Lecture 16: Software Verification

2017-07-20

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
**Introduction and Vocabulary**
- Test case, test suite, test execution.
- Positive and negative outcomes.

**Limits of Software Testing**

**Glass-Box Testing**
- Statement-, branch-, term-coverage.

**Testing: Rest**
- When to stop testing?
- Model-based testing
- Testing in the development process

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

**Other Approaches**
- Runtime verification.
- Review

**Software quality assurance wrap-up**
Content

- **Testing**: Rest
  - Model-Based Testing
  - When To Stop Testing?
  - Testing in the Development Process

- **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**
Model-Based Testing
• Does some software **implement** the given CFA model of the CoinValidator?

• **One approach: Location Coverage.**

  Check whether for each location of the model there is a **corresponding configuration** reachable in the software (needs to be observable somehow).

• Input sequences can **automatically be generated** from the model, e.g., using Uppaal’s “drive-to” feature.

  • Check “can we reach ‘idle’, ‘have_c50’, ‘have_c100’, ‘have_c150’?” by

    \[ T_1 = (\text{C50, C50, C50}; \{ \pi \mid \exists i < j < k < \ell \cdot \pi^i \sim \text{idle}, \pi^j \sim \text{h_c50}, \pi^k \sim \text{h_c100}, \pi^\ell \sim \text{h_c150} \}) \]

  • Check for ‘have_e1’ by \( T_2 = (\text{C50, C50, C50}; \ldots) \).

  • To check for ‘drink_ready’, more interaction is necessary.

• **Analogously: Edge Coverage.**

  Check whether each edge of the model has **corresponding** behaviour in the software.
If the LSC has designated **environment instance** lines, we can distinguish:

- messages expected to **originate from** the environment (driver role),
- messages expected **addressed to** the environment (monitor role).

Adjust the TBA-construction algorithm to construct a **test driver & monitor** and let it (possibly with some **glue logic** in the middle) interact with the software.

**Test passed** (i.e., test unsuccessful) if and only if TBA state $q_6$ is reached.

*Note:* We may need to **refine** the LSC by adding an activation condition; or communication which drives the system under test into the desired start state.

For example the **Rhapsody** tool directly supports this approach.
- Software-in-the-loop:
The final implementation is examined using a separate computer to simulate other system components.

- Hardware-in-the-loop:
The final implementation is running on (prototype) hardware which is connected by its standard input/output interface (e.g. CAN-bus) to a separate computer which simulates other system components.
Content

- **Testing**: Rest
  - Model-Based Testing
  - When To Stop Testing?
  - Testing in the Development Process

- **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**
When To Stop Testing?
When To Stop Testing?

- There need to be defined criteria for when to stop testing; project planning should consider these criteria (and previous experience).

- Possible “testing completed” criteria:
  - all (previously) specified test cases have been executed with negative result,
    (Special case: All test cases resulting from a certain strategy, like maximal statement coverage have been executed.)
  - testing effort time sums up to \( x \) (hours, days, weeks),
  - testing effort sums up to \( y \) (any other useful unit),
  - \( n \) errors have been discovered,
  - no error has been discovered during the last \( z \) hours (days, weeks) of testing,

Values for \( x, y, n, z \) are fixed based on experience, estimation, budget, etc.

- Of course: not all criteria are equally reasonable or compatible with each testing approach.
Another Criterion

- Another possible “testing completed” criterion:

- The average cost per error discovery exceeds a defined threshold $c$.

![Graph showing number of discovered errors and cost per discovered error over time.]

Value for $c$ is again fixed based on experience, estimation, budget, etc..
Testing in The Software Development Process
**Test Conduction: Activities & Artefacts**

(Ludewig and Lichter, 2013)

- **Test Gear**: (may need to be developed in the project!)

  - **test driver**— A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results.
  
  Synonym: test harness.

  IEEE 610.12 (1990)

  - **stub**—
    1. A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent on it.
    2. A computer program statement substituting for the body of a software module that is or will be defined elsewhere.

