Softwaretechnik / Software-Engineering

Lecture 17: Software Verification

2016-07-14

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
| VL 15 | • Introduction and Vocabulary |
| VL 16 | • Limits of Software Testing |
|       | • Glass-Box Testing |
|       |   • Statement-, branch-, term-coverage. |
| VL 17 | • Other Approaches |
|       |   • Model-based testing, |
|       |   • Runtime verification. |
| VL 18 | • Software quality assurance in a larger scope. |
|       | • Program Verification |
|       |   • partial and total correctness, |
|       |   • Proof System PD. |
|       | • Review |
**Content**

- Software quality assurance in a **larger scope**:
  - vocabulary,
  - fault, error, failure,
  - concepts of software quality assurance (next to testing)

- **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**
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)
**software quality assurance** – See: quality assurance.  
IEEE 610.12 (1990)

**quality assurance** –

(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.

(2) A set of activities designed to evaluate the process by which products are developed or manufactured.

IEEE 610.12 (1990)

**Note:** in order to trust a product, it can be built well, or proven to be good (at best: both) – both is QA in the sense of (1).
**Fault, Error, Failure**

**fault** - abnormal condition that can cause an element or an item to fail.

*Note:* Permanent, intermittent and transient faults (especially soft-errors) are considered.

*Note:* An intermittent fault occurs time and time again, then disappears. This type of fault can occur when a component is on the verge of breaking down or, for example, due to a glitch in a switch. Some systematic faults (e.g. timing marginalities) could lead to intermittent faults.

ISO 26262 (2011)

**error** - discrepancy between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition.

*Note:* An error can arise as a result of unforeseen operating conditions or due to a fault within the system, subsystem or component being considered.

*Note:* A fault can manifest itself as an error within the considered element and the error can ultimately cause a failure.

ISO 26262 (2011)

**failure** - termination of the ability of an element, to perform a function as required.

*Note:* Incorrect specification is a source of failure.

ISO 26262 (2011)

We want to avoid **failures**, thus we try to detect **faults** and **errors**.
Concepts of Software Quality Assurance

Software quality assurance

- Organisational
- Analytic
- Constructive

Project management

Software examination

- Non-mech.
- Semi-mech.
- Mechanical

Examination by humans

- Inspection
- Review
- Manual proof

Comput. aided human exam.

- Interactive prover
- E.g.: Interactive prover

Examination with computer

- Analyse
- Execute
- Prove
- Formal verification

Dynamic checking (test)

- Static checking
- Consistency checks
- E.g.: Lint

- Quantitative examination

E.g.: Code generation

Constructive software engineering

(Ludewig and Lichter, 2013)
Three Basic Approaches

- All computation paths satisfying the specification
- Expected outcomes \( S_{\text{oll}} \)

### Execution of \((\Sigma \times A)^\omega\)

- \( \in \)?
- \( \subseteq ? \)
- \( \subseteq ? \)

### Testing

- Input \( \rightarrow \) output

### Review

- Reviewer
- \( \subseteq ? \)

### Formal Verification

- Prove \( S \models \mathcal{I} \), conclude \( [S] \in [\mathcal{I}] \)

\[ \frac{\Sigma \times A}{S_{\text{oll}}} \]
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) \( \langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle \)

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

(iii) \( \frac{\langle S_1, \sigma \rangle \rightarrow \langle S_2, \tau \rangle}{\langle S_1; S, \sigma \rangle \rightarrow \langle S_2; S, \tau \rangle} \)

(iv) \( \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle, \text{ if } \sigma \models B, \)

(v) \( \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle, \text{ if } \sigma \not\models B, \)

(vi) \( \langle \text{while } B \text{ do } S \text{ od}, \sigma \rangle \rightarrow \langle S; \text{while } B \text{ do } S \text{ od}, \sigma \rangle, \text{ if } \sigma \models B, \)

(vii) \( \langle \text{while } B \text{ do } S \text{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle, \text{ if } \sigma \not\models B, \)

\( E \) 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)).
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 \( \sigma \models x = 0 \).

