Softwaretechnik / Software-Engineering

Lecture 17: Wrapup & Questions

2017-07-24

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

Albert-Ludwigs-Universität Freiburg, Germany

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**
  - **Program Verification**
    - partial and total **correctness**,
    - **Proof System PD**.
- VL 17
  - **Other Approaches**
    - **Runtime verification**.
    - **Review**
  - Software quality assurance wrap-up

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: Deterministic Programs, Correctness

Deterministic Programs

Syntax:

$$S := skip \mid u := t \mid S_1; S_2 \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od}$$

where $u \in V$ is a **variable**, t is a type-compatible **expression**, B is a Boolean **expression**.

Semantics: (is induced by the following transition relation) – $\sigma : V \to \mathcal{D}(V)$

(i) $\langle skip, \sigma \rangle \to \langle E, \sigma \rangle$ empty programs

(ii) $\langle u := t, \sigma \rangle \to \langle E, \sigma[u := \sigma(t)] \rangle$

(iii) $\dfrac{\langle S_1, \sigma \rangle \to \langle S_2, \tau \rangle}{\langle S_1; S, \sigma \rangle \to \langle S_2; S, \tau \rangle}$

(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \to \langle S_1, \sigma \rangle$, if $\sigma \models B$,

(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \to \langle S_2, \sigma \rangle$, if $\sigma \not\models B$,

(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle$, if $\sigma \models B$,

(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle \to \langle E, \sigma \rangle$, if $\sigma \not\models B$,

E denotes the **empty program**; define $E; S \equiv S; E \equiv S$.

Note: the first component of $\langle S, \sigma \rangle$ is a program (structural operational semantics (SOS)).

– 16 – 2017-07-20 – Swhile – 21/54

Example

(i) $\langle skip, \sigma \rangle \to \langle E, \sigma \rangle$ $\qquad E; S \equiv S; E \equiv S$
(ii) $\langle u := t, \sigma \rangle \to \langle E, \sigma[u := \sigma(t)] \rangle$
(iii) $\dfrac{\langle S_1, \sigma \rangle \to \langle S_2, \tau \rangle}{\langle S_1; S, \sigma \rangle \to \langle S_2; S, \tau \rangle}$
(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \to \langle S_1, \sigma \rangle$, if $\sigma \models B$,
(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \to \langle S_2, \sigma \rangle$, if $\sigma \not\models B$,
(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle$, if $\sigma \models B$,
(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma \rangle \to \langle E, \sigma \rangle$, if $\sigma \not\models B$.

Consider **program**

$$S \equiv a[0] := 1; a[1] := 0; \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ od}$$

and a **state** σ with $\sigma \models x = 0$.

$$\begin{aligned}
\langle S, \sigma \rangle &\xrightarrow{(ii),(iii)} \langle a[1] := 0; \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ od}, \sigma[a[0] := 1] \rangle \\
&\xrightarrow{(ii),(iii)} \langle \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ od}, \sigma' \rangle \\
&\xrightarrow{(vi)} \langle x := x + 1; \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ od}, \sigma' \rangle \\
&\xrightarrow{(ii),(iii)} \langle \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ od}, \sigma'[x := 1] \rangle \\
&\xrightarrow{(vii)} \langle E, \sigma'[x := 1] \rangle
\end{aligned}$$

where $\sigma' = \sigma[a[0] := 1][a[1] := 0]$.

– 16 – 2017-07-20 – Swhile – 22/54

Input/Output Semantics of Deterministic Programs

Definition.
Let S be a deterministic program.

(i) The **semantics of partial correctness** is the function

$$\mathcal{M}[\![S]\!] : \Sigma \to 2^\Sigma$$

with $(\mathcal{M}[\![S]\!])(\sigma) = \{\tau \mid \langle S, \sigma \rangle \to^* \langle E, \tau \rangle\}$. finitely many

(ii) The **semantics of total correctness** is the function

$$\mathcal{M}_{tot}[\![S]\!] : \Sigma \to 2^\Sigma \,\dot\cup\, \{\infty\}$$

with $(\mathcal{M}_{tot}[\![S]\!])(\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\infty \mid S \text{ can diverge from } \sigma\}$.
∞ is an error state representing **divergence**.

Note: $\mathcal{M}_{tot}[\![S]\!](\sigma)$ has exactly one element, $\mathcal{M}[\![S]\!](\sigma)$ at most one.

Example: $\mathcal{M}[\![S_1]\!](\sigma) = \mathcal{M}_{tot}[\![S_1]\!](\sigma) = \{\tau \mid \tau(x) = \sigma(x) \wedge \tau(y) = \sigma(x)^2\}, \quad \sigma \in \Sigma$.
(Recall: $S_1 \equiv y := x; y := (x - 1) \cdot x + y$)

– 16 – 2017-07-20 – Swhile – 25/54

Correctness of Deterministic Programs

pre-condition

post-condition

Definition.
Let S be a program over variables V, and p and q Boolean expressions over V.

