Softwaretechnik / Software-Engineering

Lecture 15: Testing

2017-07-17

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.

- Software quality assurance in a larger scope.

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

- Review
Content

• **Software Testing Introduction**
  - Test suite; Tests vs. *systematic* tests.
  - More vocabulary

• **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?

• **Choosing Test Cases**
  - Generic *requirements* on good test cases
  - Point vs. range errors
  - 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**
### Test Case

**Definition.** A test case \( T \) over \( \Sigma \) and \( A \) is a pair \( (\text{In}, \text{Soll}) \) consisting of
- a description \( \text{In} \) of sets of finite input sequences,
- a description \( \text{Soll} \) of expected outcomes,
and an interpretation \( \cdot \) of these descriptions:

\[
\text{In} = (\Sigma^* \times A)^*, \quad \text{Soll} = (\Sigma \times A)^* \cup (\Sigma \times A)^*
\]

**Examples:**
- Test case for procedure `strlen`:
  
  \[
  T = (s = "abc", r = 3)
  \]

  Shorthand notation: \( T = (\text{"abc"}, 3) \).

  - "Call `strlen()` with string \("abc"\), expect return value \(3\)."

### Executing Test Cases

- A computation path
  \[
  \pi = (\sigma_0 \xrightarrow{\sigma_1} \sigma_2 \xrightarrow{\sigma_3} \ldots)
  \]
  from \( S \) is called execution of test case \((\text{In}, \text{Soll})\) if and only if
- there is \( n \in \mathbb{N} \) such that \( \sigma_0 \xrightarrow{\sigma_{n-1}} \sigma_n \downarrow \Sigma^* \in \text{In} \).

  (A prefix of \( \pi \) corresponds to an input sequence).

Execution \( \pi \) of test case \( T \) is called
- **successful (or positive)** if and only if \( \pi \not\in \text{Soll} \).
  - Intuition: an error has been discovered.
  - Alternative: test item \( S \) failed to pass the test.
  - Confusing: "test failed".

- **unsuccessful (or negative)** if and only if \( \pi \in \text{Soll} \).
  - Intuition: no error has been discovered.
  - Alternative: test item \( S \) passed the test.
  - Okay: "test passed".

### Tell Them What You’ve Told Them...

- **Testing** is about
  - finding errors, or
  - demonstrating scenarios.

- A test case consists of
  - input sequences and
  - expected outcomes.

- A test case execution is
  - positive if an error is found,
  - negative if no error is found.

- A test suite is a set of test cases.
  - Distinguish (among others),
    - glass-box test: structure (or source code) of test item available,
    - black-box test: structure not available.
• Consider the test case

\[ T = ("", 0) \]

for procedure `strlen`.

("Empty string has length 0.")

• A tester observes the following software behaviour:

\[ \pi = \{ s \mapsto \text{NULL}, r \mapsto 0 \} \]

\[ \rightarrow \text{program-abortion} \]

\[ = \sigma_0 \rightarrow \sigma_1 \]

• Test execution **positive** or **negative**?

Note:

• If a tester does not adhere to an allowed input sequence of \( T \), \( \pi \) is **not** a test execution.

Thus \( \pi \) is neither positive nor negative (only defined for test executions).

• Same case: power outage (if continuous power supply is considered in input sequence).
High quality software should be aware of its specification and “complain” if operated outside of specification, e.g.

- throw an exception,
- abort program execution,
- (at least) print an error message,
- etc.

Not: “garbage in, garbage out”

Example: strlen(3) (C standard)

- Allowed inputs are C-strings, return value is an integer,
- NULL is not a C-string!
- Thus, on input NULL, “complain” instead of just return an arbitrary number or “crash”.
**Test Suite**

- A **test suite** is a finite set of test cases \( \{T_1, \ldots, T_n\} \).

- An **execution** of a **test suite** is a set of computation paths, such that there is at least one execution for each test case.

- An **execution** of a **test suite** is called **positive** if and only if at least one test case execution is **positive**. Otherwise, it is called **negative**.
Tests vs. Systematic Tests

Systematic Test – a test such that
- (environment) conditions are defined or precisely documented,
- inputs have been chosen systematically,
- results are documented and assessed according to criteria that have been fixed before.  

(Test) – (one or multiple) execution(s) of a program on a computer with the goal to find errors.

Not (even) a test (in the sense of this weak definition):
- any inspection of the program (no execution),
- demo of the program (other goal),
- analysis by software-tools for, e.g., values of metrics (other goal),
- investigation of the program with a debugger (other goal).

