Introduction and Vocabulary
- Test case, test suite, test execution.
- Positive and negative outcomes.

Limits of Software Testing

Glass-Box Testing
- Statement-, branch-, term-coverage.

Other Approaches
- Model-based testing,

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

Review
Proof-System PD Cont’d
Proof-System PD (for sequential, deterministic programs)

Axiom 1: Skip-Statement

$$\{ p \} \text{skip} \{ p \}$$

Axiom 2: Assignment

$$\{ p[u := t] \} \ u := t \ \{ p \}$$

Rule 3: Sequential Composition

$$\{ p \} \ S_1 \ \{ r \} , \ \{ r \} \ S_2 \ \{ q \} \ \Rightarrow \ \{ p \} \ S_1 ; \ S_2 \ \{ q \}$$

Rule 4: Conditional Statement

$$\{ p \land B \} \ S_1 \ \{ q \} , \ \{ p \land \neg B \} \ S_2 \ \{ q \} , \ \{ p \} \ \text{if} \ B \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{fi} \ \{ q \}$$

Rule 5: While-Loop

$$\{ p \land B \} \ S \ \{ p \} \ \Rightarrow \ \{ p \} \ \text{while} \ B \ \text{do} \ S \ \text{od} \ \{ p \land \neg B \}$$

Rule 6: Consequence

$$p \Rightarrow p_1 , \ \{ p_1 \} \ S \ \{ q_1 \} , \ q_1 \Rightarrow q \ \Rightarrow \ \{ p \} \ S \ \{ q \}$$

Theorem. PD is correct ("sound") and (relative) complete for partial correctness of deterministic programs, i.e. $$\vdash_{PD} \{ p \} \ S \ \{ q \}$$ if and only if $$\models \{ p \} \ S \ \{ q \}.$$
Example Proof

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

(The first (textually represented) program that has been formally verified (Hoare, 1969).

We can prove \( \models \{ x \geq 0 \land y \geq 0 \} \ \text{DIV} \{ a \cdot y + b = x \land b < y \} \)

by showing \( \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ \text{DIV} \{ a \cdot y + b = x \land b < y \} \), i.e., derivability in PD:

\[ (1) \]

\[ P \rightarrow P, \quad \{ P \} \ \text{while} \ B^D \ \text{do} \ S_1^D \ \text{od} \ \{ P \land \neg (B^D) \}, \quad P \land \neg (B^D) \rightarrow q^D \]

\[ (2) \]

\[ \{ P \land (B^D) \} S_1^D \ \{ P \} \]

\[ (3) \]

\[ \{ P \} \ \text{while} \ B^D \ \text{do} \ S_1^D \ \text{od} \ \{ q^D \} \]

\[ \{ p^D \} S_0^D \ \{ P \}, \quad P \rightarrow P, \quad \{ P \} \ \text{while} \ B^D \ \text{do} \ S_1^D \ \text{od} \ \{ q^D \} \]

\[ \{ p^D \} S_0^D \ \{ P \}, \quad \text{while} \ B^D \ \text{do} \ S_1^D \ \text{od} \ \{ q^D \} \]

\[ (A1) \ \{ p \} \ \text{skip} \ \{ p \} \]

\[ (A2) \ \{ p[u := t] \} \ u := t \ \{ p \} \]

\[ (R3) \ \{ p \} S_1 \ \{ r \}, \ \{ r \} S_2 \ \{ q \} \]

\[ (R4) \ \{ p \land B \} S_1 \ \{ q \}, \ \{ p \land \neg B \} S_2 \ \{ q \} \]

\[ (R5) \ \{ p \} \ \text{while} \ B \ \text{do} \ S \ \text{od} \ \{ p \land \neg B \} \]

\[ \text{if} \ B \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{fi} \ \{ q \} \]

\[ \text{if} \ B \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{fi} \ \{ q \} \]

\[ (R6) \ \{ p \} S \ \{ q \} \]
Example Proof

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

(The first (textually represented) program that has been formally verified (Hoare, 1969).

We can prove \(\models \{x \geq 0 \land y \geq 0\} \text{DIV} \{a \cdot y + b = x \land b < y\}\)

by showing \(\vdash_{PD} \{x \geq 0 \land y \geq 0\} \text{DIV} \{a \cdot y + b = x \land b < y\},\) i.e., derivability in PD:

\[
\begin{align*}
\text{(1)} & \quad \{x \geq 0 \land y \geq 0\} a := 0; \ b := x \{P\}, \\
\text{(2)} & \quad P \rightarrow P, \quad \{P\} \text{while} b \geq y \ \text{do} \ b := b - y; \ a := a + 1 \{P\} \quad \text{(R5)} \\
\text{(3)} & \quad P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y \quad \text{(R6)} \\
\text{ (4)} & \quad \{x \geq 0 \land y \geq 0\} a := 0; \ b := x; \ \text{while} b \geq y \ \text{do} \ b := b - y; \ a := a + 1 \{a \cdot y + b = x \land b < y\}
\end{align*}
\]