(i) The **correctness formula**

$$\{p\}\ S\ \{q\} \qquad \text{("Hoare triple")}$$

holds in the sense of partial correctness,
denoted by $\models \{p\}\ S\ \{q\}$, if and only if

$$(\mathcal{M}[\![S]\!])([\![p]\!]) \subseteq [\![q]\!].$$

We say S is **partially correct** wrt. p and q.

(ii) A **correctness formula**

$$\{p\}\ S\ \{q\}$$

holds in the sense of total correctness,
denoted by $\models_{tot} \{p\}\ S\ \{q\}$, if and only if

$$\mathcal{M}_{tot}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!].$$

We say S is **totally correct** wrt. p and q.

– 16 – 2017-07-20 – Scorrectness – 27/54

Proof-System PD

Proof-System PD (for sequential, deterministic programs)

Axiom 1: Skip-Statement

$$\{p\}\ \mathit{skip}\ \{p\}$$

Axiom 2: Assignment

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

Rule 3: Sequential Composition

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

Rule 4: Conditional Statement

$$\frac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\},}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$$

Rule 5: While-Loop

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$$

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

$$DIV \equiv \overbrace{a := 0;\ b := x;}^{=:S_0^D}\ \mathbf{while}\ \overbrace{b \geq y}^{=:B^D}\ \mathbf{do}\ \overbrace{b := b - y;\ a := a + 1}^{=:S_1^D}\ \mathbf{od}$$

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

We can prove $\models \{x \geq 0 \wedge y \geq 0\}\ DIV\ \{a \cdot y + b = x \wedge b < y\}$

by showing $\vdash_{PD} \underbrace{\{x \geq 0 \wedge y \geq 0\}}_{=:p^D}\ DIV\ \underbrace{\{a \cdot y + b = x \wedge b < y\}}_{=:q^D}$, i.e., derivability in PD:

$$\dfrac{\dfrac{\text{(1)}}{\{p^D\}\ S_0^D\ \{P\},}\qquad \dfrac{\dfrac{}{P \to P,}\quad \dfrac{\dfrac{\text{(2)}}{\{P \wedge (B^D)\}\ S_1^D\ \{P\}}}{\{P\}\ \mathbf{while}\ B^D\ \mathbf{do}\ S_1^D\ \mathbf{od}\ \{P \wedge \neg(B^D)\},}\,\text{(R5)}\quad \dfrac{\text{(3)}}{P \wedge \neg(B^D) \to q^D}}{\{P\}\ \mathbf{while}\ B^D\ \mathbf{do}\ S_1^D\ \mathbf{od}\ \{q^D\}}\,\text{(R6)}}{\{p^D\}\ S_0^D;\ \mathbf{while}\ B^D\ \mathbf{do}\ S_1^D\ \mathbf{od}\ \{q^D\}}\,\text{(R3)}$$

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$

(R6) $\dfrac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}$

Example Proof Cont'd

$$\dfrac{\dfrac{\text{(1)}}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x\ \{P\},}\qquad \dfrac{\dfrac{}{P \to P,}\quad \dfrac{\dfrac{\text{(2)}}{\{P \wedge (b \geq y)\}\ b := b - y;\ a := a + 1\ \{P\}}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{P \wedge \neg(b \geq y)\},}\,\text{(R5)}\quad \dfrac{\text{(3)}}{P \wedge \neg(b \geq y) \to a \cdot y + b = x \wedge b < y}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\,\text{(R6)}}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x;\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\,\text{(R3)}$$

In the following, we show

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

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

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

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

$$P \equiv a \cdot y + b = x \wedge b \geq 0$$

Proof of (1)

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$

(R6) $\dfrac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$

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

- $\vdash_{PD} \{\underbrace{0 \cdot y + x = x \wedge x \geq 0}_{(P[u:=t]) \ P[a:=0])}\}\ a := 0\ \{\overbrace{a \cdot y + x = x \wedge x \geq 0}^{P}\}$ by (A2),

Proof of (1)

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$

(R6) $\dfrac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$

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

- $\vdash_{PD} \{0 \cdot y + x = x \wedge x \geq 0\}\ a := 0\ \{\overbrace{a \cdot y + x = x \wedge x \geq 0}^{r}\}$ by (A2),
- $\vdash_{PD} \{\underbrace{a \cdot y + \underline{x} = x \wedge \underline{x} \geq 0}_{P[b:=x]}\}\ b := x\ \{\underbrace{a \cdot y + \underline{b} = x \wedge \underline{b} \geq 0}_{\equiv P}\}$ by (A2),

Proof of (1)

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$

(R6) $\dfrac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}$

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

- $\vdash_{PD} \{0 \cdot y + x = x \wedge x \geq 0\}\ a := 0\ \{a \cdot y + x = x \wedge x \geq 0\}$ by (A2),
- $\vdash_{PD} \{a \cdot y + x = x \wedge x \geq 0\}\ b := x\ \{\underbrace{a \cdot y + b = x \wedge b \geq 0}_{\equiv P}\}$ by (A2),
- thus, $\vdash_{PD} \{0 \cdot y + x = x \wedge x \geq 0\}\ a := 0;\ b := x\ \{P\}$ by (R3),