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

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

\[ \langle \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma' \rangle \xrightarrow{vi} \langle x := x + 1; \text{while } a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma' \rangle \]

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

\[ \langle S, \sigma \rangle \xrightarrow{(vii)} \langle E, \sigma'[x := 1] \rangle \]

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

\[ S_1 \equiv y := x; y := (x - 1) \cdot x + y \]

and a state \( \sigma \) with \( \sigma \models x = 3 \).

\[
\begin{align*}
\langle S_1, \sigma \rangle & \xrightarrow{(ii), (iii)} \langle y := (x - 1) \cdot x + y, \{x \mapsto 3, y \mapsto 3\} \rangle \\
& \xrightarrow{(ii)} \langle E, \{x \mapsto 3, y \mapsto 9\} \rangle
\end{align*}
\]

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

\[
\begin{align*}
\langle S_3, \sigma \rangle & \xrightarrow{(ii), (iii)} \langle y := (x - 1) \cdot x + y; \text{while } 1 \text{ do } \text{skip} \text{ od, } \{x \mapsto 3, y \mapsto 3\} \rangle \\
& \xrightarrow{(ii), (iii)} \langle \text{while } 1 \text{ do } \text{skip} \text{ od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
& \xrightarrow{(vi)} \langle \text{skip; while } 1 \text{ do } \text{skip} \text{ od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
& \xrightarrow{(i), (iii)} \langle \text{while } 1 \text{ do } \text{skip} \text{ od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
& \xrightarrow{(vi)} \ldots
\end{align*}
\]
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 $\tau$ 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 
$$
\mathcal{M}[S](\sigma) = \{ \tau \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \}.
$$

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

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

**Example:** 
$$
\mathcal{M}[S_1](\sigma) = \mathcal{M}_{tot}[S_1](\sigma) = \{ \tau \mid \tau(x) = \sigma(x) \land \tau(y) = \sigma(x)^2 \}, \quad \sigma \in \Sigma.
$$
(Recall: $S_1 \equiv y := x; y := (x - 1) \cdot x + y$)
Correctness of While-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 $\models_{tot} \{p\} S \{q\}$, if and only if

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

We say $S$ is totally correct wrt. $p$ and $q$. 
Example: Computing squares (of numbers 0, \ldots, 27)

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

**Program** \( S_1 \):

```
1 int y = x;
2 y = (x - 1) * x + y;
```

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

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

**Program** \( S_2 \):

```
1 int y = x;
2 int z; // uninitialised
3 y = ((x - 1) * x + y) + z;
```

\( \models \{p\} S_2 \{q\} \times \)

\( \models_{tot} \{p\} S_2 \{q\} \times \)

**Program** \( S_3 \):

```
1 int y = x;
2 y = (x - 1) * x + y;
3 while (1):
```

\( \models \{p\} S_3 \{q\} \checkmark \)

\( \models_{tot} \{p\} S_3 \{q\} \times \)

**Program** \( S_4 \):

```
1 int x = read_input();
2 int y = x + (x - 1) * x;
```

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

\( \models_{tot} \{p\} S_4 \{q\} \times \)
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 \( \models x = 0 \) for the example, which is exactly the precondition.)

- We have also shown (= proved (!)):

  \[ \models \{ x = 0 \} S \{ x = 1 \land a[x] = 0 \}. \]

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

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

---

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 \( \models x = 0 \).

\[ (S, \sigma) \xrightarrow{(v)(vi)} (E, \sigma') \]

where \( \sigma' = \sigma[a[0] := 1][a[1] := 0] \).
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**
\[
{\{p\} S_1 \{r\}, \{r\} S_2 \{q\}} \quad \frac{}{{\{p\} S_1 ; S_2 \{q\}}}
\]

**Rule 4: Conditional Statement**
\[
\begin{align*}
{\{p \land B\} S_1 \{q\}, \{p \land \neg B\} S_2 \{q\},} \\
{\{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}
\end{align*}
\]

**Rule 5: While-Loop**
\[
{\{p \land B\} S \{p\}} \quad \frac{}{{\{p\} \text{ while } B \text{ do } S \text{ od } \{p \land \neg B\}}}
\]

**Rule 6: Consequence**
\[
{p \rightarrow p_1, \{p_1\} S \{q_1\}, q_1 \rightarrow q} \quad \frac{}{{\{p\} S \{q\}}}
\]

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

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

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

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

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

\[ \text{(1)} \]

\[ P \to P, \]

\[ \{ P \} \text{while } B^D \text{ do } S^D_1 \text{ od } \{ P \land \neg(B^D) \}, \]

\[ \{ P \} \text{while } B^D \text{ do } S^D_1 \text{ od } \{ q^D \} \]

\[ \text{(2)} \]

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

\[ {P \land (B^D) \} S^D_1 \{ P \} } \]

\[ (R5) \]

\[ P \rightarrow p, \]

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

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

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

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

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

\[ (R3) \]

\[ (R4) \]

\[ (R5) \]

\[ (R6) \]

