

Softwaretechnik / Software-Engineering

Lecture 15: Testing

2017-07-17

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

-15-2017-07-17-main-

Topic Area Code Quality Assurance: Content

- VL 14
 - **Introduction and Vocabulary**
 - Test case, test suite, test execution.
 - Positive and negative outcomes.
- VL 15
 - **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 16
 - **Program Verification**
 - partial and total correctness,
 - Proof System PD.
- VL 17
 - **Review**

-15-2017-07-17-SlidesContent-

- **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**

-15-2017-07-17 - Spennett-

Recall: Test Case, Test Execution

Test Case

Definition. A test case T over Σ 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 $[\]$ of these descriptions:

- $[In] \subseteq (\Sigma_w \times A)^*$, $[Soll] \subseteq (\Sigma \times A)^* \cup (\Sigma \times A)^*$

Examples:

- Test case for procedure `strlen`: $String \rightarrow \mathbb{N}$, s denotes parameter, r return value:

$$T = (s = "abc", r = 3)$$

$[s = "abc"] = \{\sigma_0 \xrightarrow{s} \sigma_1 \mid \sigma_1(s) = "abc"\}$, $[r = 3] = \{\sigma_0 \xrightarrow{r} \sigma_1 \mid \sigma_1(r) = 3\}$.

Shorthand notation: $T = ("abc", 3)$.

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

45u

Executing Test Cases

- A computation path $\pi = (\sigma_0^i, \sigma_1^i) \xrightarrow{\sigma_2^i} (\sigma_3^i, \sigma_4^i) \xrightarrow{\sigma_5^i} \dots$

from $[S]$ is called **execution of test case** $(In, Soll)$ if and only if

- there is $n \in \mathbb{N}$ such that $\sigma_0 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} \sigma_n \downarrow \Sigma_w \downarrow \Sigma_w \in [In]$.

[A prefix of π corresponds to an input sequence.]

Execution π 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".

46u

Tell Them What You've Told Them...

- Testing is about
 - Finding errors, or
 - demonstrating scenarios.
- A test case consists of
 - **input sequence(s) and**
 - **expected outcome(s)**
- 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.

59u

-15-2017-07-17 - Small-

Not Executing Test Cases

- Consider the test case

for procedure `strlen`.

("Empty string has length 0.")

$T = ("", 0)$

\neq

- A tester observes the following software behaviour:

$$\pi = \underbrace{\{s \mapsto \text{NULL}, r \mapsto 0\}}_{=\sigma_0} \xrightarrow{\tau} \underbrace{\text{program-abortion}}_{\sigma_1}$$

- Test execution **positive** or **negative**?

// /

Note:

- If a tester does not adhere to an allowed input sequence of T , π **is not** a test execution. Thus π is neither positive nor negative (only defined for test executions).
- Same case: power outage (if continuous power supply is considered in input sequence).

By The Way... (Good Design)

- 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 $\{T_1, \dots, T_n\}$.
- 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. (Ludewig and Lichter, 2013)

Test – (one or multiple) execution(s) of a program on a computer with the goal to find errors. (Ludewig and Lichter, 2013)

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**.

- **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.

-18-2007-0717 - Striveoc -

11/60

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?

-18-2007-0717 - Striveoc -

12/60

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.

-15-2017-07-17 - Slides -

13/60

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**

-15-2017-07-17 - Content -

14/60

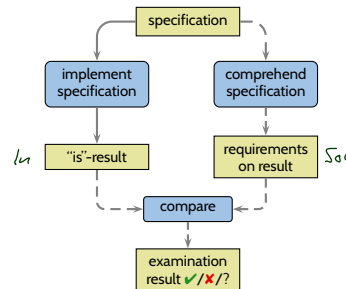
The Limits of Software Testing

-15-2007-07-17-main-

15/60

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**.



→ information flow development
 - - -> information flow examination

(Ludewig and Lichter, 2013)

Recall:

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

Handwritten notes on the table:

- SW broken, test ex. neg. (pointing to false negative)
- SW broken, test ex. pos. (pointing to true positive)
- SW okay, test ex. neg. (pointing to true negative)
- SW okay, test ex. pos. (pointing to false positive)

-15-2007-07-17-5min-

16/60

Recall: Quotes On Testing

“Software testing can be used to show the presence of bugs,
but never to show their absence!”