  IEEE 610.12 (1990)

- **Roles**: tester and developer should be different persons!
Content

- **Testing**: Rest
  - Model-Based Testing
  - When To Stop Testing?
  - Testing in the Development Process

- **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**
Customer 2

**Mmmh, Software!**

**Requirements**

\[ \mathcal{S}_1 = \{(M.C, \mathcal{T}_1), (C.M, \mathcal{T}_1)\} \]

**Design**

\[ \mathcal{S}_2 = \{(M.T_M.C, \mathcal{T}_2), (C.T_C.M, \mathcal{T}_2)\} \]

\[ \mathcal{S}_1 = \{\sigma_0^{\alpha_1} \rightarrow \sigma_1^{\alpha_2} \rightarrow \sigma_2 \cdots, \ldots\} \]

**Implementation**

\[ \mathcal{S}_2 = \{\sigma_0^{\alpha_1} \rightarrow \sigma_1^{\alpha_2} \rightarrow \sigma_2 \cdots, \ldots\} \]
validate

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)
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.

Examination with computer

Analyse

Execute

Prove

Formal verification

Manual proof

Inspection

Review

e.g. interactive prover

Static checking

Check against rules

Consistency checks

Quantitative examination

Dynamic checking (test)

E.g. code generation

(Ludewig and Lichter, 2013)
all computation paths satisfying the specification

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

defines

LSC: buy water
AC: true
AM: invariant I: strict

User
CoinValidator
ChoicePanel
Dispenser

C
50
WATER
¬ (C50! ∨ E1! ∨ pSOFT! ∨ pTEA! ∨ pFILLUP!)

water _in_stock

dWATER OK
¬ (dSoft! ∨ dTEA!)

all computation paths satisfying the specification

expected outcomes $S_{\text{oll}}$

$(\Sigma \times A)^\omega$

defines

execution of $(\text{In}, S_{\text{oll}})$

$\in \,$?

$\subseteq \,$?

Reviewer

review

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

input $\rightarrow$ output

Testing

Review

Formal Verification

prove $S \models \mathcal{I}$,
... so, off to “‘technological paradise’ where [...] everything happens according to the blueprints”.

(Kopetz, 2011; Lovins and Lovins, 2001)
Content

- **Testing**: Rest
  - Model-Based Testing
  - When To Stop Testing?
  - Testing in the Development Process

- **Formal Program Verification**
  - Deterministic Programs
    - Syntax
    - Semantics
    - Termination, Divergence
  - **Correctness** of deterministic programs
    - partial correctness,
    - total correctness.
  - Proof System PD

- **The Verifier for Concurrent C**
Sequential, Deterministic While-Programs
Deterministic Programs

Syntax:

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

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

Semantics: (is induced by the following transition relation) – \( \sigma : V \rightarrow D(V) \)

(i) \( \langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle \) \hspace{1cm} \text{empty program}

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

(iii) \( \langle S_1, \sigma \rangle \rightarrow \langle S_2, \tau \rangle \)

\( \langle S_1 ; S, \sigma \rangle \rightarrow \langle S_2 ; S, \tau \rangle \)

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

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

(vii) \( \langle \text{while } B \text{ do } S \text{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle, \text{ if } \sigma \nmid B \)

\( E \) denotes the empty program; define \( E ; S \equiv S ; E \equiv S \).

Note: the first component of \( \langle S, \sigma \rangle \) is a program (structural operational semantics (SOS)).
Example

Consider program

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

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

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

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

\[
\begin{align*}
\langle S_1, \sigma \rangle &\xrightarrow{(ii),(iii)} \langle y := (x - 1) \cdot x + y, \{x \mapsto 3, y \mapsto 3\} \rangle \\
&\xrightarrow{(ii)} \langle E, \{x \mapsto 3, y \mapsto 9\} \rangle
\end{align*}
\]

Consider program

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

\[
\begin{align*}
\langle S_3, \sigma \rangle &\xrightarrow{(ii),(iii)} \langle y := (x - 1) \cdot x + y; \text{ while } 1 \text{ do skip od, } \{x \mapsto 3, y \mapsto 3\} \rangle \\
&\xrightarrow{(ii),(iii)} \langle \text{ while } 1 \text{ do skip od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
&\xrightarrow{(vi)} \langle \text{ skip; while } 1 \text{ do skip od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
&\xrightarrow{(i),(iii)} \langle \text{ while } 1 \text{ do skip od, } \{x \mapsto 3, y \mapsto 9\} \rangle \\
&\xrightarrow{(vi)} \cdots
\end{align*}
\]
**Definition.** Let $S$ be a deterministic program.

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

$$\langle S, \sigma \rangle = \langle S_0, \sigma_0 \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \ldots$$

(that is, $\langle S_i, \sigma_i \rangle$ and $\langle S_{i+1}, \sigma_{i+1} \rangle$ are in transition relation for all $i$).

(ii) A **computation (path)** of $S$ (starting in $\sigma$) is a *maximal* transition sequence of $S$ (starting in $\sigma$), i.e. infinite or not extendible.

