Correctness

Zsolt Borsi

Correctness

Zsolt Borsi

Publication date 2014

Supported by TÁMOP-4.1.2.A/1-11/1-2011-0052.

Table of Contents

Chapter 1. Introduction

1. The syntax of structured programs

$$\begin{aligned} S ::= \quad & \mathbf{skip} \mid y \leftarrow f(x,y) \mid S_1; S_2 \mid \\ & \mathbf{if}\ \alpha(x,y)\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \mid \\ & \mathbf{while}\ \alpha(x,y)\ \mathbf{do}\ S_1\ \mathbf{od} \end{aligned}$$

2. The syntax of nondeterministic programs

$$\begin{aligned} S ::= \quad & \mathbf{skip} \mid b \to y \leftarrow f(x,y) \mid S_1; S_2 \mid \\ & \mathbf{if}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{fi} \mid \\ & \mathbf{do}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{od} \end{aligned}$$

- Upon execution of a selection all guards are evaluated. If none of the guards evaluates to true then execution of the selection aborts, otherwise one of the guards that has the value true is chosen nondeterministically and the corresponding statement is executed.
- Upon execution of a repetition all guards are evaluated. If all guards evaluate to false then skip is executed and the program terminates. Otherwise one of the guards that has the value true is chosen nondeterministically and the corresponding statement is executed. The repetition is executed again until all guards evaluate to false.

3. The semantics of deterministic programs

- $\langle \mathbf{skip}, \sigma \rangle \to \langle E, \sigma \rangle$
- $\langle y \leftarrow f(x,y), \sigma \rangle \to \langle E, \sigma(y \leftarrow f(x,y)) \rangle$
- $\frac{\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle}{\langle S_1; S, \sigma_1 \rangle \to \langle S_2; S, \sigma_2 \rangle}$
- $\sigma(\alpha) = true \Longrightarrow \langle \mathbf{if}\ \alpha\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, \sigma \rangle \to \langle S_1, \sigma \rangle$
- $\neg\sigma(\alpha) = true \Longrightarrow \langle \mathbf{if}\ \alpha\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, \sigma \rangle \to \langle S_2, \sigma \rangle$
- $\sigma(\alpha) = true \Longrightarrow \langle \mathbf{while}\ \alpha\ \mathbf{do}\ S\ \mathbf{od}, \sigma \rangle \to \langle S;\ \mathbf{while}\ \alpha\ \mathbf{do}\ S\ \mathbf{od}, \sigma \rangle$
- $\neg\sigma(\alpha) = true \Longrightarrow \langle \mathbf{while}\ \alpha\ \mathbf{do}\ S\ \mathbf{od}, \sigma \rangle \to \langle E, \sigma \rangle$

where $\sigma(\alpha) = true$, if $\alpha(x,y)$ is true in state σ

4. The semantics of nondeterministic programs

- $\sigma(b) = true \Longrightarrow \langle b \to y \leftarrow f(x,y), \sigma \rangle \to \langle E, \sigma(b \to y \leftarrow f(x,y)) \rangle$
- $\neg\sigma(b) = true \Longrightarrow \langle b \to y \leftarrow f(x,y), \sigma \rangle \to \langle E, fail \rangle$
- $(\exists i \in [1..n] : \sigma(\alpha_i = true)) \Longrightarrow \langle \mathbf{if}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{fi}, \sigma \rangle \to \langle S_i, \sigma \rangle$
- $(\forall i \in [1..n] : \sigma(\alpha_i = false)) \Longrightarrow \langle \mathbf{if}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{fi}, \sigma \rangle \to \langle E, fail \rangle$
- $(\exists i \in [1..n] : \sigma(\alpha_i = true)) \Longrightarrow \langle \mathbf{do}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{od}, \sigma \rangle \to \langle S_i; \mathbf{do}\ \alpha_1 \to S_1 \ \Box \ldots \Box\ \alpha_n \to S_n\ \mathbf{od}, \sigma \rangle$

- $(\forall i \in [1..n] : \sigma(\alpha_i = false)) \Longrightarrow \langle \mathbf{do}\ \alpha_1 \rightarrow S_1 \square \ldots \square\ \alpha_n \rightarrow S_n\ \mathbf{od}, \sigma\rangle \rightarrow E, \sigma\ \langle\rangle$

- $(\forall i \in [1..n] : \sigma(\alpha_i = false)) \Longrightarrow \langle \mathbf{do}\ \alpha_1 \rightarrow S_1 \square \ldots \square\ \alpha_n \rightarrow S_n\ \mathbf{od}, \sigma\rangle \rightarrow E, \sigma\ \langle\rangle$

