Softwaretechnik / Software-Engineering

Lecture 15: Software Quality Assurance

2015-07-09

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

– 15 – 2015-07-09 – main –

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

Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
	L 5:	11.5., Mo
Requirements Engineering	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Architecture & Design, Software Modelling	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	L 14:	2.7., Do
	T 5:	6.7., Mo
Quality Assurance	L 15:	9.7., Do
	L 16:	13.7., Mo
Invited Talks	L 17:	16.7., Do
	T 6:	20.7., Mo
Wrap-Up	L 18:	23.7., Do

– 15 – 2015-07-09 – Scontents –

Contents & Goals

Last Lecture:

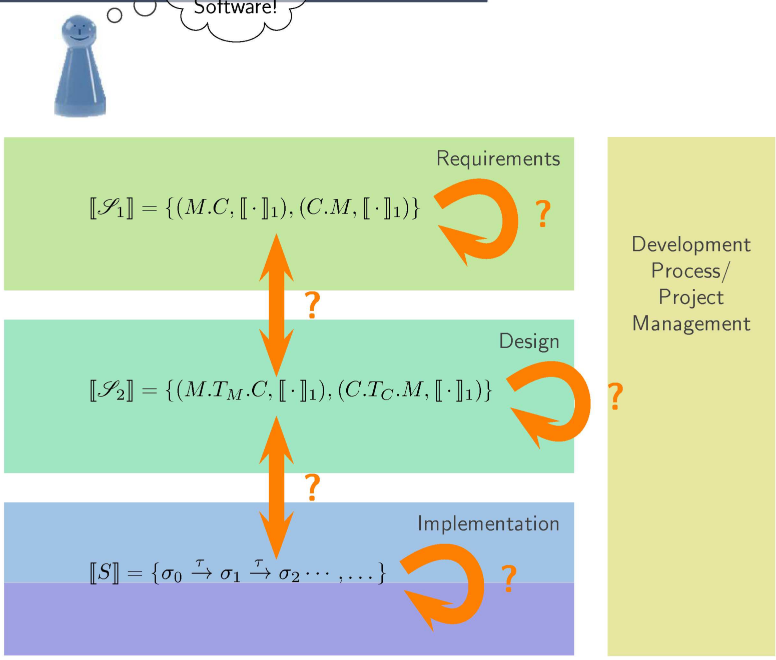
- Completed the block "Architecture & Design"

This Lecture:

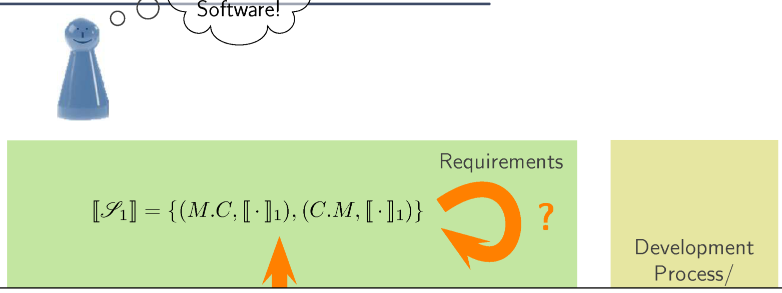
- **Educational Objectives:** Capabilities for following tasks/questions.
  - When do we call a software correct?
  - What is fault, error, failure? How are they related?
  - What is ~~formal~~ total and partial correctness?
  - What is a Hoare triple (or correctness formula)?
  - Is this program (partially) 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

Recall: Formal Software Development



Recall: Formal Software Development



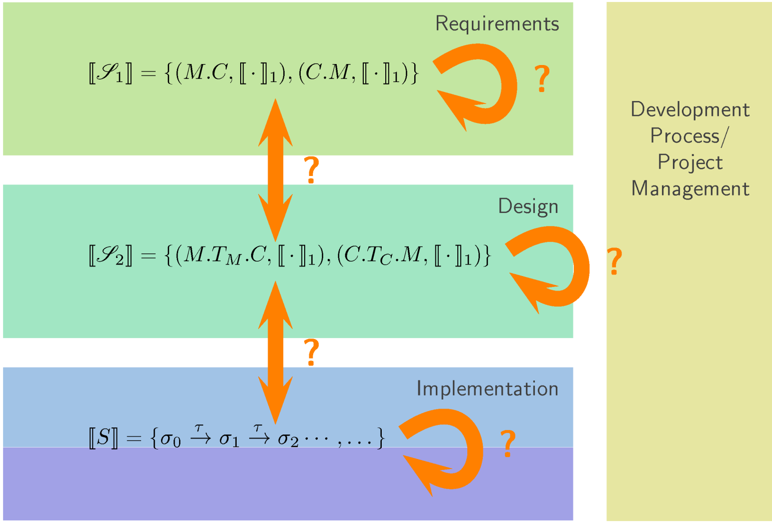
validation The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: **verification**. **IEEE 610.12 (1990)**

verification
(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: **validation**.
(2) Formal proof of program correctness. **IEEE 610.12 (1990)**