(iii) A computation of $S$ is said to

a) **terminate** in $\tau$ if and only if it is finite and ends with $\langle E, \tau \rangle$,

b) **diverge** if and only if it is infinite.

$S$ can **diverge from** $\sigma$ if and only if a diverging computation starts in $\sigma$.

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

---

**Lemma.** For each deterministic program $S$ and each state $\sigma$, there is exactly one computation of $S$ which starts in $\sigma$. 
Definition. Let $S$ be a deterministic program.

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

\[ M[S] : \Sigma \rightarrow 2^{\Sigma \times \text{finitely many}} \]

with

\[ M[S](\sigma) = \{ \tau \mid \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \} . \]

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

\[ M_{tot}[S] : \Sigma \rightarrow 2^{\Sigma} \cup \{ \infty \} \]

with

\[ M_{tot}[S](\sigma) = M[S](\sigma) \cup \{ \infty \mid S \text{ can diverge from } \sigma \} . \]

$\infty$ is an error state representing divergence.

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

**Example:**

\[ M[S_1](\sigma) = M_{tot}[S_1](\sigma) = \{ \tau \mid \tau(x) = \sigma(x) \land \tau(y) = \sigma(x)^2 \} , \quad \sigma \in \Sigma. \]

(Recall: $S_1 \equiv y := x; y := (x - 1) \cdot x + y$)
Correctness of While-Programs
Definition.
Let $S$ be a program over variables $V$, and $p$ and $q$ Boolean expressions over $V$.

(i) The correctness formula

\[
\{p\} \ S \ \{q\}
\]

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

\[
(M[S][[p]]) \subseteq [[q]].
\]

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

\[
M_{tot}[S][[p]] \subseteq [[q]].
\]

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 $S_1$:**

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

\[ \models \{ p \} S_1 \{ q \} \checkmark \]

\[ \models_{tot} \{ p \} S_1 \{ q \} \checkmark \]

**Program $S_2$:**

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

\[ \models \{ p \} S_2 \{ q \} \checkmark \]

\[ \models_{tot} \{ p \} S_2 \{ q \} \times \]

**Program $S_3$:**

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

\[ \models \{ p \} S_3 \{ q \} \times \]

\[ \models_{tot} \{ p \} S_3 \{ q \} \times \]

**Program $S_4$:**

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

\[ \models \{ p \} S_4 \{ q \} \checkmark \times \text{ (overflow) } \]

\[ \models_{tot} \{ p \} S_4 \{ q \} \checkmark \times \]
Example: Correctness

- By the example, we have shown
  \[ \models \{ x = 0 \} \text{S} \{ x = 1 \} \]

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

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

- We have also shown (= proved (!)):
  \[ \models \{ x = 0 \} \text{S} \{ x = 1 \land a[x] = 0 \}. \]

- The correctness formula \( \{ x = 2 \} \text{S} \{ \text{true} \} \) does not hold for \( \text{S} \).
  (For example, if \( \sigma \models a[i] \neq 0 \) for all \( i > 2 \).)

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

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

\[
\begin{align*}
(S, \sigma) & \xrightarrow{(vi),(vii)} (S, \sigma) \\
& \xrightarrow{(i),(vi)} (a[1] := 0; \text{while} \ a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma) \\
& \xrightarrow{(vii)} (x := x + 1; \text{while} \ a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma') \\
& \xrightarrow{(vi)} (\text{while} \ a[x] \neq 0 \text{ do } x := x + 1 \text{ od}, \sigma'[x := 1]) \\
& \xrightarrow{(vii)} (E, \sigma'[x := 1])
\end{align*}
\]

where \( \sigma' = \sigma[a[0] := 1][a[1] := 0] \).
Proof-System PD
Proof-System PD (for sequential, deterministic programs)

Axiom 1: **Skip-Statement**

\[{p} \text{ skip } \{p\}\]

Axiom 2: **Assignment**

\[{p[u := t]} \text{ } u := t \{p\}\]

