Softwaretechnik / Software-Engineering

Lecture 16: Testing

2016-07-11

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Code Quality Assurance: Content

VL 15	• **Introduction and Vocabulary**
VL 16	• **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**.
VL 17	• **Program Verification**
⋮	– partial and total **correctness**,
	– **Proof System PD**.
VL 18	• **Review**

Content

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range- vs. point errors
- **When To Stop Testing?**
- **Choosing Test Cases**
  - **Requirements** on test cases
  - The **natural habitat** of many errors
  - Test **Oracle**
- **Glass-Box Testing**
  - **Statement coverage**
  - **Branch** and **term coverage**
  - **Conclusions** from coverage measures
- **Model-Based Testing**
- **Testing in the Development Process**

Testing

Recall: Test Case

Definition. A **test case** T 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 $[\![\cdot]\!]$ of these descriptions.

A **test execution** π, i.e. $((\pi^0, \dots, \pi^n) \downarrow \Sigma_{in}) \in In$ for some $n \in \mathbb{N}_0$, is called

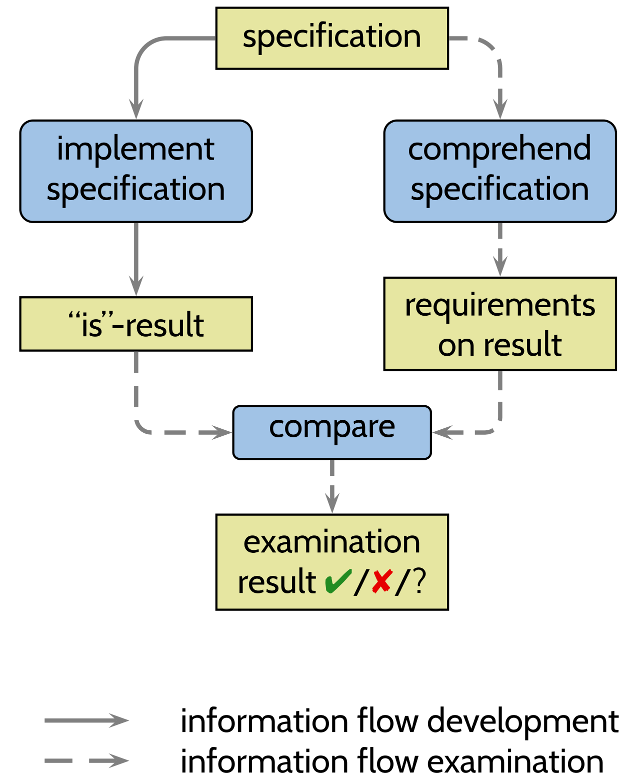
- **successful** (or **positive**)
  if it discovered an error,
  i.e., if $\pi \notin [\![Soll]\!]$.

  (Alternative: test item S **failed to pass test**; confusing: "test failed".)
- **unsuccessful** (or **negative**)
  if it did not discover an error,
  i.e., if $\pi \in [\![Soll]\!]$.

  (Alternative: test item S **passed 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 **difference detected**: examination result is **positive**.
- What is not on the paths, is not checked; crucial: **specification** and **comparison**.



(Ludewig and Lichter, 2013)

Recall:

artefact has error	checking procedure: shows no error	checking procedure: reports error
yes	false negative	true positive
no	true negative	false positive

Content

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range- vs. point errors
- **When To Stop Testing?**
- **Choosing Test Cases**
  - **Requirements** on test cases
  - The **natural habitat** of many errors
  - Test **Oracle**
- **Glass-Box Testing**
  - **Statement coverage**
  - **Branch** and **term coverage**
  - **Conclusions** from coverage measures
- **Model-Based Testing**
- **Testing in the Development Process**

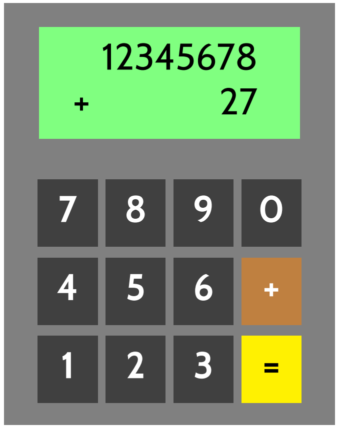
The Crux of Software Testing

Why Can't We Show The Absence of Errors (in General)?