(E. W. Dijkstra, 1970)

beweisen

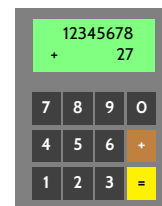
-15-2007-07-17-5mins-

17/60

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

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.)



-15-2007-07-17-5mins-

18/60

Observation: Software Usually Has Many Inputs

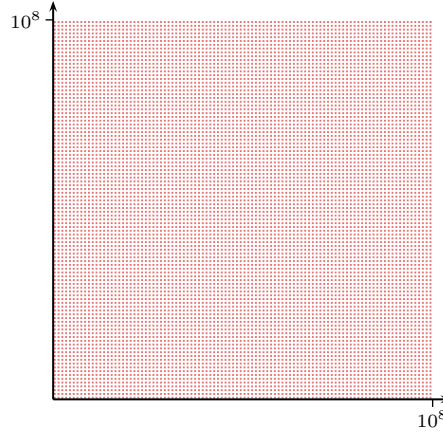
- **Example:** Simple Pocket Calculator.

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)



-15-2007-07-17 - Slides -

19/60

Observation: Software Usually Has Many Inputs

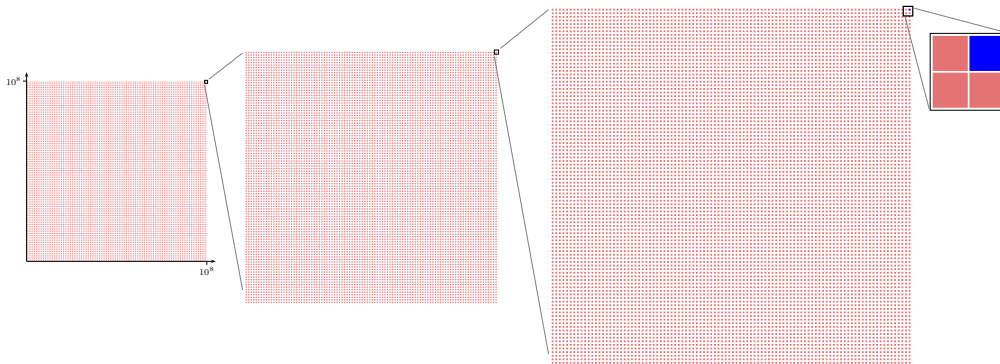
- **Example:** Simple Pocket Calculator.

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)



-15-2007-07-17 - Slides -

19/60

- **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**

20/60

-15-2007-07-17 - Segment -

Choosing Test Cases

-15-2007-07-17 - main -

22/60

Generic Requirements on Good Test Cases: Reproducibility

- 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).

-18-2007-07-17 - Steinhilber -

23/60

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.

-18-2007-07-17 - Steinhilber -

24/60

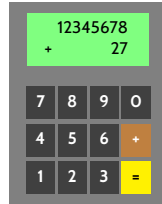
What Else Makes a Test Case a Good Test Case?

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.

-18-2007-07-17 - Steffen -

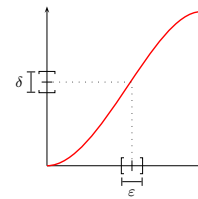
25/60

Point vs. Range Errors

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

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

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



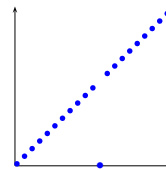
- **Physical systems** are (to a certain extent) continuous:

- 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**.

-18-2007-07-17 - Steffen -

26/60

What Else Makes a Test Case a Good Test Case?

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

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, \dots, 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)

-18-2007-07-17 - Seite 6 -

33/60

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**

-18-2007-07-17 - Content -

34/60

Habitat-based Testing

-15-2017-07-17 - main -

35/60

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”.