\[\begin{array}{l}
\text{(A1)} \quad \{p\} \text{skip} \{p\} \\
\text{(A2)} \quad \{p[u := t]\} \ u := t \{p\} \\
\text{(R3)} \quad \{p\} S_1 \{r\}, \ \{r\} S_2 \{q\} \\
\{p\} S_1; \ S_2 \{q\} \\
\text{(R4)} \quad \{p \land B\} S_1 \{q\}, \ \{p \land \neg B\} S_2 \{q\} \\
\{p\} \text{if} B \text{ then} S_1 \text{ else} S_2 \text{ fi} \{q\} \\
\text{(R5)} \quad \{p \land B\} S \{p\} \\
\{p\} \text{while} B \ \text{do} \ S \ \text{od} \ \{p \land \neg B\} \\
\text{(R6)} \quad p \rightarrow p_1, \ \{p_1\} S \{q_1\}, \ q_1 \rightarrow q \\
\{p\} S \{q\}
\end{array} \]
In the following, we show

(1) \[ P_D \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \}, \]

(2) \[ P_D \{ P \land b \geq y \} \ b := b - y; \ a := a + 1 \{ P \}, \]

(3) \[ P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y. \]

As loop invariant, we choose (creative act!):

\[ P \equiv a \cdot y + b = x \land b \geq 0 \]
(1) claims:
\[ \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} \]
where \( P \equiv a \cdot y + b = x \land b \geq 0 \).
Proof of (1)

- (1) claims:
  \[ \vdash_{PD} (x \geq 0 \land y \geq 0) \ a := 0; \ b := x \ {P} \]
  where \( P \equiv a \cdot y + b = x \land b \geq 0. \)

- \[ \vdash_{PD} (0 \cdot y + x = x \land x \geq 0) \ a := 0 \ {a \cdot y + x = x \land x \geq 0} \]

\[ (A1) \ {p} \ skip \ {p} \]
\[ (R4) \ \{p \land B\} \ S_1 \ {q}, \ \{p \land \neg B\} \ S_2 \ {q} \]
\[ \{p\} \ \text{if} \ B \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{fi} \ {q} \]
\[ (A2) \ \{p[u := t]\} \ u := t \ {p} \]
\[ (R5) \ \{p \land B\} \ S \ {p} \]
\[ \{p\} \ \text{while} \ B \ \text{do} \ S \ \text{od} \ {p \land \neg B} \]
\[ (R3) \ \{p\} \ S_1 \ {r}, \ \{r\} \ S_2 \ {q} \]
\[ \{p\} \ S_1; \ S_2 \ {q} \]
\[ (R6) \ p \rightarrow p_1, \ \{p_1\} \ S \ {q_1}, \ q_1 \rightarrow q \]
\[ \{p\} \ S \ {q} \]
Proof of (1)

- (1) claims:
  \[ \vdash_P \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \ \{ P \} \]
  where \( P \equiv a \cdot y + b = x \land b \geq 0 \).

- \[ \vdash_P \{ 0 \cdot y + x = x \land x \geq 0 \} \ a := 0 \ \{ a \cdot y + x = x \land x \geq 0 \} \] by (A2),

- \[ \vdash_P \{ a \cdot y + x = x \land x \geq 0 \} \ b := x \ \{ a \cdot y + b = x \land b \geq 0 \} \] by (A2),

\[ \equiv P \]
Proof of (1)

(1) claims:
\[\vdash_{PD} \{ x \geq 0 \land y \geq 0 \} a := 0; \ b := x \{ P \} \]
where \( P \equiv a \cdot y + b = x \land b \geq 0. \)

- \( \vdash_{PD} \{ 0 \cdot y + x = x \land x \geq 0 \} a := 0 \{ a \cdot y + x = x \land x \geq 0 \} \) by (A2),
- \( \vdash_{PD} \{ a \cdot y + x = x \land x \geq 0 \} b := x \{ a \cdot y + b = x \land b \geq 0 \} \) by (A2),

thus, \( \vdash_{PD} \{ 0 \cdot y + x = x \land x \geq 0 \} a := 0; \ b := x \{ P \} \) by (R3),

using \( x \geq 0 \land y \geq 0 \rightarrow 0 \cdot y + x = x \land x \geq 0 \) and \( P \rightarrow P \), we obtain
\[\vdash_{PD} \{ x \geq 0 \land y \geq 0 \} a := 0; \ b := x \{ P \}\]
by (R6).
The rule ‘Assignment’ uses (syntactical) substitution: \( \{p[u := t]\} u := t \{p\} \)
(In formula \(p\), replace all (free) occurrences of (program or logical) variable \(u\) by term \(t\).)

