Topic Area Code Quality Assurance: Content

- **VL 14**
  - Introduction and Vocabulary
  - Test case, test suite, test execution.
  - Positive and negative outcomes.

- **VL 15**
  - Limits of Software Testing
  - Glass-Box Testing
  - Statement-, branch-, term-coverage.

- **VL 16**
  - Testing: Rest
    - When to stop testing?
    - Model-based testing
    - Testing in the development process

- **VL 17**
  - Other Approaches
    - Runtime verification.
    - Review
  - Software quality assurance wrap-up
Recall: Deterministic Programs, Correctness

**Deterministic Programs**

- **System**
  
  \[ \mathbf{P} = \{ \mathbf{s} \cup \mathbf{a} \cup \mathbf{h} \} \quad \text{if \ there\ is\ either\ a\ skip\ or\ a\ while\ and} \quad \mathbf{v} = \{x = y, z = w\} \]

- **Semantics**
  
  Induced by the following transition relation \( \mathcal{R} \)

- **Example**
  
  \[ \mathbf{E} = \{ x = y, z = w \} \]

- **Notes**
  
  - Note: the first component of \( x = y \) is a program (structural operational semantics [2010])

**Input-Output Semantics of Deterministic Programs**

- **Definition**
  
  Let \( P \) be a deterministic program.
  
  - The semantics of partial correctness is the function:
    \[ \text{mp}(x) = \{ x = y, z = w \} \]
    
    \[ \text{mp}(x) = \{ x = y, z = w \} \]
    
  - The semantics of total correctness is the function:
    \[ \text{mt}(x) = \{ x = y, z = w \} \]
    
    \[ \text{mt}(x) = \{ x = y, z = w \} \]

**Correctness of Deterministic Programs**

- **Definition**
  
  Let \( P \) be a program over variables \( x \) and \( y \). A Boolean expression over \( x \):

  - The correctness formula:
    \[ \text{mp}(x) = \{ x = y, z = w \} \]
    
    \[ \text{mt}(x) = \{ x = y, z = w \} \]

- **Example**
  
  \[ \text{mp}(x) = \{ x = y, z = w \} \]

- **Notes**
  
  - Note: the first component of \( x = y \) is a program (structural operational semantics [2010])

- **Example**
  
  \[ \text{mp}(x) = \{ x = y, z = w \} \]

- **Notes**
  
  - Note: the first component of \( x = y \) is a program (structural operational semantics [2010])

**Input-Output Semantics of Deterministic Programs**

- **Definition**
  
  Let \( P \) be a deterministic program.
  
  - The semantics of partial correctness is the function:
    \[ \text{mp}(x) = \{ x = y, z = w \} \]
    
    \[ \text{mp}(x) = \{ x = y, z = w \} \]
  
  - The semantics of total correctness is the function:
    \[ \text{mt}(x) = \{ x = y, z = w \} \]
    
    \[ \text{mt}(x) = \{ x = y, z = w \} \]
Proof-System PD (for sequential, deterministic programs)

Axiom 1: Skip-Statement
\{p\} \text{skip} \{p\}

Axiom 2: Assignment
\{p[u := t]\} \text{u := t} \{p\}

Rule 3: Sequential Composition
\frac{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}

Rule 4: Conditional Statement
\frac{\{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
\frac{\{p \land B\} S \{p\}}{\{p\} \text{while } B \text{ do S od} \{p \land \neg B\}}

Rule 6: Consequence
\frac{p \rightarrow p_1, \{p_1\} S \{q_1\}, q_1 \rightarrow q}{\{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

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

(The first (textually represented) program that has been formally verified (7).)

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:

\[ (2) \]
\[ (1) \quad \{ P \land (BD) \} S^D_P \{ P \} \quad \text{while } B^D \text{ do } S^D_P \text{ od } \{ P \land \neg (BD) \} \]
\[ \text{while } B^D \text{ do } S^D_P \text{ od } \{ q^D \} \]
\[ (3) \]
\[ \{ p^D \} S^D_0 \{ P \}, \quad \{ P \} \text{ while } B^D \text{ do } S^D_P \text{ od } \{ q^D \} \]

Example Proof Cont’d

In the following, we show

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

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).