Recall:

> "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} possible input pairs (x, y) to be considered for **exhaustive testing**!
- 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.)

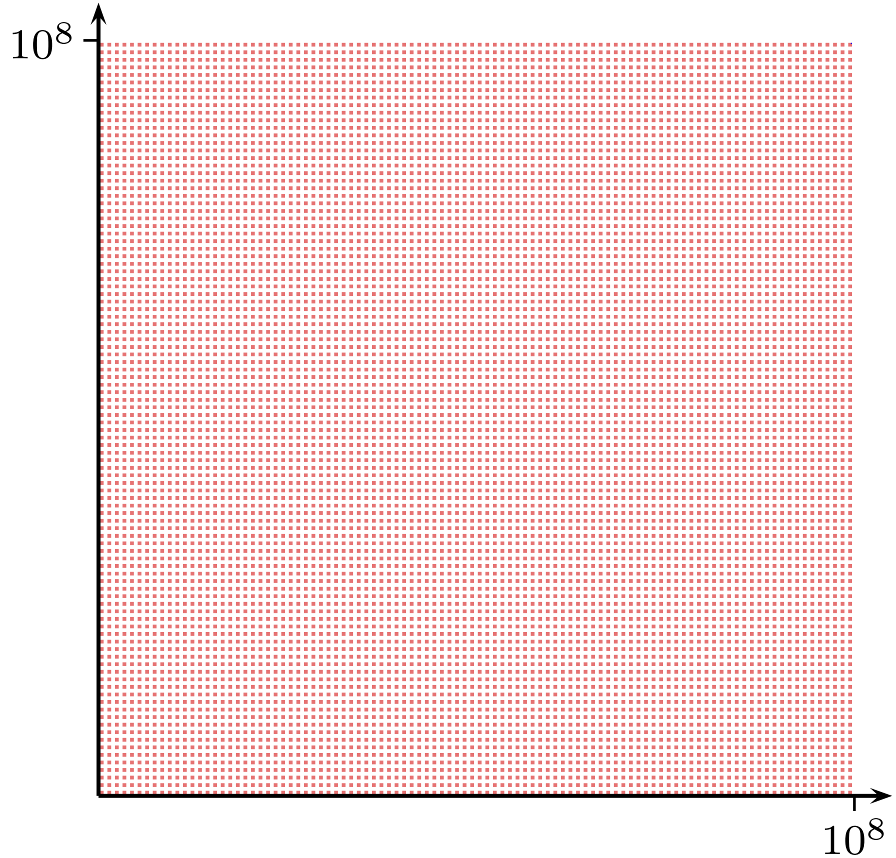
Observation: Software Usually Has Many Inputs

- **Example**: Simple Pocket Calculator.

  With **ten thousand** 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: untested, blue: tested)