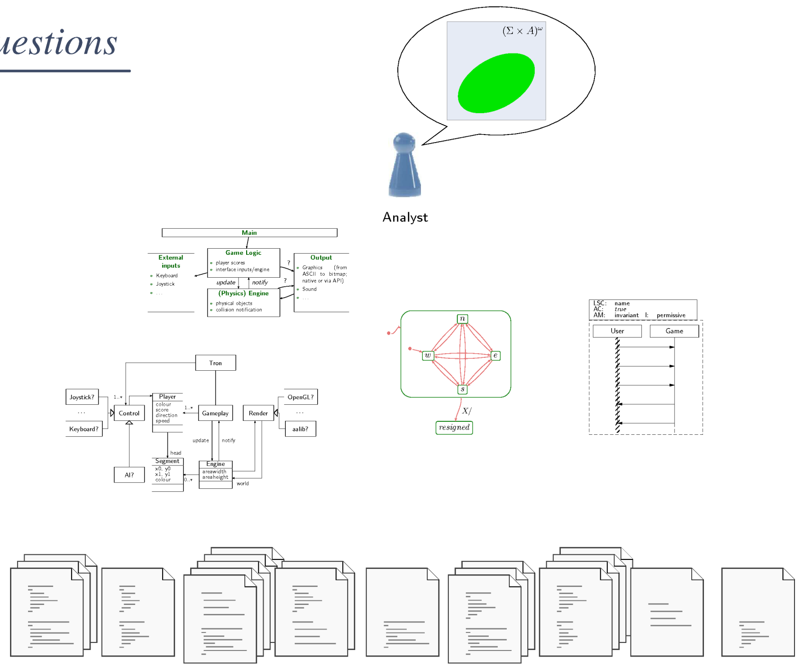
Recall: Formal Software Developm

validation The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: **verification**. **IEEE 610.12** (1990)

verification
(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: **validation**.
(2) Formal proof of program correctness. **IEEE 610.12** (1990)



Big Questions



Is the implementation "correct"? And "correct" in what sense?

Back To Lecture No. 1

Definition. A software specification is a finite description $\mathscr{S}$ of a (possibly infinite) set $[\![\mathscr{S}]\!]$ of softwares, i.e.

$$[\![\mathscr{S}]\!] = \{(S_1, [\![\cdot]\!]_1), \dots\}.$$

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

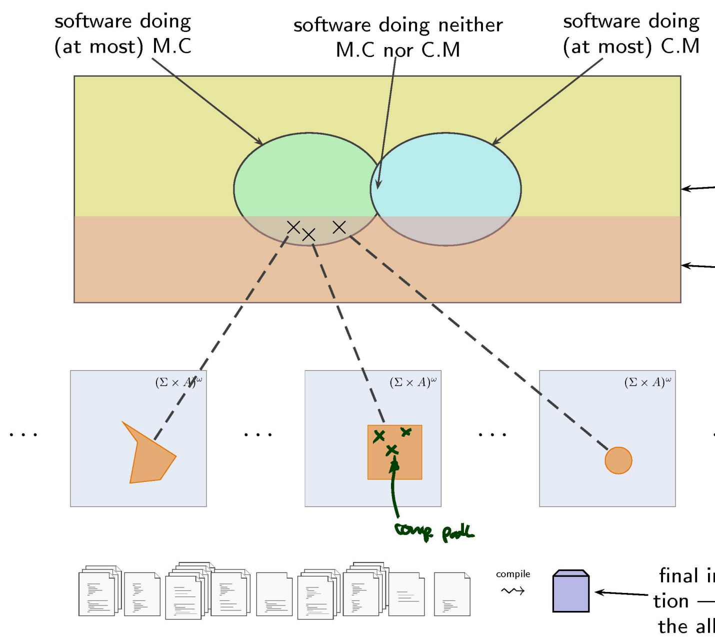
We define:

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

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

Correctness Illustrated

$$\mathscr{S} = (M.C) \text{ or } (C.M)$$



Vocabulary

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

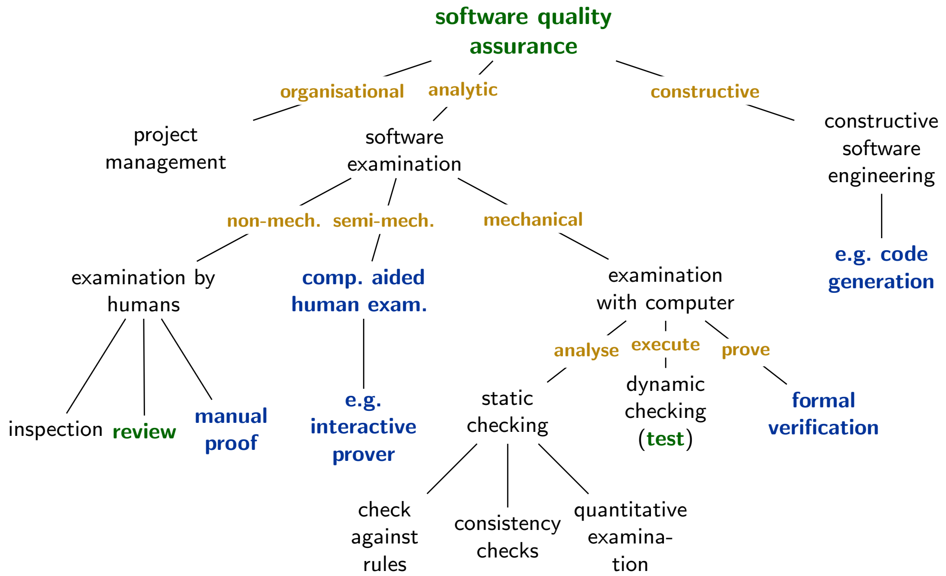
quality assurance — (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.

(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).

– 15 – 2015-07-09 – Svintro –

Concepts of Software Quality Assurance



(Ludewig and Lichter, 2013)

– 15 – 2015-07-09 – Svintro –

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)**

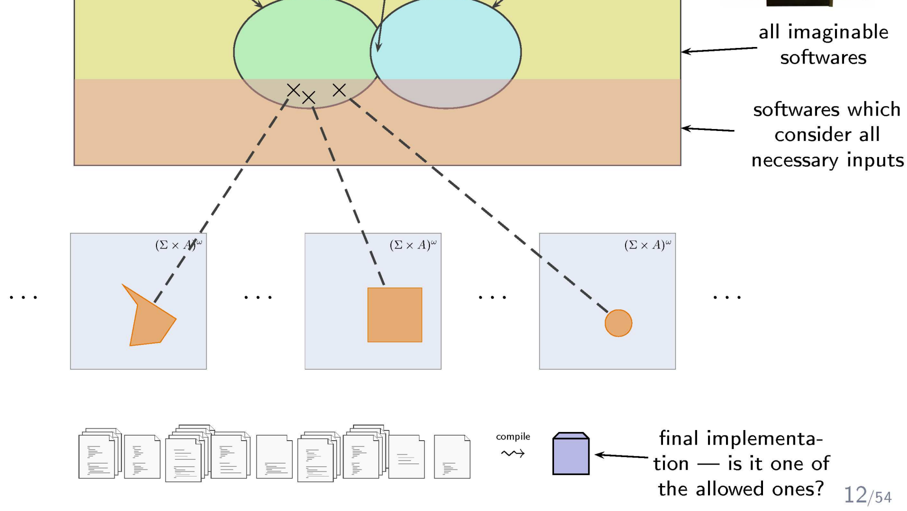
failure — termination of the ability of an element, to perform a function as required.

Note: Incorrect specification is a source of failure. **ISO 26262 (2011)**

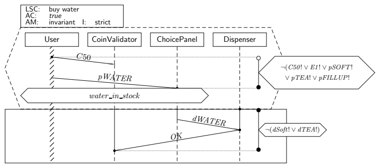
We want to avoid **failures**, thus we try to detect **faults**, e.g. by looking for **errors**.