Proof of (1)

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$

(R6) $\dfrac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}$

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

- $\vdash_{PD} \{0 \cdot y + x = x \wedge x \geq 0\}\ a := 0\ \{a \cdot y + x = x \wedge x \geq 0\}$ by (A2),
- $\vdash_{PD} \{a \cdot y + x = x \wedge x \geq 0\}\ b := x\ \{\underbrace{a \cdot y + b = x \wedge b \geq 0}_{\equiv P}\}$ by (A2),
- thus, $\vdash_{PD} \{0 \cdot y + x = x \wedge x \geq 0\}\ a := 0;\ b := x\ \{P\}$ by (R3),
- using $x \geq 0 \wedge y \geq 0 \to 0 \cdot y + x = x \wedge x \geq 0$ and $P \to P$, we obtain

$$\vdash_{PD} \{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x\ \{P\}$$

  by (R6). □

Substitution

The rule '**Assignment**' uses (syntactical) **substitution**: $\{p[u := t]\}\ u := t\ \{p\}$

(In formula p, replace all (free) occurences of (program or logical) variable u by term t.)

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

$$a \geq x\ [x := u+3] \rightsquigarrow a \geq u+3$$

$$a \geq x \wedge \forall x \bullet b \geq x\ [x := \underset{+x}{u+3}] \rightsquigarrow a \geq \underset{+x}{u+3} \wedge \forall z \bullet b \geq z$$

$$\underbrace{c[x] \geq 0}_{t}\ [c[0] := 27] \rightsquigarrow (x = 0\ ?\ 27 : c[x])$$

Substitution

The rule '**Assignment**' uses (syntactical) **substitution**: $\{p[u := t]\}\ u := t\ \{p\}$

(In formula p, replace all (free) occurences of (program or logical) variable u by term t.)

Defined as usual, only **indexed** and **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:
  $(B\ ?\ s_1 : s_2)[u := t]$
  $\equiv (B[u := t]\ ?\ s_1[u := t] : s_2[u := t])$

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.:
  $(q \wedge r)[u := t]$
  $\equiv q[u := t] \wedge r[u := t]$
- **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.

- **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**, $u \equiv a[t_1, \ldots, t_m]$:
$$(a[s_1, \ldots, s_n])[u := t] \ \equiv (\textstyle\bigwedge_{i=1}^{n} s_i[u := t] = t_i\ ?\ t\ : a[s_1[u := t], \ldots, s_n[u := t]])$$

Example Proof Cont'd

$$
\dfrac{\dfrac{\text{(1)}}{\{x \geq 0 \wedge y \geq 0\}\; a := 0;\; b := x\; \{P\},} \quad \dfrac{\dfrac{}{P \rightarrow P,} \quad \dfrac{\dfrac{\text{(2)}}{\{P \wedge (b \geq y)\}\; b := b - y;\; a := a + 1\; \{P\}}}{\{P\}\; \textbf{while}\; b \geq y\; \textbf{do}\; b := b - y;\; a := a + 1\; \textbf{od}\; \{P \wedge \neg(b \geq y)\},}\,\text{(R5)} \quad \dfrac{\text{(3)}}{P \wedge \neg(b \geq y) \rightarrow a \cdot y + b = x \wedge b < y}}{\{P\}\; \textbf{while}\; b \geq y\; \textbf{do}\; b := b - y;\; a := a + 1\; \textbf{od}\; \{a \cdot y + b = x \wedge b < y\}}\,\text{(R6)}}{\{x \geq 0 \wedge y \geq 0\}\; a := 0;\; b := x;\; \textbf{while}\; b \geq y\; \textbf{do}\; b := b - y;\; a := a + 1\; \textbf{od}\; \{a \cdot y + b = x \wedge b < y\}}\,\text{(R3)}
$$

In the following, we show

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

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

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

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

$$P \equiv a \cdot y + b = x \wedge b \geq 0$$

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

Proof of (2)

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

- **(2)** claims:
  $\vdash_{PD} \{P \wedge b \geq y\}\; b := b - y;\; a := a + 1\; \{P\}$
  where $P \equiv a \cdot y + b = x \wedge b \geq 0$.

- $\vdash_{PD} \{(a+1) \cdot y + (b - y) = x \wedge (b - y) \geq 0\}\; b := b - y\; \{(a+1) \cdot y + b = x \wedge b \geq 0\}$
  by (A2),

- $\vdash_{PD} \{(a+1) \cdot y + b = x \wedge b \geq 0\}\; a := a + 1\; \{\underbrace{a \cdot y + b = x \wedge b \geq 0}_{\equiv P}\}$ by (A2),

- $\vdash_{PD} \{(a+1) \cdot y + (b - y) = x \wedge (b - y) \geq 0\}\; b := b - y;\; a := a + 1\; \{P\}$ by (R3),

Proof of (2)

(A1) $\{p\}\ skip\ \{p\}$	(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$
(A2) $\{p[u := t]\}\ u := t\ \{p\}$	(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$
(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$	(R6) $\dfrac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}$

- **(2)** claims:
  $\vdash_{PD} \{P \wedge b \geq y\}\ b := b - y;\ a := a + 1\ \{P\}$
  where $P \equiv a \cdot y + b = x \wedge b \geq 0$.