Chapter 2. Hoare calculus

1. Hoare triple

- A Hoare triple is a proposition in the form of $\{P\}\,S\,\{Q\}$ where S is a program, P and Q are assertions about the program variables used in S.
- This notation is due to C.A.R. Hoare. The original notation was $P\,\{S\}\,Q$, not $\{P\}\,S\,\{Q\}$ but the latter form is now more widely used. The notation is introduced for specifying what a program does.
- Partial correctness: $\{P\}\,S\,\{Q\}$ means that if P is true before execution of S, then Q is true after execution of S, provided S terminates. Nothing is supposed about termination; abortion and non-termination are not ruled out.
- Total correctness: whenever S is executed in a state satisfying P the execution of S is guaranteed to terminate and after S terminates Q holds.
- In the following the total correctness meaning of a Hoare triple is denoted by $[P]S[Q]$.

2. Partial vs total correctness

- The difference between the two notions is the way how termination is dealt with: total correctness requires termination, whereas partial correctness assumes it.
- Thus the relationship between partial and total correctness can be informally expressed by the equation:

  $Total\ correctness = Termination + Partial\ correctness$

  In practice, it is usually easier to show partial correctness and termination separately.
- In the case of partial correctness, the specification $\{P\}\,S\,\{Q\}$ is partial because for $\{P\}\,S\,\{Q\}$ to be true it is not necessary for the execution of S to terminate when started in a state satisfying P. It is only required that if the execution terminates, then Q holds.

  $\{x=1\}$ $\mathbf{while}\ true\ \mathbf{do}\ x \leftarrow 5\ \mathbf{od}$ $\{x<0\}$ specification is true!

3. Statements

- Empty statement

  $\mathbf{skip}$
- Assignment statement

  $y \leftarrow g(x,y)$
- Conditional statement

  $\mathbf{if}\ t(x,y)\ \mathbf{then}\ B\ \mathbf{else}\ B'$
- While statement

  $\mathbf{while}\ t(x,y)\ \mathbf{do}\ B\ \mathbf{od}$
- Sequence

$S_1; S_2$

4. Verification method

- Logical derivation

  $$\frac{A_1 \text{ and } A_2}{B}$$

  If we have shown A_1 and A_2, then we have also shown B. In other words: to show that B holds, it suffices to show A_1 and A_2.

- Using such notation, Hoare introduced verification rules and described a deductive system for proving correctness of sequential programs.

- The derived verification rules are obtained from existing verification rules and are more convenient to use.

5. Verification rules 1.

- Skip statement axiom

  $$\{P(x,y)\}\,\mathbf{skip}\,\{P(x,y)\}$$

- Assignment axiom

  $$\{P(x,g(x,y))\}\,y \leftarrow g(x,y)\,\{P(x,y)\}$$

  Examples:

  $$\{x+2<5\}\,x \leftarrow x+2\,\{x<5\}$$

  $$\{y=2\}\,x \leftarrow 0\,\{x \geq 0 \wedge y = x+2\}$$

  $$\{x > (y-1)((y-1)-1)\}\,z \leftarrow y-1\,\{x > z(z-1)\}$$

6. Verification rules 2.

- Conditional rule

  $$\frac{\{P \wedge \alpha\}\,S_1\,\{Q\}\,\text{and}\,\{P \wedge \neg\alpha\}\,S_2\,\{Q\}}{\{P\}\ \mathbf{if}\ \alpha\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\,\{Q\}}$$

  Example:

  $$\frac{\{true \wedge x>5\}\,y \leftarrow x\,\{y \geq 5\}\,\text{and}\,\{true \wedge x \leq 5\}\,y \leftarrow 5\,\{y \geq 5\}}{\{true\}\ \mathbf{if}\ x>5\ \mathbf{then}\ y \leftarrow x\ \mathbf{else}\ y \leftarrow 5\ \mathbf{fi}\,\{y \geq 5\}}$$

- While rule

  $$\frac{\{P \wedge \alpha\}\,S\,\{P\}}{\{P\}\ \mathbf{while}\ \alpha\ \mathbf{do}\ S\ \mathbf{od}\,\{P \wedge \neg\alpha\}}$$

7. Verification rules 3.

- Concatenation rule

$$\frac{\{P\}\, S_1\, \{Q_1\} \text{ and } \{Q_1\}\, S_2\, \{Q\}}{\{P\}\, S_1; S_2\, \{Q\}}$$