Defined as usual, only indexed and bound variables need to be treated specially:

\[
\alpha \geq \chi \ [x := u+3] \quad \mapsto \quad \alpha \geq u+3
\]

\[
(\alpha \geq \chi \land \forall x \cdot b \geq x) [x := u+3] \quad \mapsto \quad ?
\]

\[
\alpha \geq \chi \land \forall z \cdot b \geq z \quad \mapsto \quad \alpha \geq u+3 \land \forall z \cdot b \geq z
\]
The rule ‘Assignment’ uses (syntactical) substitution: \( \{ p[u := t] \} u := t \{ p \} \)
(In formula \( p \), replace all (free) occurrences of (program or logical) variable \( u \) by term \( t \).)

Defined as usual, only \textbf{indexed} and \textbf{bound} variables need to be treated specially:

**Expressions:**
- plain variable \( x \): \( x[u := t] \equiv \begin{cases} t, & \text{if } x = u \\ x, & \text{otherwise} \end{cases} \)
- constant \( c \): \( c[u := t] \equiv c \).
- constant \( op \), terms \( s_i \): \( op(s_1, \ldots, s_n)[u := t] \equiv op(s_1[u := t], \ldots, s_n[u := t]) \).
- conditional expression: \( (B ? s_1 : s_2)[u := t] \equiv (B[u := t] ? s_1[u := t] : s_2[u := t]) \)
- indexed variable, \( u \) plain or \( u \equiv b[t_1, \ldots, t_m] \) and \( a \neq b \): \( (a[s_1, \ldots, s_n])[u := t] \equiv a[s_1[u := t], \ldots, s_n[u := t]] \)
- indexed variable, \( u \equiv a[t_1, \ldots, t_m] \): \( (a[s_1, \ldots, s_n])[u := t] \equiv (\bigwedge_{i=1}^{n} s_i[u := t] = t_i ? t : a[s_1[u := t], \ldots, s_n[u := t]]) \)

**Formulae:**
- boolean expression \( p \equiv s \): \( p[u := t] \equiv s[u := t] \)
- negation: \( (\neg q)[u := t] \equiv \neg(q[u := t]) \)
- conjunction etc.: \( (q \land r)[u := t] \equiv q[u := t] \land r[u := t] \)
- quantifier: \( (\forall x : q)[u := t] \equiv \forall y : q[x := y][u := t] \) \( y \) fresh (not in \( q, t, u \), same type as \( x \)).
Example Proof Cont’d

In the following, we show

(1) $\vdash_{PD} \{x \geq 0 \land y \geq 0\} a := 0; b := x \{P\},$

(2) $\vdash_{PD} \{P \land b \geq y\} b := b - y; a := a + 1 \{P\},$

(3) $\models P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y.$

As loop invariant, we choose (creative act!):

$$P \equiv a \cdot y + b = x \land b \geq 0$$
Proof of (2)

(2) claims:

\[ \vdash_{PD} \{ P \land b \geq y \} b := b - y; \ a := a + 1 \{ P \} \]

where \( P \equiv a \cdot y + b = x \land b \geq 0 \).

\[ \vdash_{PD} \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0\} b := b - y \{(a + 1) \cdot y + b = x \land b \geq 0\} \]

by (A2),
Proof of (2)

- (2) claims:

\[ \vdash_{PD} \{ P \land b \geq y \} \ b := b - y; \ a := a + 1 \ \{ P \} \]

where \( P \equiv a \cdot y + b = x \land b \geq 0 \).

\[ \vdash_{PD} \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \ b := b - y \{(a + 1) \cdot y + b = x \land b \geq 0\} \]

by (A2),

\[ \vdash_{PD} \{(a + 1) \cdot y + b = x \land b \geq 0\} \ a := a + 1 \{(a \cdot y + b = x \land b \geq 0\} \equiv_P \]

by (A2),

\[ \vdash_{PD} \{p \land B\} S_1 \{q\}, \{p \land \neg B\} S_2 \{q\} \]

(R4)

\[ \{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\} \]

\[ \{p[u := t]\} \ u := t \{p\} \]

(A2)

\[ \{p\} \ S \{p\} \]

(R5)

\[ \{p \land B\} S \{p\} \]

\[ \{p \land \neg B\} \]

\[ \text{while } B \text{ do } S \text{ od } \{p \land \neg B\} \]

(R3)

\[ \{p\} \ S_1 \{r\}, \{r\} S_2 \{q\} \]

(R6)

\[ \{p\} S_1; \ S_2 \{q\} \]

\[ \{p\} \rightarrow \{p\_1\}, \{p\_1\} S \{q\_1\}, \ q \rightarrow q \]
Proof of (2)

- (2) claims:

\[ \vdash_{PD} \{ P \land b \geq y \} \quad b := b - y; \quad a := a + 1 \{ P \} \]

where \( P \equiv a \cdot y + b = x \land b \geq 0 \).

