Lecture 16: Testing & Review

2015-07-13

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Contents of the Block “Quality Assurance”

(i) Introduction and Vocabulary
- correctness illustrated
- vocabulary: fault, error, failure
- three basic approaches

(ii) Formal Verification
- Hoare calculus
- Verifying C Compiler (VCC)
- over- / under-approximations

(iii) (Systematic) Tests
- systematic test vs. experiment
- classification of test procedures
- model-based testing
- glass-box tests: coverage measures

(iv) Runtime Verification

(v) Review

(vi) Concluding Discussion
- Dependability
Contents & Goals

Last Lecture:
• Completed the block "Architecture & Design"

This Lecture:
• Educational Objectives: Capabilities for following tasks/questions.
  • What can we conclude from the outcome of tools like VCC?
  • What is an example for not a test, non-systematic test, systematic test?
  • Given a test case and a software, is the outcome successful or unsuccessful?
  • How many test cases are necessary for exhaustive testing of a given software?

• Content:
  • The Verifying C Compiler (VCC)
  • Systematic test, test case, test suite
  • Testing notions
  • Coverage measures

The Verifying C Compiler
VCC

- The **Verifying C Compiler** (VCC) basically implements Hoare-style reasoning.

- **Special syntax:**
  - `#include <vcc.h>`
  - `#(requires p) — pre-condition, p is a C expression`
  - `#(ensures q) — post-condition, q is a C expression`
  - `#(invariant expr) — loop invariant, expr is a C expression`
  - `#(assert p) — intermediate invariant, p is a C expression`
  - `#(writes &v) — VCC considers concurrent C programs; we need to declare for each procedure which global variables it is allowed to write to (also checked by VCC)`

- **Special expressions:**
  - `\thread_local(&v) — no other thread writes to variable v (in pre-conditions)`
  - `\old(v) — the value of v when procedure was called (useful for post-conditions)`
  - `\result — return value of procedure (useful for post-conditions)`

### VCC Syntax Example

```
#include <vcc.h>

int q, r;

void div(int x, int y)
{
    #requires x >= 0 && y >= 0
    #ensures q + y + r = x && r < y
    #writes &q
    #writes &r
    q = 0;
    r = x;
    while (r >= y)
    {
        #invariant q + y + r = x && r >= 0
        r = r - y;
        q = q + 1;
    }
}
```

\[ DIV \equiv q := 0; \ r := x; \ \text{while } r \geq y \ \text{do } r := r - y; \ q := q + 1 \\text{ do} \]

\[ \{ x \geq 0 \land y \geq 0 \} \ DIV \ { q \cdot y + r = x \land r < y} \]
VCC Features

- For the exercises, we use VCC only for **sequential, single-thread programs**.
- VCC checks a number of **implicit assertions**:
  - **no arithmetic overflow** in expressions (according to C-standard),
  - **array-out-of-bounds access**,
  - **NULL-pointer dereference**,  
  - and many more.
- VCC also supports:
  - **concurrency**: different threads may write to shared global variables; VCC can check whether concurrent access to shared variables is properly managed;
  - **data structure invariants**: we may declare invariants that have to hold for, e.g., records (e.g. the length field $l$ is always equal to the length of the string field $str$); those invariants may **temporarily** be violated when updating the data structure.
  - and much more.
- Verification **does not always succeed**:
  - The backend SMT-solver may not be able to discharge proof-obligations (in particular non-linear multiplication and division are challenging);
  - In many cases, we need to provide **loop invariants** manually.

Interpretation of Results

- **VCC says: “verification succeeded**
  
  We can only conclude that the tool —  
  under its interpretation of the C-standard,  
  under its platform assumptions (32-bit), etc.  
  — “thinks” that it can prove $\models \{ p \} \text{DIV} \{ q \}$. Can be due to an error in the tool!  
  Yet we can ask for a printout of the proof and check it manually (hardly possible in practice) or  
  with other tools like interactive theorem provers.  
  
  **Note:** $\models \{ \text{false} \} f \{ q \}$ always holds  
  — so a mistake in writing down the pre-condition can provoke a false negative.
- **VCC says: “verification failed**
  - One case: “timeout” etc. — completely inconclusive outcome.  
  - The tool **does not provide counter-examples** in the form of a computation path.  
  
  It (only) gives hints on input values satisfying $p$ and causing a violation of $q$.  
  May be a false negative if these inputs are actually never used.  
  Make pre-condition $p$ stronger, and try again.
Recall: Three Basic Directions

Testing
Quotes On Testing

“Testing is the execution of a program with the goal to discover errors.”

G. J. Myers, 1979

“Testing is the demonstration of a program or system with the goal to show that it does what it is supposed to do.”

W. Hetzel, 1984

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

E. W. Dijkstra, 1970

Rule-of-thumb: (fairly systematic) tests discover half of all errors.

(Ludewig and Lichter, 2013)

Tests vs. Systematic Tests

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

(Ludewig and Lichter, 2013)

(Our) Synonyms: Experiment, ‘Rumprobieren’.

**Not (even) a test** (in the sense of this weak definition):
- any inspection of the program,
- demo of the program,
- analysis by software-tools, e.g. for values of metrics,
- investigation of the program with a debugger.

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

(Ludewig and Lichter, 2013)

In the following: experiment ::= test — test ::= systematic test.
More Formally: Test Case

- A test case $T$ is a set of pairs $\{(In_i, Soll_i), \ldots\}$ consisting of
- a (description of a) finite input sequence $In_i$ (pairwise different in $T$),
- a (description of a) finite set of expected computation path $Soll_i$.

