Topic Area Code Quality Assurance: Content

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

- **VL 15**
  - Limits of Software Testing
  - Glass-Box Testing
    - Statement-, branch-, term-coverage.
  - Other Approaches
    - Model-based testing.

- **VL 16**
  - Program Verification
    - partial and total correctness,
    - Proof System PD.
  - Runtime verification.

- **VL 17**
  - Review
**Concepts of Software Quality Assurance**

**Software Quality Assurance**

- **Project management**
  - **Examination by humans**
    - **Inspection**
      - **Review**
        - **Manual proof**
- **Software examination**
  - **Analytic**
    - **Non-mech.**
    - **Semi-mech.**
    - **Mechanical**
  - **Constructive software engineering**
  - **Formal verification**
    - **Static checking**
      - **Consistency checks**
    - **Dynamic checking (test)**
      - **Quantitative examination**
    - **Examine**
      - **Execute**
      - **Prove**
- **Constructive**
  - **Code generation**

---

**Validation**

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: **Verification**.

*IEEE 610.12 (1990)*

---

**Verification**

1. The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: **validation**.

2. Formal proof of program correctness.

*IEEE 610.12 (1990)*

---

**Ludewig and Lichter, 2013**
Content

- Formal Program Verification
  - Deterministic Programs
    - Syntax
    - Semantics
    - Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness
    - total correctness
  - Proof System PD

- The Verifier for Concurrent C
  - modular reasoning
  - return values / old values

- Assertions
Sequential, Deterministic While-Programs
Deterministic Programs

Syntax:

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

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) \(- \sigma : V \rightarrow D(V)\)

(i) \( (\text{skip}, \sigma) \rightarrow (E, \sigma) \)
(ii) \( (u ::= t, \sigma) \rightarrow (E, \sigma[u ::= \sigma(t)]) \)
(iii) \( (S_1, \sigma) \rightarrow (S_2, \tau) \)
(iv) \( (S_1; S, \sigma) \rightarrow (S_2; S, \tau) \)
(v) \( (\text{if } B \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow (S_1, \sigma), \text{if } \sigma \models B \)
(vi) \( (\text{if } B \text{ then } S_1 \text{ else } S_2, \sigma) \rightarrow (S_2, \sigma), \text{if } \sigma \not\models B \)
(vii) \( (\text{while } B \text{ do } S \text{ od}, \sigma) \rightarrow (S; \text{while } B \text{ do } S \text{ od}, \sigma), \text{if } \sigma \not\models B \), \( E \) denotes the empty program; define \( E; S \equiv S; E \equiv S \).

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

Example

Consider program

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

and a state \( \sigma \) with \( \sigma \models x = 0 \),

\[ (S, \sigma) \xrightarrow{(i),(ii)} (a[1] := 0; \text{while } a[x] \neq 0 \text{ do } x ::= x + 1 \text{ od}, \sigma[a[0] := 1]) \]
Example

Consider program
\[
S \equiv a[0] := 1; a[1] := 0; \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}
\]
and a state \( \sigma \) with \( \sigma \models x = 0 \).

\[
\langle S, \sigma \rangle \xrightarrow{(i),(iii)} \langle a[1] := 0; \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma[a[0] := 1] \rangle
\]

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

Example

Consider program
\[
S \equiv a[0] := 1; a[1] := 0; \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}
\]
and a state \( \sigma \) with \( \sigma \models x = 0 \).

\[
\langle S, \sigma \rangle \xrightarrow{(ii),(iii)} \langle a[1] := 0; \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma[a[0] := 1] \rangle
\]

where \( \sigma' = [a[0] := 1][a[1] := 0] \).
Consider program
\[ S_1 \equiv y := x; \ y := (x - 1) \cdot x + y \]
and a state \( \sigma \) with \( \sigma \models x = 3 \). 

\[ \langle S_1, \sigma \rangle \xrightarrow{(i),(iii)} \langle y := (x - 1) \cdot x + y, \{x \mapsto 3, y \mapsto 3\} \rangle \xrightarrow{(i)} \langle E, \{x \mapsto 3, y \mapsto 9\} \rangle \]