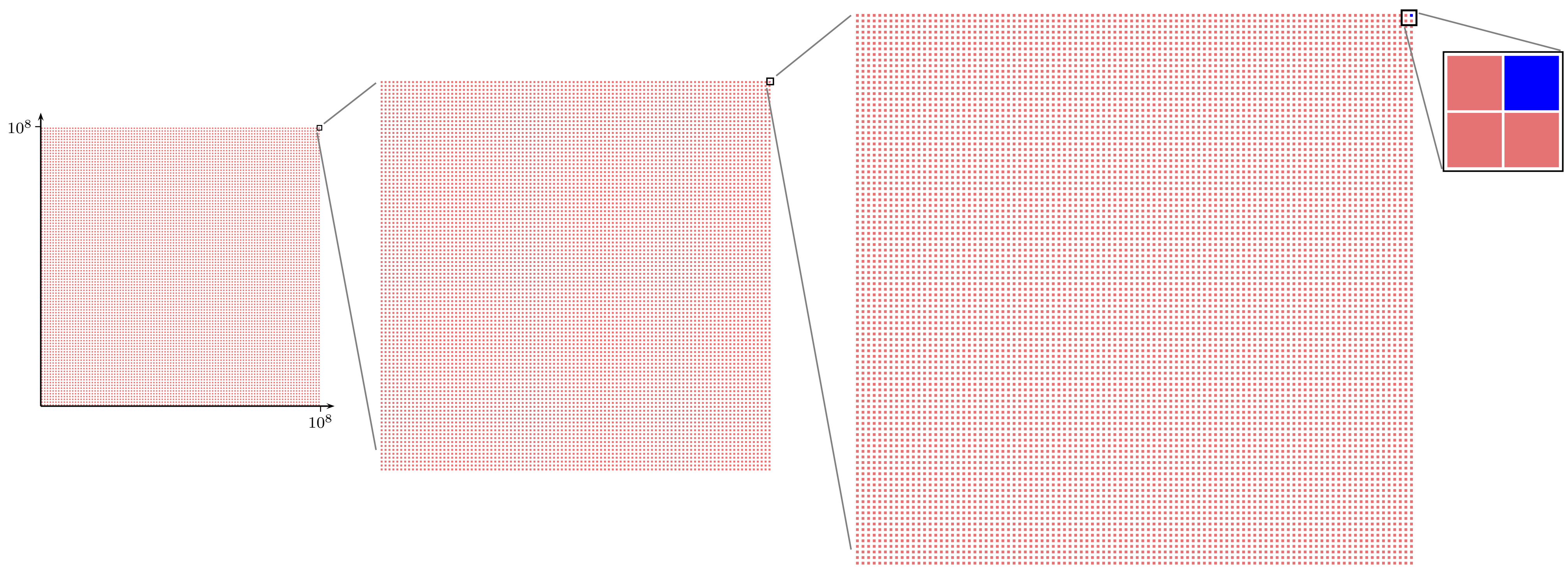
Observation: Software Usually Has Many Inputs

- **Example**: Simple Pocket Calculator.

  With **ten thousand** 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: untested, blue: tested)



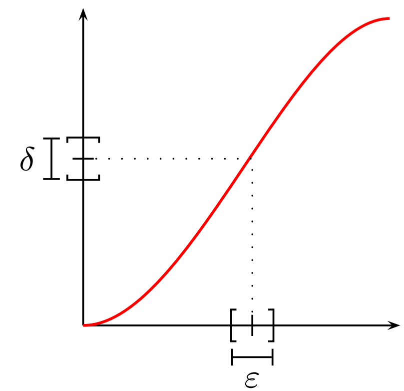
Conclusion

* **Question 1**:
  If we **cannot consider all** test cases, are there **clever choices** of test cases?

More Observations

- Software is (in general) **not continous**.
  - Consider a continuous function, e.g.

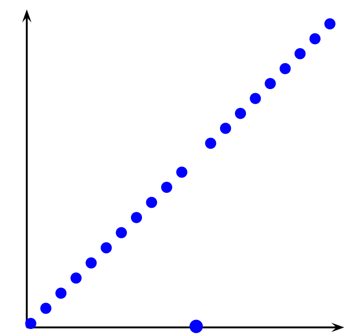
    For sufficiently small ε-environments of an input, the outputs **differ only by a small amount** δ.



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

    Vocabulary:

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



- For Software, we can (in general) not **conclude from some values to others**:
  - For example, if a bridge endures a single car of 1000 kg, we strongly expect the bridge to hold a single person of 100 kg.
  - If the pocket calculator is correct for 12345678 + 27, we can (in general) not expect anything on the other numbers.

Conclusion Cont'd

- **Question 1**:
  If we **cannot consider all** test cases,
  are there **clever choices** of test cases?

- **Question 2**:
  If we **cannot conclude** from few test cases to all inputs,
  when should we **stop testing**?

When To Stop Testing?