- $\vdash_{PD} \{(a+1) \cdot y + (b-y) = x \wedge (b-y) \geq 0\}\ b := b - y\ \{(a+1) \cdot y + b = x \wedge b \geq 0\}$
  by (A2),

- $\vdash_{PD} \{(a+1) \cdot y + b = x \wedge b \geq 0\}\ a := a + 1\ \{\underbrace{a \cdot y + b = x \wedge b \geq 0}_{\equiv P}\}$ by (A2),

- $\vdash_{PD} \{(a+1) \cdot y + (b-y) = x \wedge (b-y) \geq 0\}\ b := b - y;\ a := a + 1\ \{P\}$ by (R3),

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

$$\vdash_{PD} \{P \wedge b \geq y\}\ b := b - y;\ a := a + 1\ \{P\}$$

  by (R6). $\square$

Example Proof Cont'd

$$\dfrac{\dfrac{(1)}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x\ \{P\},}\qquad \dfrac{\dfrac{}{P \to P,}\quad \dfrac{\dfrac{(2)}{\{P \wedge (b \geq y)\}\ b := b - y;\ a := a + 1\ \{P\}}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{P \wedge \neg(b \geq y)\},}\,\text{(R5)}\quad \dfrac{(3)}{P \wedge \neg(b \geq y) \to a \cdot y + b = x \wedge b < y}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\,\text{(R6)}}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x;\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\,\text{(R3)}$$

In the following, we show

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

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

(3) $\models P \wedge \neg(b \geq y) \to \underbrace{a \cdot y + b = x \wedge b < y}_{I}$.

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

$$P \equiv a \cdot y + b = x \wedge b \geq 0$$

(A1) $\{p\}\ skip\ \{p\}$	(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$	(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{p \wedge \neg B\}}$
(A2) $\{p[u := t]\}\ u := t\ \{p\}$	(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$	(R6) $\dfrac{p \to p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \to q}{\{p\}\ S\ \{q\}}$

Proof of (3)

(3) claims

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

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

Proof: easy.

Back to the Example Proof

We have shown:

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

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

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

and

$$\dfrac{\dfrac{(1)}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x\ \{P\}},\qquad \dfrac{\dfrac{}{P \rightarrow P},\quad \dfrac{\dfrac{(2)}{\{P \wedge (b \geq y)\}\ b := b - y;\ a := a + 1\ \{P\}}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{P \wedge \neg(b \geq y)\}}\ \text{(R5)},\quad \dfrac{(3)}{P \wedge \neg(b \geq y) \rightarrow a \cdot y + b = x \wedge b < y}}{\{P\}\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\ \text{(R6)}}{\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x;\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}\ \{a \cdot y + b = x \wedge b < y\}}\ \text{(R3)}$$

thus

$$\vdash_{PD} \{x \geq 0 \wedge y \geq 0\}\ \underbrace{a := 0;\ b := x;\ \mathbf{while}\ b \geq y\ \mathbf{do}\ b := b - y;\ a := a + 1\ \mathbf{od}}_{\equiv DIV}\ \{a \cdot y + b = x \wedge b < y\}$$

and thus (since PD is sound) DIV is **partially correct** wrt.

- **pre-condition**: $x \geq 0 \wedge y \geq 0$,
- **post-condition**: $a \cdot y + b = x \wedge b < y$.

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

Once Again

(A1) $\{p\}\ skip\ \{p\}$

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

(R3) $\dfrac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \neg B\}}$

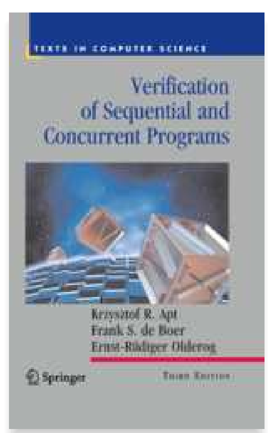
(R6) $\dfrac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$

- $P \equiv a \cdot y + b = x \wedge b \geq 0$

  $\{x \geq 0 \wedge y \geq 0\}$
  $\{0 \cdot y + x = x \wedge x \geq 0\}$
- $a := 0;$
  $\{a \cdot y + x = x \wedge x \geq 0\}$
- $b := x;$
  $\{a \cdot y + b = x \wedge b \geq 0\}$
  $\{P\}$
- **while** $b \geq y$ **do**
  $\{P \wedge b \geq y\}$
  $\{(a+1) \cdot y + (b-y) = x \wedge (b-y) \geq 0\}$
- $b := b - y;$
  $\{(a+1) \cdot y + b = x \wedge b \geq 0\}$
- $a := a + 1$
  $\{a \cdot y + b = x \wedge b \geq 0\}$
  $\{P\}$
- **od**
  $\{P \wedge \neg(b \geq y)\}$
  $\{a \cdot y + b = x \wedge b < y\}$