Consider program
\[ S_3 \equiv y := x; \ y := (x - 1) \cdot x + y; \ \text{while} \ 1 \ \text{do} \ \text{skip} \ \text{od}. \]

\[ \langle S_3, \sigma \rangle \xrightarrow{(i),(iii)} \langle y := (x - 1) \cdot x + y; \ \text{while} \ 1 \ \text{do} \ \text{skip} \ \text{od}, \{x \mapsto 3, y \mapsto 3\} \rangle \xrightarrow{(vi)} \langle \text{skip; while} \ 1 \ \text{do} \ \text{skip} \ \text{od}, \{x \mapsto 3, y \mapsto 9\} \rangle \]

\[ \langle S_3, \sigma \rangle \xrightarrow{(i),(iii)} \langle \text{while} \ 1 \ \text{do} \ \text{skip} \ \text{od}, \{x \mapsto 3, y \mapsto 9\} \rangle \xrightarrow{(vi)} \langle \text{while} \ 1 \ \text{do} \ \text{skip} \ \text{od}, \{x \mapsto 3, y \mapsto 9\} \rangle \xrightarrow{(vi)} \ldots \]

### Computations of Deterministic Programs

**Definition.** Let \( S \) be a deterministic program.

(i) A **transition sequence** of \( S \) (starting in \( \sigma \)) is a finite or infinite sequence

\[ \langle S, \sigma \rangle = \langle S_0, \sigma_0 \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \ldots \]

(that is, \( \langle S_i, \sigma_i \rangle \) and \( \langle S_{i+1}, \sigma_{i+1} \rangle \) are in transition relation for all \( i \)).

(ii) A **computation (path)** of \( S \) (starting in \( \sigma \)) is a maximal transition sequence of \( S \) (starting in \( \sigma \)), i.e. infinite or not extendible.

(iii) A computation of \( S \) is said to

a) **terminate** in \( \sigma \) if and only if it is finite and ends with \( \langle E, \tau \rangle \),

b) **diverge** if and only if it is infinite.

\( S \) **can diverge from** \( \sigma \) if and only if a diverging computation starts in \( \sigma \).

(iv) We use \( \rightarrow^* \) to denote the transitive, reflexive closure of \( \rightarrow \).

**Lemma.** For each deterministic program \( S \) and each state \( \sigma \), there is exactly one computation of \( S \) which starts in \( \sigma \).
Definition.
Let $S$ be a deterministic program.

(i) The semantics of partial correctness is the function
$$
\mathcal{M}[S] : \Sigma \rightarrow 2^{\Sigma}
$$
with
$$\mathcal{M}[S](\sigma) = \{ \tau | \langle S, \sigma \rangle \rightarrow^{*} \langle E, \tau \rangle \}.
$$

(ii) The semantics of total correctness is the function
$$
\mathcal{M}_{\text{tot}}[S] : \Sigma \rightarrow 2^{\Sigma} \cup \{ \infty \}
$$
with \(\mathcal{M}_{\text{tot}}[S](\sigma) = \mathcal{M}[S](\sigma) \cup \{ \infty \ | \ S \text{ can diverge from } \sigma \} \).

$\infty$ is an error state representing divergence.

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

Example: $\mathcal{M}[S_1](\sigma) = \mathcal{M}_{\text{tot}}[S_1](\sigma) = \{ \tau | \tau(x) = \sigma(x) \land \tau(y) = \sigma(x)^2 \}$, $\sigma \in \Sigma$.

(Recall: $S_1 \equiv y := x; y := (x - 1) \cdot x + y$)

Content

- Formal Program Verification
  - Deterministic Programs
    - Syntax
    - Semantics
    - Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness,
    - total correctness.
  - Proof System PD
- The Verifier for Concurrent C
  - modular reasoning
  - return values / old values
- Assertions
Correctness of While-Programs

Correctness of Deterministic Programs

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

(i) The correctness formula
\[ \{ p \} S \{ q \} \]
holds in the sense of partial correctness, denoted by $\models \{ p \} S \{ q \}$, if and only if
\[ \mathcal{M}[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket. \]
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 $\models_{\text{tot}} \{ p \} S \{ q \}$, if and only if
\[ \mathcal{M}_{\text{tot}}[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket. \]
We say $S$ is totally correct wrt. $p$ and $q$. 
Example: Computing squares (of numbers 0, . . . , 27)

- **Pre-condition**: \( p \equiv 0 \leq x \leq 27 \).
- **Post-condition**: \( q \equiv y = x^2 \).