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

\[ \{ p \} S \{ q \} \]
Example Proof

DIV ≡ \( a := 0; b := x; \text{ while } b \geq y \text{ do } \begin{array}{l}
\quad b := b - y; \quad a := a + 1
\end{array} \text{ od} \)

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

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

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

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

(2) \[ \begin{array}{l}
\text{P \rightarrow P,} \\
\{ \text{P while } b \geq y \text{ do } b := b - y; \quad a := a + 1 \text{ od } \{ P \land \neg (b \geq y) \},}
\end{array} \]

(3) \[ \begin{array}{l}
P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y
\end{array} \]

(4) \[ \begin{array}{l}
\{ P \land (b \geq y) \} \quad \text{b := b - y; a := a + 1} \{ P \}
\end{array} \]

(5) \[ \begin{array}{l}
P \land (b \geq y) \rightarrow a \cdot y + b = x \land b < y
\end{array} \]

(6) \[ \begin{array}{l}
\{ x \geq 0 \land y \geq 0 \} \text{ a := 0 \; b := x} \{ P \}
\end{array} \]

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

(R3) \[ \begin{array}{l}
\{ p \} \text{ S}_1 \{ r \}
\{ r \} \text{ S}_2 \{ q \}
\{ p \} \text{ S}_1; \text{ S}_2 \{ q \}
\end{array} \]

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

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

(R4) \[ \begin{array}{l}
\{ p \land B \} \text{ S}_1 \{ q \}
\{ p \land \neg B \} \text{ S}_2 \{ q \}
\{ p \} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{ q \}
\end{array} \]

(R6) \[ \begin{array}{l}
p \rightarrow p_1, \{ p_1 \} \text{ S} \{ q_1 \}, \quad q_1 \rightarrow q
\{ p \} \text{ S} \{ q \}
\end{array} \]
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) \( \models P \land \neg (b \geq y) \rightarrow a \cdot y + b = x \land b < y \).

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

\[ P \equiv a \cdot y + b = x \land b \geq 0 \]
Proof of (1)

- (1) claims:

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

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

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

by (A2),

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

\[ p[b:=x] \]

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

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

- \[ \vdash \mathcal{PD} \{ a \cdot y + x = x \land x \geq 0 \} \ a := 0 \ {a \cdot y + x = x \land x \geq 0} \quad \text{by (A2)}, \]

- \[ \vdash \mathcal{PD} \{ a \cdot y + x = x \land x \geq 0 \} \ b := x \ {a \cdot y + b = x \land b \geq 0} \quad \equiv P \quad \text{by (A2)}, \]

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

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

The rule ‘Assignment’ uses (syntactical) sub\text{\textbf{tu}}\text{\textion}: \{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:
  \((B ? s_1 : s_2)[u := t] \equiv (B[u := t] ? s_1[u := t] : s_2[u := t]).\)

- 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]::
  \(a[s_1, \ldots, s_n])[u := t] \equiv (\land_{i=1}^n s_i[u := t] = t_i ? t : a[s_1[u := t], \ldots, s_n[u := t]]).\)

Formulae:

- boolean expression \(p \equiv s\):
  \(p[u := t] \equiv s[u := t]\)

- negation:
  \((\neg q)[u := t] \equiv \neg(q[u := t])\)

- conjunction etc.:
  \((q \land r)[u := t] \equiv q[u := t] \land r[u := t]\)

- quantifier:
  \((\forall x : q)[u := t] \equiv \forall y : q[x := y][u := t]\)
  \(y\) fresh (not in \(q, t, u\), same type as \(x\)).
Proof of (2)

- (2) claims:

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

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

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

by (A2).
Proof of (2)

- **(2)** claims:

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

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

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

by (A2),

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

by (A2).
Proof of (2)

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

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

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

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

Using \( P \land b \geq y \rightarrow (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \) and \( P \to P \) we obtain,
\[
\vdash_{PD} \{ P \land b \geq y \} \ b := b - y; \ a := a + 1 \ \{ P \}
\]
by (R6).
Proof of (3)

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

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

Proof: easy.
We have shown:

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

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

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

and

\[
\begin{align*}
\{ x \geq 0 \land y \geq 0 \} & a := 0; \ b := x \ \{P\}, & \{ P \land (b \geq y) \} & b := b - y; \ a := a + 1 \ \{P\} & (R5) \quad \{ P \land \neg(b \geq y) \} & \rightarrow & a \cdot y + b = x \land b < y & (R6) \\
\{ P \} & \text{while } b \geq y \ do \ b := b - y; \ a := a + 1 \ \{P \land \neg(b \geq y)\}, & \{ P \} & \text{while } b \geq y \ do \ b := b - y; \ a := a + 1 \ \{a \cdot y + b = x \land b < y\} & (R5) \quad \{ P \land \neg(b \geq y) \} & \rightarrow & a \cdot y + b = x \land b < y & (R6) \\
\{ x \geq 0 \land y \geq 0 \} & a := 0; \ b := x; \ \text{while } b \geq y \ do \ b := b - y; \ a := a + 1 \ \{a \cdot y + b = x \land b < y\} & (R3)
\end{align*}
\]

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) \textit{DIV} is \textbf{partially correct} wrt.