Back to the Illustration

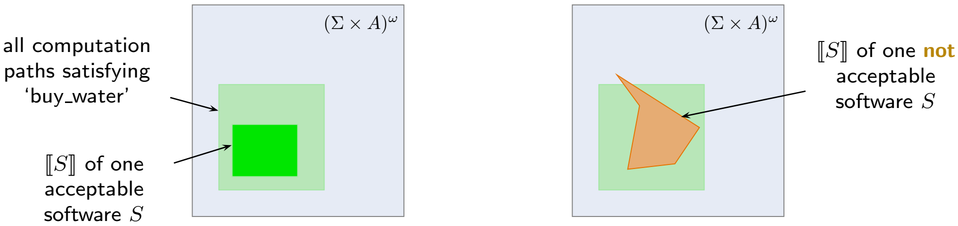
$\mathscr{S} = (M.C)$ or $(C.M)$



So, What Do We Do?

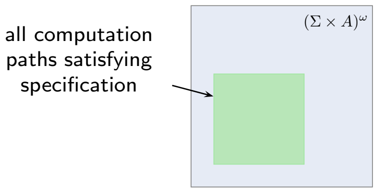


- If we are lucky, the requirement specification is a constraint on **computation paths**.
- LSC 'buy_water' is such a software specification $\mathscr{S}$.
- It denotes all controller softwares which "faithfully" sell water.
  (Or which refuse to accept C50 coins, or block the 'WATER' button).
- Formally
  $$[\![\text{buy\_water}]\!]_{spec} = \{S \mid [\![S]\!] \text{ satisfies 'buy\_water'}\}.$$
- In pictures:

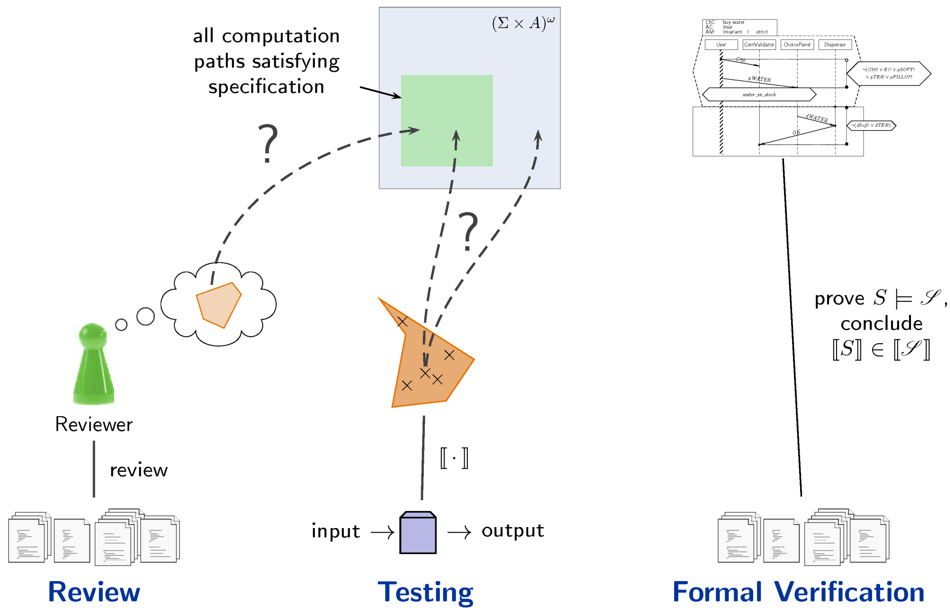


- Then we can check correctness of a given software S by examining its computation paths $[\![S]\!]$.

Three Basic Directions



Three Basic Directions



Formal Verification

Correctness Formulae ("Hoare Triples")

- **One style of requirements specifications**: **pre-** and **post-conditions** (on whole programs or on procedures).
- Let S be a program with states from Σ and let p and q be formulae such that there is a **satisfaction relation** $\models \subseteq \Sigma \times \{p, q\}$.

  correctness formula, Hoare triple

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

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

  ("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
  - $\forall\, \pi \in [\![S]\!] \bullet \pi^0 \models p \implies |\pi| \in \mathbb{N}_0$

    (S terminates from all states satisfying p; length of paths: $|\cdot| : \Pi \to \mathbb{N}_0 \,\dot\cup\, \{\bot\}$).

Example

Computing squares (of numbers $0, \dots, 27$).

