Softwaretechnik / Software-Engineering

Lecture 17: Software Verification

2016-07-14

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Code Quality Assurance: Content

- Introduction and Vocabulary
- Limits of Software Testing
- Glass-Box Testing
  - Statement-, branch-, term-coverage.
- Other Approaches
  - Model-based testing,
  - Runtime verification.
- 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

Software Quality Assurance
Vocabulary


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.

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).
We want to avoid failures, thus we try to detect faults and errors.
Three Basic Approaches

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) - \( \sigma : V \rightarrow \mathcal{D}(V) \)

(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) \( \langle S_1, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle \) (empty program)

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

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

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

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

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

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

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

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

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

\[
\langle E, \sigma'[x := 1] \rangle
\]

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

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

Consider program  

\[ S_2 \equiv y := x; \ y := (x - 1) \cdot x + y; \ \text{while } 1 \ \text{do skip od.} \]

\[ \langle S_2, \sigma \rangle \xrightarrow{(i),(iii)} \langle y := (x - 1) \cdot x + y; \ \text{while } 1 \ \text{do skip od, } \{ x \mapsto 3, y \mapsto 3 \} \rangle \]

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

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

\[ \xrightarrow{(vi)} \ldots \]

Computation 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

$$\mathcal{M}_{tot}[S] : \Sigma \rightarrow 2^\Sigma \cup \{\infty\}$$

with

$$\mathcal{M}_{tot}[S](\sigma) = \mathcal{M}[S](\sigma) \cup \{\infty | S \text{ can diverge from } \sigma\}.$$  

$\infty$ is an error state representing divergence.

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

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

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

$$\mathcal{M}(S)((p)) \subseteq \llbracket q \rrbracket.$$

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

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

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 $S_1$:**

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

$\models_{\text{tot}} \{p\} S_1 \{q\}$

---

**Program $S_2$:**

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

$\models_{\text{tot}} \{p\} S_2 \{q\}$

---

**Program $S_3$:**

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

$\models_{\text{tot}} \{p\} S_3 \{q\}$

---

**Program $S_4$:**

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

$\models_{\text{tot}} \{p\} S_4 \{q\}$
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 \( S \).
  (For example, if \( \sigma \models a[i] \neq 0 \) for all \( i > 2 \))

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

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

**Rule 4: Conditional Statement**
\[
\{ p \land B \} S_1 \{ q \}, \{ p \land \lnot B \} S_2 \{ q \}, \{ p \} \text{if } B \text{ then } S_1 \text{ else } S_2 \{ q \}
\]

**Rule 5: While-Loop**
\[
\{ p \land B \} S \{ p \}
\]

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

**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**

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

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

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

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

\[
(1) \quad P \rightarrow P, \quad \{ P \} \text{while } B^D \text{ do } S^D \{ P \}, \quad \text{while } B^D \text{ do } S^D \text{ od } \{ q^D \}
\]

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

\[
(3) \quad P \land (B^D) \Rightarrow q^D
\]

\[
(4) \quad (R3) \quad (R5) \quad (R6)
\]

\[
(5) \quad (R3) \quad \{ p \} S^D \{ p \}, \quad \text{while } B^D \text{ do } S^D \text{ od } \{ q^D \}
\]

\[
(6) \quad (A0) \quad \{ p \} \text{skip} \{ p \}
\]

\[
(7) \quad (R3) \quad \{ p \} S_1 \{ r \}, \{ r \} S_2 \{ q \}
\]

\[
(8) \quad (R5) \quad \{ p \} \text{if } B \text{ then } S_1 \text{ else } S_2 \{ q \}
\]

\[
(9) \quad (R6) \quad p \rightarrow p_1, \{ p_1 \} S \{ q_1 \}, q_1 \rightarrow q
\]

\[
(10) \quad \{ p \} \text{if } B \text{ then } S_1 \text{ else } S_2 \{ q \}
\]

\[
(11) \quad \{ p \} \text{ if and only if } \models \{ p \} S \{ q \} \]

Example Proof

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

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

We can prove \[\vdash \{x \geq 0 \land y \geq 0\} DIV \{a \cdot y + b = x \land b < y\}\] by showing \[\vdash_{PD}\{x \geq 0 \land y \geq 0\} DIV \{a \cdot y + b = x \land b < y\}\] i.e., derivability in PD:

\[\frac{P \land \neg(b \geq y) \Rightarrow a \cdot y + b = x \land b < y}{P \Rightarrow a \cdot y + b = x \land b < y}\]

Example Proof Cont’d

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

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

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

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

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

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

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

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

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).

\[ \square \]