Example:

If one has deduced the two following propositions: $\{true\}\, d \leftarrow 1\, \{d = 1\}$ and $\{d = 1\}\, h \leftarrow 10\, \{h = 10^d\}$,

then one can deduce: $\{true\}\, d \leftarrow 1; h \leftarrow 10\, \{h = 10^d\}$

Example:

$$\frac{\begin{array}{l}\{x - y - 1 > 0\}\, y \leftarrow y + 1\, \{x - y > 0\} \text{ and} \\ \{x - y > 0\}\, y \leftarrow x - y\, \{y > 0\}\end{array}}{\{x - y - 1 > 0\}\, y \leftarrow y + 1; y \leftarrow x - y\, \{y > 0\}}$$

- Consequence rule

$$\frac{P \Rightarrow P_1 \text{ and } \{P_1\}\, S\, \{Q_1\} \text{ and } Q_1 \Rightarrow Q}{\{P\}\, S\, \{Q\}}$$

8. Derived verification rules

- Assignment rule (assignment axiom + consequence rule)

$$\frac{P(x, y) \Rightarrow Q(x, g(x, y))}{\{P(x, y)\}\, y \leftarrow g(x, y)\, \{Q(x, y)\}}$$

- Modified concatenation rule (concatenation rule + consequence rule)

$$\frac{\begin{array}{l}P \Rightarrow P_1 \text{ and } \{P_1\}\, S_1\, \{Q_1\} \text{ and } Q_1 \Rightarrow P_2 \text{ and} \\ \{P_2\}\, S_2\, \{Q_2\} \text{ and } Q_2 \Rightarrow Q\end{array}}{\{P\}\, S_1; S_2\, \{Q\}}$$

- Modified conditional rule (conditional rule + consequence rule)

$$\frac{\begin{array}{l}P \Rightarrow P_1 \text{ and } \{P_1 \wedge \alpha\}\, S_1\, \{Q_1\} \text{ and} \\ \{P_1 \wedge \neg\alpha\}\, S_2\, \{Q_1\} \text{ and } Q_1 \Rightarrow Q\end{array}}{\{P\} \textbf{ if } \alpha \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi } \{Q\}}$$

9. Modified while rule

- The loop invariant I is such a property, that
  - it is true when the loop is reached (i.e. P implies I)
  - when the test α also holds, after the execution of the loop body the invariant still holds
  - when the loop terminates (i.e. if α does not hold) the desired result is given (i.e. I implies Q)
- With the introduction of the notion of loop invariant, the while rule can be expressed as

$$\frac{\begin{array}{l} P \Rightarrow I \\ \{I \land \alpha\}\, S\, \{I\} \\ I \land \neg\alpha \Rightarrow Q \end{array}}{\{P\}\ \textbf{while}\ \alpha\ \textbf{do} S\ \textbf{od}\ \{Q\}}$$

10. Annotations

- Before starting the proof, it is helpful to insert some assertions to some certain points of the program. Annotations are enclosed in curly brackets and are intended to hold whenever control reaches them.
- Using the properties of a loop invariant Inv an annotated loop looks like the following:

  $\{Inv\}$

  $\textbf{while}\ \alpha\ \textbf{do}$

  $\{Inv \land \alpha\}$

  S

  $\{Inv\}$

  od

  $\{Inv \land \neg\alpha\}$

11. Backward reasoning 1.

- If C is a command and Q a predicate

  $wp(C,Q)$ is a precondition for C that ensures Q as a postcondition (i.e. $\{wp(C,Q)\}\, C\, \{Q\}$)
- $wp(C,Q)$ is the weakest such precondition

  It means, that it suffices to prove $P \Rightarrow wp(C,Q)$ when we want to prove $\{P\}\, C\, \{Q\}$.
- Consider the following program and specification:

  $S: \qquad x \leftarrow 3; y \leftarrow y-1; z \leftarrow x*(y+1)$

  We want to determine a condition P so that $\{P\}\, S\, \{z<15\}$ holds.

  step 1: Due to the assignment rule it suffices to make sure that $x*(y+1<15)$ holds before the assignment $z \leftarrow x*(y+1)$

  step 2: Due to the assignment rule it suffices to make sure that $x*y<15$ holds before the assignment $y \leftarrow y-1$

  step 3: Due to the assignment rule it suffices to make sure that $y<5$ holds before the assignment $x \leftarrow 3$

12. Backward reasoning 2.

The annotated program:

$\{y<5\}$

$x \leftarrow 3$

$\{x * y < 15\}$

$y \leftarrow y - 1$

$\{x * (y + 1) < 15\}$

$z \leftarrow x * (y + 1)$

$\{z < 15\}$

13. Partial correctness example 1.

Proove the partial correctness of the following program with respect to the input predicate $\varphi(x)$ and the output predicate $\psi(x, z)$!