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 is done**" criteria:
  - all (previously) **specified test cases**
    have been executed with negative result,

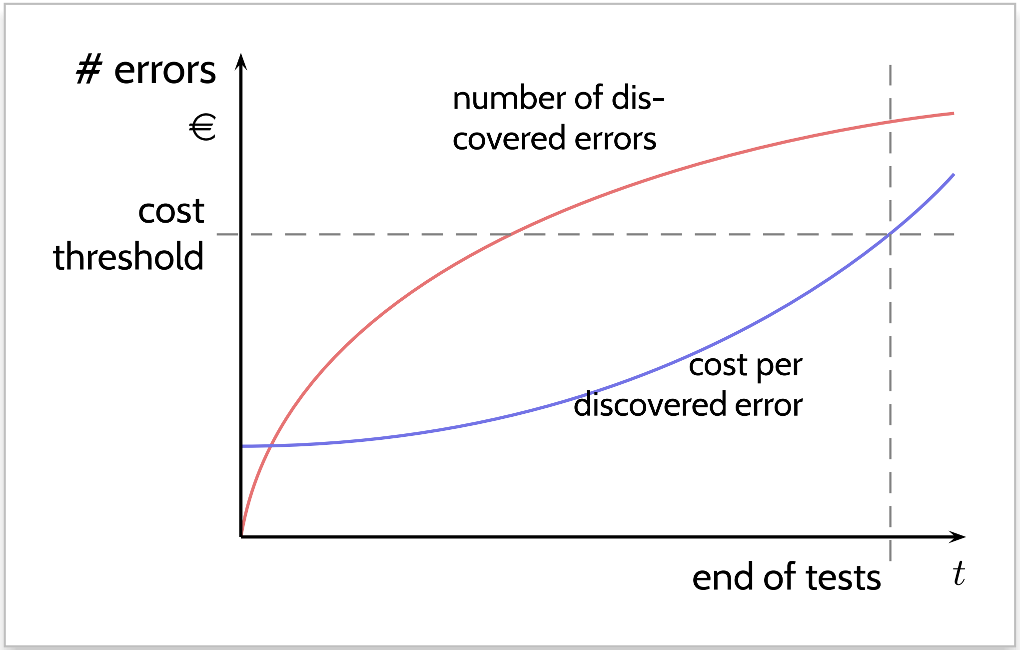
    (**Special case**: All test cases resulting from a certain strategy,
    like maximal statement coverage ($\rightarrow$ in a minute) have been executed.)
  - **testing effort time** sums up to x (hours, days, weeks),
  - **testing effort** sums up to y (any other useful unit),
  - n **errors** have been 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 is done**" criterion:
  - The **average cost per error discovery** exceeds a defined threshold c.



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

Content

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range- vs. point errors
- **When To Stop Testing?**
- **Choosing Test Cases**
  - **Requirements** on test cases
  - The **natural habitat** of many errors
  - Test **Oracle**
- **Glass-Box Testing**
  - **Statement coverage**
  - **Branch** and **term coverage**
  - **Conclusions** from coverage measures
- **Model-Based Testing**
- **Testing in the Development Process**

Choosing Test Cases

Choosing Test Cases

A test case is a **good test case** if 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 wish for representative test cases is **particularly problematic**:

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

Still, 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 "buy water", "buy soft-drink", "buy tea" vs. "buy beverage".

Recall: strive to have at least one test case per feature.

> "Who is hunting lions, should know how a lion looks like.
> One should also know where the lion likes to stay, which traces the lion leaves behind, and which sounds the lion makes." (Ludewig and Lichter, 2013)

Lion and Error Hunting

Hunting errors in software is (basically) the same.

Some traditional popular belief on software error habitat:

- Software errors (in contrast to lions?) **(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),
  - **special cases** of the problem (empty list, use-case without actor, $\ldots$),
  - special cases of the programming language semantics,
  - **complex implementations**.

$\rightarrow$ **Good idea**: for each test case, note down why it has been chosen.
For example, "demonstrate that error handling is not completely broken".

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 the **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)

Glass-Box Testing: Coverage

Statements and Branches by Example

Definition. Software is a finite description S of a (possibly infinite) set $[\![S]\!]$ 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**).