• \( \vdash_{PD} \{ a \cdot y + x = x \land x \geq 0 \} \ b := x \{ a \cdot y + b = x \land b \geq 0 \} \equiv P \)
  by (A2).
Proof of (1)

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

\[ \vdash_{PD} \{ 0 \cdot y + x = x \land x \geq 0 \} a := 0 \{ a \cdot y + x = x \land x \geq 0 \} \quad \text{by (A2).} \]
\[ \vdash_{PD} \{ a \cdot y + x = x \land x \geq 0 \} b := x \{ a \cdot y + b = x \land b \geq 0 \} \quad \text{by (A2),} \]

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

Proof of (1)

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

\[ \vdash_{PD} \{ 0 \cdot y + x = x \land x \geq 0 \} a := 0 \{ a \cdot y + x = x \land x \geq 0 \} \quad \text{by (A2).} \]
\[ \vdash_{PD} \{ a \cdot y + x = x \land x \geq 0 \} b := x \{ a \cdot y + b = x \land b \geq 0 \} \quad \text{by (A2),} \]

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

\[ \text{using} \quad x \geq 0 \land y \geq 0 \rightarrow 0 \cdot y + x = x \land x \geq 0 \quad \text{and} \quad P \rightarrow P, \quad \text{we obtain} \]
\[ \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} a := 0; b := x \{ P \} \quad \text{by (R6).} \]
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:

\[
\begin{align*}
\text{Expressions:} & \\
\text{• plain variable } x : [x := u] & \equiv \begin{cases} 
  t & \text{if } x = u \\
  x & \text{otherwise} 
\end{cases} \\
\text{• constant } c : [u := t] & \equiv c \\
\text{• constant op, terms } s_1 : [u := t] & \equiv op(s_1[u := t], \ldots, s_n[u := t]) \\
\text{• conditional expression: } (B \Rightarrow s_1 : s_2) & [u := t] \\
& \equiv (B[u := t] \Rightarrow s_1[u := t] : s_2[u := t]) \\
\text{• indexed variable, u plain or } u \equiv b[t_1, \ldots, t_m] & \text{ and } a \neq b: \\
(a[s_1, \ldots, s_n]) & [u := t] \equiv a[s_1[u := t], \ldots, s_n[u := t]] \\
\text{• indexed variable, u } \equiv a[t_1, \ldots, t_m]: \\
(a[s_1, \ldots, s_n]) & [u := t] \equiv (\wedge_{i=1}^n s_i[u := t] = t_i \ ? \ t : a[s_1[u := t], \ldots, s_n[u := t]]) \\
\text{Formulae:} & \\
\text{• boolean expression } p \equiv s: \\
p & [u := t] \equiv s[u := t] \\
\text{• negation:} \\
(\neg q) & [u := t] \equiv \neg(q[u := t]) \\
\text{• conjunction etc.:} \\
(q \land r) & [u := t] \\
& \equiv q[u := t] \land r[u := t] \\
\text{• quantifier:} \\
(\forall x : q) & [u := t] \\
& \equiv \forall y : q[x := y][u := t] \\
\text{y fresh (not in } q, t, u), \text{ same type as } x.
\end{align*}
\]
In the following, we show 

(1) $\vdash_D \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \ \{ P \}$.

(2) $\vdash_D \{ P \land b \geq y \} \ b := b - y; \ a := a + 1 \ \{ P \}$.

(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:

  $\vdash_D \{ a + 1 \cdot y + (b - y) = x \land (b - y) \geq 0 \} \ b := b - y; \ a := a + 1 \ \{ P \}$

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

- $\vdash_D \{ (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \ b := b - y \ \{ (a + 1) \cdot y + b = x \land b \geq 0 \}$ by (A2).

