Softwaretechnik / Software-Engineering

Lecture 17: Software Verification

2018-07-12

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
**Topic Area Code Quality Assurance: Content**

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

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

- **Other Approaches**
  - Model-based testing,
  - Runtime verification.

- **Program Verification**
  - Partial and total correctness,
  - Proof System PD.

- Review
Testing, Review, Verification Illustrated

all computation paths satisfying the specification

expected outcomes \( S_{\text{oll}} \)

\( \in? \)

\( \subseteq? \)

\( \subseteq? \)

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

Reviewer

review

input \(\rightarrow\) output

Testing

Review

Formal Verification

prove \( S \models \mathcal{S} \), conclude \([S] \in [\mathcal{S}]\)
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
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) \( \langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle \) — empty program

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

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

(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

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

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

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

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

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

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

\[
\langle \textbf{while} a[x] \neq 0 \textbf{do} x := x + 1 \textbf{od}, \sigma'[x := 1] \rangle \xrightarrow{(vii)} \langle E, \sigma'[x := 1] \rangle
\]

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 \( \sigma \models x = 3 \).

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

\[
\xrightarrow{(ii)} \langle E, \{x \mapsto 3, y \mapsto 9\} \rangle
\]

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

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

\[
\xrightarrow{(ii),(iii)} \langle \text{while } 1 \text{ do } \text{skip} \text{ od}, \{x \mapsto 3, y \mapsto 9\} \rangle
\]

\[
\xrightarrow{(vi)} \langle \text{skip}; \text{ while } 1 \text{ do } \text{skip} \text{ od}, \{x \mapsto 3, y \mapsto 9\} \rangle
\]

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

\[
\xrightarrow{(vi)} \ldots
\]
**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 $\mathcal{M}[S](\sigma) = \{ \tau \mid \langle S, \sigma \rangle \xrightarrow{*} \langle E, \tau \rangle \}$.

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

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

\[\models ? \left\{ p \right\} S_1 \left\{ q \right\} \checkmark
\]

\[\models_{\text{tot}} ? \left\{ p \right\} S_1 \left\{ q \right\} \checkmark
\]

**Program \( S_2 \):**

\[
\begin{align*}
1 & \text{int } y = x; \\
2 & y = (x - 1) \ast x + y; \\
3 & \textbf{while } (1);
\end{align*}
\]

\[\models ? \left\{ p \right\} S_2 \left\{ q \right\} \checkmark
\]

\[\models_{\text{tot}} ? \left\{ p \right\} S_2 \left\{ q \right\} \times
\]

**Program \( S_3 \):**

\[
\begin{align*}
1 & \text{int } y = x; \\
2 & \text{int } z; \quad \text{// uninitialised} \\
3 & y = ((x - 1) \ast x + y) + z;
\end{align*}
\]

\[\models ? \left\{ p \right\} S_3 \left\{ q \right\} \times \quad \text{e.g. } z=1
\]

\[\models_{\text{tot}} ? \left\{ p \right\} S_3 \left\{ q \right\} \times
\]

**Program \( S_4 \):**

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

\[\models ? \left\{ p \right\} S_4 \left\{ q \right\} \checkmark
\]

\[\models_{\text{tot}} ? \left\{ p \right\} S_4 \left\{ q \right\} \times
\]
**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 \( \sigma \models x = 0 \) 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 \( S \). (in the sense of total correctness.
    (For example, if \( \sigma \models a[i] \neq 0 \) for all \( i > 2 \).

  - In the sense of **partial correctness**, \( \{ x = 2 \land \forall i \geq 2 \cdot a[i] = 1 \} S \{ \text{false} \} \) 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
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
\[
\frac{\{p \land B\} S_1 \{q\}, \{p \land \lnot B\} S_2 \{q\},}{\{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}
\]

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

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

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 \(\models \{p\} S \{q\}\).
(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:
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 \( P \equiv a \cdot y + b = x \land b \geq 0 \).

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

\[ p \left[ u := t \right] \]

\[ (a \cdot y + x = x \land x \geq 0) \; [a := 0] \]

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

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

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

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

\[ \equiv P \]

• thus, \( \vdash_{PD} \{ 0 \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
\[ \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} a := 0; \ b := x \{ P \} \]
by (R6).
The rule ‘Assignment’ uses (syntactical) substitution: \( \{p[u := t]\} u := t \{p\} \)