$\varphi(x) = x \geq 0$

$\psi(x, z) = z^2 \leq x \leq (z + 1)^2$

$\{x \geq 0\}$

$(y_1, y_2, y_3) \leftarrow (0, 1, 1)$; **while** $y_2 \leq y$ **do** $(y_1, y_2, y_+) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;

$(z) \leftarrow (y_1)$

$\{z^2 \leq x < (z + 1)^2\}$

14. Partial correctness example 2.

Let us introduce the following assertion: $R(x, y_1, y_2, y_3) = (y_1^2 \leq x) \wedge (y_2 = (y_1 + 1)^2) \wedge (y_3 = 2 \cdot y_1 + 1)$

The proof will rely on the following lemmas:

1. lemma1: $x \geq 0 \Rightarrow R(x, 0, 1, 1)$

2. lemma2: $(R(x, y_1, y_2, y_3) \wedge y_2 \leq x) \Rightarrow R(x, y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$

3. lemma3: $(R(x, y_1, y_2, y_3) \wedge y_2 > x) \Rightarrow y_1^2 \leq x < (y_1 + 1)^2$

We give the proof only of lemma3:

$y_1^2 \leq x$ since it is contained in $R(x, y_1, y_2, y_3)$ that we know.

$x < (y_1 + 1)^2$ holds since $x < y_2 \wedge y_2 = (y_1 + 1)^1$, where the latter is included in $(R(x, y_1, y_2, y_3)$

15. Partial correctness example 3.

Full proof of partial correcness

- 1. step

  $\{x \geq 0\}$

  $(y_1, y_2, y_3) \leftarrow (0, 1, 1)_{;}\ \{R(x, y_1, y_2, y_3)\}$

  Proof: lemma1 and assignment axiom

- 2. step

  $\{R(x, y_1, y_2, y_3) \wedge y_2 \leq x\}$

  $(y_1, y_2, y_+) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$

  $\{R(x, y_1, y_2, y_3)$

  Proof: lemma2 and assignment axiom

- 3. step

  $\{R(x, y_1, y_2, y_3)\}$

  $\textbf{while}\ \ y_2 \leq y\ \ \textbf{do}\ (y_1, y_2, y_+) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)_{;}$

  $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$

  Proof: while rule

16. Partial correctness example 4.

- 4. step

  $\{x \geq 0\}$

  $(y_1, y_2, y_3) \leftarrow (0, 1, 1)_{;}$

  $\textbf{while}\ \ y_2 \leq y\ \ \textbf{do}\ (y_1, y_2, y_+) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)_{;}$

  $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$

  Proof: concatenation rule for program fragments given in step 1. and step 3.

- 5. step

  $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$

  $(z) \leftarrow (y_1)$

  $\{z^2 \leq x < (z+1)^2\}$

  Proof: lemma3 and assignment axiom

17. Partial correctness example 5.

- 6. step

  By applying the concatenation rule for specifications given in step 4. and step 5. we get:

$\{x \geq 0\}$

$(y_1, y_2, y_3) \leftarrow (0, 1, 1)$; $\textbf{while}\ y_2 \leq y\ \textbf{do}\ (y_1, y_2, y_+) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;

$(z) \leftarrow (y_1)$

$\{z^2 \leq x < (z+1)^2\}$

Chapter 3. A relational model of sequential programs

1. Basic notions 1.

- Sets

  Let $\mathbb{N}$ denote the set of all natural numbers, $\mathbb{N}_0$ the set of all nonnegative integers, $\mathbb{Z}$ the set of all integers, $\mathbb{L}$ the set of logical values. $\emptyset$ denotes the empty set.

- Sequences

  $< \alpha_1, \ldots, \alpha_n >$, $\alpha_i \in A$ denotes a finite sequence of length n of elements of A .

  $< \alpha_1, \ldots >$, $\alpha_i \in A$ denotes an infinite sequence of elements of A .

  A^* : the set of finite sequences constructed from the elements of A .

  A^∞ : the set of infinite sequences constructed from the elements of A .