Examples:
- $T_1 = \{\text{FILLUP}, \text{C50; water} \text{button} \text{on}\}$
  (shorthand notation)
  (fill up vending machine (at any time after power on), insert C50 coin (at any time),
  expect water button is enabled (some time later))
- $T_2 = \{(\sigma_0 \xrightarrow{\alpha_1} \sigma_1; \sigma_0 \xrightarrow{\alpha_1} \sigma_1) \mid \sigma_0(x) = 7 \land \sigma_1(y) = 49\}$
  (input 7, expect output 49, don’t care for other variables’ values; shorthand notation: (7; 49))
- $T_3 = \{(\sigma_0 \xrightarrow{\tau_1} \sigma_1; \sigma_0 \xrightarrow{\tau_1} \sigma_1) \mid \sigma_0 = \sigma_0 = 0[x := 7], \sigma_0 = 0, \sigma_1 = 0[y := 49]\}$
  (each and every variable value at start and at end fixed)

Test Case Execution, Test Suite
Test Case Execution, Test Suite

• An execution of test case $T$ for software $S$ is a computation path of $S$

$$\pi = \left( \begin{array}{c} \sigma_0' \\ \sigma_0 \\ \alpha_1 \\ \sigma_1 \\ \alpha_2 \\ \sigma_2 \\ \vdots \end{array} \right)$$

where $\sigma_0' \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \xrightarrow{\cdots} \sigma_i = \text{In}_i$ for some $i$ in $T$.

• The test case execution is called

  • **successful** (or positive) if it discovered an error, i.e. if $\pi \notin S_{\text{oll}}$.
    (Alternative: test item failed to pass test; confusing: "test failed").
  
  • **unsuccesful** (or negative) if it did not discover an error, i.e. if $\pi \in S_{\text{oll}}$.
    (Alternative: test item passed test; okay: "test passed").

  **Note:** if input sequence not adhered to, or power outage, etc., it is not a test execution.

• A test suite is a set of test cases.

  Execution, positive, and negative are lifted canonically.

The Outcome of Systematic Tests Depends on...

• **inputs:**
  • the input vector of the test case (of course), possibly with timing constraints,
  • other interaction, e.g., from network,
  • initial memory content,
  • etc.

• **(environmental) conditions:**
  any aspects which could have an effect on the outcome of the test 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 tested when?
  • etc.

  ... so strictly speaking all of them need to be specified within (or as an extension to) $\text{In}$.

• In practice, this is hardly possible — but one wants to specify as much as possible in order to achieve **reproducibility**.

• One approach:
  • have a fixed build environment, a fixed test host which does not do any other jobs, etc.
Software Examination (in Particular Testing)

- In each check, there are **two paths** from specification to result:
  - the **production path** (using model, source code, executable, etc.), and
  - the **examination path** (using requirements specification).
- A check can only discover errors on **exactly one** of the paths.
- What is not on the paths, is not checked; crucial: **specification** and **comparison**.
- Difference detected: examination result is **positive**.

**Recall:**

<table>
<thead>
<tr>
<th>Checking Procedure</th>
<th>Reports Error</th>
<th>Shows No Error</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true positive</td>
<td>✔</td>
<td>□</td>
<td>✔</td>
</tr>
<tr>
<td>true negative</td>
<td></td>
<td>□</td>
<td>✗</td>
</tr>
<tr>
<td>false positive</td>
<td></td>
<td>□</td>
<td>?</td>
</tr>
<tr>
<td>false negative</td>
<td></td>
<td>□</td>
<td>✗</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)

**Test Conduction**

- **Test Gear**:
  - **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.  
    \[\text{IEEE 610.12 (1990)}\]
  - **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.  
    \[\text{IEEE 610.12 (1990)}\]
  - **hardware-in-the-loop, software-in-the-loop**: the final implementation is running on (prototype) hardware, other system component are simulated by a separate computer.
Specific Testing Notions

- How are the test cases chosen?
  - Considering the structure of the test item (glass-box or structure test).
  - Considering only the specification (black-box or function 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,
  - systematic test — somebody (not author!) derives test cases, defines input/output, documents test execution.

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 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.
  - regression test — does the new version of the software behave like the old one on inputs where no behaviour change is expected?
  - response time, minimal hardware (software) requirements, etc.

- Which roles are involved in testing?
  - only the developer, or selected (potential) customers (alpha and beta test),
  - acceptance test — the customer tests whether the system (or parts of it, at milestones) test whether the system is acceptable.
• **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.

\[\begin{array}{|c|c|c|c|}
\hline
1 & 2 & 3 & = \\
\hline
\end{array}\]

\[\begin{array}{|c|c|c|c|}
\hline
1 & 2 & 3 & = \\
\hline
\end{array}\]
Testing the Pocket Calculator

Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small, e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error), e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...

Testing the Pocket Calculator

Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small, e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error), e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...

23/65
Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,  
  e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error),  
  e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error),  
  e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error),  
  e.g. $99999999 + 1$
- ...

Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,  
  e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error),  
  e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error),  
  e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error),  
  e.g. $99999999 + 1$
- ...

Test some representatives of "equivalence classes":

- \( n + 1 \), \( n \) small, e.g. 27 + 1
- \( n + m \), \( n \) small, \( m \) small (for non error), e.g. 13 + 27
- \( n + m \), \( n \) big, \( m \) big (for non error), e.g. 12345 + 678
- \( n + m \), \( n \) huge, \( m \) small (for error), e.g. 99999999 + 1
- ...

Test some representatives of "equivalence classes":

- \( n + 1 \), \( n \) small,
- \( n + m \), \( n \) small, \( m \) small (for non error),
- \( n + m \), \( n \) big, \( m \) big (for non error),
- \( n + m \), \( n \) huge, \( m \) small (for error),
- ...

23/65
Test some representatives of "equivalence classes":

- $n + 1$, $n$ small, e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error), e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...
Test some representatives of "equivalence classes":

- $n + 1$, $n$ small,
- $n + m$, $n$ small, $m$ small (for non error), e.g. $27 + 1$ e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...

Testing the Pocket Calculator: One More Try
Testing the Pocket Calculator: One More Try

• Oops...

Behind the Scenes: Test “99999999 + 1” Failed Because...

```c
int add( int x, int y )
{
    if (y == 1) // be fast
        return ++x;
    int r = x + y;
    if (r > 99999999)
        r = -1;
    return r;
}
```
A continuous function: we can conclude from a point to its environment.

Software is (in general) not continuous...

```c
int f(int x) {
    int r = 0;
    if (0 <= x && x < 128)
        r = fast_f(x); // only for [0,127]
    else if (128 < x && x < 1024)
        r = slow_f(x); // only for [128,1023]
    else
        r = really_slow_f(x); // only for [1024,...]
    return r;
}
```

- fast_f, slow_f, really_slow_f correct
- f(x) ≤ 0 for all x is required

• Range error: multiple “neighbouring” inputs trigger the error.
• Point error: an isolated input value triggers the error.
And Software Usually Has Many Inputs

- **Example**: Simple Pocket Calculator.