(Our) Synonyms for non-systematic tests: Experiment, ‘Rumprobieren’.

In the following: test means systematic test; if not systematic, call it experiment.
Content

- Software Testing Introduction
  - Test suite; Tests vs. systematic tests.
  - More vocabulary

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?

- Choosing Test Cases
  - Generic requirements on good test cases
  - Point vs. range errors
  - 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
Testing Vocabulary
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.
• 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?
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.
Content

- **Software Testing Introduction**
  - Test suite; Tests vs. *systematic* tests.
  - More vocabulary

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?

- **Choosing Test Cases**
  - Generic *requirements* on good test cases
  - Point vs. range errors
  - 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**
The Limits of Software Testing
In each examination, there are two paths from the specification to results:

- the production path (using model, source code, executable, etc.), and
- the examination path (using requirements specifications).

A check can only discover errors on exactly one of the paths.

If a difference is detected: examination result is positive.

What is not on the paths, is not checked; crucial: specification and comparison.

Recall:

<table>
<thead>
<tr>
<th>Artfact has error</th>
<th>Checking Procedure</th>
<th>Artfact is OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Shows no error</td>
<td>OK</td>
</tr>
<tr>
<td>Yes</td>
<td>Reports error</td>
<td>OK</td>
</tr>
</tbody>
</table>

- SW broken: test ex. neg.
- SW broken: test ex. pos.
- SW broken: test ex. neg.
- SW broken: test ex. pos.

(Ludewig and Lichter, 2013)
“Software testing can be used to show the presence of bugs, but never to show their absence!”

(E. W. Dijkstra, 1970)
Consider a simple pocket calculator for adding 8-digit decimals:

- **Requirement**: If the display shows \( x, +, \) and \( y \), then after pressing \( = \),
  - the sum of \( x \) and \( y \) is displayed if \( x + y \) has at most 8 digits,
  - otherwise “-E-” is displayed.

- With 8 digits, both \( x \) and \( y \) range over \([0, 10^8 - 1]\).

- Thus there are \( 10^{16} = 10,000,000,000,000,000 \) possible input pairs \((x, y)\) to be considered for **exhaustive testing**, i.e. testing every possible case!

- And if we restart the pocket calculator for each test, we do not know anything about problems with sequences of inputs...

  (Local variables may not be re-initialised properly, for example.)
**Observation: Software Usually Has Many Inputs**

- **Example:** Simple Pocket Calculator.

  With ten thousand (10,000) different test cases (that’s a lot!), 9,999,999,999,990,000 of the $10^{16}$ possible inputs remain **uncovered**.

  **In other words:**
  Only 0.00000000001% of the possible inputs are covered, 99.9999999999% not touched.

- **In diagrams:** (red: uncovered, blue: covered)
**Observation: Software Usually Has Many Inputs**

- **Example**: Simple Pocket Calculator.
  
  With **ten thousand** (10,000) **different** test cases (that’s a lot!),
  
  9,999,999,999,990,000 of the $10^{16}$ possible inputs remain **uncovered**.
  
  **In other words**:
  
  Only 0.0000000001% of the possible inputs are covered, 99.9999999999% not touched.

- **In diagrams**: (red: uncovered, blue: covered)
Content

- Software Testing Introduction
  - Test suite; Tests vs. systematic tests.
  - More vocabulary

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?

- Choosing Test Cases
  - Generic requirements on good test cases
  - Point vs. range errors

  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
Choosing Test Cases
Test executions should be (as) **reproducible** and **objective** (as possible).

So, **strictly speaking**, a test case is a triple \((In, Soll, Env)\)
comprising a description \(Env\) of (environmental) **conditions**.

\(Env\) describes any aspects which **could have an effect**
on the outcome of a test execution and cannot be specified as part of \(In\), such as:

- Which **program** (version) is tested?
- **Built** with which compiler, linker, etc.?
- **Test host** (OS, architecture, memory size, connected devices (configuration?), etc.)?
- Which **other software** (in which version, configuration) is involved?
- **Who** is supposed to test **when**?
- etc. etc.

**Full reproducibility** is hardly possible **in practice** – obviously (err, why…?).

**Steps** towards **reproducibility** and **objectivity**:

- have a fixed build environment,
- use a fixed test host which does not do any other jobs,
- execute test cases **automatically** (test scripts).
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 \((0, 27; 27), (1, 27; 28), \text{ etc.}\)
- then from a **negative** execution of test case \((12345678, 27; 12345705)\)
- we **could** conclude that \((0, 27; 27), \text{ etc.}\) will be negative as well.
Software is (in general) **not continuous**.