Program \( S_1 \):

\[
\begin{align*}
\text{int } y = x; \\
y = (x - 1) \times x + y;
\end{align*}
\]

\( \models \{ p \} S_1 \{ q \} \)

\( \models_{\text{tot}} \{ p \} S_1 \{ q \} \)

Program \( S_3 \):

\[
\begin{align*}
\text{int } x; \\
\text{while } (1): \\
y = x \times x; \\
\text{int } y = x \times (x-1) \times x + y;
\end{align*}
\]

\( \not\models \{ p \} S_3 \{ q \} \)

\( \not\models_{\text{tot}} \{ p \} S_3 \{ q \} \)

Program \( S_4 \):

\[
\begin{align*}
\text{int } x = \text{read_input}(); \\
y = x + (x - 1) \times x;
\end{align*}
\]

\( \models \{ p \} S_4 \{ q \} \)

\( \models_{\text{tot}} \{ p \} S_4 \{ q \} \)

\( \not\models \{ x \} \)

\( \not\models_{\text{tot}} \{ x \} \)

Example: Correctness

- By the example, we have shown
  \[
  \models \{ x = 0 \} S \{ x = 1 \}
  \]
  and
  \[
  \models_{\text{tot}} \{ x = 0 \} S \{ x = 1 \}.
  \]
  (because we only assumed \( \sigma \models x = 0 \) for the example, which is exactly the pre-condition.)

- We have also shown (= proved (!)):
  \[
  \models \{ x = 0 \} S \{ x = 1 \wedge a[x] = 0 \}.
  \]

- The correctness formula \( \{ x = 2 \} S \{ \text{true} \} \) does not hold for \( S \).
  (For example, if \( \sigma \models a[i] \neq 0 \) for all \( i > 2 \).)

- In the sense of partial correctness, \( \{ x = 2 \wedge \forall i \geq 2 \rightarrow a[i] = 1 \} S \{ \text{false} \} \) also holds.
Content

- Formal Program Verification
  - Deterministic Programs
    - Syntax
    - Semantics
    - Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness,
    - total correctness.

- Proof System PD
  - The Verifier for Concurrent C
    - modular reasoning
    - return values / old values

- Assertions

Proof-System PD
Proof-System PD (for sequential, deterministic programs)

Axiom 1: **Skip-Statement**

\[
\{p\} \text{skip} \{p\}
\]

Axiom 2: **Assignment**

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

Rule 3: **Sequential Composition**

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

Rule 4: **Conditional Statement**

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

Rule 5: **While-Loop**

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

Rule 6: **Consequence**

\[
p \rightarrow p_1, \{p_1\} S \{q_1\}, q_1 \rightarrow q, \{p\} S \{q\}
\]

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

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

(2)

(3)

We can prove \( \vdash \{ x \geq 0 \land y \geq 0 \} \; \text{DIV} \; \{ a \cdot y + b = x \land b < y \} \)
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:
In the following, we show

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

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

\[ P \equiv a \cdot y + b = x \land b \geq 0. \]

Proof of (1)

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

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

• (1) claims:
  \[ \vdash_D \{ 0 \cdot y + x = x \land x \geq 0 \} a := 0 \{ a \cdot y + x = x \land x \geq 0 \} \]
  by \( A2 \).

\[ a \cdot y + x \]

\[ p \]

\[ \left[ x := t \right] \]

\[ p[\]
Proof of (1)

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

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

• \[ \vdash_{PD} \{ a \cdot y + x = x \land x \geq 0 \} \ b := x \ \{ a \cdot y + b = x \land b \geq 0 \} \]
  by (A2),
  \[ \equiv P \]

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

\[ \square \]
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:

**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:
  \( (B ? s_1 : s_2)[u := t] \equiv (B[u := t] ? s_1[u := t] : s_2[u := t]) \)
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:

**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.: \( (q \land r)[u := t] \equiv q[u := t] \land r[u := t] \)
- quantifier: \( (\forall x : q)[u := t] \equiv \forall y : q[x := y][u := t] \)
  \( y \) fresh (not in \( q \), \( t \), \( u \), same type as \( x \).)
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:

**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: \( (B ? s_1 : s_2)[u := t] \equiv (B[u := t] \land s_1[u := t] \lor \neg s_2[u := t]) \).