  With **one million** different test cases, 9,999,999,999,000 of the $10^{16}$ possible inputs remain **uncovered**.

  IOW: only 0.00000001% of the possible inputs covered, 99.99999999% not touched.

And if we restart the pocket calculator for each test, we do not know anything about problems with **sequences** of inputs...
The natural criterion “when everything has been done” does not apply for testing — at least not for testing pocket calculators.

So there need to be defined criteria to stop testing; project planning considers these criteria and experience with them.

Possible testing is done criteria:
- all (previously) specified test cases have been executed with negative result,
- testing effort 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,
- the average cost per error discovery exceeds a defined threshold $c$. 

---

![Diagram](attachment:image.png)
When To Stop Testing?

- The natural criterion “when everything has been done” does not apply for testing — at least not for testing pocket calculators.

- So there need to be defined criteria to stop testing; project planning considers these criteria and experience with them.

- Possible testing is done criteria:
  - all (previously) specified test cases have been executed with negative result,
  - testing effort 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,
  - the average cost per error discovery exceeds a defined threshold $c$,

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

- Of course: not all equally reasonable or compatible with each testing approach.

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 should be

- **representative**, i.e. represent a whole class of inputs,
- **error sensitive**, i.e. has high probability to detect an error,
- **of low redundancy**, i.e. it does not test what other test cases also test.

The wish for representative test cases is **particularly problematic**:

- Recall **point errors** (pocket calculator, fast/slow, ...).

In general, we do not know which inputs lie in an equivalence class wrt. errors.

Yet there is a large body on literature on how to construct representative test cases, **assuming** we know the equivalence classes.

“Acceptable” equivalence classes: Based on requirement specification, e.g.

- valid and invalid inputs (to check whether input validation works),
- different classes of inputs considered in the requirements, e.g. “buy water”, “buy soft-drink”, “buy tea” vs. “buy beverage”.

Lion and Error Hunting

“He/she who is hunting lions, should know how a lion looks like. He/she 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)*

**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, ...),
    - special cases of the programming language semantics,
    - complex implementations.
Where Do We Get The “Soll”-Values From?

• In an ideal world, all test cases are pairs \((\text{In}, \text{Soll})\) with proper “soll”-values. As, for example, defined by the formal requirements specification.

  Advantage: we can mechanically, objectively check for positive/negative.

• 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 an oracle.

• I prefer not to call decisions based on formally defined test cases “oracle”... ;-)

Glass-Box Testing: Coverage
Glass-Box Testing: Coverage

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

- Recall: An execution of test case $T = (In, S_{\text{call}})$ for software $S$ is a computation path
  \[
  (\sigma_0^T) \xrightarrow{a_1^T} (\sigma_1^T) \xrightarrow{a_2^T} \cdots \text{ where } \sigma_0^T \xrightarrow{a_1^T} \sigma_1^T \xrightarrow{a_2^T} \sigma_2^T \cdots = In.
  \]

- Let $S$ be a program (or model) consisting of statements $S_{\text{stmt}}$, conditions $S_{\text{cond}}$, and a control flow graph $(V, E)$ (as defined by the programming language).

- Assume that each state $\sigma$ gives information on statements, conditions, and control flow graph edges which were executed right before obtaining $\sigma$:
  
  \[
  \text{stm} : \Sigma \rightarrow 2^{S_{\text{stmt}}} , \quad \text{end} : \Sigma \rightarrow 2^{S_{\text{cond}}} , \quad \text{edg} : \Sigma \rightarrow 2^{E}
  \]

- $T$ achieves $p\%$ **statement coverage** if and only if $p = \frac{|\bigcup_{i \in \mathbb{N}} \text{stm}(\sigma_i)|}{|S_{\text{stmt}}|}, |S_{\text{stmt}}| \neq 0$.

- $T$ achieves $p\%$ **branch coverage** if and only if $p = \frac{|\bigcup_{i \in \mathbb{N}} \text{edg}(\sigma_i)|}{|E|}, |E| \neq 0$.

- Define: $p = 100$ for empty program.

- Statement/branch coverage canonically extends to test suite $T$.

### 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: $\{\text{true}\} f \{\text{true}\}$ (no abnormal termination)


**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)

<table>
<thead>
<tr>
<th>x, y, z</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>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>
</tbody>
</table>

---

**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)

<table>
<thead>
<tr>
<th>x, y, z</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>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>75</td>
<td>50</td>
<td>25</td>
</tr>
</tbody>
</table>

---
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: \( \{ \text{true} \} f \{ \text{true} \} \) (no abnormal termination)

<table>
<thead>
<tr>
<th>x, y, z</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>% stm</th>
<th>% cfg</th>
<th>i2/% term</th>
</tr>
</thead>
<tbody>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>501, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>0, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>

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: \( \{ \text{true} \} f \{ \text{true} \} \) (no abnormal termination)

<table>
<thead>
<tr>
<th>x, y, z</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>% stm</th>
<th>% cfg</th>
<th>i2/% term</th>
</tr>
</thead>
<tbody>
<tr>
<td>501, 11, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>75</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>501, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>0, 0, 0</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>100</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>

- 0, 51, 0
### 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: \( \{\text{true}\} f \{\text{true}\} \) (no abnormal termination)

### Term Coverage

- Consider the statement

  \[
  \text{if } (A ∧ (B ∨ (C ∧ D)) ∨ E) \text{ then } \ldots;
  \]

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

- **Branch coverage** is easy: use \((A = 0, \ldots, E = 0)\) and \((A = 0, \ldots, E = 1)\).

