k \sim h_{c100}, \pi_\ell \sim h_{c150} \}) \]
  - Check for ‘have_e1’ by \( T_2 = (C50, C50, C50, \ldots) \).
  - To check for ‘drink_ready’, more interaction is necessary.

- **Analogously: Edge Coverage.**
  - Check whether each edge of the model has corresponding behaviour in the software.
Existential LSCs as Test Driver & Monitor (Lettrari and Klose, 2001)

- If the LSC has designated environment instance lines, we can distinguish:
  - messages expected to originate from the environment (driver role),
  - messages expected addressed to the environment (monitor role).

- Adjust the TBA-construction algorithm to construct a test driver & monitor and let it (possibly with some glue logic in the middle) interact with the software.

- Test passed (i.e., test unsuccessful) if and only if TBA state $q_6$ is reached.
  - Note: We may need to refine the LSC by adding an activation condition; or communication which drives the system under test into the desired start state.

- For example the Rhapsody tool directly supports this approach.

Vocabulary

- Software-in-the-loop:
The final implementation is examined using a separate computer to simulate other system components.

- Hardware-in-the-loop:
The final implementation is running on (prototype) hardware which is connected by its standard input/output interface (e.g., CAN-bus) to a separate computer which simulates other system components.
Some more vocabulary

Choosing Test Cases
- Generic requirements on good test cases
- Approaches:
  - Statistical testing
  - Expected outcomes: Test Oracle : - /
  - Habitat-based
- Glass-Box Testing
  - Statement / Branch / term coverage
  - Conclusions from coverage measures

When To Stop Testing?

Model-Based Testing

Testing in the Development Process

Formal Program Verification
- Deterministic Programs
  - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
• **Test Gear**: (may need to be developed in the project!)

  **test driver**—A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results.

  **Synonym**: test harness.

  IEEE 610.12 (1990)

  **stub**—

  1. A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent on it.

  2. A computer program statement substituting for the body of a software module that is or will be defined elsewhere.

  IEEE 610.12 (1990)

• **Roles**: tester and developer should be different persons!

---

**Content**

- Some more vocabulary
- Choosing Test Cases
  - Generic requirements on good test cases
  - Approaches:
    - Statistical testing
    - Expected outcomes: Test Oracle : - /
    - Habitat-based
    - Glass-Box Testing
      - Statement / Branch / term coverage
      - Conclusions from coverage measures
- When To Stop Testing?
- Model-Based Testing
- Testing in the Development Process
- Formal Program Verification
  - Deterministic Programs
    - Syntax, Semantics, Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness, total correctness.
Formal Methods in the Software Development Process

---

**validation**

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: **verification**.

IEEE 610.12 (1990)

---

**verification**

(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: **validation**.

(2) Formal proof of program correctness.

IEEE 610.12 (1990)
Concepts of Software Quality Assurance

software quality assurance

organisational
analytic
constructive

project management
software examination

examination by humans

comp. aided
human exam.

inspection

review

manual proof

comp. aided human exam.

e.g. interactive prover

check against rules

consistent checks

quantitative examination

dynamic checking (test)

examine

execute

prove

formal verification

(Ludewig and Lichter, 2013)

Testing, Review, Verification Illustrated

all computation paths satisfying the specification

expected outcomes $S_{all}$

defines

$\in$?

$\subseteq$?

$\subseteq$?

proof $S \models \forall$, conclude $[S] \in [\forall]$

execution of $(In, S_{all})$

input $\rightarrow$ output

Testing

Reviewer

review

Formal Verification
• Some more vocabulary

• Choosing Test Cases
  • Generic requirements on good test cases
  • Approaches:
    • Statistical testing
    • Expected outcomes: Test Oracle : -/
    • Habitat-based
    • Glass-Box Testing
      • Statement / Branch / term coverage
      • Conclusions from coverage measures

• When To Stop Testing?

• Model-Based Testing

• Testing in the Development Process

• Formal Program Verification
  • Deterministic Programs
    • Syntax, Semantics, Termination, Divergence
  • Correctness of deterministic programs
    • partial correctness, total correctness.
Deterministic Programs

**Syntax:**

\[ S ::= \text{skip} \mid u := t \mid S_1 ; S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } B \text{ do } S_1 \text{ do} \]

where \( u \in V \) is a variable, \( t \) is a type-compatible expression, \( B \) is a Boolean expression.
• There is a vast amount of literature on how to choose test cases.  
  A good starting point:
  • at least one test case per feature,
  • corner-cases, extremal values,
  • error handling, etc.

• Glass-box testing
  • considers the control flow graph,
  • defines coverage measures.

• Other approaches:
  • statistical testing, model-based testing,
  • Define criteria for “testing done” (like coverage, or cost per error).

  • Process: tester and developer should be different persons.

Formal Verification:
• There are more approaches to software quality assurance
  than (just) testing.
• For example, program verification.

References