**Indexed variable:**
- \( u \) plain or \( u \equiv b[t_1, \ldots, t_m] \) and \( a \neq b \):
  \( (a[s_1, \ldots, s_n])[u := t] \equiv a[s_1[u := t], \ldots, s_n[u := t]] \)
- \( u \) \( \equiv a[t_1, \ldots, t_m] \):
  \( (a[s_1, \ldots, s_n])[u := t] \equiv (\land_{i=1}^n s_i[u := t] = t_i \land t : a[s_1[u := t], \ldots, s_n[u := t]]) \)

**Formulae:**
- boolean expression \( p \equiv \land \)
  \( p[u := t] \equiv s[u := t] \)
- negation: \( \neg q[u := t] \equiv \neg(q[u := t]) \)
- conjunction etc.: \( (q \land r)[u := t] \equiv q[u := t] \land r[u := t] \)
- quantifier: \( (\forall x : q)[u := t] \equiv (\forall x : q(x := u))[u := t] \)

**Example Proof Cont’d**

```
(1) P \rightarrow P \rightarrow P \land (x \geq y \land y \geq 0) \\
(2) P \land (x \geq y \land y \geq 0) \\
(3) P \land (b \geq y) \land b = x \land b < y
```

In the following, we show

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

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

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

As loop invariant, we choose (creative act):

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

```
(\{A\} (p) skip [p] \{R3\} \( p \lor \{ r \} \lor \{ q \} \{R5\} (p \land \{ B \} S \{ p \} \{R6\} (p \land \{ B \} S \{ p \} \lor \{ p \} S \{ q \} \lor \{ q \} S \{ p \} \lor \{ q \} \rightarrow \{ p \} \rightarrow \{ q \} \{ p \} \lor \{ p \} \rightarrow \{ q \})
```
Proof of (2)

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

\[ \vdash_{PD} \{ (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} b := b - y \ \{ (a + 1) \cdot y + b = x \land b \geq 0 \} \]
by (A2),
Proof of (2)

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]

Proof of (2)

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]

Proof of (2)

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

\[ b := b - y; a := a + 1 \{P\} \]

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]

Proof of (2)

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]

Proof of (2)

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]

Proof of (2)

• (2) claims:
  \[ \vdash_{PD} \{(a+1) \cdot y + (b - y) = x \land (b - y) \geq 0\} \]
  by (A2),

\[ b := b - y; a := a + 1 \{P\} \]
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),

\[ \vdash_{PD} \{(a + 1) \cdot y + b = x \land b \geq 0 \} \ a := a + 1 \ \{ a \cdot y + b = x \land b \geq 0 \} \]
by (A2),

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

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

\[ \vdash_{PD} \{ P \land b \geq y \} b := b - y; \ a := a + 1 \ \{ P \} \]
by (R6).

Example Proof Cont’d

\[ \begin{array}{c}
| (1) \quad P \rightarrow P \text{ while } \forall x \forall y \exists k \forall a \exists b (x + y = a + 1 \lor y + x = b + k \land a < b) \\
| (2) \quad P \rightarrow P \text{ while } \forall x \forall y \forall k \forall a \exists b (x + y = a + 1 \lor y + x = b + k \land a < b) \\
| (3) \quad P \rightarrow P \text{ while } \forall x \forall y \forall k \forall a \exists b (x + y = a + 1 \lor y + x = b + k \land a < b) \\
| (4) \quad P \rightarrow P \text{ while } \forall x \forall y \forall k \forall a \exists b (x + y = a + 1 \lor y + x = b + k \land a < b) \\
\end{array} \]

In the following, we show

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

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

(3) \[ \vdash_{PD} \{ P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y \}, \]

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

\[ P \equiv a \cdot y + b = x \land b \geq 0 \]
Proof of (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) \( P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y. \)

and

\[
\begin{array}{l}
\{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} \\
\{ P \land (b \geq y) \} \ b := b - y; \ a := a + 1 \{ P \} \\
\{ P \land (b < y) \} \\
\{ a \cdot y + b = x \land b < y \}
\end{array}
\]

thus

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

and thus (since PD is sound) \( DIV \) 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 \( DIV \) is called with \( x \) and \( y \) such that \( x \geq 0 \land y \geq 0 \),
then (if \( DIV \) terminates) \( a \cdot y + b = x \land b < y \) will hold.
Once Again

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

\[
\begin{align*}
\{ x \geq 0 \land y \geq 0 \} \\
\{ 0 \cdot y + x = x \land x \geq 0 \}
\end{align*}
\]

- \( a := 0; \)

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

- \( b := x; \)

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

\[
\{ P \}
\]

- while \( b \geq y \) do

\[
\{ P \land b \geq y \}
\]

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

- \( b := b - y; \)

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

- \( a := a + 1 \)

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

\[
\{ P \}
\]

- od

\[
\{ P \land \neg (b \geq y) \}
\]

\[
\{ a \cdot y + b = x \land b < y \}
\]

Literature Recommendation

Programmverifikation
Sequentielle, parallele und verteilte Programme
Springer-Lehrbuch

Verification of Sequential and Concurrent Programs
Kees Doets
Frank de Boer
Erik-Jan Oudeg
Springer

31/47

32/47
Content

- Formal Program Verification
  - Deterministic Programs
    - Syntax
    - Semantics
    - Termination, Divergence
  - Correctness of deterministic programs
    - partial correctness,
    - total correctness.
  - Proof System PD

- The Verifier for Concurrent C
  - modular reasoning
  - return values / old values

- Assertions
- The Verifier for Concurrent C (VCC) basically implements Hoare-style reasoning.