(A2) (A2) (R3) (R6) (A2) (A2) (R3) (R6) (R5) (R3) (R6)

Literature Recommendation



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

$$S ::= \cdots \mid \textbf{assert}(B)$$

- and the **semantics** by rule

$$\langle \textbf{assert}(B),\, \sigma \rangle \to \langle E,\, \sigma \rangle \text{ if } \sigma \models B.$$

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

Extend PD by axiom:

$$\text{(A7) } \{p\}\ \textbf{assert}(p)\ \{p\}$$

- 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 $\{\textit{true}\}\ x := 0;\ \texttt{assert}(x = 27)\ \{\textit{true}\}$ in PD.

Modular Reasoning

Modular Reasoning

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

$$\text{(R7)}\ \frac{\{p\}\ F\ \{q\}}{\{p\}\ f()\ \{q\}}$$

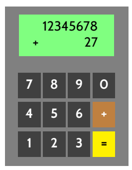
"If we have $\vdash \{p\}\ F\ \{q\}$ for the **implementation** of function f,
then if f is **called** in a state satisfying p, the state after return of f will satisfy q."

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

Example: if we have

- $\{\mathit{true}\}$ `read_number` $\{0 \leq \mathit{result} < 10^8\}$
- $\{0 \leq x \wedge 0 \leq y\}$ `add` $\{(\mathit{old}(x) + \mathit{old}(y) < 10^8 \wedge \mathit{result} = \mathit{old}(x) + \mathit{old}(y)) \vee \mathit{result} < 0\}$
- $\{\mathit{true}\}$ `display` $\{(0 \leq \mathit{old}(\mathit{sum}) < 10^8 \implies \text{"}\mathit{old}(\mathit{sum})\text{"}) \wedge (\mathit{old}(\mathit{sum}) < 0 \implies \text{"-E-"})\}$

we may be able to prove our pocket calculator correct.




```
int x, y, sum;

int main() {

  while (true) {
    x = read_number();
    y = read_number();

    sum = add(); // add 'x' and 'y'

    display(); // display 'sum'
  }
}
```


Return Values and Old Values

- For **modular reasoning**, it's often useful to refer in the post-condition to
  - the **return value** as result,
  - the **values** of variable x **at calling time** as $\mathit{old}(x)$.
- Can be defined using **auxiliary variables**:
  - Transform function

$$T\ f()\ \{\ldots; \mathbf{return}\ \mathit{expr};\}$$

(over variables $V = \{v_1, \ldots, v_n\}$; where $\mathit{result}, v_i^{old} \notin V$) into

$$\begin{array}{l} T\ f()\ \{ \\ \quad v_1^{old} := v_1; \ldots; v_n^{old} := v_n; \\ \quad \ldots; \\ \quad \mathit{result} := \mathit{expr}; \\ \quad \mathbf{return}\ \mathit{result}; \\ \} \end{array}$$

over $V' = V \cup \{v^{old} \mid v \in V\} \cup \{\mathit{result}\}$.

- Then $\mathit{old}(x)$ is just an abbreviation for x^{old}.

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)

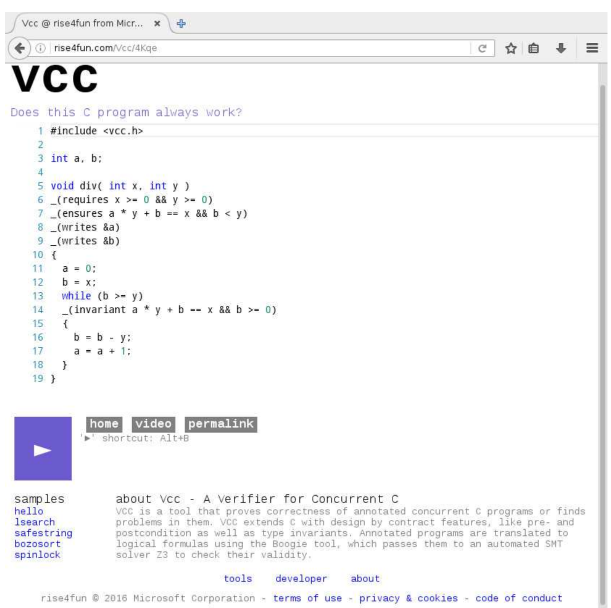
VCC Syntax Example

```
#include <vcc.h>

int a, b;

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

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

$$\{x \geq 0 \wedge y \geq 0\}\ DIV\ \{x \geq 0 \wedge y \geq 0\}$$

VCC Web-Interface



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 $\models \{p\}\ DIV\ \{q\}$.
  - 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**: $\models \{\mathit{false}\}\ f\ \{q\}$ **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.
  - $\rightarrow$ 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.

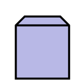
  This approach considers **all inputs** inside the specification!
- Tools like **VCC** implement this approach.

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

Run-Time Verification: Idea


Software S

- 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:

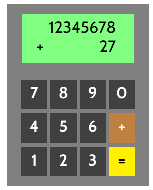
$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{call\ f} \sigma_{n+1} \cdots \sigma_m \xrightarrow{f\ returns} \sigma_{m+1} \cdots$$

- Assume there are functions $check_p$ and $check_q$, 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
  - (i) extending S by implementations of $check_p$ and $check_q$,
  - (ii) call $check_p$ right after entering f,
  - (iii) call $check_q$ right before returning from f.
- For S', obtain computation paths like:

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{call\ f} \sigma_{n+1} \xrightarrow{check_p} \sigma'_{n+1} \cdots \sigma_m \xrightarrow{check_q} \sigma'_m \xrightarrow{f\ returns} \sigma_{m+1} \cdots$$

- If $check_p$ and $check_q$ 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