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

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

\[
\begin{align*}
\lambda x \cdot x \cdot [x := u + 3] & \mapsto \lambda x \cdot u + 3 \\
\lambda x \cdot \forall x \cdot b \cdot x \cdot [x := u + 3] & \mapsto \ ? \\
\lambda x \cdot \forall z \cdot b \cdot z \cdot \lambda z \cdot \forall z \cdot b \cdot z \ & \mapsto \lambda z \cdot u + 3 \cdot \forall z \cdot b \cdot z
\end{align*}
\]
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** \( \text{op} \), terms \( s_i \):
  \( \text{op}(s_1, \ldots, s_n)[u := t] \equiv \text{op}(s_1[u := t], \ldots, s_n[u := t]) \).
- **Conditional expression**:
  \( (B \ ? \ s_1 : s_2)[u := t] \equiv (B[u := t] \ ? \ s_1[u := t] : s_2[u := t]) \)
- **Indexed variable**, \( u \) plain or \( u \equiv b[t_1, \ldots, t_m] \) and \( a \neq b \):
  \( (a[s_1, \ldots, s_n])[u := t] \equiv a[s_1[u := t], \ldots, s_n[u := t]] \)
- **Indexed variable**, \( u \equiv a[t_1, \ldots, t_m] \):
  \( (a[s_1, \ldots, s_n])[u := t] \equiv (\bigwedge_{i=1}^n s_i[u := t] = t_i \ ? \ t : a[s_1[u := t], \ldots, s_n[u := t]]) \)

### Formulae:
- **Boolean expression** \( p \equiv s \):
  \( p[u := t] \equiv s[u := t] \)
- **Negation**:
  \( (\neg q)[u := t] \equiv \neg(q[u := t]) \)
- **Conjunction etc.**:
  \( (q \land r)[u := t] \equiv q[u := t] \land r[u := t] \)
- **Quantifier**:
  \( (\forall x : q)[u := t] \equiv \forall y : q[x := y][u := t] \)
  \( y \) fresh (not in \( q, t, u \)), same type as \( x \).
In the following, we show

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

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

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

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

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

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

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

- \[ \vdash_{PD} \{(a + 1) \cdot y + b = x \land b \geq 0\} \ a := a + 1 \ \{(a + 1) \cdot y + b = x \land b \geq 0\} \]
  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).
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) $\models P \land \neg(b \geq y) \to a \cdot y + b = x \land b < y$.

As loop invariant, we choose (creative act!): $P \equiv a \cdot y + b = x \land b \geq 0$.

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

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

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

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

(R5) $\{p \land B\} S \{p\}$

(R6) $\{p \land B\} \text{while } B \text{ do } S \text{ od } \{p \land \neg B\}$

$\Rightarrow p \to p_1, \{p_1\} S \{q_1\}, q_1 \to q \{p\} S \{q\}$
**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.
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

\[
\begin{align*}
\{x \geq 0 \land y \geq 0\} & \ a := 0; \ b := x \ \{P\}, \\
\{P \land (b \geq y)\} & \ b := b - y; \ a := a + 1 \ \{P\}, \\
P \land (b \geq y) & \rightarrow a \cdot y + b = x \land b < y
\end{align*}
\]

thus

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

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

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

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

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

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

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

- and the **semantics** by rule

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

(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:

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

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

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

“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

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

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

• Can be defined using **auxiliary variables**:
  • Transform function
    \[
    T \ f() \ {\ldots} \ \text{return} \ \text{expr};
    \]
    (over variables \( V = \{v_1, \ldots, v_n\}; \text{where \( \text{result}, v_i^{old} \notin V \) } \) into
    \[
    T \ f() \ \{
    v_1^{old} := v_1; \ldots; \ v_n^{old} := v_n;
    \ldots;
    \text{result} := \text{expr};
    \text{return} \ \text{result};
    \}
    \]
    over \( V' = V \cup \{v^{old} | v \in V\} \cup \{\text{result}\} \).

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

- `\texttt{thread\_local}(&v)` — no other thread writes to variable \( v \) (in pre-conditions)

- `\texttt{old}(v)` — the value of \( v \) when procedure was called (useful for post-conditions)

- `\texttt{result}` — return value of procedure (useful for post-conditions)
#include <vcc.h>

int a, b;

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

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\}
VCC

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

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

- VCC result: “verification failed”
  - May be a false positive (wrt. the goal of finding errors).
    The tool does not provide counter-examples in the form of a computation path,
    it (only) gives hints on input values satisfying $p$ and causing a violation of $q$.
  - → try to construct a (true) counter-example from the hints.
    or: make loop-invariant(s) (or pre-condition $p$) stronger, and try again.

- Other case: “timeout” etc. — completely inconclusive outcome.
VCC Features

- For the exercises, we use VCC only for **sequential, single-thread programs**.
- VCC checks a number of **implicit assertions**:
  - **no arithmetic overflow** in expressions (according to C-standard),
  - **array-out-of-bounds access**, 
  - **NULL-pointer dereference**, 
  - and many more.

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

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

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