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

- **Term Coverage** (for an expression \(\text{expr}\)):
  - Let \(\beta : \{A_1, \ldots, A_n\} → B\) be a valuation of the terms.
  - Term \(A_i\) is \(b\)-**effective** in \(\beta\) for \(\text{expr}\) if and only if
    \[
    \beta(A_i) = b \text{ and } \llbracket\text{expr}\rrbracket(\beta[A_i/\text{true}]) \neq \llbracket\text{expr}\rrbracket(\beta[A_i/\text{false}]).
    \]
  - \(\Xi \subseteq (\{A_1, \ldots, A_n\} → B)\) achieves \(p\%\) term coverage if and only if
    \[p = \frac{|\{A_i \mid \exists \beta \in \Xi \cdot A_i \text{ is } b\text{-effective in } \beta\}|}{2n}.
    \]
Unreachable Code

```c
int f(int x, int y, int z)
{
    i1: if (x != x)
        z = y/0;
    i2: if (x = x ∨ z/0 = 27)
        z = z * 2;
    i3: return z;
}
```

- Statement \( s_1 \) is **never executed** \((x \neq x \iff \text{false})\), thus **100\% coverage not achievable**.
- Is statement \( s_1 \) an **error** anyway...?
- Term \( y/0 \) is never evaluated either (short-circuit evaluation)

Conclusions from Coverage Measures

- Assume, we are testing property \( \varphi = \{ p \} f \{ q \} \) (maybe just \( q = \text{true} \) with \( \downarrow \)),
- assume our test suite \( T \) achieved **100\% statement / branch / term coverage**.

What does this tell us about \( f \)? 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 \( T \), e.g. differing in variables’ valuation.)
  - “there is no unreachable statement”

- **100\% branch (term) coverage**:
  - “there is no single branch (term) which necessarily causes violations of \( \varphi \)”

    IOW: “for each condition (term), there is one computation path satisfying \( \varphi \) where the condition (term) evaluates to \text{true/false}”
  - “there is no unused condition (term)”

**Not more \((\rightarrow\text{exercises})\)!**

That’s **something**, but not as much as “100\%” may sound...
• (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 certain *coverage results*. (Next to development process requirements, reviews, unit testing, etc.)
• Currently, the standard moves towards accepting certain verification or static analysis tools to support (or even replace?) some testing obligations.

*Model-Based Testing*
• Does some software **implement** the given CFA model of the CoinValidator?

• **One approach**: check whether each state of the model has some reachable **corresponding configuration** in the software.
  
  \[ T_1 = \{ (C50, C50, C50); \{ \pi \mid \exists i < j < k < \ell \cdot \pi^i \sim \text{idle}, \pi^j \sim \text{have_c50}, \pi^k \sim \text{have_c100}, \pi^\ell \sim \text{have_c150} \} \} \]
  
  checks: can we reach 'idle', 'have_c50', 'have_c100', 'have_c150'?
Does some software implement the given CFA model of the CoinValidator?

One approach: check whether each state of the model has some reachable corresponding configuration in the software.

$T_1 = (C50, C50, C50; 
\{ \pi \mid \exists i < j < k < \ell \bullet \pi^0 \sim \text{idle}, \pi^i \sim \text{have}_c50, \pi^k \sim \text{have}_c100, \pi^\ell \sim \text{have}_c150 \})$

can we reach 'idle', 'have_c50', 'have_c100', 'have_c150'?

$T_2 = (C50, C50, C50; \dots) $ checks for 'have_e1'.

To check for 'drink ready', more interaction is necessary.
• Does some software implement the given CFA model of the CoinValidator?

• **One approach**: check whether each state of the model has some reachable corresponding configuration in the software.

  - $T_1 = \{C50, C50, C50; \{\pi | \exists i < j < k < \ell : \pi^i \sim \text{idle}, \pi^j \sim \text{have c50}, \pi^k \sim \text{have c100}, \pi^\ell \sim \text{have c150}\}\}$
    checks: can we reach 'idle', 'have c50', 'have c100', 'have c150'?  

  - $T_2 = \{C50, C50, C50; \ldots\}$ checks for 'have e1'.

  - To check for 'drink ready', more interaction is necessary.

• **Or**: Check whether each edge of the model has corresponding behaviour in the software.

• **Advantage**: input sequences can automatically be generated from the model.
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 have it (possibly with some glue logic in the middle) interact with the software (or a model of it).

- **Test passed** (i.e., test unsuccessful) if and only if TBA state $q_6$ is reached.
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 have it (possibly with some glue logic in the middle) interact with the software (or a model of it).

- Test passed (i.e., test unsuccessful) if and only if TBA state $q_6$ is reached.

- 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.

Statistical Testing
Another Approach: Statistical Tests

One proposal to deal with the uncertainty of tests, and to avoid bias (people tend to choose expected inputs): classical statistical testing.

- Randomly choose and apply test cases $T_1, \ldots, T_n$,
  - 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 confidence interval.

Needs stochastical assumptions on error distribution and truly random test cases.
(Confidence interval may get large — reflecting the low information tests give.)
Another Approach: Statistical Tests

One proposal to deal with the uncertainty of tests, and to avoid bias (people tend to choose expected inputs): classical statistical testing.

- Randomly choose and apply test cases $T_1, \ldots, T_n$,
  - 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 confidence interval.

Needs stochastic assumptions on error distribution and truly random test cases. (Confidence interval may get large — reflecting the low information tests give.)

(Ludewig and Lichter, 2013) name the following objections against statistical testing:

• In particular for interactive software, the primary goal is often that the “typical user” does not experience failures. Statistical testing (in general) may also cover a lot of “untypical user behaviour”, unless user-models are used.
Another Approach: Statistical Tests

One proposal to deal with the uncertainty of tests, and to avoid bias (people tend to choose expected inputs): classical statistical testing.

- Randomly choose and apply test cases $T_1, \ldots, T_n$,
  - 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 confidence interval.

Needs stochastical assumptions on error distribution and truly random test cases. (Confidence interval may get large — reflecting the low information tests give.)

(Ludewig and Lichter, 2013) name the following objections against statistical testing:

- In particular for interactive software, the primary goal is often that the “typical user” does not experience failures. Statistical testing (in general) may also cover a lot of "untypical user behaviour", unless user-models are used.
- Statistical testing needs a method to compute “soll”-values for the randomly chosen inputs; that is easy for "does not crash" but can be difficult in general.
- There is a high risk for not finding point or small-range errors which do live in their "natural habitat" as expected by testers.

Findings in the literature can at best be called inconclusive.
One Approach: Black-Box-Testing of Filter-like Software

- A low profile approach† when a formal (requirements) specification is not available, not even “agile-style” in form of test cases
- whenever∗ a feature∗∗ is considered finished,
  (i) make up inputs for (at least one) test case,
  (ii) create script which runs the program on these inputs,
  (iii) carefully examine the outputs for whether they are acceptable,
  (iv) if no: repair,
  (v) if yes: define the observed output as “soll”,
  (vi) extend script to compare ist/soll and add to test suite.

