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 \( T \) over \( \Sigma \) and \( A \) is a pair \( (In, Soll) \) consisting of

- a description \( In \) of sets of finite input sequences,
- a description \( Soll \) of expected outcomes,

and an interpretation \( I \) of these descriptions:

- \( [In] \subseteq (\Sigma^* \times A)^* \),
- \( [Soll] \subseteq (\Sigma \times A)^* \),
- \( [In] \cup [Soll] \subseteq (\Sigma \times A)^* \),
- \( I(\sigma_1, \sigma_2) = I(\sigma_2, \sigma_2) \).

**Executing Test Cases**

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

Execution \( \pi \) of test case \( T \) is called
- successful (or positive) if and only if \( \pi \notin [Soll] \).
  - 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 \( \pi \in [Soll] \).
  - Intuition: no error has been discovered.
  - Alternative: test item \( S \) passed the test.
  - Okay: “test passed.”

**Software Examination (in Particular 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.

**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)
Point vs. Range Errors

- For software, (in general, without extra information) we **cannot conclude** from some values to others.

- Software behaviour is (in general) **not continuous**.
  - 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.

- 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.
• 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
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.
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?
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
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 \((0, 27; 27), (1, 27; 28), \text{ etc.}\)
- 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.
- 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 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.
• 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
Statistical Testing
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 \( T \).
- Execute test suite \( T \).
- **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)
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
Habitat-based Testing
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
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 $\sigma_0 \xrightleftharpoons{\alpha_1} \sigma_1 \xrightleftharpoons{\alpha_2} \sigma_2 \cdots$ 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 $\sigma_0 \xrightleftharpoons{\alpha_1} \sigma_1 \xrightleftharpoons{\alpha_2} \sigma_2 \cdots \xrightleftharpoons{\alpha_n} \sigma_n$ gives information on statements and control flow graph branch edges **which were executed** right before obtaining $\sigma_n$:

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

```
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 $(V, E)\subseteq S$ 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 $\sigma_n$:

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

1: int $f$(int $x$, int $y$, int $z$)
2: {
3:   $i_1$: if ($x > 100 \land y > 10$)
4:     $s_1$: $z = z \times 2$;
5:   else
6:     $s_2$: $z = z / 2$;
7:   $i_2$: if ($x > 500 \lor y > 50$)
8:     $s_3$: $z = z \times 5$;
9:     $s_4$: 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 \textbf{statements} \( Stm_S \subseteq V \) and \textbf{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 \textbf{which were executed} right before obtaining \( \sigma_n \):

\[
\begin{align*}
stm : (\Sigma \times A)^* &\rightarrow 2^{Stm_S}, \\
cnd : (\Sigma \times A)^* &\rightarrow 2^{Cnd_S},
\end{align*}
\]