- \[ \vdash_{PD} \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \quad b := b - y; \quad (a + 1) \cdot y + b = x \land b \geq 0 \]

by (A2),

- \[ \vdash_{PD} \{(a + 1) \cdot y + b = x \land b \geq 0 \} \quad a := a + 1 \{ a \cdot y + b = x \land b \geq 0 \} \]

by (A2),

- \[ \vdash_{PD} \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \quad b := b - y; \quad a := a + 1 \{ P \} \]

by (R3),

- using \( P \land b \geq y \rightarrow (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \) and \( P \rightarrow P \) we obtain,

\[ \vdash_{PD} \{ P \land b \geq y \} \quad b := b - y; \quad a := a + 1 \{ P \} \]

by (R6).
Example Proof Cont’d

In the following, we show

(1) \( \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \}, \)

(2) \( \vdash_{PD} \{ P \land b \geq y \} \ b := b - y; \ a := a + 1 \{ P \}, \)

(3) \( \models P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y. \)

As loop invariant, we choose (creative act!):

\[
P = a \cdot y + b = x \land b \geq 0
\]
Proof of (3)

(3) claims

\[ \models P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y. \]

where \( P \equiv a \cdot y + b = x \land b \geq 0. \)

Proof: easy.
We have shown:

1. \( \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \; a := 0 ; \; b := x \{ P \} , \)
2. \( \vdash_{PD} \{ P \land b \geq y \} \; b := b - y ; \; a := a + 1 \{ P \} , \)
3. \( \models P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y . \)

and

\[
\begin{align*}
\{ x \geq 0 \land y \geq 0 \} & \vdash a := 0 ; \; b := x \{ P \} , & (1) \\
\vdash_{PD} \{ P \} \; \text{while} \; b \geq y \; \text{do} \; b := b - y ; \; a := a + 1 \od \{ P \land \neg (b \geq y) \} , & (2) \\
\vdash_{PD} \{ P \} \; \text{while} \; b \geq y \; \text{do} \; b := b - y ; \; a := a + 1 \od \{ a \cdot y + b = x \land b < y \} , & (3) \\
\{ x \geq 0 \land y \geq 0 \} & \vdash a := 0 ; \; b := x ; \; \text{while} \; b \geq y \; \text{do} \; b := b - y ; \; a := a + 1 \od \{ a \cdot y + b = x \land b < y \} , & (R3) \\
\vdash_{PD} \{ x \geq 0 \land y \geq 0 \} & \vdash a := 0 ; \; b := x ; \; \text{while} \; b \geq y \; \text{do} \; b := b - y ; \; a := a + 1 \od \{ a \cdot y + b = x \land b < y \} \equiv DIV . & (R6)
\end{align*}
\]

thus

\( \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \; a := 0 ; \; b := x ; \; \text{while} \; b \geq y \; \text{do} \; b := b - y ; \; a := a + 1 \od \{ a \cdot y + b = x \land b < y \} \equiv DIV . \)

and thus (since PD is sound) \( DIV \) is \textit{partially correct} wrt.

- **pre-condition**: \( x \geq 0 \land y \geq 0 \),
- **post-condition**: \( a \cdot y + b = x \land b < y \).

IOW: whenever \( DIV \) is called with \( x \) and \( y \) such that \( x \geq 0 \land y \geq 0 \), then (if \( DIV \) terminates) \( a \cdot y + b = x \land b < y \) will hold.
\[ P \equiv a \cdot y + b = x \land b \geq 0 \]

\[ \{ x \geq 0 \land y \geq 0 \} \]

\[ \{ 0 \cdot y + x = x \land x \geq 0 \} \]

\[ a := 0; \]
\[ \{ a \cdot y + x = x \land x \geq 0 \} \]

\[ b := x; \]
\[ \{ a \cdot y + b = x \land b \geq 0 \} \]
\[ \{ P \} \]

\[ \textbf{while } b \geq y \textbf{ do} \]
\[ \{ P \land b \geq y \} \]
\[ \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \]

\[ b := b - y; \]
\[ \{(a + 1) \cdot y + b = x \land b \geq 0 \} \]

\[ a := a + 1 \]
\[ \{ a \cdot y + b = x \land b \geq 0 \} \]
\[ \{ P \} \]

\[ \textbf{od} \]
\[ \{ P \land \neg(b \geq y) \} \]
\[ \{ a \cdot y + b = x \land b < y \} \]
Literature Recommendation

Apt · Olderog
Programmverifikation
Sequentielle, parallele und verteilte Programme
Springer-Lehrbuch

Verification of Sequential and Concurrent Programs
Krzysztof R. Apt
Frank S. de Boer
Ernst-Rüdiger Olderog
Springer Third Edition
Content

• Formal Program Verification
  • Proof System PD

• The Verifier for Concurrent C
  • Assertions, Modular Verification, VCC

• Runtime-Verification
  • Assertions, LSC-Observers

• Reviews
  • Roles and artefacts
  • Review procedure
  • Stronger and weaker variants

• Code QA Techniques Revisited
  • Test, Runtime-Verification, Review,
  • Static Checking, Formal Verification

• Do’s and Don’ts in Code QA

• Dependability
The Verifier for Concurrent C
The **Verifier for Concurrent C** (VCC) basically implements Hoare-style reasoning.