†: best for pipe/filter style software, where comparing output with “soll” is trivial.
*: if test case creation is postponed too long, chances are high that there will not be any test cases at all. Experience: “too long” is very short.
∗∗: error handling is also a feature.

Discussion

Advantages of testing (in particular over inspection):
**Discussion**

**Advantages** of testing (in particular over inspection):

- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and **objective**
  (if the start configuration is reproducible and the test environment deterministic).
Discussion

Advantages of testing (in particular over inspection):

- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and objective
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be re-used: properly prepared and documented tests can be re-executed with
  low effort, in particular fully automatic tests; important in maintenance.
- The test environment is (implicitly) subject of testing;
  errors in additional components and tools may show up.
Discussion

Advantages of testing (in particular over inspection):

- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and objective
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be re-used: properly prepared and documented tests can be re-executed with low effort, in particular fully automatic tests; important in maintenance.
- The test environment is (implicitly) subject of testing; errors in additional components and tools may show up.
- System behaviour (efficiency, usability) becomes visible, even if not explicitly subject of a test.

Disadvantages:
Discussion

Advantages of testing (in particular over inspection):
- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and objective
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be re-used: properly prepared and documented tests can be re-executed with low effort, in particular fully automatic tests; important in maintenance.
- The test environment is (implicitly) subject of testing;
  errors in additional components and tools may show up.
- System behaviour (efficiency, usability) becomes visible, even if not explicitly subject of a test.

Disadvantages:
- A proof of correctness is practically impossible, tests are seldomly exhaustive.

Discussion

Advantages of testing (in particular over inspection):
- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and objective
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be re-used: properly prepared and documented tests can be re-executed with low effort, in particular fully automatic tests; important in maintenance.
- The test environment is (implicitly) subject of testing;
  errors in additional components and tools may show up.
- System behaviour (efficiency, usability) becomes visible, even if not explicitly subject of a test.

Disadvantages:
- A proof of correctness is practically impossible, tests are seldomly exhaustive.
- It can be extremely hard to provoke environment conditions like interrupts or critical timings
  ("two buttons pressed at the same time").
Discussion

Advantages of testing (in particular over inspection):
- Testing is a “natural” checking procedure; “everybody can test”.
- The systematic test is reproducible and \textit{objective}
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be \textit{re-used}: properly prepared and documented tests can be re-executed with low effort, in particular fully automatic tests; important in maintenance.
- The test \textit{environment} is (implicitly) subject of testing;
  errors in additional components and tools may show up.
- System behaviour (efficiency, usability) becomes \textit{visible}, even if not explicitly subject of a test.

Disadvantages:
- A proof of correctness is practically \textit{impossible}, tests are seldomly \textit{exhaustive}.
- It can be \textit{extremely hard} to provoke environment conditions like interrupts or critical timings (“two buttons pressed at the same time”),
- Other properties of the implementation (like readability, maintainability)
  are not subject of the tests (but, e.g., of reviews),
- Tests tend to focus only on the code, other \textit{artefacts} (documentation, etc.) are hard to test.
  (Some say, developers tend to focus (too much) on coding, anyway.)

Recall: some agile methods turn this into a feature: there’s only requirements, tests, and code.
**Discussion**

**Advantages** of testing (in particular over inspection):

- Testing is a "natural" checking procedure; "everybody can test".
- The systematic test is reproducible and **objective**
  (if the start configuration is reproducible and the test environment deterministic).
- Invested effort can be **re-used**: properly prepared and documented tests can be re-executed with low effort, in particular fully automatic tests; important in maintenance.
- The **test environment** is (implicitly) subject of testing; errors in additional components and tools may show up.
- System behaviour (efficiency, usability) becomes **visible**, even if not explicitly subject of a test.

**Disadvantages**:

- A proof of correctness is practically **impossible**, tests are seldomly **exhaustive**.
- It can be **extremely hard** to provoke environment conditions like interrupts or critical timings ("two buttons pressed at the same time"),
- Other **properties** of the implementation (like readability, maintainability)
  are not subject of the tests (but, e.g., of reviews),
- Tests tend to focus only on the code, **other artefacts** (documentation, etc.) are hard to test.
  (Some say, developers tend to focus (too much) on coding, anyway.)
  Recall: some agile methods turn this into a feature: there’s only requirements, tests, and code.
- Positive tests show the **presence** of errors, but not their cause;
  the positive result may be false, caused by flawed test gear.

---

**Run-Time Verification**
• If we have an implementation for checking whether an output is correct wrt. a given input (according to requirements),
• we can just embed this implementation into the actual software, and
• thereby check satisfaction of the requirement during each run.
• → run-time verification.

### Run-Time Verification

```c
int main() {
    while (true) {
        int x = read_number();
        int y = read_number();
        int sum = add(x, y);
        display(sum);
    }
}
```

```c
void verify_sum(int x, int y, int sum) {
    if (sum != (x+y)
        || (x + y > 99999999
            && (sum < 0)))
    {
        fprintf(stderr, "verify_sum: error\n");
        abort();
    }
}
```

• If we have an implementation for checking whether an output is correct wrt. a given input (according to requirements),
• we can just embed this implementation into the actual software, and
• thereby check satisfaction of the requirement during each run.
• → run-time verification.
Run-Time Verification

```c
int main() {
    while (true) {
        int x = read_number();
        int y = read_number();
        int sum = add(x, y);
        verify_sum(x, y, sum);
        display(sum);
    }
}
```

```c
void verify_sum(int x, int y, int sum)
{
    if (sum != (x+y) || (x + y > 99999999 && (sum < 0)))
    {
        printf(stderr, "verify_sum: error\n");
        abort();
    }
}
```

- If we have an implementation for checking whether an output is correct wrt. a given input (according to requirements),
- we can just embed this implementation into the actual software, and
- thereby check satisfaction of the requirement during each run.
- → run-time verification.