```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: }
```

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

\( Cnd_f = \{e_1, e_2, e_3, e_4\} \)

<table>
<thead>
<tr>
<th>( \sigma_0 )</th>
<th>( \alpha_1 )</th>
<th>( \sigma_1 )</th>
<th>( \alpha_2 )</th>
<th>( \sigma_2 )</th>
<th>( \alpha_3 )</th>
<th>( \sigma_3 )</th>
<th>( \alpha_4 )</th>
<th>( \sigma_4 )</th>
<th>( \alpha_5 )</th>
<th>( \sigma_5 )</th>
<th>( \alpha_6 )</th>
<th>( \sigma_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc: 1</td>
<td>pc: 3</td>
<td>pc: 4</td>
<td>pc: 7</td>
<td>pc: 8</td>
<td>pc: 9</td>
<td>pc: 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y: 11</td>
<td>y: 11</td>
<td>y: 11</td>
<td>y: 11</td>
<td>y: 11</td>
<td>y: 11</td>
<td>y: 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z: 0</td>
<td>z: 0</td>
<td>z: 0</td>
<td>z: 0</td>
<td>z: 0</td>
<td>z: 0</td>
<td>z: 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( stm: \) \{\} \{\} \{\} \{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 $\pi = \sigma_0^{\alpha_1} \cdots$ of test case $T$ achieves $p\%$ statement coverage if and only if
  $$p = \text{cov}_{\text{stm}}(\pi) := \frac{\left| \bigcup_{i \in \mathbb{N}_0} \text{stm}(\sigma_0 \cdots \sigma_i) \right|}{|\text{Stm}_S|}, |\text{Stm}_S| \neq 0.$$

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

- Execution $\pi$ of $T$ achieves $p\%$ branch coverage if and only if
  $$p = \text{cov}_{\text{cnd}}(\pi) := \frac{\left| \bigcup_{i \in \mathbb{N}_0} \text{cnd}(\sigma_0 \cdots \sigma_i) \right|}{|\text{Cnd}_S|}, |\text{Cnd}_S| \neq 0.$$

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

- 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 $\pi_1 = \sigma_0^1 \cdots$ and $\pi_n = \sigma_0^n \cdots$, then $T$ achieves
  $$p = \frac{\left| \bigcup_{1 \leq j \leq n} \bigcup_{i \in \mathbb{N}_0} \text{stm}(\sigma_0^j \cdots \sigma_i^j) \right|}{|\text{Stm}_S|}, |\text{Stm}_S| \neq 0,$$ statement coverage.
Coverage Example

int \( f(\text{int } x, \text{int } y, \text{int } z) \)
{
    i_1: \text{if } (x > 100 \land y > 10)
    s_1: \quad z = z \times 2;
    \text{else}
    s_2: \quad z = z / 2;
    i_2: \text{if } (x > 500) \lor (y > 50)
    s_3: \quad z = z \times 5;
    s_4: \text{return } z;
}

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

<table>
<thead>
<tr>
<th>In</th>
<th>i1/t</th>
<th>i1/f</th>
<th>s1</th>
<th>s2</th>
<th>i2/t</th>
<th>i2/f</th>
<th>c1</th>
<th>c2</th>
<th>s3</th>
<th>s4</th>
<th>%</th>
<th>%</th>
<th>i2/%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></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>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>
<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>✔</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 minimal boolean terms, e.g. \( x > 0 \), but not \( a \lor b \).

Branch coverage is easy in this case:

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

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

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

  • Let \( \beta : \{A_1, \ldots, A_n\} \to B \) be a valuation of the terms.

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

\[
\beta(A_i) = b \text{ and } \llbracket expr \rrbracket(\beta[A_i/true]) \neq \llbracket expr \rrbracket(\beta[A_i/false]).
\]

  • \( \Xi \subseteq (\{A_1, \ldots, A_n\} \to B) \) achieves \( p \% \) term coverage if and only if

\[
p = \frac{\left| \{A_i^b \mid \exists \beta \in \Xi \bullet A_i \text{ is } b \text{-effective in } \beta \} \right|}{2n}.
\]
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** (→ 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.
• 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
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 $c$.

Value for $c$ is again fixed based on experience, estimation, budget, etc.
• 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
Model-Based Testing
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

$$T_1 = (C50, C50, C50; \{ \pi \mid \exists i < j < k < \ell \cdot \pi^i \sim \text{idle}, \pi^j \sim \text{h_c50}, \pi^k \sim \text{h_c100}, \pi^\ell \sim \text{h_c150} \})$$

- 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.
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
Testing in The Software Development Process
Test Conduction: Activities & Artefacts

- Planning
- Preparation
- Execution
- Evaluation
- Analysis

- Test Plan
- Test Cases
- Test Directions
- Test Gear
- Test Protocol
- Test Report

(Ludewig and Lichter, 2013)

- **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
Tell Them What You’ve Told Them…

- 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.

- There are more approaches to code quality assurance than (just) testing.

- For example, program verification.
References