```
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();
  }
}
```


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:

```
ASSERT(3)            Linux Programmer's Manual            ASSERT(3)

NAME
    assert – abort the program if assertion is false

SYNOPSIS
    #include <assert.h>

    void assert(scalar expression);

DESCRIPTION
          [...] the macro assert() prints an error message to stan–
    dard 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`)

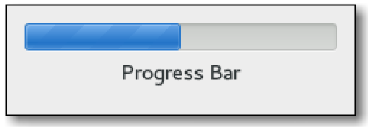
Assertions At Work

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

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

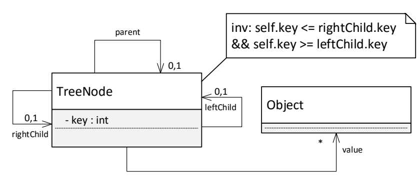
```
ASSERT(3)          Linux Programmer's Manual          ASSERT(3)

NAME
     assert — abort the program if assertion is false

SYNOPSIS
     #include <assert.h>

     void assert(scalar expression);

DESCRIPTION
            [...] the macro assert() prints an error message to stan—
     dard 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.
```

- Compute the width of a progress bar:




```

int progress_bar_width( int progress, int window_left, int window_right )
{
  assert( window_left <= window_right ); /* pre-condition */
  ...                      progress
  /* 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;
}
```


Assertions At Work II

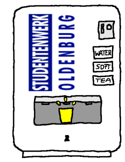


- Recall the **structure model** with Proto-OCL constraint from Exercise Sheet 4.
- Assume, we add a method `set_key()` to class **TreeNode**:

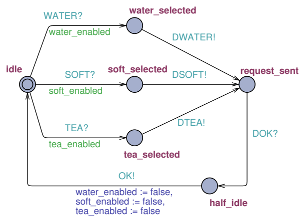
```
class TreeNode {

  private int key;
  TreeNode parent, leftChild, rightChild;

  public int get_key() { return key; }

  public void set_key( int new_key ) {
    key = new_key;
  }
}
```

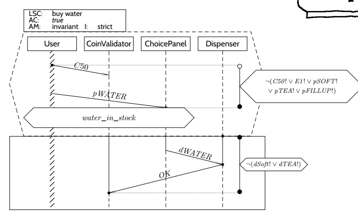
- We can **check consistency** with the Proto-OCL constraint at runtime by using assertions:

```
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;
}
```

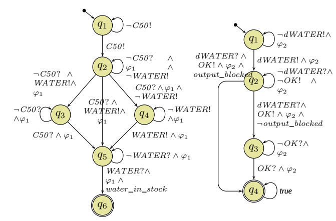


ChoicePanel:






```
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();
          ;;
} }
```



- **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. `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…

  assert (x = 0); ↯

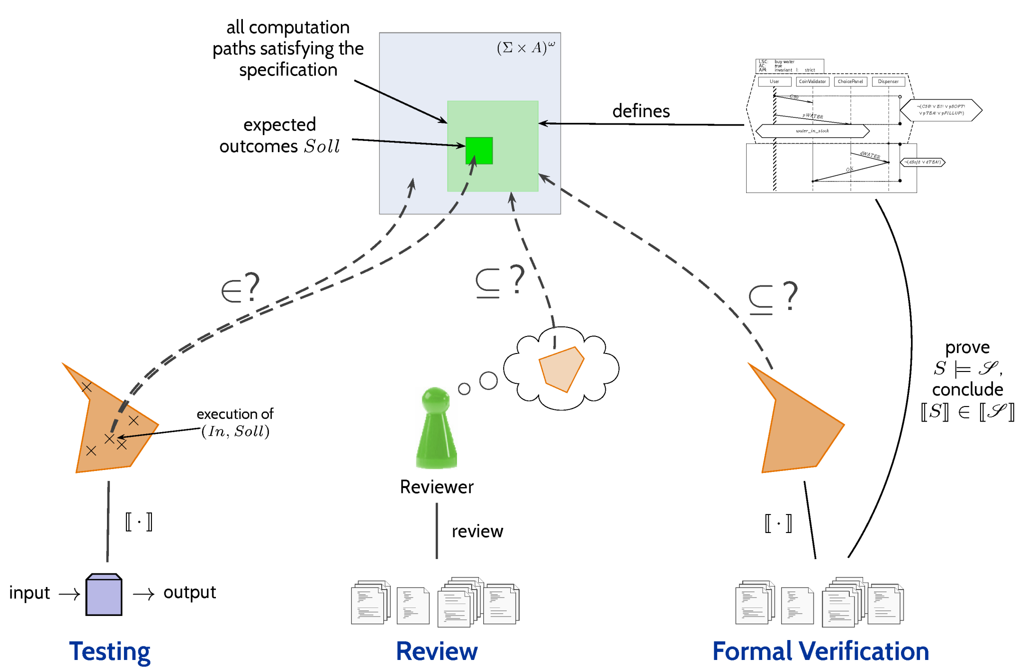
  assert (x == 0); :)