Simplest Case: Assertions

- Maybe the simplest instance of runtime verification: Assertions.
- Available in standard libraries of many programming languages, e.g. C:
Maybe the simplest instance of runtime verification: **Assertions.**

Available in standard libraries of many programming languages, e.g. C:

```
#include <assert.h>

void assert(scalar expression);
```

**DESCRIPTION**

The purpose of this macro is to help the programmer find bugs in his program. The message “assertion failed in file foo.c, function do_bar(), line 1287” is of no help at all to a user.

- **Assertions at work:**

```c
int square(int x)
{
    assert(x < sqrt(x));
    return x * x;
}
```

```c
void f(...)
{
    assert(p);
    ...
    assert(q);
}
```
More Complex Case: LSC Observer

ChoicePanel:

```c
st : { idle, wsel, ssel, tsel, reqs, half };

take_event( E : { TAU, WATER, SOFT, TEA, ... } ) { 
  bool stable = 1;
  switch (st) { 
    case idle : 
      switch (E) { 
        case WATER : { 
          if (water_enabled) { 
            st := wsel; stable := 0; 
          } 
        } 
        case SOFT : 
          ... 
        case TAU : 
          ... 
        case reqs : 
          ... 
        case half : 
          ... 
      } 
  } 
}
```
More Complex Case: LSC Observer

ChoicePanel:

```java
st : { idle, wsel, ssel, tsel, reqs, halfIdle }

take(event( E : { TAU, WATER, SOFT, TEA, ... } ) )
{
   bool stable = 1;
   switch (st)
   {
      case idle :
         switch (E)
         {
            case WATER :
               if (water
                   enabled)
               {
                  st := wsel; stable := 0;
               }
            ...
            case TAU :
               send(DWATER()); st := reqs;
            ...
         } }
```
More Complex Case: LSC Observer

ChoicePanel:

```java
st : { idle, use1, use2, tool, req1, half };
task_wood( E : { TW, WATER, SOFT, TEA, ... }) {
    bool stable = 1;
    switch (E) {
        case idle:
            switch (E) {
                case WATER :
                    if (water_enabled) { st = use1; stable = 0; } 
                    case SOFT: ...
                ...
            }
        case use1:
            switch (E) {
                case WATER:
                    st := wsel; stable := 0;
                ...
            }
        ...
    }
}
```
Run-Time Verification: Discussion

- **Experience:**
  During development, **assertions** for pre/post conditions and intermediate invariants are an extremely powerful tool with **very good gain/effort ratio** (low effort, high gain).

  - Effectively work as safe-guard against unexpected use of functions and regression, e.g. during later maintenance or efficiency improvement.
  - Can serve as **formal** (support of) documentation: “Dear reader, at this point in the program, I expect this condition to hold, because…”.

- **Usually:**
  Development version **with** (cf. `assert(3)`) / release version **without** run-time verification.

  If run-time verification enabled in release version,
  - software should terminate as gracefully as possible (e.g. try to save data);
  - save information from assertion failure if possible.
Run-Time Verification: Discussion

- **Experience:**
  During development, **assertions** for pre/post conditions and intermediate invariants are an extremely powerful tool with **very good gain/effort ratio** (low effort, high gain).
  - Effectively work as safe-guard against unexpected use of functions and regression, e.g. during later maintenance or efficiency improvement.
  - Can serve as **formal** (support of) documentation: 
    "Dear reader, at this point in the program, I expect this condition to hold, because...".

- **Usually:**
  Development version **with** (cf. `assert(3)`) / release version **without** run-time verification.
  If run-time verification enabled in release version,
  - software should terminate as gracefully as possible (e.g. try to save data),
  - save information from assertion failure if possible.
  - Run-time verification can be arbitrarily complicated and complex, e.g., construction of observers for LSCs or temporal logic, e.g., expensive checking of data, etc.

- **Drawback:** development and release software have different computation paths — with bad luck, the software only behaves well because of the run-time verification code...
Recall: Three Basic Directions

Review

Testing

Formal Verification

prove $S \models \mathcal{S}$, conclude $[S] \in [\mathcal{S}]$
Reviews

- **Review item**: can be every closed, human-readable part of software (document, module, test data, installation manual, etc.)
  
  **Social aspect**: it is an *artefact* which is examined, not the *human* (who created it).

- **Input to Review Session**:
  
  - the *review item*, and *reference documents* which enable an assessment (requirements specification, guidelines (e.g. coding conventions), catalogue of questions (“all variables initialised?”), etc.)
Reviews

- **Review item**: can be every closed, human-readable part of software (document, module, test data, installation manual, etc.)
  
  **Social aspect**: it is an *artefact* which is examined, not the *human* (who created it).

- **Input to Review Session**:
  
  - the *review item*, and *reference documents* which enable an assessment (requirements specification, guidelines (e.g. coding conventions), catalogue of questions (“all variables initialised?”), etc.)

- **Roles**:
  
  **Moderator**: leads session, responsible for properly conducted procedure.
  
  **Author**: (representative of the) creator(s) of the artefact under review; is present to listen to the discussions, can answer questions; does not speak up if not asked.
  
  **Reviewer(s)**: person who is able to judge the artefact under review; maybe different reviewers for different aspects (programming, tool usage, etc.), at best experienced in detecting inconsistencies or incompleteness.
  
  **Transcript Writer**: keeps minutes of review session, can be assumed by author.

  The **review team** consists of everybody but the author(s).

---

**Review Procedure**

1. **Planning**
2. **Analysis**
3. **Preparation (2w)**
   - **Review Session (2h)**
   - **“3rd hour” (1h)**
4. **Postparation (2w)**
5. **Initiation**
   - review organisation under guidance of moderator
6. **Approval of review item**
**Review Procedure**

- review triggered, e.g., by submission to revision control system: moderator invites (include review item in invitation), state review missions,
- preparation: reviewers investigate review item,
- review session: reviewers report, evaluate and document issues; solve open questions,
- “3rd hour”: time for informal chat, reviewers may state proposals for solutions or improvements,
- postparation, rework: responsibility of author(s),
- reviewers re-assess reworked review item (until approval).
- planning: reviews need time in project plan; analysis: improve development and review process.