### Special syntax:
- `#include <vcc.h>`
- `_(requires \ p) — **pre-condition**, \ p is (basically) a C expression`
- `_(ensures \ q) — **post-condition**, \ q is (basically) a C expression`
- `_(invariant \ expr) — **loop invariant**, \ expr is (basically) a C expression`
- `_(assert \ p) — **intermediate invariant**, \ p is (basically) 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

```c
#include <vcc.h>

int a, b;

void div(int x, int y)
  _(requires x >= 0 && y >= 0)
  _(ensures a * y + b == x && b < y)
  _(writes &a)
  _(writes &b)
{
  a = 0;
  b = x;
  while (b >= y)
    _(invariant a * y + b == x && b >= 0)
    {
      b = b - y;
      a = a + 1;
    }
}
```

\[ \text{DIV} \equiv a := 0; \; b := x; \; \textbf{while} \; b \geq y \; \textbf{do} \; b := b - y; \; a := a + 1 \; \textbf{od} \]

\[ \{ x \geq 0 \land y \geq 0 \} \; \text{DIV} \; \{ x \geq 0 \land y \geq 0 \} \]
VCC Web-Interface

VCC

Example program DIV: http://rise4fun.com/Vcc/4Kqe
Interpretation of Results

- VCC result: “verification succeeded”
  - We can only conclude that the tool — under its interpretation of the C-standard, under its platform assumptions (32-bit), etc. — claims that there is a proof for $\models \{p\} \text{DIV} \{q\}$.
  - May be due to an error in the tool! (That’s a false negative then.) 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.
    - That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

- VCC result: “verification failed”
  - May be a false positive (wrt. the goal of finding errors).
    - 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 $q$.
  - → try to construct a (true) counter-example from the hints.
    - or: make loop-invariant(s) (or pre-condition $p$) stronger, and try again.
  - Other case: “timeout” etc. — completely inconclusive outcome.
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.

- 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.

- 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.
Modular Reasoning
We can add another rule for calls of functions $f : F$ (simplest case: only global variables):

\[
\begin{array}{c}
\{ p \} F \{ q \} \\
\{ p \} f() \{ q \}
\end{array}
\]

"If we have $\vdash \{ p \} F \{ q \}$ for the implementation of function $f$, then if $f$ is called in a state satisfying $p$, the state after return of $f$ will satisfy $q$.”

$p$ is called **pre-condition** and $q$ is called **post-condition** of $f$.

**Example:** if we have

- $\{ \text{true} \} \text{read\_number} \{ 0 \leq \text{result} < 10^8 \}$
- $\{ 0 \leq x \land 0 \leq y \} \text{add} \{ (\text{old}(x) + \text{old}(y) < 10^8 \land \text{result} = \text{old}(x) + \text{old}(y)) \lor \text{result} < 0 \}$
- $\{ \text{true} \} \text{display} \{ (0 \leq \text{old}(\text{sum}) < 10^8 \implies \text{"old(\text{sum})"}) \land (\text{old}(\text{sum}) < 0 \implies \text{"-E-"}) \}$

we may be able to prove our pocket calculator correct.
For **modular reasoning**, it’s often useful to refer in the post-condition to

- the **return value** as \( \text{result} \),
- the **values** of variable \( x \) at calling time as \( \text{old}(x) \).

Can be defined using **auxiliary variables**:

- Transform function

\[
T f() \{ \ldots ; \text{return expr}; \}
\]

(over variables \( V = \{v_1, \ldots, v_n\}; \text{where } \text{result, } v_i^{\text{old}} \not\in V \) into

\[
T f() \{
\begin{align*}
v_1^{\text{old}} & := v_1; \ldots; v_n^{\text{old}} := v_n; \\
\ldots; & \\
\text{result} & := \text{expr}; \\
\text{return } \text{result};
\end{align*}
\}
\]

over \( V' = V \cup \{v_i^{\text{old}} \mid v \in V\} \cup \{\text{result}\} \).