- $\vdash_D \{ (a + 1) \cdot y + b = x \land b \geq 0 \} \ a := a + 1 \ \{ a \cdot y + b = x \land b \geq 0 \}$ by (A2).

- $\vdash_D \{ (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \ b := b - y; \ a := a + 1 \ \{ P \}$ by (R3).
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 \} \quad \text{by \((A2)\)}.
\]

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

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

by \((R6)\).

Example Proof Cont’d

In the following, we show

\[ \begin{array}{ll}
(1) & \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \; a := 0; \; b := x \{ P \}, \\
(2) & \vdash_{PD} \{ P \land b \geq y \} \; b := b - y; \; a := a + 1 \{ P \}, \\
(3) & \vdash P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y.
\end{array} \]

As loop invariant, we choose (creative act):

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

(3) claims

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

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

Proof: easy.

---

Back to the Example Proof

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

\[
\begin{array}{ccc}
\{ x \geq 0 \land y \geq 0 \} & a := 0; \ b := x \{ P \} & P \rightarrow P. \\
\{ P \} \text{ while } b \geq y \ do \ b := b - y; \ a := a + 1 \{ P \} & \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} & P \rightarrow P
\end{array}
\]

\[
\begin{array}{ccc}
\{ P \} \text{ while } b \geq y \ do \ b := b - y; \ a := a + 1 \{ P \} & \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} & P \rightarrow P
\end{array}
\]

\[
\begin{array}{ccc}
\{ P \} \text{ while } b \geq y \ do \ b := b - y; \ a := a + 1 \{ P \} & \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} & P \rightarrow P
\end{array}
\]

thus

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

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

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

IOW: whenever \textbf{DIV} is called with \( x \) and \( y \) such that \( x \geq 0 \land y \geq 0, \) then (if \textbf{DIV} terminates) \( a \cdot y + b = x \land b < y \) will hold.
Once Again

- $P \equiv a \cdot y + b = x \land b \geq 0$
  - $\{ x \geq 0 \land y \geq 0 \}$
  - $\{ 0 \cdot y + x = x \land x \geq 0 \}$
- $a := 0$
  - $\{ a \cdot y + x = x \land x \geq 0 \}$
- $b := x$
  - $\{ a \cdot y + b = x \land b \geq 0 \}$
  - $\{ P \}$
- while $b \geq y$
  - $\{ P \land b \geq y \}$
  - $\{ (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \}$
- $b := b - y$
  - $\{ (a + 1) \cdot y + b = x \land b \geq 0 \}$
- $a := a + 1$
  - $\{ a \cdot y + b = x \land b \geq 0 \}$
  - $\{ P \}$
- od
  - $\{ P \land \neg (b \geq y) \}$
  - $\{ a \cdot y + b = x \land b < y \}$

Literature Recommendation

- Programmverifikation
- Verification of Sequential and Concurrent Programs
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

assertions
**Assertions**

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

- and the semantics by rule

  \[ \langle \text{assert}(B), \sigma \rangle \rightarrow \langle E, \sigma \rangle \text{ if } \sigma \models B. \]

  (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 \( \{ \text{true} \} \ x := 0; \text{ assert}(x = 27) \{ \text{true} \} \) in PD.

---

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

\[
\begin{align*}
\text{(R7)} & \quad \{p\} \ F \ \{q\} \\
& \quad \{p\} \ f() \ \{q\}
\end{align*}
\]

"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

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

we may be able to prove our pocket calculator correct.

\begin{verbatim}
12345678 + 27 7 8 9 0 4 5 6 + 1 2 3 =
\end{verbatim}

\begin{verbatim}
int x, y, sum;
int main() {
  while (true) {
    x = read_number();
    y = read_number();
    sum = add(); // add 'x' and 'y'
    display(); // display 'sum'
  }
}
\end{verbatim}
The Verifier for Concurrent C

VCC

- 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)
```
#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;
            }
    }
```

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

**VCC Web-Interface**

Example program \texttt{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: $\models \{false\} f \{q\}$ 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.
Tell Them What You’ve Told Them...