**Review Rules** *(Ludewig and Lichter, 2013)*

(i) moderator organises, invites to, conducts review,

(ii) the review session is **limited to 2 hours** — if needed: more sessions

(iii) moderator may terminate review if conduction not possible (inputs, preparation, or people missing),

(iv) the **review item** is under review, not the author(s),
    reviewers choose their wording accordingly,
    authors neither defend themselves nor the review item,

(v) roles are **not mixed up**, the moderator does not act as reviewer,

(vi) **style** issues (outside fixed conventions) are not discussed,

(vii) the review team is **not** supposed to **develop solutions**, issues are **not** noted in form of tasks for the author(s),

(viii) each **reviewer** gets the opportunity to present her/his findings appropriately,

(ix) reviewers need to reach **consensus** on issues, consensus is noted down,

(x) issues are classified as: **critical** (review unusable for purpose), **major** (usability severely affected), **minor** (usability hardly affected), **good** (no problem).

(xi) review team declares: accept **without changes**, accept **with changes**, do not accept.

(xii) **protocol** is signed by all participants.
Weaker and Stronger Variants

- Review

  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more faults found.
Weaker and Stronger Variants

- **Structured Walkthrough**
  - simple variant of review: developer moderates walkthrough-session, presents artefact, reviewer pose (prepared or spontaneous) questions, issues are noted down,
  - variants: with or without preparation (do reviewers see the artefact before the session?)
  - less effort, less effective.

  **disadvantages**: unclear responsibilities; “salesman”-author may trick reviewers.

- **Review**

- **Design and Code Inspection** *(Fagan, 1976, 1986)*
  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more faults found.

- **Comment** *(‘Stellungnahme’)*
  - colleague(s) of developer read artefacts,
  - developer considers feedback,

  **advantage**: low organisational effort; **disadvantages**: choice of colleagues may be biased; no protocol; consideration of comments at discretion of developer.

- **Structured Walkthrough**
  - simple variant of review: developer moderates walkthrough-session, presents artefact, reviewer pose (prepared or spontaneous) questions, issues are noted down,
  - variants: with or without preparation (do reviewers see the artefact before the session?)
  - less effort, less effective.

  **disadvantages**: unclear responsibilities; “salesman”-author may trick reviewers.

- **Review**

- **Design and Code Inspection** *(Fagan, 1976, 1986)*
  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more faults found.
Weaker and Stronger Variants

- **Careful Reading** (‘Durchsicht’)
  - done by developer,
  - recommendation: “away from screen” (use print-out or different device and situation)

- **Comment** (‘Stellungnahme’)
  - colleague(s) of developer read artefacts,
  - developer considers feedback,
  - **advantage**: low organisational effort; **disadvantages**: choice of colleagues may be biased; no protocol; consideration of comments at discretion of developer.

- **Structured Walkthrough**
  - simple variant of review: developer moderates walkthrough-session, presents artefact, reviewer pose (prepared or spontaneous) questions, issues are noted down,
  - variants: with or without preparation (do reviewers see the artefact before the session?)
  - less effort, less effective.
  - **disadvantages**: unclear responsibilities; “salesman”-author may trick reviewers.

- **Review**

- **Design and Code Inspection** (Fagan, 1976, 1986)
  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more faults found.
## Concluding Discussion

### Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove &quot;can run&quot;</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Review</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Strengths:**
- can be fully automatic (yet not easy for GUI programs);
- negative test proves "program not completely broken", "can run" (or positive scenarios);
- final product is examined, thus toolchain and platform considered;
- one can stop at any time and take partial results;
- few, simple test cases are usually easy to obtain;
- provides reproducible counter-examples (good starting point for repair).

**Weaknesses:**
- (in most cases) vastly incomplete, thus no proofs of correctness;
- creating test cases for complex functions (or complex conditions) can be difficult;
- maintaining many, complex test cases be challenging.
- executing many tests may need substantial time (but: can be run in parallel);
## Techniques Revisited

<table>
<thead>
<tr>
<th>Technique</th>
<th>Automatic</th>
<th>prove &quot;can run&quot;</th>
<th>Toolchain considered</th>
<th>Exhaustive</th>
<th>Prove correct</th>
<th>Partial results</th>
<th>Entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Runtime-Verification</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Review</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static Checking</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Strengths:
- fully automatic (once observers are in place);
- provides counter-example, not necessarily reproducible;
- (nearly) final product is examined, thus toolchain and platform considered;
- one can stop at any time and take partial results;
- assert-statements have a very good effort/effect ratio.

### Weaknesses:
- may negatively affect performance;
- code is changed, program may only run because of the observers;
- completeness depends on usage, may also be vastly incomplete, so no correctness proofs;
- constructing observers for complex properties may be difficult, one needs to learn how to construct observers.

## Techniques Revisited

<table>
<thead>
<tr>
<th>Technique</th>
<th>Automatic</th>
<th>prove &quot;can run&quot;</th>
<th>Toolchain considered</th>
<th>Exhaustive</th>
<th>Prove correct</th>
<th>Partial results</th>
<th>Entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Runtime-Verification</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Review</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static Checking</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Strengths:
- human readers can understand the code, may spot point errors;
- reported to be highly effective;
- one can stop at any time and take partial results;
- intermediate entry costs; good effort/effect ratio achievable.

### Weaknesses:
- no tool support;
- no results on actual execution, toolchain not reviewed;
- human readers may overlook errors; usually not aiming at proofs.
- does (in general) not provide counter-examples, developers may deny existence of error.
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove &quot;can run&quot;</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Review</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Static Checking</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Verification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Strengths:**
- there are (commercial), fully automatic tools (lint, Coverity, Polyspace, etc.);
- some tools are complete (relative to assumptions on language semantics, platform, etc.);
- can be faster than testing (at the price of many false positives);
- one can stop at any time and take partial results.

**Weaknesses:**
- no results on actual execution, toolchain not reviewed;
- can be very resource consuming (if few false positives wanted);
- many false positives can be very annoying to developers (if fast checks wanted);
- distinguish false from true positives can be challenging;
- configuring the tools (to limit false positives) can be challenging.

## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove &quot;can run&quot;</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Review</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Static Checking</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Verification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Strengths:**
- some tool support available (few commercial tools);
- complete (relative to assumptions on language semantics, platform, etc.);
- thus can provide correctness proofs;
- can prove correctness for multiple language semantics and platforms at a time;
- can be more efficient than other techniques.