Then \( \text{old}(x) \) is just an abbreviation for \( x^{\text{old}} \).
Assertions
Assertions

- Extend the syntax of deterministic programs by
  \[ S := \cdots \mid \text{assert}(B) \]

- and the semantics by rule
  \[ \langle \text{assert}(B), \sigma \rangle \rightarrow \langle E, \sigma \rangle \] if \( \sigma \models B \).

(If the asserted boolean expression \( B \) does not hold in state \( \sigma \), the empty program is not reached; otherwise the assertion remains in the first component: abnormal program termination).

Extend PD by axiom:

\[ (A7) \ {p} \ \text{assert}(p) \ {p} \]

- That is, if \( p \) holds before the assertion, then we can continue with the derivation in PD. If \( p \) does not hold, we “get stuck” (and cannot complete the derivation).

- So we cannot derive \( \{true\} \ x := 0; \ \text{assert}(x = 27) \ {true} \) in PD.
- Formal Program Verification
  - Proof System PD

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
    - Static Checking, Formal Verification

- Do’s and Don’ts in Code QA

- Dependability
Run-Time Verification
A Very Useful Special Case: Assertions

- Maybe the simplest instance of runtime verification: Assertions.
- Available in standard libraries of many programming languages (C, C++, Java, ...).

- For example, the C standard library manual reads:

```c
#include <assert.h>

void assert(scalar expression);
```

- In C code, assert can be disabled in production code (-D NDEBUG).
- Use java -ea ... to enable assertion checking (disabled by default).
  (cf. https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html)
Assertions At Work

• The abstract $f$-example from run-time verification:
  (specification: $\{p\} f \{q\}$)

```c
void f(...) {
  assert(p);
  ...
  assert(q);
}
```

• Compute the width of a progress bar:

```c
int progress_bar_width(int progress, int window_left, int window_right)
{
  assert(window_left <= window_right);  /* pre-condition */
  ...
  /* treat special cases 0 and 100 */
  assert(0 < progress && progress < 100);  // extremal cases already treated
  ...
  assert(window_left <= r && r <= window_right);  /* post-condition */
  return r;
}
```
• Recall the **structure model** with Proto-OCL constraint from Exercise Sheet 4/2017.

• Assume, we add a method `set_key()` to class **TreeNode**:

```java
class TreeNode {
  private int key;
  TreeNode parent, leftChild, rightChild;
  
  public int get_key() { return key; }
  
  public void set_key( int new_key ) {
    key = new_key;
  }
}
```

• We can **check consistency** with the Proto-OCL constraint at runtime by using assertions:

```java
public void set_key( int new_key ) {
  assert( parent == null || parent.get_key() <= new_key );
  assert( leftChild == null || new_key <= leftChild.get_key() );
  assert( rightChild == null || new_key <= rightChild.get_key() );
  key = new_key;
}
```
Run-Time Verification: Idea

- Assume, there is a function \( f \) in software \( S \) with the following specification:
  - pre-condition: \( p \), post-condition: \( q \).

- Computation paths of \( S \) may look like this:

\[
\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{\mathit{call \, f}} \sigma_{n+1} \cdots \sigma_m \xrightarrow{\mathit{f \, returns}} \sigma_{m+1} \cdots
\]

- Assume there are functions \( \mathit{check}_p \) and \( \mathit{check}_q \), which check whether \( p \) and \( q \) hold at the current program state, and which do not modify the program state (except for program counter).

- **Idea**: create software \( S' \) by
  1. extending \( S \) by implementations of \( \mathit{check}_p \) and \( \mathit{check}_q \),
  2. call \( \mathit{check}_p \) right after entering \( f \),
  3. call \( \mathit{check}_q \) right before returning from \( f \).

- For \( S' \), obtain computation paths like:

\[
\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{\mathit{call \, f}} \sigma_{n+1} \xrightarrow{\mathit{check}_p} \sigma'_{n+1} \cdots \sigma_m \xrightarrow{\mathit{check}_q} \sigma'_m \xrightarrow{\mathit{f \, returns}} \sigma_{m+1} \cdots
\]

- If \( \mathit{check}_p \) and \( \mathit{check}_q \) notify us of violations of \( p \) or \( q \), then we are notified of \( f \) violating its specification when running \( S' \) (\( = \) at run-time).
Run-Time Verification: Example

```c
int x, y, sum;

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

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();
    }
}
```

```
12345678
+ 27
7 8 9 0
4 5 6 +
1 2 3 =
```
More Complex Run-Time Verification: LSC Observers

ChoicePanel:

```
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 wsel:
            switch (E) {
                case TAU :
                    send_DWATER(); st := reqs;
                    hey_observer_I_just_sent_DWATER();
                    break;
            }
        }
    }
```
Experience. **Assertions** for pre/post conditions and intermediate invariants are an **extremely powerful** tool with a **very attractive gain/effort ratio** (low effort, high gain).