Testing:
- Define criteria for “testing done” (like coverage, or cost per error).
- Process: tester and developer should be different persons.

Formal Verification:
- There are more approaches to software quality assurance than just testing.
- For example, program verification.
- Proof System PD can be used
  - to prove
  - that a given program is
  - correct wrt. its specification.
  This approach considers all inputs inside the specification!
- Tools like VCC implement this approach.

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
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:

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

• Assume there are functions \( \text{check}_p \) and \( \text{check}_q \), 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
  \begin{enumerate}
  \item extending \( S \) by implementations of \( \text{check}_p \) and \( \text{check}_q \),
  \item call \( \text{check}_p \) right after entering \( f \),
  \item call \( \text{check}_q \) right before returning from \( f \).
  \end{enumerate}

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

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

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

```
#include <assert.h>

void assert(scalar expression);
```

```
DESCRIPTION

[...] the macro assert prints an error message to standard error and terminates the program by calling abort(3) if expression is false (i.e., compares equal to zero).

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.
```

- 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: $\{p\} f \{q\}$)

- Compute the width of a progress bar:

\begin{center}
\begin{tabular}{l}
\textbf{TreeNode} - key : int \\
\hspace{1cm} leftChild : TreeNode \\
\hspace{1cm} rightChild : TreeNode \\
\hspace{1cm} parent : TreeNode
\end{tabular}
\end{center}

\begin{center}
Object
\end{center}

- Recall the structure model with Proto-OCL constraint from Exercise Sheet 4.
- Assume, we add a method set_key() to class TreeNode:

\begin{verbatim}
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;
  }
}
\end{verbatim}

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

\begin{verbatim}
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;
}
\end{verbatim}
Run-Time Verification: Discussion

- **Experience:**
  During development, 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.

- Can serve as formal (support of) documentation:
  “Dear reader, at this point in the program, I expect condition expr to hold, because…”
By the Way: Development vs. Release Versions

- Development vs. Release Versions:
  - Common practice:
    - development version with run-time verification enabled (cf. \texttt{assert(3)}).
    - release version without run-time verification.
  
  If run-time verification is enabled in a release version,
  - software should terminate as gracefully as possible (e.g. try to save data),
  - save information from assertion failure if possible for future analysis.

  Risk: with bad luck, the software only behaves well because of the run-time verification code...

  Then disabling run-time verification "breaks" the software. Yet very complex run-time verification
  may significantly slow down the software, so needs to be disabled...


code snippet:

\begin{verbatim}
assert ( x > 0 );
assert ( x < 0 );
\end{verbatim}

---

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
Recall: Three Basic Directions

- Testing: input → output
- Review: execution of \("(s, S)\)" by Reviewer
- Formal Verification: prove \(S \models \phi\), conclude \(\{S\} \in \psi\)
Reviews

- 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
  - 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.

- Preparation
  - Planning: reviews need time in the project plan.
  - Preparation: reviewers investigate review item.

- Review Session
  - Review organisation under guidance of moderator
  - Approval of review item
  - Review: report, evaluate, and document issues; resolve open questions.

- “3rd hour”
  - Time for informal chat; reviewers may state proposals for solutions or improvements.

- Postparation
  - Postparation: rework review item; responsibility of the author(s).

- Analysis
  - Analysis: improve development and review process.

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

- Reviewers re-assess reworked review item (until approval is declared).
## Review Rules

1. The **moderator** organises the review, issues invitations, supervises the review session.

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

3. The review session is **limited to 2 hours**. If needed: organise more sessions.

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

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

6. **Style** issues (outside fixed conventions) are **not discussed**.

7. The **review team** is not supposed to **develop solutions**. Issues are **not** noted down in form of **tasks** for the author(s).