Content

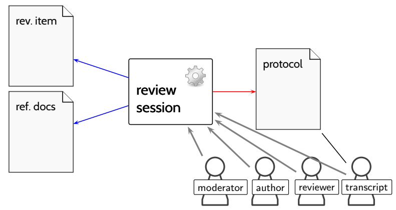
- **Formal Program Verification**
  - **Proof System PD**
- **The Verifier for Concurrent C**
  - Assertions, Modular Verification, VCC
- **Runtime-Verification**
  - **Assertions**, LSC-Observers
- **Reviews**
  - **Roles** and **artefacts**
  - Review **procedure**
  - Stronger and weaker **variants**
- **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



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.

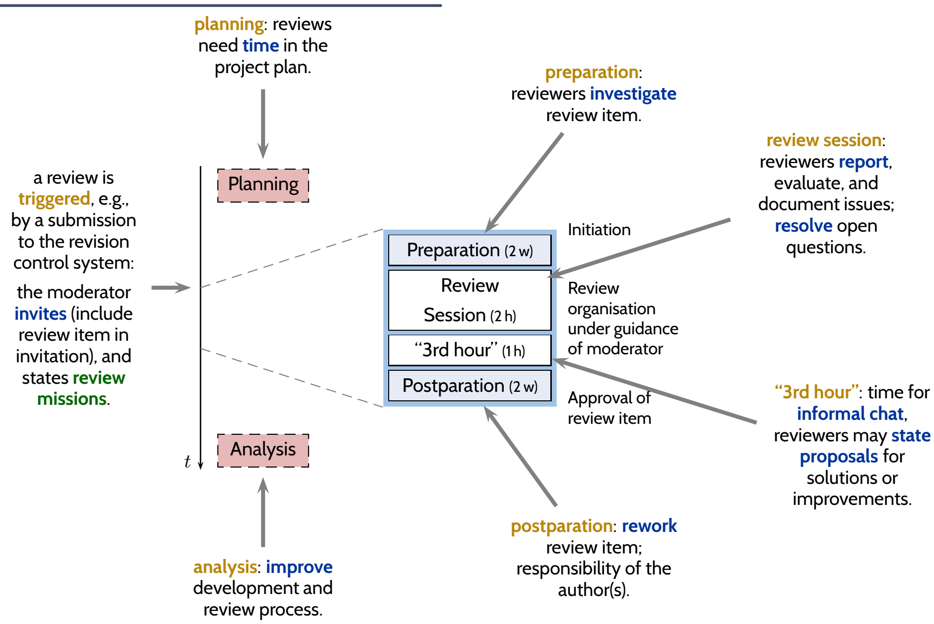
  Author: (representative of the) creator(s) of the artefact under review; is present to listen to the discussions; can answer questions; does not speak up if not asked.

  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



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

Review Rules (?)

(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

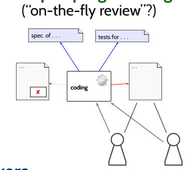
more effort more effect

less effort, less effect

- **Design and Code Inspection** (??)
  - 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 reponsibilities; "salesman"-**developer** may trick **reviewers**.

XP's pair programming
("on-the-fly review"?)



- **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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification							
Review							
Static Checking							
Verification							

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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification	✔	(✔)	✔	(✘)	✘	✔	(✔)
Review							
Static Checking							
Verification							

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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification	✔	(✔)	✔	(✘)	✘	✔	(✔)
Review	✘	✘	✘	(✔)	(✔)	✔	(✔)
Static Checking							
Verification							

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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification	✔	(✔)	✔	(✘)	✘	✔	(✔)
Review	✘	✘	✘	(✔)	(✔)	✔	(✔)
Static Checking	✔	(✘)	✘	✔	(✔)	✔	(✘)
Verification							

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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification	✔	(✔)	✔	(✘)	✘	✔	(✔)
Review	✘	✘	✘	(✔)	(✔)	✔	(✔)
Static Checking	✔	(✘)	✘	✔	(✔)	✔	(✘)
Verification	(✔)	✘	✘	✔	✔	(✘)	✘

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

	auto-matic	prove "can run"	toolchain considered	exhaus-tive	prove correct	partial results	entry cost
Test	(✔)	✔	✔	✘	✘	✔	✔
Runtime-Verification	✔	(✔)	✔	(✘)	✘	✔	(✔)
Review	✘	✘	✘	(✔)	(✔)	✔	(✔)
Static Checking	✔	(✘)	✘	✔	(✔)	✔	(✘)
Verification	(✔)	✘	✘	✔	✔	(✘)	✘

– 17 – 2017-07-24 – Sqwrapup –

50/69

Dependability Case

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.