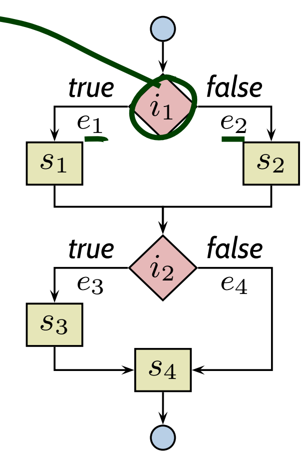
- In the following, we assume that
  - S has a **control flow graph** $(V, E)_S$, and **statements** $Stm_S \subseteq V$ and **branches** $Cnd_S \subseteq E$,
  - each state σ gives information on statements and control flow graph branch edges **which were executed** right before obtaining σ:

$$stm : \Sigma \to 2^{Stm_S}, \qquad cnd : \Sigma \to 2^{Cnd_S},$$

```
int f( int x, int y, int z )
{
i1: if (x > 100 ∧ y > 10)      e1
s1:     z = z * 2;
        else
s2:     z = z/2;               e2
i2: if (x > 500 ∨ y > 50)
s3:     z = z * 5;
s4: return z;
}
```

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

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



Statements and Branches by Example

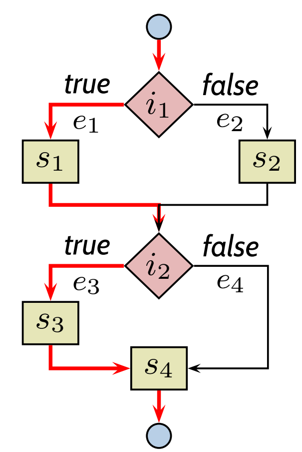
- In the following, we assume that
  - S has a **control flow graph** $(V, E)_S$, and **statements** $Stm_S \subseteq V$ and **branches** $Cnd_S \subseteq E$,
  - each state σ gives information on statements and control flow graph branch edges **which were executed** right before obtaining σ:

$$stm : \Sigma^{\omega} \to 2^{Stm_S}, \qquad cnd : \Sigma^{\omega} \to 2^{Cnd_S},$$

```
 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: }
```

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

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



$$\ldots \to \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \xrightarrow{\alpha_3} \sigma_3 \xrightarrow{\alpha_4} \sigma_4 \xrightarrow{\alpha_5} \sigma_5$$

	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5
pc	3	4	7	8	9	10
x	501	501	501	501	501	501
y	11	11	11	11	11	11
z	0	0	0	0	0	0
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 π of test case T achieves $p\,\%$ **statement coverage** if and only if

$$p = cov_{stm}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} stm(\sigma_i)|}{|Stm_S|}, |Stm_S| \neq 0.$$

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

- Execution π of T achieves $p\,\%$ **branch coverage** if and only if

$$p = cov_{cnd}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} cnd(\sigma_i)|}{|Cnd_S|}, |Cnd_S| \neq 0.$$

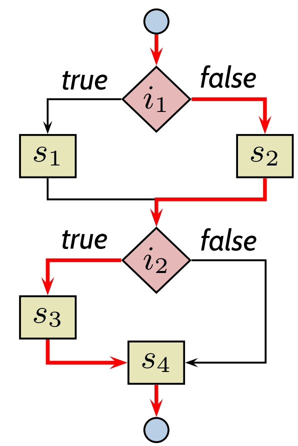
  Test case T achieves $p\,\%$ **branch coverage** if and only if $p = \min_{\pi \text{ execution of } T} cov_{cnd}(\pi)$.