- **Pre-condition**: $p \equiv 0 \le x \le 27$, **post-condition**: $q \equiv y = x^2$.
- **Program** S_1:

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

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

- **Program** S_2:

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

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

- **Program** S_3:

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

  never terminates

  $\models^? \{p\}\ S_3\ \{q\}$ ✓ *(trivially)*, $\models^?_{tot} \{p\}\ S_3\ \{q\}$ ✗

- **Program** S_4:

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

  7:1 NO YES

  $\models^? \{p\}\ S_4\ \{q\}$ ✓, $\models^?_{tot} \{p\}\ S_4\ \{q\}$ ✗

Deterministic Programs

Syntax:

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

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

Semantics: (is induced by the following transition relation)

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$

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

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

(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,

(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,

(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,

(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, 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).

Computations of Deterministic Programs

Definition.

(i) A **transition sequence** of S (starting in σ) is a finite or infinite sequence

$$\langle S, \sigma\rangle = \langle S_0, \sigma_0\rangle \to \langle S_1, \sigma_1\rangle \to \dots$$

(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 σ) is a maximal transition sequence of S (starting in σ), i.e. infinite or not extendible.

(iii) A computation of S is said to

a) **terminate** in τ 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** σ if and only if there is a diverging computation starting in σ.

(iv) We use $\to^*$ to denote the transitive, reflexive closure of $\to$.

Lemma. For each deterministic program S and each state σ, there is exactly one computation of S which starts in σ.

Example

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$ $\qquad E; S \equiv S; E \equiv S$

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

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

(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,

(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,

(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,

(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, if $\sigma \not\models B$,

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** σ with $\sigma \models x = 0$.

Example

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$ $\qquad E; S \equiv S; E \equiv S$

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

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

(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,

(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,

(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,

(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, if $\sigma \not\models B$,

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** σ with $\sigma \models x = 0$.

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

Example

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$ $\quad E; S \equiv S; E \equiv S$
(ii) $\langle u := t, \sigma\rangle \to \langle E, \sigma[u := \sigma(t)]\rangle$
(iii) $\dfrac{\langle S_1, \sigma\rangle \to \langle S_2, \tau\rangle}{\langle S_1; S, \sigma\rangle \to \langle S_2; S, \tau\rangle}$
(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,
(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,
(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,
(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, if $\sigma \not\models B$,

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** σ with $\sigma \models x = 0$.

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

Example

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$ $\quad E; S \equiv S; E \equiv S$
(ii) $\langle u := t, \sigma\rangle \to \langle E, \sigma[u := \sigma(t)]\rangle$
(iii) $\dfrac{\langle S_1, \sigma\rangle \to \langle S_2, \tau\rangle}{\langle S_1; S, \sigma\rangle \to \langle S_2; S, \tau\rangle}$
(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,
(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,
(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,
(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, if $\sigma \not\models B$,

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** σ with $\sigma \models x = 0$.

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

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

Example

(i) $\langle skip, \sigma\rangle \to \langle E, \sigma\rangle$ $\qquad E; S \equiv S; E \equiv S$
(ii) $\langle u := t, \sigma\rangle \to \langle E, \sigma[u := \sigma(t)]\rangle$
(iii) $\dfrac{\langle S_1, \sigma\rangle \to \langle S_2, \tau\rangle}{\langle S_1; S, \sigma\rangle \to \langle S_2; S, \tau\rangle}$
(iv) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle$, if $\sigma \models B$,
(v) $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle$, if $\sigma \not\models B$,
(vi) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle S; \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle$, if $\sigma \models B$,
(vii) $\langle \textbf{while } B \textbf{ do } S \textbf{ do}, \sigma\rangle \to \langle E, \sigma\rangle$, if $\sigma \not\models B$,

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** σ with $\sigma \models x = 0$.

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

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

Input/Output Semantics of Deterministic Programs

Definition.
Let S be a deterministic program.

(i) The **semantics of partial correctness** is the function

$$\mathcal{M}[\![S]\!] : \Sigma \to 2^\Sigma$$

with $\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid \langle S, \sigma\rangle \to^* \langle E, \tau\rangle\}$.

(ii) The **semantics of total correctness** is the function

$$\mathcal{M}_{tot}[\![S]\!] : \Sigma \to 2^\Sigma \,\dot\cup\, \{\bot\}$$

with $\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\bot \mid S \text{ can diverge from } \sigma\}$.
$\bot$ is an error state representing divergence.

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

Correctness of Deterministic Programs

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

$$\mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!].$$

$\{\sigma \mid \sigma \models q\}$

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

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

$\bot$ is not in!

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

Example: Correctness

- By the previous example, **we have shown**

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

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

- **We have also shown**:

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

- The following correctness formula **does not hold** for S:

$$\not\models_{tot} \{x = 2\}\ S\ \{\mathit{true}\}.$$

(e.g., if $\sigma \models a[i] \neq 0$ for all $i > 2$.)