Substitution

The rule ‘Assignment’ uses (syntactical) substitution: \( \{ u[= t] \} u := t \{ p \} \)

(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:
  \[ (\forall x : q)[u := t] \equiv (\forall y : q[x := y])[u := t] \]

**Indexed terms:**
- indexed variable, 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, \( u \equiv a[t_1, \ldots, t_m] \):
  \[ (a[s_1, \ldots, s_n])[u := t] \equiv (\Lambda_{i=1}^{n} s_i[u := t] = t_i ? t : a[s_1[u := t], \ldots, s_n[u := t]]) \]
Proof of (2)

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

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

Proof of (2)

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

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

(2) claims:

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

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

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

\[ \vdash PD \{ P \land b \geq y \} b := b - y; \quad a := a + 1 \{ P \} \]
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.
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) \( \vdash P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y \).

and

\[
\begin{align*}
(P \land b \geq y) \land \neg y & \leq a + 1 (P) \Rightarrow P \land \neg(b \geq y) \rightarrow a \cdot y + b = x \land b < y \\
\{x \geq 0 \land y \geq 0\} a & := 0, \ b := x; \ 	ext{while} \ y \geq b \ 	ext{do} \ b := b - y; \ a := a + 1 \ 	ext{od} \{ a \cdot y + b = x \land b < y \} = \text{DIV}
\end{align*}
\]

thus

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

and thus (since PD is sound) \( \text{DIV} \) 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 \( \text{DIV} \) 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

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

\[
\begin{align*}
\{x \geq 0 \land y \geq 0\} & \\
\{0 \cdot y + x = x \land x \geq 0\} & \\
\{a := 0;\} & \\
\{a \cdot y + x = x \land x \geq 0\} & \\
\{b := x;\} & \\
\{a \cdot y + b = x \land b \geq 0\} & \\
\{P\} &
\end{align*}
\]

while \( b \geq y \) do

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

od

\[
\begin{align*}
\{P \land \neg(b \geq y)\} & \\
\{a \cdot y + b = x \land b < y\} &
\end{align*}
\]

\[
\begin{align*}
(A1) \{p\} \text{skip } \{q\} & \\
(A2) \{v := t\} \{w := t\} \{p\} & \\
(R3) \{p\} S_1 (r) \cdot (r) S_2 (q); & \\
\{p\} S_1; S_2 (q) & \\
(R4) \{p \land B\} S_1 (q); \{p \land \neg B\} S_2 (q); \{q\} & \\
\{p\} \text{if } B \text{ then } S_1 \text{ else } S_2; \{q\} & \\
(R5) \{p \land B\} S (q); & \\
\{p\} \text{while } B \text{ do } S \text{ od } \{p \land \neg B\} & \\
(R6) p \rightarrow p_1; \{p_1\} S (q_1); q \rightarrow q & \\
\{p\} S (q) &
\end{align*}
\]
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 \( \{ \text{true} \} x := 0; \text{assert}(x = 27) \{ \text{true} \} \) in PD.

Modular Reasoning
Modular Reasoning

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

\[
(R7) \quad \{ p \} F \{ q \} \quad \{ p \} f() \{ q \}
\]

“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

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

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 \( \text{result} \),
  - 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 \( V = \{ v_1, \ldots, v_n \}, \text{result}, v_i^{\text{old}} \not\in V \) to
    \[
    T f() \{
    \begin{align*}
    v_1^{\text{old}} & := v_1; \ldots; v_n^{\text{old}} := v_n; \\
    \ldots & ; \\
    \text{result} & := \text{expr;} \\
    \text{return} \; \text{result};
    \end{align*}
    \}
    \]
    over \( V' = V \cup \{v^{\text{old}} | v \in V\} \cup \{ \text{result} \} \).
  - Then \( \text{old}(x) \) is just an abbreviation for \( x^{\text{old}} \).
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:**
  - \	hread_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 = x && b = y)
  // (writes &a)
  // (writes &b)
  {
    a = 0;
    b = x;
    while (b >= y)
      // (invariant a = x && b = y)
      {
        b = b - y;
        a = a + 1;
      }
  }
```

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

\{ x \geq 0 \land y \geq 0 \} DIV \{ x \geq 0 \land y \geq 0 \}

VCC Web-Interface

Example program \( DIV \): [http://rise4fun.com/Vcc/4Kqe](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 $\vdash \{p\} \text{DIV} \{q\}$.

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: $\vdash \{false\} \not\vdash \{q\}$ 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 $q$.

→ try to construct a (true) counter-example from the hints.
or: → make pre-condition $p$ or loop-invariant(s) 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.
• 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.
Tell Them What You’ve Told Them...

- 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