- **Define**: $p = 100$ for empty program.
- Statement/branch coverage canonically extends to test suite $\mathcal{T} = \{T_1, \dots, T_n\}$, e.g. given executions $\pi_1, \dots, \pi_n$, $\mathcal{T}$ achieves

$$p = \frac{|\bigcup_{1 \leq j \leq n} \bigcup_{i \in \mathbb{N}_0} stm(\pi^i_j)|}{|Stm_S|}, |Stm_S| \neq 0, \text{ **statement coverage**}.$$

Coverage Example

$\texttt{int}\ f(\ \texttt{int}\ x,\ \texttt{int}\ y,\ \texttt{int}\ z\ )$
$\{$
i_1: $\texttt{if}\ (x > 100 \land y > 10)$
s_1: $\quad z = z * 2;$
$\quad \texttt{else}$
s_2: $\quad z = z/2;$
i_2: $\texttt{if}\ (\overbrace{x > 500}^{c_1} \lor \overbrace{y > 50}^{c_2})$
s_3: $\quad z = z * 5;$
s_4: $;$
$\}$



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

test suite coverage

In x, y, z	i_1/t	i_1/f	s_1	s_2	i_2/t	i_2/f	c_1	c_2	s_3	s_4	% stm	% cnd	i_2/% term
$501, 11, 0$	✔		✔		✔		✔		✔	✔	75	50	25
$501, 0, 0$		✔		✔	✔		✔		✔	✔	100	75	25
$0, 0, 0$		✔		✔		✔				✔	100	100	75
$0, 51, 0$		✔		✔	✔			✔		✔	100	100	100

Term Coverage

- Consider the statement

$$\texttt{if } (\overbrace{A \wedge (B \vee (C \wedge D)) \vee E}^{expr}) \texttt{ then } \dots;$$

where A, …, E are **minimal** boolean terms, e.g. $x > 0$, but not $a \vee b$.

Branch coverage is easy in this case:

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

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

	A	B	C	D	E	b	%
β_1	1	1	0	0	0	1	20
β_2	1	0	0	1	0	0	50
β_3	1	0	1	1	0	1	70
β_4	0	0	1	0	1	1	80

red: b-effective, black: otherwise

- **Term Coverage** (for an expression $expr$):
  - Let $\beta : \{A_1, \dots, A_n\} \to \mathbb{B}$ be a valuation of the terms.
  - Term A_i is b-**effective** in β for $expr$ if and only if

$$\beta(A_i) = b \text{ and } [\![expr]\!](\beta[A_i/\textit{true}]) \neq [\![expr]\!](\beta[A_i/\textit{false}]).$$

  - $\Xi \subseteq (\{A_1, \dots, A_n\} \to \mathbb{B})$ achieves $p\,\%$ **term coverage** if and only if

$$p = \frac{|\{A_i^b \mid \exists\, \beta \in \Xi \bullet A_i \text{ is } b\text{-effective in } \beta\}|}{2n}.$$

Unreachable Code

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{int}\ f(\ \texttt{int}\ x,\ \texttt{int}\ y,\ \texttt{int}\ z\ )} \\
\multicolumn{2}{l}{\{} \\
i_1: & \texttt{if}\ (x \neq x) \\
s_1: & \quad z = y/0; \\
i_2: & \texttt{if}\ (x = x \lor z/0 = 27) \\
s_2: & \quad z = z * 2; \\
s_3: & \texttt{return}\ z; \\
\multicolumn{2}{l}{\}}
\end{array}
$$

- Statement s_1 is **never executed** (because $x \neq x \iff$ *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 $\mathcal{T}$ tests software S for the following property φ:
  - **pre-condition**: p, **post-condition**: q,

  and S passes (!) $\mathcal{T}$, 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 φ"

    (Still, there may be many, many computation paths which violate φ, and which just have not been touched by $\mathcal{T}$.)
  - "there is no unreachable statement"

- 100 % **branch** (**term**) coverage:
  - "there is no single branch (term) which **necessarily causes** violations of φ"

    In other words: "for each condition (term), there is one computation path satisfying φ 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...

Coverage Measures in Certification

- (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

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range- vs. point errors
- **When To Stop Testing?**
- **Choosing Test Cases**
  - **Requirements** on test cases
  - The **natural habitat** of many errors
  - Test **Oracle**
- **Glass-Box Testing**
  - **Statement coverage**
  - **Branch** and **term coverage**
  - **Conclusions** from coverage measures
- **Model-Based Testing**
- **Testing in the Development Process**

Statistical Testing

Another Approach: Statistical Tests

Classical **statistical testing** is another approach to deal with

- in practice not exhaustively testable **huge input space**,
- **tester bias**.