Consider a continuous function, e.g. the one to the right:

For sufficiently small $\varepsilon$-environments of an input, the outputs **differ only by a small amount** $\delta$.

Physical systems are (to a certain extent) continuous:

- For example, if a bridge endures a single car of 1000 kg, we strongly expect the bridge to endure cars of 990 kg or 1010 kg.
- And anything of weight smaller than 1000 kg can be expected to be endured.

For software, adjacent inputs **may yield arbitrarily distant** output values.

**Vocabulary**:

- **Point error**: an isolated input value triggers the error.
- **Range error**: multiple “neighbouring” inputs trigger the error.

For software, we can (in general, **without extra information**) not **conclude from some values to others**.
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 on 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

- Software Testing Introduction
  - Test suite; Tests vs. **systematic** tests.
  - More vocabulary

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?

- Choosing Test Cases
  - Generic **requirements** on good test cases
  - Point vs. range errors
  - 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
Statistical Testing
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.)
  (And: 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)
Software Testing Introduction
- Test suite; Tests vs. systematic tests.
- More vocabulary

Limits of Software Testing
- Software examination paths
- Is exhaustive testing feasible?

Choosing Test Cases
- Generic requirements on good test cases
  - Point vs. range errors
- 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
Habitat-based Testing
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”.
Content

- Software Testing Introduction
  - Test suite; Tests vs. systematic tests.
  - More vocabulary

- Limits of Software Testing
  - Software examination paths
  - Is exhaustive testing feasible?

- Choosing Test Cases
  - Generic requirements on good test cases
  - Point vs. range errors
  - 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
Glass-Box Testing: Coverage
Definition. **Software** is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ of (finite or infinite) computation paths of the form $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$ where
- $\sigma_i \in \Sigma, i \in \mathbb{N}_0$, is called **state** (or **configuration**), and
- $\alpha_i \in A, i \in \mathbb{N}_0$, is called **action** (or **event**).

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 $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_n} \sigma_n$ gives information on statements and control flow graph branch edges which were executed right before obtaining $\sigma_n$:

$$stm : (\Sigma \times A)^* \rightarrow 2^{Stm_S}, \quad cnd : (\Sigma \times A)^* \rightarrow 2^{Cnd_S},$$

```c
1: int f(int x, int y, int z)
2: {
3:     i1: if (x > 100 && y > 10)
4:         s1: z = z * 2;
5:     else
6:         s2: z = z / 2;
7:     i2: if (x > 500 || y > 50)
8:         s3: z = z * 5;
9:     s4: return z;
10: }
```

$Stm_f = \{s_1, s_2, s_3, s_4\}$
$Cnd_f = \{e_1, e_2, e_3, e_4\}$
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 $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_n} \sigma_n$ gives information on statements and control flow graph branch edges which were executed right before obtaining $\sigma_n$:

$$stm : (\Sigma \times A)^* \rightarrow 2^{Stm_S}, \quad cnd : (\Sigma \times A)^* \rightarrow 2^{Cnd_S},$$

```plaintext
1: int f(int x, int y, int z)
2: {
3:   i1: if (x > 100 \land y > 10)  
4:     s1: z = z * 2;
5:   else
6:     s2: z = z / 2;
7:   i2: if (x > 500 \lor y > 50)  
8:     s3: z = z * 5;
9:   s4: return z;
10: }
```

$Stm_f = \{ s_1, s_2, s_3, s_4 \}$

$Cnd_f = \{ e_1, e_2, e_3, e_4 \}$

```
\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \xrightarrow{\alpha_3} \sigma_3 \xrightarrow{\alpha_4} \sigma_4 \xrightarrow{\alpha_5} \sigma_5 \xrightarrow{\alpha_6} \sigma_6
```

```
stm: {} {} {} {} {s_1} {} {} {s_3} {s_4}
```

```
cnd: {} {} {e_1} {} {} {e_3} {} {}
```
Glass-Box Testing: Coverage

- **Coverage** is a property of test cases and test suites.

- Execution $\pi = \sigma_0^{\alpha_1} \cdot \cdot \cdot$ of test case $T$ achieves $p\%$ **statement coverage** if and only if
  $$p = cov_{stm}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} stm(\sigma_0 \cdot \cdot \cdot \sigma_i)|}{|Stm_S|}, |Stm_S| \neq 0.$$  
  Test case $T$ achieves $p\%$ **statement coverage** if and only if $p = \min_{\pi \text{ execution of } T} cov_{stm}(\pi)$.

