Softwaretechnik / Software-Engineering

Lecture 15: Software Quality Assurance

2015-07-09

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
**Contents of the Block “Quality Assurance”**

(i) **Introduction and Vocabulary**
- correctness illustrated
- vocabulary: fault, error, failure
- three basic approaches

(ii) **Formal Verification**
- Hoare calculus
- Verifying C Compiler (VCC)
- over- / under-approximations

(iii) **(Systematic) Tests**
- systematic test vs. experiment
- classification of test procedures
- model-based testing
- glass-box tests: coverage measures

(iv) **Runtime Verification**

(v) **Review**

(vi) **Concluding Discussion**
- Dependability

<table>
<thead>
<tr>
<th>Introduction</th>
<th>L 1: 20.4., Mo</th>
<th>T 1: 23.4., Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process, Metrics</td>
<td>L 2: 27.4., Mo</td>
<td>L 3: 30.4., Do</td>
</tr>
<tr>
<td></td>
<td>L 4: 4.5., Mo</td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Engineering</td>
<td>T 2: 7.5., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 5: 11.5., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 6: 18.5., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 7: 21.5., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 3: 1.6., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 8: 8.6., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 9: 11.6., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 10: 15.6., Mo</td>
<td></td>
</tr>
<tr>
<td>Architecture &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design, Software</td>
<td>T 4: 18.6., Do</td>
<td></td>
</tr>
<tr>
<td>Modelling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality Assurance</td>
<td>L 11: 22.6., Mo</td>
<td></td>
</tr>
<tr>
<td>Invited Talks</td>
<td>L 12: 25.6., Do</td>
<td></td>
</tr>
<tr>
<td>Wrap-Up</td>
<td>L 13: 29.6., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 14: 2.7., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 5: 6.7., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 15: 9.7., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 16: 13.7., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 17: 16.7., Do</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 6: 20.7., Mo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L 18: 23.7., Do</td>
<td></td>
</tr>
</tbody>
</table>
Contents & Goals

Last Lecture:
- Completed the block “Architecture & Design”

This Lecture:
- **Educational Objectives:** Capabilities for following tasks/questions.
  - When do we call a software correct?
  - What is fault, error, failure? How are they related?
  - What is formal and partial correctness?
  - What is a Hoare triple (or correctness formula)?
  - Is this program (partially) correct?
  - Prove the (partial) correctness of this WHILE-program using PD.
  - What can we conclude from the outcome of tools like VCC?

- **Content:**
  - Introduction, Vocabulary
  - WHILE-program semantics, partial & total correctness
  - Correctness proofs with the calculus PD.
  - The Verifying C Compiler (VCC)
Introduction
Mmmh, Software!

Recall: Formal Software Development

\[ [S] = \{\sigma_0 \xrightarrow{\tau} \sigma_1 \xrightarrow{\tau} \sigma_2 \ldots, \ldots\} \]

\[ [S'] = \{\tau \xrightarrow{\tau} \sigma_1 \xrightarrow{\tau} \sigma_2 \ldots, \ldots\} \]

\[ [S''] = \{\sigma_0 \xrightarrow{\tau} \sigma_1 \xrightarrow{\tau} \sigma_2 \ldots, \ldots\} \]

Development Process/Project Management

Requirements

\[ [\mathcal{L}_1] = \{(M.C, [\cdot]_1), (C.M, [\cdot]_1)\} \]

Design

\[ [\mathcal{L}_2] = \{(M.T_M.C, [\cdot]_1), (C.T_C.M, [\cdot]_1)\} \]
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: validation.

IEEE 610.12 (1990)

(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.
Mmmh, Software!

### Development Process/Project Management

- **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**.

- **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.

---

**Requirements**

\[
[S_1] = \{(M.C, [\cdot]_1), (C.M, [\cdot]_1)\}
\]

**Design**

\[
[S_2] = \{(M.T_M.C, [\cdot]_1), (C.T_C.M, [\cdot]_1)\}
\]

**Implementation**

\[
[S] = \{\sigma_0 \xrightarrow{\tau} \sigma_1 \xrightarrow{\tau} \sigma_2 \cdots, \cdots\}
\]
Is the implementation “correct”? And “correct” in what sense?
Definition. A software specification is a finite description $\mathcal{I}$ of a (possibly infinite) set $[\mathcal{I}]$ of softwares, i.e.

$$[\mathcal{I}] = \{(S_1, [\cdot]_1), \ldots \}.$$ 

The (possibly partial) function $[\cdot] : \mathcal{I} \mapsto [\mathcal{I}]$ is called interpretation of $\mathcal{I}$.