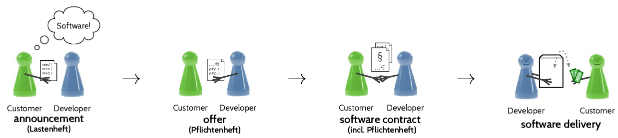
- **Runtime Verification**
  - (as the name suggests) checks properties at **program run-time**,
  - generous use of `assert`'s can be a valuable safe-guard against
    - **regressions**, usage **outside specification**, etc.

    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**.

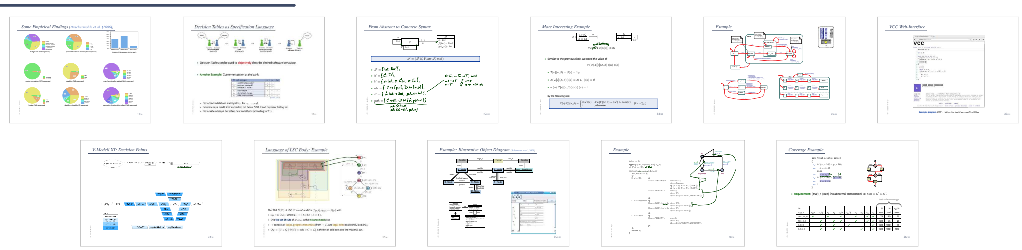
Looking Back:
17 Lectures on Software Engineering

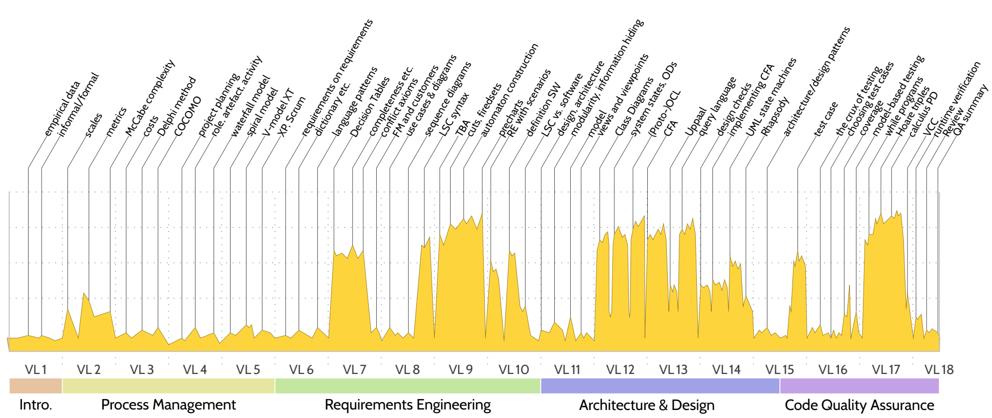
Contents of the Course



Introduction	L 1:	18.4., Mon
Scales, Metrics, Costs	L 2:	21.4., Thu
	L 3:	25.4., Mon
	T 1:	28.4., Thu
Development	L 4:	2.5., Mon
	-	5.5., Thu
Process	L 5:	9.5., Mon
	L 6:	12.5., Thu
	-	16.5., Mon
	-	19.5., Thu
	T 2:	23.5., Mon
	-	26.5., Thu
Requirements Engineering	L 7:	30.5., Mon
	L 8:	2.6., Thu
	L 9:	6.6., Mon
	T 3:	9.6., Thu
	L10:	13.6., Mon
Architecture & Design	L 11:	16.6., Thu
	L12:	20.6., Mon
	T 4:	23.6., Thu
Software Modelling	L13:	27.6., Mon
	L14:	30.6., Thu
	L15:	4.7., Mon
	T 5:	7.7., Thu
Quality Assurance (Testing, Formal Verification)	L16:	11.7., Mon
	L17:	14.7., Thu
	L18:	18.7., Mon
Wrap-Up	L19:	21.7., Thu

What Did We Do?





That’s Today’s Software Engineering — More or Less. . .

Questions?

Advertisements

Advertisement

- **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`

Advertisement

- **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).

References

References

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211.

Fagan, M. (1986). Advances in software inspections. *IEEE Transactions On Software Engineering*, 12(7):744–751.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Jackson, D. (2009). A direct path to dependable software. *Comm. ACM*, 52(4).

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