  Let $A^{**} = A^* \cup A^\infty$. A^{**} denotes the set of all finite and infinite sequences of the elements of A

2. Basic notions 2.

- Relations

  The relations are applicable to describe nondeterministic programs. Any subset of any direct product is called a relation. $R \subseteq A \times B$ is called binary relation. Relation means binary relation in the following.

  $R \subseteq A \times B$. $R(a) = \{b \in B \mid (a, b) \in R\}$

  The domain of R is $D_R = \{a \in A \mid R(a) \neq \emptyset\}$

  The range of R is $R_R = \bigcup_{a \in A} R(a)$

  The relation $R \subseteq A \times B$ is a deterministic relation, if $\forall a \in A : |R(a)| \leq 1$.

  The relation $R \subseteq A \times B$ is a function, if $\forall a \in A : |R(a)| = 1$. Let denote such a relation by $R \colon A \to B$.

  $R_2 \circ R_1 = \{(a, c) \in A \times C \mid \exists b \in B \colon (a, b) \in R_1 \wedge (b, c) \in R_2\}$ is the rational composition of the relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$.

3. Abstract mathematical definition of programming notions 1.

Definition 3.1. Let $A_1, \ldots, A_n$ be arbitrary finite or numerable sets. The set $A = A_1 \times \ldots \times A_n$ is called the state space. The components of the state space, i.e. the sets A_i are called type value sets. This name refers to the fact that each component of the state space is the range of a type. A state is a comound of the values of the important data types of the problem.

Definition 3.2. The projections $v_i \colon A \to A_i$ of the state space $A = A_1 \times \ldots \times A_n$ are called variables.

Definition 3.3. Any homogeneous binary relation $F \subseteq A \times A$ is called a problem. The problem is a relation over the state space that maps from the possible initial states to the expected goal states.

4. Abstract mathematical definition of programming notions 2.

An execution of a program is a sequence of states. The program is a relation, which associates a sequence of the points of the state space to the points of the state space. The program is defined as all of its executions so it can be described by the relation that maps form any state to the executions starting from the given point. This model allows nondeterminism: if several executions start from the same state it means that the program is non-deterministic: any execution may happen.

Definition 3.4. A relation $S \subseteq A \times A^{**}$ is called a program, if

1.

$$D_S = A$$

2.

$$\forall \alpha \in R_S : \alpha = red(\alpha)$$

3.

$$\forall a \in A : \forall \alpha \in S(a) : |\alpha| \geq 1 \wedge \alpha_1 = a$$

The reduced sequence $red(\alpha)$ of $\alpha \in A^{**}$ is obtained by replacing each finite stationary subsequence by one of its single element.

5. Abstract mathematical definition of programming notions 3.

To determine, whether a program is a solution of a problem, we introduce the concept of the program function:

Definition 3.5. The effect of the program is defined by a relation called program function. The domain of the program function contains the states from which the program surely terminates (the executions starting from these states are finite). The program function of the program S is the relation $p(S) \subseteq A \times A$, if

1.

$$D_{p(S)} = \{a \in A \mid S(a) \subseteq A^*\}$$

2.

$$\forall a \in D_{p(S)} : p(S)(a) = \left\{b \in A \mid \exists \alpha \in S(a) : \alpha_{|\alpha|} = b\right\}$$

Definition 3.6. The program S is correct with respect to the problem F (or the program S is a solution of the problem F), if

1.

$$D_F \subseteq D_{p(S)}$$

2.

$$\forall a \in D_F : p(S)(a) \subseteq F(a)$$

6. The weakest precondition 1.

Definition 3.7. Let R be a logical function over the state space A. The set $\lceil R \rceil$ is called the truth-set of R. $\lceil R \rceil = \{a \in A \mid R(a) = true\}$

Let Q and R be logical functions. Let $Q \Rightarrow R$ denote that $\lceil Q \rceil \subseteq \lceil R \rceil$.

Definition 3.8. Let R be a logical function over the state space A and let $S \subseteq A \times A^{**}$ be a program over the state space. The logical function $wp(S, R)$ is called the weakest precondition of the postcondition R in respect of the program S, if

$$\lceil wp(S,R) \rceil = \{a \in A \mid a \in D_{p(S)} \wedge p(S)(a) \subseteq \lceil R \rceil\}$$

This means, that the image of a point $a \in A$ by the function $wp(S, R)$ is true, if starting from this point the program S terminates surely, all the sequences which are associated to a by S are finite and the program terminates in a state for which R holds.