We define:

Software $S$ is correct wrt. software specification $\mathcal{I}$ if and only if $(S, [\cdot]) \in [\mathcal{I}]$.

- **Note**: no specification, no correctness. Without specification, $S$ is neither correct nor not correct — it’s just some software then.
Correctness Illustrated

\[ \mathcal{S} = (M.C) \text{ or } (C.M) \]

- Software doing (at most) M.C
- Software doing neither M.C nor C.M
- Software doing (at most) C.M

All imaginable softwares

Softwares which consider all necessary inputs

\[ (\Sigma \times A)^{\omega} \]

Final implementation — is it one of the allowed ones?

compile
Vocabulary

**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).
Concepts of Software Quality Assurance

- **software quality assurance**
  - organisational
  - analytic
  - constructive

- project management

- software examination
  - non-mech.
  - semi-mech.
  - mechanical

- examination by humans
  - comp. aided human exam.
    - e.g. interactive prover
      - check against rules
      - consistency checks

- mechanical examination with computer
  - analyse
  - execute
  - prove
    - dynamic checking (test)
    - quantitative examination

- comp. aided human exam.
  - manual proof
    - inspection
    - review

- constructive software engineering
  - e.g. code generation

- formal verification

(Ludewig and Lichter, 2013)
**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**, e.g. by looking for **errors**.
\[ S = (M.C) \text{ or } (C.M) \]

- Software doing (at most) M.C
- Software doing neither M.C nor C.M
- Software doing (at most) C.M

All imaginable softwares

Softwares which consider all necessary inputs

\[ (\Sigma \times M)^\omega \]
\[ (\Sigma \times A)^\omega \]
\[ (\Sigma \times A)^\omega \]

... final implementation — is it one of the allowed ones?
If we are lucky, the requirement specification is a constraint on computation paths.

LSC ‘buy_water’ is such a software specification $\mathcal{S}$.

It denotes all controller softwares which “faithfully” sell water. (Or which refuse to accept C50 coins, or block the ‘WATER’ button).

Formally

$$\llbracket \text{buy\_water} \rrbracket_{\text{spec}} = \{ S \mid \llbracket S \rrbracket \text{ satisfies ‘buy\_water’} \}.$$ 

In pictures:

Then we can check correctness of a given software $S$ by examining its computation paths $\llbracket S \rrbracket$. 
all computation paths satisfying specification

$$\left( \Sigma \times A \right)^\omega$$
Three Basic Directions

**Review**

Reviewer

review

**Testing**

all computation paths satisfying specification

\[(\Sigma \times A)^\omega\]

input → output

\[\rightarrow\]

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

**Formal Verification**
Formal Verification
One style of requirements specifications: pre- and post-conditions (on whole programs or on procedures).

Let $S$ be a program with states from $\Sigma$ and let $p$ and $q$ be formulae such that there is a satisfaction relation $\models \subseteq \Sigma \times \{ p, q \}$.

$S$ is called partially correct wrt. $p$ and $q$, denoted by $\models \{ p \} S \{ q \}$, if and only if

$$\forall \pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \sigma_{n-1} \xrightarrow{\alpha_n} \sigma_n \in [S] \cdot \sigma_0 \models p \implies \sigma_n \models q$$

("if $S$ terminates from a state satisfying $p$, then the final state of that computation satisfies $q$")

$S$ is called totally correct wrt. $p$ and $q$, denoted by $\models_{tot} \{ p \} S \{ q \}$, if and only if

- $\{ p \} S \{ q \}$ ($S$ is partially correct), and
- $\forall \pi \in [S] \cdot \pi^0 \models p \implies |\pi| \in \mathbb{N}_0$

($S$ terminates from all states satisfying $p$; length of paths: $|\cdot| : \Pi \rightarrow \mathbb{N}_0 \cup \{ \perp \}$).
Computing squares (of numbers 0, . . . , 27).