**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**

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

**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 \(\models \{p\} S \{q\}\).
Example Proof

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
\begin{align*}
\text{(1)} & \quad \{ p^D \} \ S_0^D \ \{ P \}, \ \text{while } B^D \ \text{do } S_1^D \ \text{od} \ \{ p^D \} \\
\text{(2)} & \quad \{ P \} \ \text{while } B^D \ \text{do } S_1^D \ \text{od} \ \{ p^D \} \\
\text{(3)} & \quad \{ p^D \} \ S_0^D \ \{ P \}, \ \text{while } B^D \ \text{do } S_1^D \ \text{od} \ \{ q^D \}
\end{align*}
\]

\[
\begin{align*}
\text{(A1)} & \quad \{ p \} \ \text{skip} \ \{ p \} \\
\text{(A2)} & \quad \{ p[u := t] \} \ u := t \ \{ p \} \\
\text{(R3)} & \quad \{ p \} \ S_1 \ \{ r \}, \ \{ r \} \ S_2 \ \{ q \} \\
\text{(R4)} & \quad \{ p \} \ S_1; \ S_2 \ \{ q \} \\
\text{(R5)} & \quad \{ p \} \ \text{while } B \ \text{do } S \ \text{od} \ \{ p \land \neg B \} \\
\text{(R6)} & \quad \{ p \} \ \text{if } B \ \text{then } S_1 \ \text{else } S_2 \ \text{fi} \ \{ q \} \\
\end{align*}
\]
In the following, we show

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

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

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

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

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

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

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

thus, \[ \vdash_{PD} \{ 0 \cdot y + x = x \land x \geq 0 \} \ a := 0; \ b := x \{ P \} \quad \text{by (R3)}, \]

using \( x \geq 0 \land y \geq 0 \rightarrow 0 \cdot y + x = x \land x \geq 0 \) and \( P \rightarrow P \), we obtain
\[ \vdash_{PD} \{ x \geq 0 \land y \geq 0 \} \ a := 0; \ b := x \{ P \} \]
by (R6).
The rule ‘Assignment’ uses (syntactical) substitution: \( \{ p[u := t] \} u := t \{ p \} \)
(In formula \( p \), replace all (free) occurences 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, \( u \equiv a[t_1, \ldots, t_m] \):
  \( (a[s_1, \ldots, s_n])[u := t] \equiv (\land_{i=1}^n s_i[u := t] = t \ ? t : a[s_1[u := t], \ldots, s_n[u := t]]) \)

**Formulae:**

- boolean expression \( p \equiv s \):
  \( p[u := t] \equiv s[u := t] \)
- negation:
  \( (\neg q)[u := t] \equiv \neg(q[u := t]) \)
- conjunction etc.:
  \( (q \land r)[u := t] \equiv q[u := t] \land r[u := t] \)
- quantifier:
  \( (\forall x : q)[u := t] \equiv \forall y : q[x := y][u := t] \)
  \( y \) fresh (not in \( q, t, u \), same type as \( x \).)
Proof of (2)

• (2) claims:

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

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

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

by (A2).

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

by (A2).

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

by (R3).

• using \( P \land b \geq y \rightarrow (a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \) and \( P \rightarrow P \) we obtain,

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

by (R6).
Proof of (3)

(3) claims

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

where $P \equiv a \cdot y + b = x \land b \geq 0$.

Proof: easy.
We have shown:

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

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

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

and

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

thus

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

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

- \( P \equiv a \cdot y + b = x \land b \geq 0 \)
  - \( \{ x \geq 0 \land y \geq 0 \} \)
  - \( \{ 0 \cdot y + x = x \land x \geq 0 \} \)
- \( a := 0; \)
  - \( \{ a \cdot y + x = x \land x \geq 0 \} \)
- \( b := x; \)
  - \( \{ a \cdot y + b = x \land b \geq 0 \} \)
  - \( \{ P \} \)
- **while** \( b \geq y \) **do**
  - \( \{ P \land b \geq y \} \)
  - \( \{(a + 1) \cdot y + (b - y) = x \land (b - y) \geq 0 \} \)
  - \( b := b - y; \)
  - \( \{(a + 1) \cdot y + b = x \land b \geq 0 \} \)
  - \( a := a + 1 \)
  - \( \{ a \cdot y + b = x \land b \geq 0 \} \)
  - \( \{ P \} \)
- **od**
  - \( \{ P \land \neg(b \geq y) \} \)
  - \( \{ a \cdot y + b = x \land b < y \} \)
Literature Recommendation

Tell Them What You’ve Told Them…

Testing:
- Define criteria for “testing done” (like coverage, or cost per error).
- Process: tester and developer should be different persons.

Formal Verification:
- There are more approaches to software quality assurance than just testing.
- For example, program verification.
- Proof System PD can be used
  - to prove
  - that a given program is correct wrt. its specification.

  This approach considers all inputs inside the specification!
- Tools like VCC implement this approach.
References
References