7. The weakest precondition 2.

Properties of the weakest precondition:

Theorem 3.9. Let S be a program, let R, Q be logical functions, and denote $FALSE$ the constant false logical function over A.

- $wp(S, FALSE) = FALSE$
- if $Q \Rightarrow R$ then $wp(S,Q) \Rightarrow wp(S,R)$
- $wp(S,Q) \wedge wp(S,R) = wp(S, Q \wedge R)$
- $wp(S,Q) \vee wp(S,R) \Rightarrow wp(S, Q \vee R)$

8. Weakest precondition examples 1.

- $wp(ABORT, R) = FALSE$

  where R is an arbitrary logical function over A and $ABORT \subseteq A \times A^{**}$ is such a program that $\forall a \in A : ABORT(a) = \{<a, a, a, \ldots>\}$

  Proof:

  $$\lceil wp(ABORT,R) \rceil = \{a \in A \mid a \in D_{p(ABORT)} \wedge p(ABORT)(a) \subseteq \lceil R \rceil\} = \emptyset$$,

  since $D_{p(ABORT)} = \emptyset$

- $wp(SKIP, R) = R$

where R is an arbitrary logical function over A and $SKIP \subseteq A \times A^{**}$ is such a program that $\forall a \in A : SKIP(a) = \{<a, a, a, \dots>\}$

Proof:

$\lceil wp(SKIP, R) \rceil = \{a \in A \mid a \in D_{p(SKIP)} \wedge p(SKIP)(a) \subseteq \lceil R \rceil\} = \{a \in A \mid a \in \lceil R \rceil\}$,

since $D_{p(SKIP)} = A$ and $\forall a \in A : p(SKIP)(a) = \{a\}$

9. Weakest precondition examples 2.

$$A = \underset{x}{\mathbb{Z}} \times \underset{y}{\mathbb{Z}}$$

$R \colon A \to \mathbb{L} \qquad R = (x < y)$

$S \subseteq A \times A^{**} \qquad S \colon x := x - y$

We are interested in $wp(S, R)$.

$$\begin{aligned}\lceil wp(S,R) \rceil &= \{(x,y) \in D_{p(S)} \mid p(S)\,((x,y)) \subseteq \lceil R \rceil\} \\ &= \{(x,y) \in A \mid \{((x-y,y))\} \subseteq \lceil R \rceil\} \\ &= \{(x,y) \in A \mid x - y < y\}\end{aligned}$$

since $A = D_{p(S)} \wedge \forall (x,y) \in A : p(S)((x,y)) = (x-y,y)$. It means that $wp(S,R) = (x < 2y)$

In other way: $wp(S,R) = R^{x \leftarrow x-y} = (x - y < y) = (x < 2y)$

10. Weakest precondition examples 3.

$$A = \underset{x}{\mathbb{N}} \times \underset{y}{\mathbb{N}}$$

$R \colon A \to \mathbb{L} \qquad R = (x < y)$

$S \subseteq A \times A^{**} \qquad S \colon x := x - y$

We are interested in $wp(S, R)$.

$$\begin{aligned}\lceil wp(S,R) \rceil &= \{(x,y) \in D_{p(S)} \mid p(S)\,((x,y)) \subseteq \lceil R \rceil\} \\ &= \{(x,y) \in A | x > y \wedge \{((x-y,y))\} \subseteq \lceil R \rceil\} \\ &= \{(x,y) \in A | x > y \wedge x - y < y\}\end{aligned}$$

since $\forall (x,y) \in D_{p(S)} : p(S)((x,y)) = (x-y,y)$, but this time $D_{p(S)} \neq A$

It means that $wp(S,R) = (y < x \wedge x < 2y)$

In other way: $wp(S,R) = R^{x \leftarrow x-y} \wedge x > y = (x - y < y \wedge x > y) = (y < x < 2y)$

11. The theorem of the specification

The following theorem makes a connection between the weakest precondition and the solution. It formulates a sufficient condition of the solution:

Theorem 3.10. Let $F \subseteq A \times A$ be a problem, $F_1 \subseteq A \times B$ and $F_2 \subseteq B \times A$ be relations such that F is the composition of F_1 and F_2. B is called the parameter space of the problem. Let define the sets $\lceil Q_b \rceil, \lceil R_b \rceil$ on the following manner:

$\forall b \in B : \lceil Q_b \rceil = \{a \in A \mid (a, b) \in F_1\} \qquad \lceil R_b \rceil = \{a \in A \mid (b, a) \in F_2\}$

Let $S \subseteq A \times A^{**}$ a program over the state space A.