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

- **Program** $S_1$:

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

  $\models \{p\} S_1 \{q\}$, $\models_{tot} \{p\} S_1 \{q\}$

- **Program** $S_2$:

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

  $\not\models \{p\} S_2 \{q\}$, $\models_{tot} \{p\} S_2 \{q\}$

- **Program** $S_3$:

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

  $\models \{p\} S_3 \{q\}$, $\models_{tot} \{p\} S_3 \{q\}$

- **Program** $S_4$:

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

  $\models \{p\} S_4 \{q\}$, $\models_{tot} \{p\} S_4 \{q\}$
Deterministic Programs

Syntax:

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

where \( u \) is a variable, \( t \) a type-compatible expression, \( B \) a Boolean expression.

Semantics: (is induced by the following transition relation)

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

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

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

(vii) \( \langle \text{while } B \text{ do } S \text{ do}, \sigma \rangle \rightarrow \langle E, \sigma \rangle, \) 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).
Definition.

(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 there is a diverging computation starting 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 $\sigma$. 
Consider **program** $S \equiv a[0] := 1; a[1] := 0; \textbf{while} a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ do}$ and a **state** $\sigma$ with $\sigma \models x = 0$. 
Example

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

\[ \langle S, \sigma \rangle \xrightarrow{(ii),(i)} \langle E; S, \sigma[ a[0] := 1 ] \rangle \]
Example

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

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

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

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

\[ (i) \langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle \quad \text{E; } S \equiv S; \ E \equiv S \]

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

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

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

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

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

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

Consider program \( S \equiv a[0] := 1; a[1] := 0; \text{while } a[x] \not= 0 \text{ do } x := x + 1 \text{ do} \) and a state \( \sigma \) with \( \sigma \models x = 0 \).

\[ \langle S, \sigma \rangle \xrightarrow{(ii),(iii)} \langle a[1] := 0; \text{while } a[x] \not= 0 \text{ do } x := x + 1 \text{ do}, \sigma[a[0] := 1] \rangle \]

\[ \xrightarrow{(ii),(iii)} \langle \text{while } a[x] \not= 0 \text{ do } x := x + 1 \text{ do, } \sigma' \rangle \]

\[ \xrightarrow{(vi)} \langle x := x + 1; \text{while } a[x] \not= 0 \text{ do } x := x + 1 \text{ do, } \sigma' \rangle \]

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

\[
\begin{align*}
\langle S, \sigma \rangle & \xrightarrow{(ii),(iii)} \langle a[1] := 0; \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ do}, \sigma[a[0] := 1] \rangle \\
& \xrightarrow{(ii),(iii)} \langle \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ do}, \sigma' \rangle \\
& \xrightarrow{(vi)} \langle x := x + 1; \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ do}, \sigma' \rangle \\
& \xrightarrow{(ii),(iii)} \langle \textbf{while } a[x] \neq 0 \textbf{ do } x := x + 1 \textbf{ do}, \sigma'[x := 1] \rangle \\
& \xrightarrow{(vii)} \langle E, \sigma'[x := 1] \rangle \\
\end{align*}
\]

where \( \sigma' = [\sigma[a[0] := 1]][a[1] := 0] \).
Definition.
Let $S$ be a deterministic program.

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

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

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

(i) A correctness formula \( \{p\} S \{q\} \) holds in the sense of partial correctness, denoted by \( \models \{p\} S \{q\} \), if and only if

\[
\mathcal{M}[S](\lceil p \rceil) \subseteq \lceil q \rceil.
\]

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

\[
\mathcal{M}_{\text{tot}}[S](\lceil p \rceil) \subseteq \lceil q \rceil.
\]

We say \( S \) is totally correct wrt. \( p \) and \( q \).
Example: Correctness

• By the previous example, we have shown

\[ \models \{ x = 0 \} S \{ x = 1 \} \text{ and } \models_{\text{tot}} \{ x = 0 \} S \{ x = 1 \}. \]

(because we only assumed \( \sigma \models x = 0 \) for the example, which is exactly the precondition.)