- In the sense of **partial correctness**,

$$\{x = 2 \wedge \forall\, i \geq 2 \bullet a[i] = 1\}\ S\ \{\mathit{false}\}$$

also holds.

Proof-System PD (*for sequential, deterministic programs*)

Axiom 1: Skip-Statement

$$\{p\}\ skip\ \{p\}$$

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

$$\frac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\},}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$$

Rule 5: While-Loop

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{do}\ \{p \wedge \neg B\}}$$

Rule 6: Consequence

$$\frac{p \to p_1, \{p_1\}\ S\ \{q_1\}, q_1 \to q}{\{p\}\ S\ \{q\}}$$

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 $\models \{p\}\ S\ \{q\}$.

correct

Substitution

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

(In formula p, replace all (free) occurences 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:
  $op(s_1, \dots, s_n)[u := t]$
  $\equiv op(s_1[u := t], \dots, s_n[u := t])$.
- indexed variable, u plain
  or $u \equiv b[t_1, \dots, t_m]$ and $a \neq b$:
  $(a[s_1, \dots, s_n])[u := t] \equiv a[s_1[u := t], \dots, s_n[u := t]])$
- indexed variable, $u \equiv a[t_1, \dots, t_m]$:
  $(a[s_1, \dots, s_n])[u := t]$
  $\equiv \textbf{if}\ \bigwedge_{i=1}^{n} s_i[u := t] = t_i\ \textbf{then}\ t$
  $\textbf{else}\ a[s_1[u := t], \dots, s_n[u := t]]\ \textbf{fi}$
- conditional expression:
  $\textbf{if}\ B\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}[u := t]$
  $\equiv \textbf{if}\ B[u := t]\ \textbf{then}\ s_1[u := t]\ \textbf{else}\ s_2[u := t]\ \textbf{fi}$

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.:
  $(q \wedge r)[u := t]$
  $\equiv q[u := t] \wedge r[u := t]$
- 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

$$DIV \equiv q := 0;\ r := x;\ \mathbf{while}\ r \geq y\ \mathbf{do}\ r := r - y;\ q := q + 1\ \mathbf{do}$$

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

We want to prove

$$\models \{x \geq 0 \wedge y \geq 0\}\ DIV\ \{q \cdot y + r = x \wedge r < y\}$$

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!**):

$$P \equiv q \cdot y + r = x \wedge r \geq 0$$

We prove

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

Example Proof

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

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

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

(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\},}{\{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{q\}}$

(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{do}\ \{p \wedge \neg B\}}$

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

Assume:

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

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

$$\vdash \{P\}\ \mathbf{while}\ r \geq y\ \mathbf{do}\ r := r - y;\ q := q + 1\ \mathbf{do}\ \{P \wedge \neg(r \geq y)\}$$

Example Proof

(A1) $\{p\}\ skip\ \{p\}$	(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\},}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$
(A2) $\{p[u := t]\}\ u := t\ \{p\}$	(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{do}\ \{p \wedge \neg B\}}$
(R3) $\dfrac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$	(R6) $\dfrac{p \to p_1, \{p_1\}\ S\ \{q_1\}, q_1 \to q}{\{p\}\ S\ \{q\}}$

Assume:

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

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

$$\vdash \{P\}\ \textbf{while}\ r \geq y\ \textbf{do}\ r := r - y;\ q := q + 1\ \textbf{do}\ \{P \wedge \neg(r \geq y)\}$$

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

$$\vdash \{x \geq 0 \wedge y \geq 0\}\ DIV\ \{P \wedge \neg(r \geq y)\}$$

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

$$\vdash \{x \geq 0 \wedge y \geq 0\}\ DIV\ \{q \cdot y + r = x \wedge r < y\}$$

Proof: (2)

(A1) $\{p\}\ skip\ \{p\}$	(R4) $\dfrac{\{p \wedge B\}\ S_1\ \{q\}, \{p \wedge \neg B\}\ S_2\ \{q\},}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{q\}}$
(A2) $\{p[u := t]\}\ u := t\ \{p\}$	(R5) $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{do}\ \{p \wedge \neg B\}}$
(R3) $\dfrac{\{p\}\ S_1\ \{r\}, \{r\}\ S_2\ \{q\}}{\{p\}\ S_1;\ S_2\ \{q\}}$	(R6) $\dfrac{p \to p_1, \{p_1\}\ S\ \{q_1\}, q_1 \to q}{\{p\}\ S\ \{q\}}$