If $\forall b \in B : Q_b \Rightarrow wp(S, R_b)$ then the program S is a solution of the problem F.

To simplify the verifying of the condition of the theorem, when solving a problem, we construct the program in a form for which the proof can be done independently from the $b \in B$ points.

12. Specification example 1.

Give the specification of the following problem: Find a positive divisor of a given natural number.

Every natural number has a positive divisor. So the state space should contain two components, one for the given number and the other for the divisor:

$$A = \underset{x}{\mathbb{N}} \times \underset{d}{\mathbb{N}}$$

We know that F can be written in the form of a relation: $F = \{((a, b), (c, d)) \mid a = c \wedge d|a\}$

In the following the specification of the problem will be given in the form of (A, B, Q, R) where A is a state space of the problem, B is the parameter space. Q is called precondition and R is called postcondition, respectively. The notion of parameter space, precondition and postcondition are defined by the theorem of the specification.

13. Specification example 2.

The complete specification of the previous problem:

$$A = \underset{x}{\mathbb{N}} \times \underset{d}{\mathbb{N}}$$

$$B = \underset{x'}{\mathbb{N}}$$

$\forall b \in B : Q_b(a) = (x(a) = x'(b))$, where $a \in A$

$\forall b \in B : R_b(a) = (x(a) = x'(b) \wedge d(a)|x(a))$, where $a \in A$

which can be simplyfied to the form

$$A = \underset{x}{\mathbb{N}} \times \underset{d}{\mathbb{N}}$$

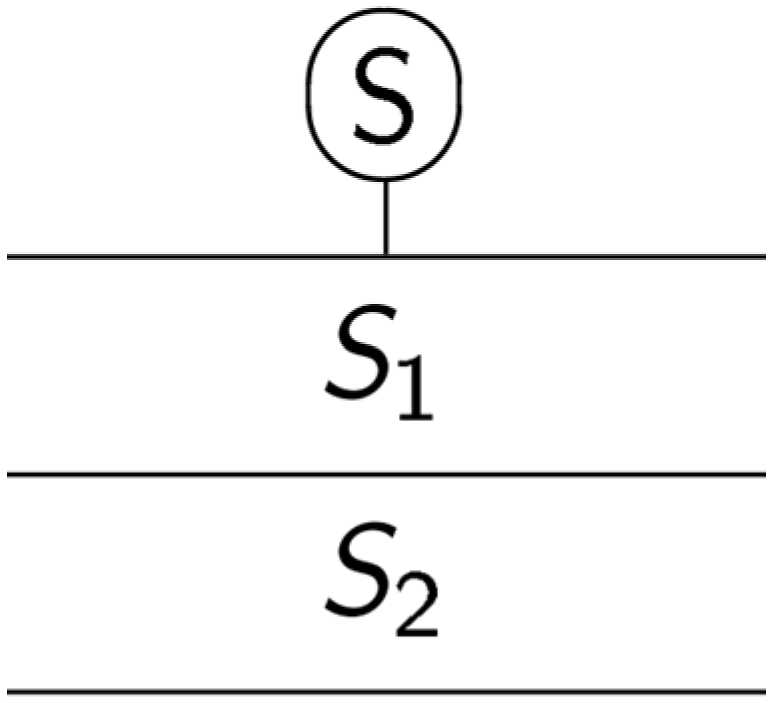
$$B = \underset{x'}{\mathbb{N}}$$

$$Q = (x = x')$$

$$R = (Q \wedge d|x)$$

14. Program constructs and their derivation rules 1.

Structogram of the sequential construction:



Theorem 3.11. $S = (S_1; S_2)$ is a program. Q , R and Q' are logical functions over A . If

1.