• We have also shown:

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

• The following correctness formula does not hold for \( S \):

\[ \not\models_{\text{tot}} \{ x = 2 \} S \{ \text{true} \}. \]

(e.g., 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 \( PD \) (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**

\[
\begin{align*}
\{p\} & \ S_1 \ \{r\}, \ \{r\} \ S_2 \ \{q\} \\
\{p\} & \ S_1; \ S_2 \ \{q\}
\end{align*}
\]

**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**

\[
\begin{align*}
\{p \land B\} & \ S \ \{p\} \\
\{p\} & \ \text{while} \ B \ \text{do} \ S \ \text{do} \ \{p \land \neg B\}
\end{align*}
\]

**Rule 6**: **Consequence**

\[
\begin{align*}
p \rightarrow p_1, \ \{p_1\} & \ S \ \{q_1\}, \ q_1 \rightarrow q \\
\{p\} & \ S \ \{q\}
\end{align*}
\]

**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 \( \models \{p\} \ S \ \{q\} \).
Substitution

In PD uses substitution of the form $p[u := t]$.

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

Usually straightforward, but indexed and bound variables need to be treated specially:

**Expressions:**
- plain variable: $x[u := t] \equiv \begin{cases} t & \text{if } x = u \\ x & \text{otherwise} \end{cases}$
- constant $c$: $c[u := t] \equiv c$.
- constant $op$, terms $s_i$:
  $op(s_1, \ldots, s_n)[u := t]$
  $\equiv op(s_1[u := t], \ldots, s_n[u := t])$.
- 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, $u \equiv a[t_1, \ldots, t_m]$:
  $(a[s_1, \ldots, s_n])[u := t]$
  $\equiv \begin{cases} \text{if } \bigwedge_{i=1}^{n} s_i[u := t] = t_i \text{ then } t \\ \text{else } a[s_1[u := t], \ldots, s_n[u := t]] \text{ fi} \end{cases}$
- conditional expression:
  if $B$ then $s_1$ else $s_2$ fi$[u := t]$
  $\equiv \begin{cases} \text{if } B[u := t] \text{ then } s_1[u := t] \text{ else } s_2[u := t] \text{ fi} \end{cases}$

**Formulae:**
- boolean expression $p \equiv s$: $p[u := t] \equiv s[u := t]$
- negation:
  $(-q)[u := t] \equiv \neg(q[u := t])$
- 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 $x$. 
Example Proof

\[ \text{DIV} \equiv q := 0; \ r := x; \ \textbf{while} \ r \geq y \ \textbf{do} \ r := r - y; \ q := q + 1 \ \textbf{do} \]

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

We want to prove

\[ \models \{x \geq 0 \land y \geq 0\} \ \text{DIV} \ \{q \cdot y + r = x \land r < y\} \]

**Note:** writing a program \(S\) which satisfies this correctness formula is much easier if \(S\) may change \(x\) and \(y\)...

The proof needs a loop invariant, we choose (creative act!):

\[ P \equiv q \cdot y + r = x \land r \geq 0 \]

We prove

- (1) \(\{x \geq 0 \land y \geq 0\} \ q := 0; \ r := x \ \{P\} \) and
- (2) \(\{P \land r \geq y\} \ r := r - y; \ q := q + 1 \ \{P\} \) in PD, and
- (3) \(P \land \neg(r \geq y) \rightarrow q \cdot y + r = x \land r < y \) “by hand”.

\[ Q \]

\[ S_1 \]

\[ S_2 \]
### Example Proof

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A1)</td>
<td>( { p } \text{ skip } { p } )</td>
</tr>
<tr>
<td>(A2)</td>
<td>( { p[u := t] } u := t { p } )</td>
</tr>
<tr>
<td>(R3)</td>
<td>( { p } S_1 { r }, { r } S_2 { q } )</td>
</tr>
<tr>
<td>(R4)</td>
<td>( { 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 } )</td>
</tr>
<tr>
<td>(R5)</td>
<td>( { p \land B } S { p } )</td>
</tr>
<tr>
<td>(R6)</td>
<td>( p \rightarrow p_1, { p_1 } S { q_1 }, q_1 \rightarrow q )</td>
</tr>
</tbody>
</table>

#### Assume:

- (1) \( \{ x \geq 0 \land y \geq 0 \} q := 0; r := x \{ P \}, \)
- (2) \( \{ P \land r \geq y \} r := r - y; q := q + 1 \{ P \}, \) and
- (3) \( P \land \neg(r \geq y) \rightarrow q \cdot y + r = x \land r < y. \)