- $P \equiv q \cdot y + r = x \wedge r \geq 0$,
- (2): $\{P \wedge r \geq y\}\ r := r - y;\ q := q + 1\ \{P\}$

- $\{(q + 1) \cdot y + r = x \wedge x \geq 0\}\ q := q + 1\ \{P\}$ by (A2),

Proof: (2)

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

- $P \equiv q \cdot y + r = x \wedge r \geq 0$,
- (2): $\{P \wedge r \geq y\}\ r := r - y;\ q := q + 1\ \{P\}$

- $\{(q+1) \cdot y + r = x \wedge x \geq 0\}\ q := q + 1\ \{P\}$ by (A2),
- $\{(q+1) \cdot y + (r - y) = x \wedge (r - y) \geq 0\}\ r := r - y\ \{(q+1) \cdot y + r = x \wedge x \geq 0\}$ by (A2),

Proof: (2)

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

- $P \equiv q \cdot y + r = x \wedge r \geq 0$,
- (2): $\{P \wedge r \geq y\}\ r := r - y;\ q := q + 1\ \{P\}$

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

$$P \wedge r \geq y \rightarrow (q+1) \cdot y + (r - y) = x \wedge (r - y) \geq 0.$$

Proof: (1)

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

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

$$x \geq 0 \wedge y \geq 0 \rightarrow 0 \cdot y + x = x \wedge x \geq 0.$$

Once Again

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

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

Modular Reasoning

We can add a rule for function calls (simplest case: only global variables):

$$\text{(R7)}\ \frac{\{p\}\ f\ \{q\}}{\{p\}\ f()\ \{q\}}$$

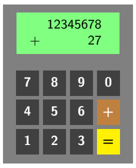
"If we have $\vdash \{p\}\ f\ \{q\}$ for the **implementation** of function f, then if f is **called** in a state satisfying p, the state after return of f will satisfy q."

p is called **pre-condition** of f, q is called **post-condition**.

Example: if we have

- $\{\mathit{true}\}$ `read_number` $\{0 \leq \mathit{ret} < 10^8\}$
- $\{0 \leq x \wedge 0 \leq y\}$ `add` $\{(\mathit{old}(x) + \mathit{old}(y) < 10^8 \wedge \mathit{ret} = \mathit{old}(x) + \mathit{old}(y)) \vee \mathit{ret} < 0\}$
- $\{\mathit{true}\}$ `display` $\{(0 \leq \mathit{old}(x) < 10^8 \implies \text{"}\mathit{old}(x)\text{"}) \wedge (\mathit{old}(x) < 0 \implies \text{"-E-"})\}$

we may be able to prove our ($\rightarrow$ later) pocket calculator correct.




```
int main() {

  while (true) {
    int x = read_number();
    int y = read_number();

    int sum = add( x, y );

    display(sum);
  }
}
```


Assertions

We add another rule for **assertions**:

$$\text{(A3)}\ \{p\}\ \texttt{assert}(p)\ \{p\}$$

- That is, if p holds **before** the assertion, then we can **continue** with the proof.
- Otherwise we **"get stuck"**.

  So we **cannot** even prove

  $$\{\mathit{true}\}\ x := 0;\ \texttt{assert}(x = 27)\ \{\mathit{true}\}.$$

  to hold (it is not derivable).

- Which is exactly what we want — if we add
  - $\langle \texttt{assert}(B), \sigma\rangle \rightarrow \langle E, \sigma\rangle$ 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:

```
ASSERT(3)                Linux Programmer's Manual                ASSERT(3)

NAME
       assert − abort the program if assertion is false

SYNOPSIS
       #include <assert.h>

       void assert(scalar expression);

DESCRIPTION
                  [...] the macro assert() prints an error message to stan
       dard 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:

```
ASSERT(3)                Linux Programmer's Manual                ASSERT(3)

NAME
       assert − abort the program if assertion is false

SYNOPSIS
       #include <assert.h>

       void assert(scalar expression);

DESCRIPTION
                  [...] the macro assert() prints an error message to stan
       dard 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.
```


- Assertions at work:

```
int square( int x )
{
  assert( x < sqrt(x) );

  return x * x;
}
```

```
void f( ... ) {
  assert( p );
  ...
  assert( q );
}
```

The Verifying C Compiler

VCC

- 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` $expr$`)` — looop 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**:
    - `\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