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

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

### VCC Syntax Example

```c
#include <vcc.h>

int a, b;

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

\[
DIV \equiv a := 0; \ b := x; \ while \ b \geq y \ do \ b := b - y; \ a := a + 1 \ od \ \\
\{ x \geq 0 \land y \geq 0 \} \ DIV \{ x \geq 0 \land y \geq 0 \}
\]
**Interpretation of Results**

- VCC result: "verification succeeded"
- VCC result: "verification failed"
- Other case: "timeout" etc.
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 $\models \{p\} \text{DIV} \{q\}$.

- VCC result: "verification failed"

- Other case: "timeout" etc.

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.

- VCC result: "verification failed"

- Other case: "timeout" etc.
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:** $\models \{false\} \not{f} \{q\}$ always holds.
    That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

- VCC result: “verification failed”

- Other case: “timeout” etc.

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:** $\models \{false\} \not{f} \{q\}$ 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 $q$.

- Other case: “timeout” etc.
Interpretation of Results

- **VCC result: “verification succeeded”**
  - We can **only** conclude that the tool
    — under its interpretation of the C-standard, under its platform assumptions (32-bit), etc. —
    claims that there is a proof for \( \models \{ p \} \text{DIV} \{ q \} \).
  - May be due to an error in the tool! (That’s a **false negative** then.)
    Yet we can ask **for a printout of the proof** and check it manually
    (hardly possible in practice) or with other tools like interactive theorem provers.
  - **Note:** \( \models \{ \text{false} \} f \{ q \} \) **always** holds.
    That is, a **mistake** in writing down the pre-condition can make errors in the program go undetected!

- **VCC result: “verification failed”**
  - May be a **false positive** (wrt. the goal of finding errors).
    The tool **does not provide counter-examples** in the form of a computation path,
    it (only) gives **hints on input values** satisfying \( p \) and causing a violation of \( q \).
  - \( \rightarrow \) 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.**
• 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 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 \( s \)); those invariants may temporarily be violated when updating the data structure.
  - and much more.

Modular Reasoning
Modular Reasoning

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

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

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

\( p \) is called pre-condition and \( q \) is called post-condition of \( f \).

Example: if we have

- \{true\} read_number \{0 ≤ result < 10^8\}
- \{0 ≤ x ∧ 0 ≤ y\} add \{(old(x) + old(y) < 10^8 ∧ result = old(x) + old(y)) ∨ result < 0\}
- \{true\} display \{(0 ≤ old(sum) < 10^8 ⇒ "old(sum)") ∧ (old(sum) < 0 ⇒ "-E-"\}

we may be able to prove our pocket calculator correct.

![Pocket Calculator Example](image)

Return Values and Old Values

- For modular reasoning, it's often useful to refer in the post-condition to
  - the return value as \( \text{result} \),
  - the values of variable \( x \) at calling time as \( \text{old}(x) \).

- Can be defined using auxiliary variables:
  - Transform function
    \[
    T f() \{ \ldots ; \text{return expr}; \}
    \]
    (over variables \( V = \{ v_1, \ldots, v_n \} \); where \( \text{result}, v_i^{\text{old}} \notin V \)) into
    \[
    T f() \{
    v_1^{\text{old}} := v_1; \ldots ; v_n^{\text{old}} := v_n;
    \ldots ;
    \text{result} := \text{expr};
    \}
    \]
    return \( \text{result} \);
  - over \( V' = V \cup \{ v^{\text{old}} \mid v \in V \} \cup \{ \text{result} \} \).
  - Then \( \text{old}(x) \) is just an abbreviation for \( x^{\text{old}} \).
• Extend the syntax of deterministic programs by

\[ S := \cdots | \text{assert}(B) \]

• and the semantics by rule

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

(If the asserted boolean expression \( B \) does not hold in state \( \sigma \), 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.
Formal Verification:

- **Program verification** is another approach to software quality assurance.

- **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.

References
References