- By rule (R5), we obtain, using (2),

\[
\vdash \{ P \} \text{ while } r \geq y \text{ do } r := r - y; q := q + 1 \text{ do } \{ P \land \neg(r \geq y) \}
\]
Example Proof

\[(A1) \{p\} \text{skip} \{p\} \quad (R4) \{p \land B\} S_1 \{q\}, \{p \land \neg B\} S_2 \{q\}, \{p\} \text{if} B \text{ then } S_1 \text{ else } S_2 \text{ fi} \{q\}\]

\[(A2) \{p[u := t]\} u := t \{p\} \quad (R5) \{p \land B\} S \{p\}\]

\[(R3) \begin{array}{l}
\{p\} S_1 \{r\}, \{r\} S_2 \{q\} \\
\{p\} S_1; S_2 \{q\} 
\end{array} \quad (R6) \begin{array}{l}
\{p \rightarrow p_1, \{p_1\} S \{q_1\}\}, q_1 \rightarrow q \\
\{p\} S \{q\} 
\end{array}\]

Assume:

- (1) \(\{x \geq 0 \land y \geq 0\} \ q := 0; \ r := x \ \{P\}\),
- (2) \(\{P \land r \geq y\} \ r := r - y; \ q := q + 1 \ \{P\}\), and
- (3) \(P \land \neg(r \geq y) \rightarrow q \cdot y + r = x \land r < y\).

By rule (R5), we obtain, using (2),

\[\vdash \{P\} \text{while} r \geq y \text{ do } r := r - y; \ q := q + 1 \text{ do } \{P \land \neg(r \geq y)\}\]

By rule (R3), we obtain, using (1),

\[\vdash \{x \geq 0 \land y \geq 0\} \text{DIV} \ \{P \land \neg(r \geq y)\}\]

By rule (R6), we obtain, using (3),

\[\vdash \{x \geq 0 \land y \geq 0\} \text{DIV} \ \{q \cdot y + r = x \land r < y\}\]
\textbf{Proof: (2)}

\begin{enumerate}
\item \( P \equiv q \cdot y + r = x \land r \geq 0, \)
\item (2): \( \{ P \land r \geq y \} \ r := r - y; \ q := q + 1 \ \{ P \} \)
\item \( \{ (q + 1) \cdot y + r = x \land x \geq 0 \} \ q := q + 1 \ \{ P \} \) by (A2),
\end{enumerate}
Proof: (2)

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

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

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

S_1; S_2 \{q\}

(R4) \{p \land B\} S_1 \{q\}, \{p \land \neg B\} S_2 \{q\}, \{p\} \mathtt{if} B \mathtt{then} S_1 \mathtt{else} S_2 \mathtt{fi} \{q\}

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

\{p\} \mathtt{while} B \mathtt{do} S \mathtt{do} \{p \land \neg B\}

(R6) p \rightarrow p_1, \{p_1\} S \{q_1\}, q_1 \rightarrow q

\{p\} S \{q\}

\bullet \quad P \equiv q \cdot y + r = x \land r \geq 0,

\bullet \quad (2): \{P \land r \geq y\} r := r - y; \quad q := q + 1 \{P\}

\bullet \quad \{(q + 1) \cdot y + r = x \land r \geq 0\} q := q + 1 \{P\} \text{ by (A2)},

\bullet \quad \{(q + 1) \cdot y + (r - y) = x \land (r - y) \geq 0\} r := r - y \{(q + 1) \cdot y + r = x \land r \geq 0\} \text{ by (A2)},
**Proof: (2)**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A1)</td>
<td>{p} skip {p}</td>
</tr>
<tr>
<td>(A2)</td>
<td>{p[u := t]} u := t {p}</td>
</tr>
<tr>
<td>(R3)</td>
<td>{p} S_1 {r}, {r} S_2 {q}</td>
</tr>
<tr>
<td>(R4)</td>
<td>{p \land B} S_1 {q}, {p \land \neg B} S_2 {q}, {p} if B then S_1 else S_2 fi {q}</td>
</tr>
<tr>
<td>(R5)</td>
<td>{p \land B} S {p}</td>
</tr>
<tr>
<td>(R6)</td>
<td>{p \land \neg B} while B do S do {p} S {q}</td>
</tr>
</tbody>
</table>