-15-2017-07-17 - Habitat -

36/60

- **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**

37/60

-15-2017-07-17 - Segment -

Glass-Box Testing: Coverage

-15-2017-07-17 - main -

38/60

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

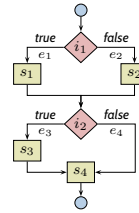
$$stm : (\Sigma \times A)^* \rightarrow 2^{Stm_S}, \quad cnd : (\Sigma \times A)^* \rightarrow 2^{Cnd_S}$$

```

1: int f(int x, int y, int z)
2: {
3:   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:}
    
```

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

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



-18-2017-07-17 - Saver -

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

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

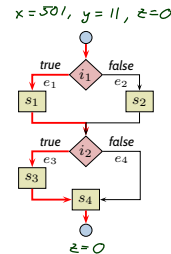
```

1: int f(int x, int y, int z)
2: {
3:   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:}
    
```

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

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

i1/false, i2/false



	σ_0	$\xrightarrow{\alpha_1}$	σ_1	$\xrightarrow{\alpha_2}$	σ_2	$\xrightarrow{\alpha_2}$	σ_3	$\xrightarrow{\alpha_3}$	σ_4	$\xrightarrow{\alpha_4}$	σ_5	$\xrightarrow{\alpha_5}$	σ_6
	pc: 1		pc: 3		pc: 4		pc: 7		pc: 8		pc: 9		pc: 10
	x: 501		x: 501		x: 501		x: 501		x: 501		x: 501		x: 501
	y: 11		y: 11		y: 11		y: 11		y: 11		y: 11		y: 11
	z: 0		z: 0		z: 0		z: 0		z: 0		z: 0		z: 0
stm:	{}		{}		{}		{s1}		{}		{s3}		{s4}
cnd:	{}		{}		{e1}		{}		{e3}		{}		{}
					<i>i1/true</i>				<i>i2/true</i>				

-18-2017-07-17 - Saver -

Glass-Box Testing: Coverage

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

- Execution $\pi = \sigma_0 \xrightarrow{\alpha_1} \dots$ of test case T achieves $p\%$ statement coverage if and only if

$$p = cov_{stm}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} stm(\sigma_0 \dots \sigma_i)|}{|Stm_S|}, |Stm_S| \neq 0.$$

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

$$p = cov_{cnd}(\pi) := \frac{|\bigcup_{i \in \mathbb{N}_0} cnd(\sigma_0 \dots \sigma_i)|}{|Cnd_S|}, |Cnd_S| \neq 0.$$

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.)

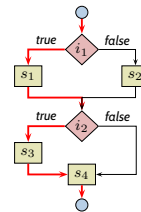
- Statement/branch coverage canonically extends to test suite $\mathcal{T} = \{T_1, \dots, T_n\}$.
For example, given $\pi_1 = \sigma_0^1 \dots, \dots, \pi_n = \sigma_0^n \dots$, then \mathcal{T} achieves

$$p = \frac{|\bigcup_{1 \leq j \leq n} \bigcup_{i \in \mathbb{N}_0} stm(\sigma_0^j \dots \sigma_i^j)|}{|Stm_S|}, |Stm_S| \neq 0, \text{ statement coverage.}$$

-18-2017-07-17 - Sover -

Coverage Example

```
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. $Soll = \Sigma^* \cup \Sigma^\omega$.

In	x, y, z	i1/t	i1/f	s1	s2	i2/t	i2/f	c1	c2	s3	s4	test suite coverage		
												% stm	% cnd	i2/% term
501, 11, 0	✓		✓			✓		✓		✓	✓	75	50	25
12, 11, 0			✓		✓		✓				✓	100	100	

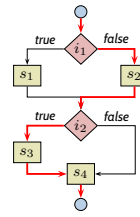
-18-2017-07-17 - Sover -

Coverage Example

```

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. $Soll = \Sigma^* \cup \Sigma^\omega$.

In											test suite coverage		
	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

-18-2017-07-17 - Solver -

41/60

Term Coverage

- Consider the statement

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

where A, \dots, 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 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 $\beta : \{A_1, \dots, A_n\} \rightarrow \mathbb{B}$ be a valuation of the terms.
 - Term A_i is **b -effective** in β for $expr$ if and only if

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

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

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

-18-2017-07-17 - Solver -

42/60

Unreachable Code

```
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 $x \neq x \iff \text{false}$), 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.)

-18-2017-07-17 - Saver -

43/60

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...

-18-2017-07-17 - Saver -

44/60

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.

-18-2007-0717 - Server -

45/60

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-continuous**,
 - 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.

-18-2007-0717 - Shuyi -

58/60

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.