- Assertions effectively work as **safe-guard** against
  - unexpected use of functions and
  - regression,
  e.g. during later maintenance or efficiency improvement.

- Assertions can serve as **formal (support of) documentation**:
  - `assert(expr);`
  means
  - “Dear reader, at this point in the program, I expect condition `expr` to hold.”

Be good to your readers: **add a comment** that explains the **why**...
Content

- Formal Program Verification
  - Proof System PD

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Do’s and Don’ts in Code QA

- Dependability
Review
Recall: Three Basic Directions

- All computation paths satisfying the specification
- Expected outcomes \( S_{\text{oll}} \)
- \((\Sigma \times A)^\omega\) defines

Execution of \((I_{\text{in}}, S_{\text{oll}})\)

Reviewer

Input \[\to\] Output

Testing

Review

Formal Verification

\[
\text{prove } S \models \mathcal{I}, \text{ conclude } [S] \in [\mathcal{I}]
\]
• **Input to Review Session:**
  - **Review item**: can be every closed, human-readable part of software (documentation, module, test data, installation manual, etc.)
  - **Social aspect**: it is an artefact which is examined, not the human (who created it).
  - **Reference documents**: need to 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 Over Time

planning: reviews need **time** in the project plan.

preparation: reviewers **investigate** review item.

review session: reviewers **report**, evaluate, and document issues; **resolve** open questions.

postparation: **rework** review item; responsibility of the author(s).

- Reviewers **re-assess** reworked review item (until **approval** is declared).

a review is **triggered**, e.g., by a submission to the revision control system: the moderator **invites** (include review item in invitation), and states **review missions**.

analysis: improve development and review process.

**Review organisation under guidance of moderator**

**Approval of review item**

**Initiation**

**“3rd hour”**: time for informal chat, reviewers may state **proposals** for solutions or improvements.
(i) The moderator organises the review, issues invitations, supervises the review session.

(ii) The moderator may terminate the review if conduction is not possible, e.g., due to inputs, preparation, or people missing.

(iii) The review session is limited to 2 hours. If needed: organise more sessions.

(iv) The review item is under review, not the author(s). Reviewers choose their words accordingly. Authors neither defend themselves nor the review item.

(v) Roles are not mixed up, e.g., the moderator does not act as reviewer. (Exception: author may write transcript.)

(vi) Style issues (outside fixed conventions) are not discussed.

(vii) The review team is not supposed to develop solutions. Issues are not noted down 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) The review team declares:
  - accept without changes,
  - accept with changes,
  - do not accept.

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

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

- **Review**
  - **Structured Walkthrough**
    - simple variant of review:
      - *developer* moderates walkthrough-session,
      - *developer* presents artefact(s),
      - *reviewer* poses (prepared or spontaneous) questions,
      - issues are noted down,
    - **Variation point**: do reviewers see the artefact before the session?
    - **less effort, less effective**.

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

- **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.

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

- Formal Program Verification
  - Proof System PD

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Do’s and Don’ts in Code QA

- Dependability
Code Quality Assurance Techniques Revisited
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</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>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**;
- **maintenance** of many, complex test cases be **challenging**.
- executing many tests may need **substantial time** (but: can sometimes be run in parallel);
# Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</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;
- (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:**

- counter-examples not necessarily reproducible;
- 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></th>
<th>automatic</th>
<th>prove “can run”</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-Veri-</td>
<td>✔</td>
<td>(✔)</td>
<td>✔</td>
<td>(✘)</td>
<td>✘</td>
<td>✔</td>
<td>(✔)</td>
</tr>
<tr>
<td>fication</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 Check-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ing</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:**
- 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 “can run”</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**;
- 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), e.g., code may need to be “designed for static analysis”.
- 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 “can run”</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:**
- 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 challenging: failing to find a proof does not allow any useful conclusion;
- **false negatives** (broken program “proved” correct) hard to detect.
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</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>
Some Final, General Guidelines
Do’s and Don’ts in Code Quality Assurance

Avoid using special examination versions for examination. (Test-harness, stubs, etc. may have errors which may cause false positives and (!) negatives.)

Avoid to stop examination when the first error is detected.

Clear: Examination should be aborted if the examined program is not executable at all.

Do not modify the artefact under examination during examination.

• otherwise, it is unclear what exactly has been examined (“moving target”), (examination results need to be uniquely traceable to one artefact version.)

• fundamental flaws are 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 examinor are different anyway: an examinor fixing flaws would violate the role assignment.

Do not switch (fine grained) between examination and debugging.
Dependability Case
Proposal: Dependability Cases (Jackson, 2009)

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

  “Developers [should] **express the critical properties** and **make an explicit argument** that the system satisfies them.”

**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**, i.e. an **argument**, that the software, in concert with other components, establishes the **critical properties**.