- \(P \equiv q \cdot y + r = x \land r \geq 0\),
- \((2): \{P \land r \geq y\} r := r - y; q := q + 1 \{P\}\)
- \\{(q + 1) \cdot y + r = x \land x \geq 0\} q := q + 1 \{P\} by (A2),
- \\{(q + 1) \cdot y + (r - y) = x \land (r - y) \geq 0\} r := r - y \{(q + 1) \cdot y + r = x \land x \geq 0\} by (A2),
- \\{(q + 1) \cdot y + (r - y) = x \land (r - y) \geq 0\} r := r - y; q := q + 1 \{P\} by (R3),
- \(2)\) by (R6), using

\[ P \land r \geq y \rightarrow (q + 1) \cdot y + (r - y) = x \land (r - y) \geq 0.\]
Proof: (1)

- \( P \equiv q \cdot y + r = x \land r \geq 0, \)

- \( (1) \{ x \geq 0 \land y \geq 0 \} q := 0; \quad r := x \{ P \} \)

- \( \{ q \cdot y + x = x \land x \geq 0 \} r := x \{ P \} \text{ by (A2)}, \)

- \( \{ 0 \cdot y + x = x \land x \geq 0 \} q := 0 \{ q \cdot y + x = x \land x \geq 0 \} \text{ by (A2)}, \)

- \( \{ 0 \cdot y + x = x \land x \geq 0 \} q := 0; \quad r := x \{ P \} \text{ by (R3)}, \)

- (1) by (R6) using \( x \geq 0 \land y \geq 0 \rightarrow 0 \cdot y + x = x \land x \geq 0. \)
- **Once Again**

- \( P \equiv q \cdot y + r = x \land r \geq 0 \)
  \[ \{x \geq 0 \land y \geq 0\} \]
  \[\{0 \cdot y + x = x \land x \geq 0\}\]
- \( q := 0; \)
  \[\{q \cdot y + x = x \land x \geq 0\}\]
- \( r := x; \)
  \[\{q \cdot y + r = x \land x \geq 0\}\]
  \(\{P\}\)
- **while** \( r \geq y \) **do**
  \[\{P \land r \geq y\}\]
  \[\{(q + 1) \cdot y + (r - y) = x \land (r - y) \geq 0\}\]
- \( r := r - y; \)
  \[\{(q + 1) \cdot y + r = x \land x \geq 0\}\]
- \( q := q + 1 \)
  \[\{q \cdot y + r = x \land x \geq 0\}\]
  \(\{P\}\)
- **do**
  \[\{P \land \neg(r \geq y)\}\]
  \[\{q \cdot y + r = x \land r < y\}\]

(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\} if \(B\) then \(S_1\) else \(S_2\) fi \{q\}

(R5) \{p \land B\} S \{p\}
  \{p\} while \(B\) do \(S\) do \{p \land \neg B\}

(R6) \(p \rightarrow p_1, \{p_1\} S \{q_1\}, q_1 \rightarrow q\)
  \{p\} S \{q\}
We can add a rule for function calls (simplest case: only global variables):

\[
\begin{align*}
\text{(R7)} \quad & \quad \{p\} \ f \ \{q\} \\
& \quad \{p\} \ f() \ \{q\}
\end{align*}
\]

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

\(p\) is called \textbf{pre-condition} of \(f\), \(q\) is called \textbf{post-condition}.

\textbf{Example}: if we have

- \(\{\text{true}\}\) \text{read}\_\text{number} \(\{0 \leq \text{ret} < 10^8\}\)
- \(\{0 \leq x \wedge 0 \leq y\}\) add \(\{\text{old}(x) + \text{old}(y) < 10^8 \wedge \text{ret} = \text{old}(x) + \text{old}(y)\} \lor \text{ret} < 0\}\)
- \(\{\text{true}\}\) 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 (\(\rightarrow\) later) pocket calculator correct.

```c
int main() {
    int x = read_number();
    int y = read_number();
    int sum = add( x, y );
    display(sum);
}
```
Assertions

We add another rule for assertions:

\[(A3) \{p\} \text{assert}(p) \{p\}\]

- That is, if \(p\) holds before the assertion, then we can continue with the proof.
- Otherwise we “get stuck”.