  (People tend to choose "good-will" inputs and disregard 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.)

Statistical Testing: Discussion

(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.

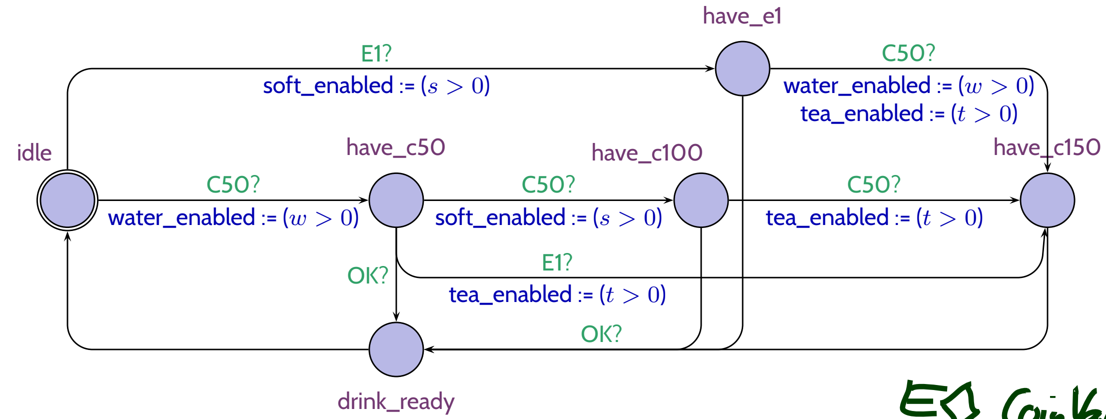
- There is a high risk for **not finding** **point** or **small-range** errors.

  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**.

Model-Based Testing

Model-based Testing



$E\Diamond$ CoinVal. have_c150

- 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

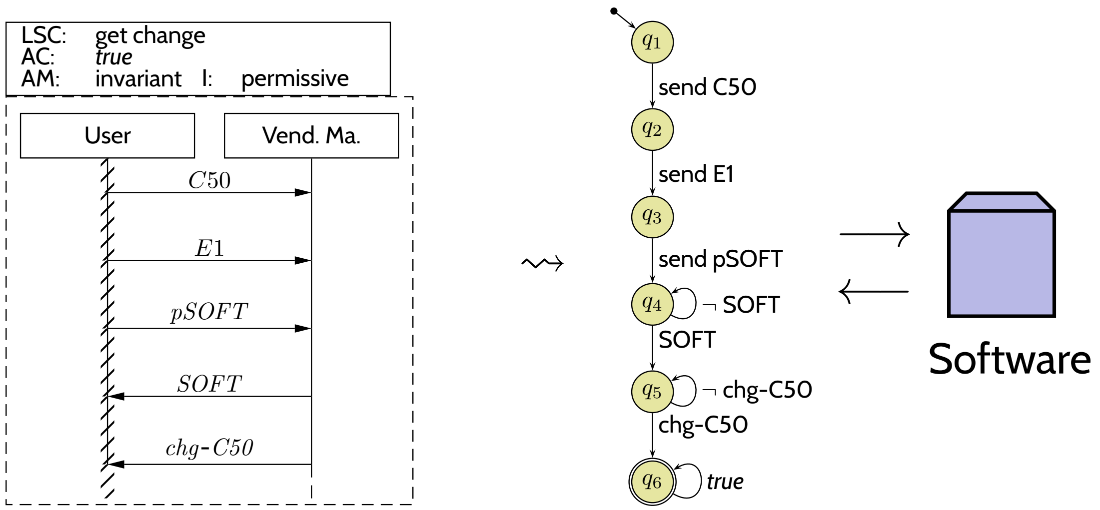
    $$T_1 = (\text{C50}, \text{C50}, \text{C50}; \{\pi \mid \exists\, i < j < k < \ell \bullet \pi^i \sim \text{idle}, \pi^j \sim \text{h\_c50}, \pi^k \sim \text{h\_c100}, \pi^\ell \sim \text{h\_c150}\})$$

  - Check for 'have_e1' by $T_2 = (\text{C50}, \text{C50}, \text{C50}; \dots)$.
  - 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.