**Weaknesses:**
- no results on actual execution, toolchain not reviewed;
- not many intermediate results: “half of a proof” may not allow any useful conclusions;
- entry cost high: significant training is useful to know how to deal with tool limitations;
- proving things is difficult: failing to find a proof does not allow any useful conclusion;
- false negatives (broken program “proved” correct) hard to detect.
Concluding Recommendations

• Not having at least one (systematic) test for each feature is (grossly?) negligent.
IOW: without at least one test for each feature, it is not software engineering.

General Guidelines: Do’s and Don’ts

• Do not use special examination versions for examination.
  (Test-harness, stubs, etc. can be used; yet may have errors which may undermine results.)
General Guidelines: Do’s and Don’ts

• Do not use special examination versions for examination.  
  (Test-harness, stubs, etc. can be used; yet may have errors which may undermine results.)

• Do not stop examination when first error is detected.  
  Clear: Examination can (and should) be aborted if the examined program is not executable at all.

• Do not modify the artefact under examination during examination.
  • changes/corrections during examination:  
    in the end unclear what exactly has been examined (“moving target”),  
    (results need to be uniquely traceable to one artefact version.)
  • fundamental flaws sometimes easier to detect  
    with a complete picture of unsuccessful/successful tests,
  • changes are particularly error-prone, should not happen “en passant” in examination,
  • fixing flaws during examination may cause them to go uncounted in the statistics  
    (which we need for all kinds of estimation),
  • roles developer and examiner are different anyway:  
    an examiner fixing flaws would violate the role assignment.
General Guidelines: Do’s and Don’ts

- Do not use special examination versions for examination. (Test-harness, stubs, etc. can be used; yet may have errors which may undermine results.)

- Do not stop examination when first error is detected. Clear: Examination can (and should) be aborted if the examined program is not executable at all.

- Do not modify the artefact under examination during examination.
  - Changes/corrections during examination:
    - In the end unclear what exactly has been examined (“moving target”), (results need to be uniquely traceable to one artefact version.)
  - Fundamental flaws sometimes easier to detect with a complete picture of unsuccessful/successful tests,
  - Changes are particularly error-prone, should not happen “en passant” in examination,
  - Fixing flaws during examination may cause them to go uncounted in the statistics (which we need for all kinds of estimation),
  - Roles developer and examiner are different anyway: an examiner fixing flaws would violate the role assignment.

- In particular: Do not switch (fine grained) between examination and debugging.

So All Hope is Lost...?

- Seems like computer systems more or less inevitably have errors.
So All Hope is Lost...?

- Seems like computer systems more or less inevitably have errors.
- So why does my (heavily computerised) Airbus fly at all?

Firstly, aerospace software maybe has the lowest error rate of all softwares due to very careful development, very thorough analysis (e.g. fault tree analysis), and strong regulatory obligations (“no proof of correctness, no take-off”).
So All Hope is Lost...?

- Seems like computer systems more or less inevitably have errors.

- So why does my (heavily computerised) Airbus fly at all?
  - Firstly, aerospace software maybe has the lowest error rate of all softwares due to very careful development, very thorough analysis (e.g. fault tree analysis), and strong regulatory obligations (“no proof of correctness, no take-off”).
  - Plus: classical engineering wisdom for high reliability: Redundancy.
    Highly-critical components may be present 3-times redundant, developed by 3 different teams, compiled by 3 different compilers, running on 3 different platforms, ...

- And why does my (heavily computerised) car, infusion pump, etc. not do harm?
So All Hope is Lost...?

- Seems like computer systems more or less inevitably have errors.

- So why does my (heavily computerised) Airbus fly at all?
  - Firstly, aerospace software maybe has the lowest error rate of all softwares due to very careful development, very thorough analysis (e.g. fault tree analysis), and strong regulatory obligations (“no proof of correctness, no take-off”).
  - Plus: classical engineering wisdom for high reliability: **Redundancy**.
    - Highly-critical components may be present 3-times redundant, developed by 3 different teams, compiled by 3 different compilers, running on 3 different platforms, ...

- And why does my (heavily computerised) car, infusion pump, etc. not do harm?
  - Again, classical engineering wisdom for high reliability: **Run-time monitoring**.

---

**Proposal: Dependability Cases** *(Jackson, 2009)*

- A **dependable** system is one you can **depend** on — that is, you can place your trust in it.
Proposal: Dependability Cases (Jackson, 2009)

• A dependable system is one you can depend on — that is, you can place your trust in it.

• Proposed Approach:
  • identify the critical requirements, and determine what level of confidence is needed.
    Most systems do also have non-critical requirements.
Proposal: Dependability Cases (Jackson, 2009)

- A **dependable** system is one you can **depend** on — that is, you can place your trust in it.

- **Proposed Approach:**
  - identify the **critical requirements**, and determine what **level of confidence** is needed.
  Most systems do also have **non-critical** requirements.
  - Construct a **dependability case:**
    - an argument, that the software, in concert with other components, establishes the critical properties.

- The case should be
  - **auditable**: can (easily) be evaluated by third-party certifier.
  - **complete**: no holes in the argument, any assumptions that are not justified should be noted (e.g. assumptions on compiler, on protocol obeyed by users, etc.)
  - **sound**: e.g. should not claim full correctness [...] based on nonexhaustive testing; should not make unwarranted assumptions on independence of component failures; etc.
• A **dependable** system is one you can **depend** on — that is, you can place your trust in it.

• **Proposed Approach:**
  • identify the **critical requirements**, and determine what **level of confidence** is needed.
    Most systems do also have **non-critical** requirements.
  • Construct a **dependability case:**
    • an argument, that the software, in concert with other components, establishes the critical properties.
  • The case should be
    • **auditable**: can (easily) be evaluated by third-party certifier.
    • **complete**: no holes in the argument, any assumptions that are not justified should be noted (e.g. assumptions on compiler, on protocol obeyed by users, etc.)
    • **sound**: e.g. should not claim full correctness [...] based on nonexhaustive testing; should not make unwarranted assumptions on independence of component failures; etc.

• IOW: “Developers [should] express the critical properties and make an explicit argument that the system satisfies them.”
  (As opposed to, e.g. requiring **term coverage** (which is usually not exhaustive), or requiring only coding conventions and procedure models, which may support, but do not **prove** dependability.)

**References**
References