So we cannot even prove

\[\{\text{true}\} \ x := 0; \ \text{assert}(x = 27) \ \{\text{true}\}.\]

to hold (it is not derivable).

- Which is exactly what we want — if we add
  - \langle\text{assert}(B), \sigma\rangle \rightarrow \langle E, \sigma\rangle\ if \sigma \models B,
  to the transition relation.
(If the assertion does not hold, the empty program is not reached; the assertion remains in the first component: abnormal program termination).
Why Assertions?

- Available in standard libraries of many programming languages, e.g. C:

```c
#include <assert.h>

void assert(scalar expression);
```

**DESCRIPTION**

[...] the macro `assert()` prints an error message to standard error and terminates the program by calling `abort(3)` if expression is false (i.e., compares equal to zero).

The purpose of this macro is to help the programmer find bugs in his program. The message "assertion failed in file foo.c, function do_bar(), line 1287" is of no help at all to a user.
Why Assertions?

- Available in standard libraries of many programming languages, e.g. C:

```
void assert(scalar expression);
```

- Assertions at work:

```
int square(int x) {
    assert(x < sqrt(x));
    return x * x;
}
```
The Verifying C Compiler
The Verifying C Compiler (VCC) basically implements Hoare-style reasoning.

**Special syntax:**
- `#include <vcc.h>`
- `_(requires p)` — pre-condition, `p` is a C expression
- `_(ensures q)` — post-condition, `q` is a C expression
- `_(invariant expr)` — loop invariant, `expr` is a C expression
- `_(assert p)` — intermediate invariant, `p` is 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 q, r;

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

DIV ≡ q := 0; r := x; while r ≥ y do r := r - y; q := q + 1 do

{x ≥ 0 ∧ y ≥ 0} DIV {q · y + r = x ∧ r < y}
Does this C program always work?

```c
#include <stdio.h>

int q, r;

void div(int x, int y)
{
    (requires x > 0 && y > 0)
    (ensures q * y + r == x && r < y)
    (writes &q)
    (writes &r)

    q = 0;
    r = x;

    while (r >= y)
    {
        (invariant q * y + r == x && r >= 0)
        r = r - y;
        q = q + 1;
    }
}
```

VCC is a tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and postcondition as well as type invariants. Annotated programs are translated to logical formulas using the Boogie tool, which passes them to an automated SMT solver z3 to check their validity.
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.
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!

  Yet we can ask for a printout of the proof and check it manually (hardly possible in practice) or with other tools like interactive theorem provers.

  **Note:** $\models \{false\} \not\models \{q\}$ always holds
  — so a mistake in writing down the pre-condition can provoke a false negative.

- VCC says: “verification failed

  - One case: “timeout” etc. — completely inconclusive outcome.
  - 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 $q$.

    May be a false negative if these inputs are actually never used.
    Make pre-condition $p$ stronger, and try again.
Investigate All Paths

(like Uppaal; possible for finite-state software; no false positives or negatives)
Investigate All Paths

(like Uppaal; possible for finite-state software; no false positives or negatives)
Investigate All Paths
(like Uppaal; possible for finite-state software; no false positives or negatives)

Over-Approximation
(some Software model-checkers; goal: verify correctness; false positives, no false negatives)
Investigate All Paths
(like Uppaal; possible for finite-state software; no false positives or negatives)

Over-Approximation
(some Software model-checkers; goal: verify correctness; false positives, no false negatives)
(Automatic) Formal Verification Techniques

\[(\Sigma \times A)^\omega\]

all computation paths satisfying specification

**Investigate All Paths**
(like Uppaal; possible for finite-state software; no false positives or negatives)

**Over-Approximation**
(some Software model-checkers; goal: verify correctness; false positives, no false negatives)

**Under-Approximation**
(e.g. bounded model-checking; goal: find errors; false negatives, no false positives)
Investigate All Paths
(like Uppaal; possible for finite-state software; no false positives or negatives)

Over-Approximation
(some Software model-checkers; goal: verify correctness; false positives, no false negatives)

Under-Approximation
(e.g. bounded model-checking; goal: find errors; false negatives, no false positives)
References
References