8. Each **reviewer** gets the opportunity to present her/his findings appropriately.

9. **Reviewers** need to reach **consensus** on issues, consensus is noted down.

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

11. The **review team** declares:
    - accept **without changes**,
    - accept **with changes**,
    - do not accept.

12. The **protocol** is signed by all participants.

---

### Stronger and Weaker Review Variants

<table>
<thead>
<tr>
<th>Design and Code Inspection (??)</th>
<th><strong>more effort</strong></th>
<th><strong>more effect</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>deluxe variant of review,</td>
<td></td>
<td>approx. 50% more time, approx. 50% more errors found.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Review</th>
<th><strong>less effort, less effective</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structured Walkthrough</strong></td>
<td></td>
</tr>
<tr>
<td>simple variant of review:</td>
<td></td>
</tr>
<tr>
<td><strong>developer</strong> moderates walkthrough-session,</td>
<td></td>
</tr>
<tr>
<td>developer presents artefact(s),</td>
<td></td>
</tr>
<tr>
<td>reviewer poses (prepared or spontaneous) questions,</td>
<td></td>
</tr>
<tr>
<td>issues are noted down,</td>
<td></td>
</tr>
<tr>
<td>Variation point: do reviewers see the artefact before the session?</td>
<td></td>
</tr>
<tr>
<td>less effort, less effective.</td>
<td></td>
</tr>
</tbody>
</table>

→ 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>auto-matic</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-Review</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 can be challenging;
- executing many tests may need substantial time (but: can sometimes be run in parallel).

---

### Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>auto-matic</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-Review</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:**
- 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>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>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>
</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>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>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>
</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>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 challenging: failing to find a proof does not allow any useful conclusion;
- false negatives (broken program “proved” correct) hard to detect.
Proposal: Dependability Cases

- 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.
• **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

### What Did We Do?

### Intoduction
- L 1: 18.4, Mon
- L 2: 18.4, Thu
- L 3: 18.4, Thu
- T 1: 18.4, Thu

### Scales, Metrics, Costs
- L 1: 21.4, Mon
- L 2: 21.4, Thu
- L 3: 21.4, Thu
- T 1: 21.4, Thu

### Development
- L 4: 2.5, Mon
- L 5: 5.5, Thu
- L 6: 12.5, Thu
- T 2: 19.5, Mon

### Process
- L 7: 26.5, Thu
- L 8: 30.5, Mon
- L 9: 6.6, Mon
- T 3: 9.6, Thu

### Requirements Engineering
- L 10: 13.6, Mon
- L 11: 16.6, Thu
- L 12: 20.6, Mon
- T 4: 23.6, Thu

### Architecture & Design
- L 13: 27.6, Mon
- L 14: 30.6, Thu
- L 15: 4.7, Mon
- T 5: 7.7, Thu

### Software Modelling
- L 16: 11.7, Mon
- L 17: 14.7, Thu
- L 18: 18.7, Mon
- L 19: 21.7, Thu

### Quality Assurance (Testing, Formal Verification)
- L 16:
- L 17:
- L 18:
- L 19:

---

**What Did We Do?**

- Project planning
- McCabe complexity
- Delphi method
- Requirements on requirements
- Modularity, information hiding
- Empirical data
- Early deadlines missed
- System states, ODs
- Coverage example

---

**What Did We Do?**

- Process Management
- Requirements Engineering
- Architecture & Design
- Code Quality Assurance

---

**What Did We Do?**

- Process Management
- Requirements Engineering
- Architecture & Design
- Code Quality Assurance
That’s Today’s Software Engineering — More or Less...
• **Further studies:**
  • **Real-Time Systems** → *Winter 2017/18*  
    (specification and verification of real-time systems)
  • **Software Design, Modelling, and Analysis in UML**  
    (not in 2017/18)  
    (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 2018?  

→ **contact us** (around September / March).
Thanks For Your Participation...
References