```
#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;
  }
}
```

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

$$\{x \geq 0 \wedge y \geq 0\}\ DIV\ \{q \cdot y + r = x \wedge r < y\}$$

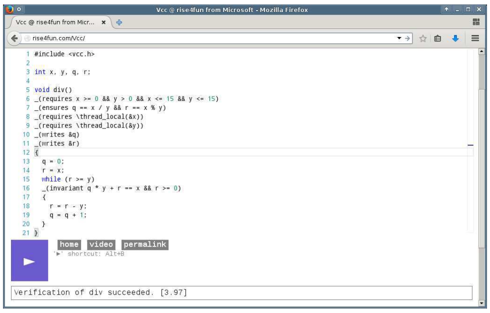
VCC Web-Interface

VCC Architecture

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 $\models \{p\}\ DIV\ \{q\}$. 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.

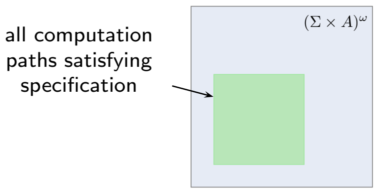
  Note: $\models \{\mathit{false}\}\ f\ \{q\}$ **always holds**
  — so a mistake in writing down the pre-condition can provoke a **false negative**.

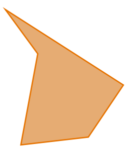
- VCC says: "**verification failed**
  - One case: "timeout" etc. — completely inconclusive outcome.
  - The tool **does not provide counter-examples** in the form of a computation path.

    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.

(Automatic) Formal Verification Techniques

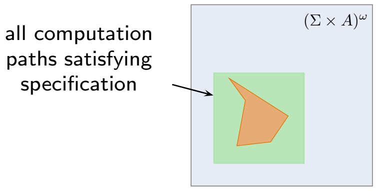


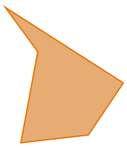


Investigate All Paths

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

(Automatic) Formal Verification Techniques

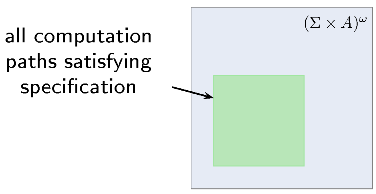


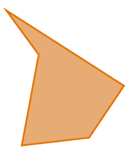


Investigate All Paths

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

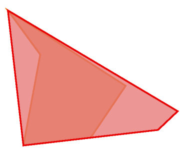
(Automatic) Formal Verification Techniques





Investigate All Paths

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

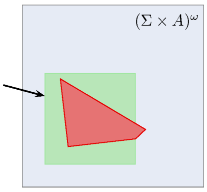


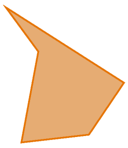
Over-Approximation

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

(Automatic) Formal Verification Techniques

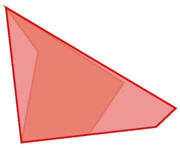
all computation paths satisfying specification





Investigate All Paths

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

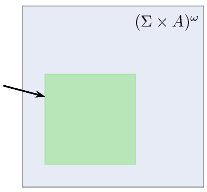


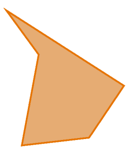
Over-Approximation

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

(Automatic) Formal Verification Techniques

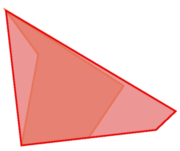
all computation paths satisfying specification





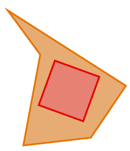
Investigate All Paths

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



Over-Approximation

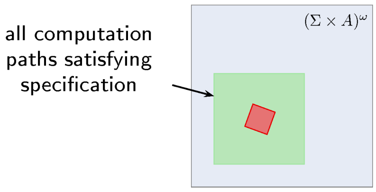
(some Software model-checkers; goal: verify correctness; false positives, no false negatives)

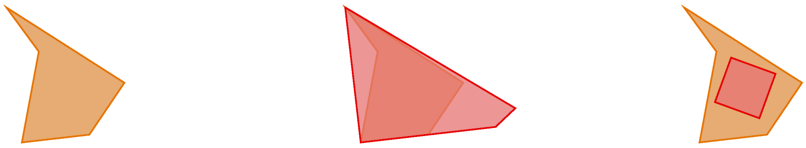


Under-Approximation

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

(Automatic) Formal Verification Techniques





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

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

ISO (2011). *Road vehicles – Functional safety – Part 1: Vocabulary*. 26262-1:2011.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.