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**
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
Deterministic Programs

Syntax:
\[ S ::= \text{skip} \mid u := t \mid S_1; S_2 \mid \text{if} B \text{ then } S_1 \text{ else } S_2 \text{ fl } \mid \text{while} B \text{ do } S_1 \text{ od} \]
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) \(- \sigma : V \rightarrow D(V)\)

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

Note: the first component of \( (S, \sigma) \) is a program (structural operational semantics (SOS)).
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 \( \sigma \models x = 0 \).

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

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

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

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

---

**Another Example**

Consider program

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

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

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

\[ \langle E, \{x \mapsto 3, y \mapsto 9\} \rangle \]

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

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

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

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

\[ \ldots \]
Definition. Let \( S \) be a deterministic program.

(i) A \textbf{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 \textbf{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
\begin{itemize}
  \item[a)] terminate in \( \tau \) if and only if it is finite and ends with \( \langle E, \tau \rangle \),
  \item[b)] diverge if and only if it is infinite.
\end{itemize}

\( S \) 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 \( \sigma \).

Input/Output Semantics of Deterministic Programs

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

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

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

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

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

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

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

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

(hoare triple)

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

$$M[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket.$$

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

$$M_{\text{tot}}[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket.$$

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

Example: Computing squares (of numbers $0, \ldots, 27$)

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

Program $S_1$:

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

$\models^? \{p\} S_1 \{q\}$ ✓

$\models^?_{\text{tot}} \{p\} S_1 \{q\}$ ✓

Program $S_2$:

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

$\models^? \{p\} S_2 \{q\}$ ✓

$\models^?_{\text{tot}} \{p\} S_2 \{q\}$ ✗

Program $S_3$:

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

$\models^? \{p\} S_3 \{q\}$ ✗

$\models^?_{\text{tot}} \{p\} S_3 \{q\}$ ✗

Program $S_4$:

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

$\models^? \{p\} S_4 \{q\}$ ✓

$\models^?_{\text{tot}} \{p\} S_4 \{q\}$ ✗
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 \( S \). (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

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

**Rule 5:** While-Loop

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

**Rule 6:** Consequence

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

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

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

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

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

\begin{align*}
\text{(1)} & \quad \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \text{ while } b \geq y \text{ do } b := b - y; \ a := a + 1 \text{ od} \{ P \}\text{ (R3)} \\
\text{(2)} & \quad \vdash_{PD} \{ P \land (b \geq y) \} \text{ while } b \geq y \text{ do } b := b - y; \ a := a + 1 \text{ od} \{ P \}\text{ (R4)} \\
\text{(3)} & \quad \vdash_{PD} \{ P \land (b \geq y) \} \text{ while } b \geq y \text{ do } b := b - y; \ a := a + 1 \text{ od} \{ P \}\text{ (R5)}
\end{align*}

\( \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \) \( a := 0; \ b := x \{ P \} \text{ (R1)} \)

\( \vdash_{PD} \{ P \land b \geq y \} \) \( b := b - y; \ a := a + 1 \{ P \} \text{ (R2)} \)

\( \vdash_{PD} \{ P \land \neg (b \geq y) \} \rightarrow a \cdot y + b = x \land b < y \text{ (R6)} \)

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

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

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

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

**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) ⊢_{P} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \},

(2) ⊢_{P} \{ 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!):

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

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

\[ (p \land b \geq y) \ b := b - y; \ a := a + 1 \{ P \} \]

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

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

\[ (a + 1) \cdot y + b = x \land b \geq 0 \] by (A2),

\[ (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \] by (R3),

\[ \text{using } P \land b \geq y \to (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \text{ and } P \to P' \text{ we obtain} \]

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

by (R6).
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 \(3\)**

\(3\) claims

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

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

and

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

thus

\[
\vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x; \ while \ b \geq y \ do \ b := b - y; \ a := a + 1 \ od \{ a \cdot y + b = x \land b < y \}
\]

\[\equiv \text{DIV}\]

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

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

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

while \( b \geq y \)

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

\( \text{od} \)

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

---

\[\text{R5} \]

\[\text{R6} \]

\[\text{R1} \]

\[\text{R2} \]

\[\text{R3} \]

\[\text{R4} \]

\[\text{R0} \]

\[\text{R1} \]

\[\text{R2} \]

\[\text{R3} \]

\[\text{R4} \]

\[\text{R5} \]

\[\text{R6} \]

\[\text{R0} \]
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**

- 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

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

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

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 $V = \{ v_1, \ldots, v_n \}$; where result, $v^\text{old} \notin V$) into
    \[
    T f() \{
    v^\text{old} := v_1; \ldots; v^\text{old} := v_n;
    \ldots ;
    \text{result} := expr;
    \text{return } \text{result};
    \}
    \]
    over $V' = V \cup \{ v^\text{old} | v \in V \} \cup \{ \text{result} \}$.
  - 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;
    }
}
```

\[
DIV \equiv a := 0;\ b := x;\ \textbf{while} b \geq y\ \textbf{do} b := b - y;\ a := a + 1 \ \textbf{od}
\{x \geq 0 \land y \geq 0\} \textbf{DIV} \{x \geq 0 \land y \geq 0\}
\]
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 $\vdash \{p\} \text{DIV} \{q\}$.
  - 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: $\vdash \{\text{false}\} f \{q\}$ 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. — 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 \( i \) 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...
References