- The **dependability 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.
Tell Them What You’ve Told Them…

- **Runtime Verification**
  - (as the name suggests) checks properties at program run-time,
  - generous use of `assert`s can be a valuable safe-guard against
    - regressions, usage outside specification, etc.
  and serve as formal documentation of (intermediate) assumptions.
  Very attractive effort / effect ratio.

- **Review** (structured examination of artefacts by humans)
  - (mild variant) advocated in the XP approach,
  - not uncommon:
    lead programmer reviews all commits from team members,
  - literature reports good effort/effect ratio achievable.

- All approaches to code quality assurance have their
  - advantages and drawbacks.
  - Which to use? It depends!

- Overall: Consider Dependability Cases
  - an (auditable, complete, sound) argument,
  that a software has the critical properties.
Looking Back:
17 Lectures on Software Engineering
Contents of the Course

- Customer Developer announcement (Lastenheft)
  → Customer Developer offer (Pflichtenheft)
  → Customer Developer software contract (incl. Pflichtenheft)
  → Developer Customer software delivery

| Introduction | L 1: 25.4., Thu |
| Metrics, Costs, Development Process | L 2: 29.4., Mon |
| | L 3: 2.5., Thu |
| | L 4: 6.5., Mon |
| | T 1: 9.5., Thu |
| Requirements Engineering | L 5: 13.5., Mon |
| | L 6: 16.5., Thu |
| | L 7: 20.5., Mon |
| | T 2: 23.5., Thu |
| | L 8: 27.5., Mon |
| | L 9: 30.5., Thu |
| | T 3: 6.6., Thu |
| Arch. & Design, (Testing, Formal Verification) | L 10: 17.6., Mon |
| | L 11: 24.6., Mon |
| | T 4: 27.6., Thu |
| Software-Modelling, Patterns | L 12: 1.7., Mon |
| | L 13: 4.6., Thu |
| QA | L 14: 8.7., Mon |
| | T 5: 11.7., Thu |
| Wrap-Up | L 15: 15.7., Mon |
| | L 16: 18.7., Thu |
| | T 6: 25.7., Thu |

Introduction L 1: 25.4., Thu
Metrics, Costs, Development Process
Requirements Engineering
Development Process
Arch. & Design, (Testing, Formal Verification)
Software-Modelling, Patterns
QA
Wrap-Up
What Did We Do?
The Software-Engineering Course on One Slide

- **Topic Area: Project Management**
- **Topic Area: Requirements Engineering**
- **Topic Area: Architecture & Design**
- **Topic Area: Software Quality Assurance**
The Software-Engineering Course on One Slide

**Topic Area: Project Management**
- **measure**, know what you measure (scales, pseudo-metrics)
- estimate, measure, improve estimation — it’s about **experience**
- describe processes in terms of **artefact, activity, role**, etc. — and **risk**

**Topic Area: Requirements Engineering**
- requirements characterise **acceptable** and **unacceptable** softwares (there may be a gray zone)
- **formal requirements**: unambiguous, exact analysis methods
- requirements engineers see the **absence of meaning**

**Topic Area: Architecture & Design**
- Model: “Nobody builds a house without a **plan.”** (L. Lamport)
- software has **structural** and **behavioural** aspects
- there are **methods and tools** to analyse software models (know how to interpret analysis outcomes)

**Topic Area: Software Quality Assurance**
- testing is **almost always incomplete**; testing is **necessary** (know how to interpret the outcomes: true/false positive/negative)
- there are methods and tools to **prove correctness** code (correctness is relative: correct wrt. specification (and assumptions))
That’s Today’s Software Engineering — More or Less...
Questions?
Advertisements
Further studies:

- **Real-Time Systems**  (not in 2019/20)
  (specification and verification of real-time systems)

- **Software Design, Modelling, and Analysis in UML**  (not in 2019/20)
  (a formal, in-depth view on structural and behavioural modelling)

- **Cyber-Physical Systems I - Discrete Models**
  (more on variants of CFA and queries (LTL, CTL, CTL*))

- **Cyber-Physical Systems - Hybrid Models**
  (Modelling and analysis of cyber-physical systems with hybrid automata)

- **Program Verification**
  (the theory behind tools like VCC)

- **Formal Methods for Java**
  (JML and “VCC for Java”)

- **Decision Procedures**
  (the basis for program verification)

→ [https://swt.informatik.uni-freiburg.de/teaching](https://swt.informatik.uni-freiburg.de/teaching)
Individual Projects
(BSc/MSc project, Lab Project, BSc/MSc thesis)
• formal modelling of industrial case studies
• improving analysis techniques
• own topics
→ contact us (3–6 months before planned start).

Want to be a tutor, e.g. Software Engineering 2020,
→ contact us (around early September / early March).

Want to be a scientific student assistant?
→ contact us.
Thanks For Your Participation...
References
References