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

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

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

  $\{x \geq 0 \land y \geq 0\}$
  $\{0 \cdot y + x = x \land x \geq 0\}$

  - $a := 0;$
    $\{a \cdot y + x = x \land x \geq 0\}$
    $\{P\}$

  - 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

  $\{P \land \neg(b \geq y)\}$
  $\{a \cdot y + b = x \land b < y\}$

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

Apt - Olderog

Programmverifikation
Sequentielle, parallele und verteilte Programme

Springer-Lehrbuch

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

Springer
Assertions
Assertions

- Extend the **syntax** of **deterministic programs** by

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

- 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 \( \{true\} x := 0; \text{assert}(x = 27) \{true\} \) in PD.
Modular Reasoning
We can add another rule for calls of functions $f : F$ (simplest case: only global variables):

$$
\begin{align*}
\text{(R7)} & \frac{\{p\} F \{q\}}{\{p\} f() \{q\}} \\
\end{align*}
$$

“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

- $\{true\}$ read_number $\{0 \leq result < 10^8\}$
- $\{0 \leq x \land 0 \leq y\}$ add $\{(old(x) + old(y) < 10^8 \land result = old(x) + old(y)) \lor result < 0\}$
- $\{true\}$ display $\{(0 \leq old(x) < 10^8 \implies ”old(x)” \land (old(x) < 0 \implies ”-E-“))\}$

we may be able to prove our pocket calculator correct.
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 old\((x)\).

- Can be defined using auxiliary variables:
  - Transform function
    
    \[
    T f() \{ \ldots; \text{return } expr; \}
    \]

    (over variables \( V = \{v_1, \ldots, v_n\}, result, v_i^{\text{old}} \notin V \) to
     \[
     T f() \{
     v_1^{\text{old}} := v_1; \ldots; v_n^{\text{old}} := v_n;
     \ldots;
     result := expr;
     \text{return } result;
     \}
     \]

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

- Then old\((x)\) is just an abbreviation for \(x^{\text{old}}\).
The Verifier for Concurrent C
The **Verifier for Concurrent C (VCC)** basically implements Hoare-style reasoning.

**Special syntax:**

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

**Special expressions:**

- `thread_local(&v)` – no other thread writes to variable `v` (in pre-conditions)
- `old(v)` – the value of `v` when procedure was called (useful for post-conditions)
- `result` – return value of procedure (useful for post-conditions)
**VCC Syntax Example**

```c
#include <vcc.h>

int a, b;

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

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

\(\{x \geq 0 \land y \geq 0\}\ DIV \{x \geq 0 \land y \geq 0\}\)
Example program DIV:  http://rise4fun.com/Vcc/4Kqe
Interpretation of Results

- **VCC says: “verification succeeded”**
  
  We can only conclude that the tool
  – under its interpretation of the C-standard, under its platform assumptions (32-bit), etc.
  – “thinks” that it can prove $\models \{p\} \text{DIV} \{q\}$.
  
  Can be due to an error in the tool! (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 says: “verification failed”**

  - May be a false positive.
    
    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: $\rightarrow$ make pre-condition $p$ or loop-invariant(s) stronger, and try again.

  - Other case: “timeout” etc. – completely inconclusive outcome.
For the exercises, we use VCC only for **sequential, single-thread programs**.

VCC checks a number of **implicit assertions**:
- **no arithmetic overflow** in expressions (according to C-standard),
- array-out-of-bounds access,
- NULL-pointer dereference,
- and many more.

VCC also supports:
- **concurrency**: different threads may write to shared global variables; VCC can check whether concurrent access to shared variables is properly managed;
- **data structure invariants**: we may declare invariants that have to hold for, e.g., records (e.g. the length field $l$ is always equal to the length of the string field $str$); those invariants may **temporarily** be violated when updating the data structure.
- and much more.

Verification **does not always succeed**:
- The backend SMT-solver may not be able to discharge proof-obligations (in particular non-linear multiplication and division are challenging);
- In many cases, we need to provide **loop invariants** manually.
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.
References
References