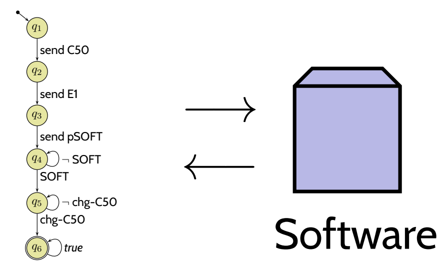
Existential LSCs as Test Driver & Monitor (Lettrari and Klose, 2001)



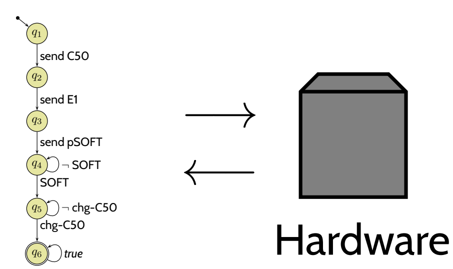
- If the LSC has designated **environment instance** lines, we can distinguish:
  - messages expected to **originate from** the environemnt (**driver role**),
  - messages expected **adressed to** the environemnt (**monitor role**).
- Adjust the TBA-construction algorithm to construct a **test driver & 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.



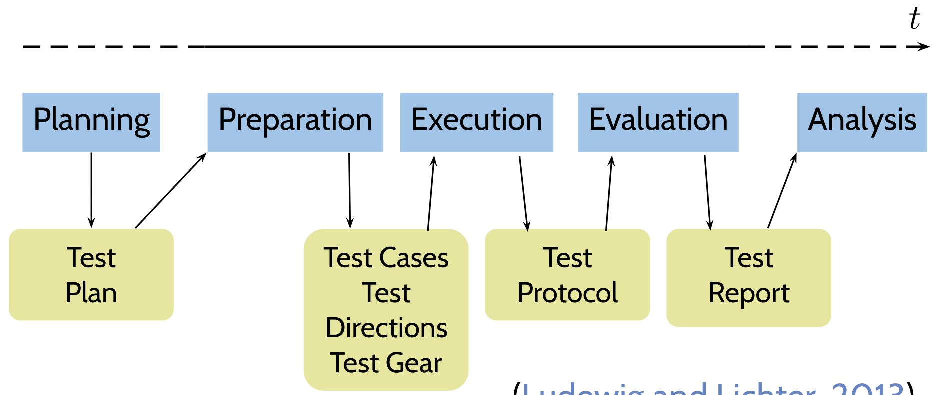
- **Hardware-in-the-loop**:
  The final implementation is running on (prototype) hardware
  which is connected by its standard input/output interface (e.g. CAN-bus)
  to a separate computer which simulates other system components.

Content

- **Limits of Software Testing**
  - Software examination paths
  - Is exhaustive testing feasible?
  - Range- vs. point errors
- **When To Stop Testing?**
- **Choosing Test Cases**
  - **Requirements** on test cases
  - The **natural habitat** of many errors
  - Test **Oracle**
- **Glass-Box Testing**
  - **Statement coverage**
  - **Branch** and **term coverage**
  - **Conclusions** from coverage measures
- **Model-Based Testing**
- **Testing in the Development Process**

Testing in The Software Development Process

Test Conduction: Activities & Artefacts



(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!

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**,
- Define criteria for **"testing done"** (like coverage, or cost per error).
- 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,
  - ~~**runtime verification** (`assert!`)~~ → VL 18
- **Process**: tester and developer should be different persons.

References

References

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

Lettrari, M. and Klose, J. (2001). Scenario-based monitoring and testing of real-time UML models. In Gogolla, M. and Kobryn, C., editors, *UML*, number 2185 in Lecture Notes in Computer Science, pages 317–328. Springer-Verlag.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.