- Execution $\pi$ of $T$ achieves $p\%$ **branch coverage** if and only if
  $$p = cov_{cnd}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} cnd(\sigma_0 \cdot \cdot \cdot \sigma_i)|}{|Cnd_S|}, |Cnd_S| \neq 0.$$  
  Test case $T$ achieves $p\%$ **branch coverage** if and only if $p = \min_{\pi \text{ execution of } T} cov_{cnd}(\pi)$.

- **Define:** $p = 100$ for empty program. (More precisely: $Stm_S = \emptyset$ and $Cnd_S = \emptyset$, respectively.)

- Statement/branch coverage canonically extends to test suite $T = \{T_1, \ldots, T_n\}$. For example, given $\pi_1 = \sigma_0^{\alpha_1} \cdot \cdot \cdot$, $\pi_n = \sigma_0^{\alpha_n} \cdot \cdot \cdot$, then $T$ achieves
  $$p = \frac{|\bigcup_{1 \leq j \leq n} \bigcup_{i \in \mathbb{N}_0} stm(\sigma_0^{\alpha_j} \cdot \cdot \cdot \sigma_i)|}{|Stm_S|}, |Stm_S| \neq 0,$$  
  **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 ∨ y > 50)
    s3:    z = z * 5;
    s4: return z;
}
```

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

<table>
<thead>
<tr>
<th>In</th>
<th>(x, y, z)</th>
<th>(i_1/t)</th>
<th>(i_1/f)</th>
<th>(s_1)</th>
<th>(s_2)</th>
<th>(i_2/t)</th>
<th>(i_2/f)</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(s_3)</th>
<th>(s_4)</th>
<th>%</th>
<th>%</th>
<th>(i_2/%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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></td>
<td>12, 11, 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></td>
</tr>
</tbody>
</table>
```

---

- **Test suite 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 ∨ y > 50)  
        s3: z = z * 5;
    s4: ;
}
```

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

<table>
<thead>
<tr>
<th>In ((x, y, z))</th>
<th>(i_1/t)</th>
<th>(i_1/f)</th>
<th>(s_1)</th>
<th>(s_2)</th>
<th>(i_2/t)</th>
<th>(i_2/f)</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(s_3)</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>✔</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>

*test suite coverage*
Term Coverage

• Consider the statement
  \[
  \text{if } (A \land (B \lor (C \land D)) \lor E) \text{ then } \ldots;
  \]
  where \( A, \ldots, E \) are \textbf{minimal} boolean terms, e.g. \( x > 0 \), but not \( a \lor b \).

\textbf{Branch coverage} is easy in this case:

\textit{Use} \( In_1 \) such that \( (A = 0, \ldots, E = 0) \), and \( In_2 \) such that \( (A = 0, \ldots, E = 1) \).

• \textbf{Additional goal:}
  check whether there are useless terms, or terms causing abnormal program termination.

• \textbf{Term Coverage} (for an expression \( expr \)):
  
  \begin{itemize}
  \item Let \( \beta : \{A_1, \ldots, A_n\} \rightarrow \mathbb{B} \) be a valuation of the terms.
  \item Term \( A_i \) is \textbf{\textit{b-effective}} in \( \beta \) for \( expr \) if and only if
    \[
    \beta(A_i) = b \text{ and } \llbracket expr \rrbracket(\beta[A_i/true]) \neq \llbracket expr \rrbracket(\beta[A_i/false]).
    \]
  \end{itemize}

  \begin{itemize}
  \item \( \Xi \subseteq (\{A_1, \ldots, A_n\} \rightarrow \mathbb{B}) \) achieves \( p \% \textbf{term coverage} \) if and only if
    \[
    p = \frac{|\{A_i^b | \exists \beta \in \Xi \bullet A_i \text{ is \textit{b-effective} in } \beta\}|}{2n}.
    \]
  \end{itemize}


Unreachable Code

```c
int f(int x, int y, int z) {
    i1: if (x ≠ x)
    s1:    z = y/0;
    i2: if (x = x ∨ z/0 = 27)
    s2:    z = z * 2;
    s3: return z;
}
```

- Statement $s_1$ is never executed (because $x ≠ x ⇔ false$), 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.
Tell Them What You’ve Told Them...

- A check can **only discover** errors on **exactly one** path.

- Software testing is **challenging** because
  - typically **huge input space**,
  - software is **non-continous**,

- 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.
References
References