$Q \Rightarrow wp(S_1, Q')$ and

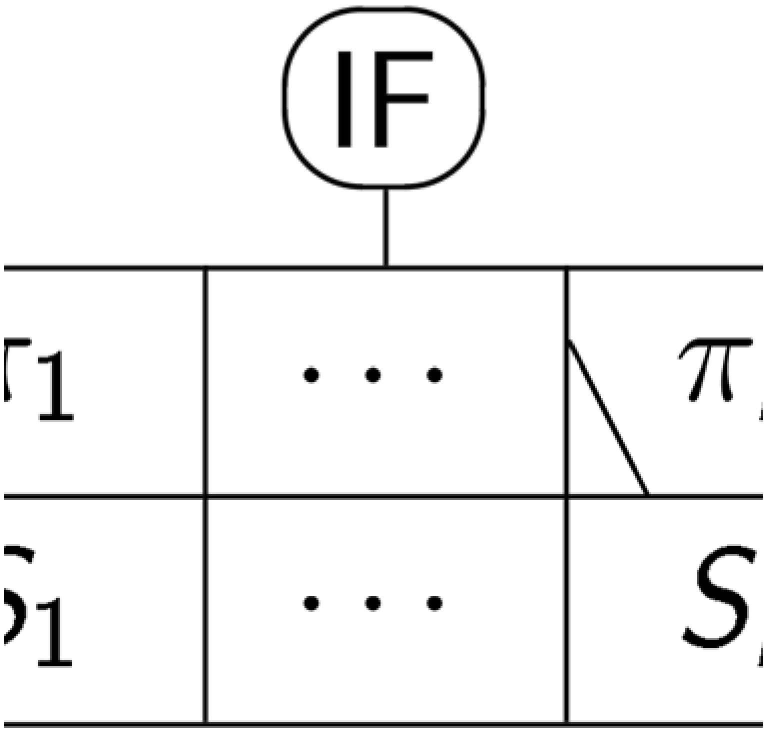
2.

$Q' \Rightarrow wp(S_2, R)$

then $Q \Rightarrow wp(S, R)$

15. Program constructs and their derivation rules 2.

Structogram of the branch construction:



Theorem 3.12. $IF = (\pi_1 : S_1, \ldots, \pi_n : S_n)$ is a program. Q , R and Q' are logical functions over A . If

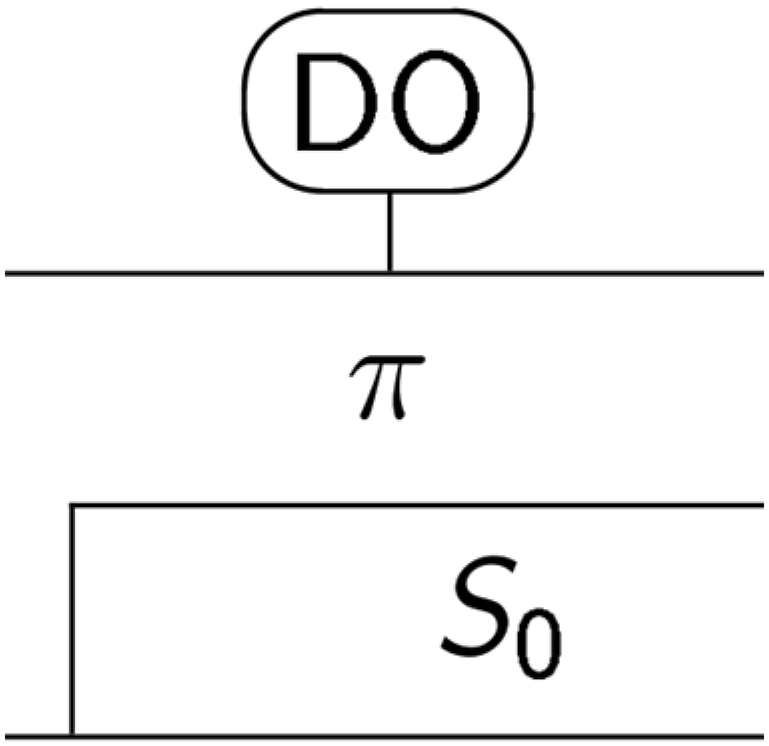
1. $$Q \Rightarrow \bigvee_{i=1}^{n} \pi_i \text{ and}$$

2. $$\forall i \in [1..n] : Q \wedge \pi_i \Rightarrow wp(S_i, R)$$

then $Q \Rightarrow wp(IF, R)$

16. Program constructs and their derivation rules 3.

Structogram of the loop construction:



Theorem 3.13. $DO = (\pi, S_0)$ is a program. P, Q , R are logical functions over A and $t\colon A \to \mathbb{Z}$ is a function. If

1.

   $Q \Rightarrow P$ and

2.

   $P \wedge \neg\pi \Rightarrow R$ and

3.

   $P \wedge \pi \Rightarrow t > 0$ and

4.

   $P \wedge \pi \wedge t = t' \Rightarrow wp(S_0, P \wedge t < t')$

then $Q \Rightarrow wp(DO, R)$

17. Extension of a problem and extension of a program

Let the state space B be a subpace of the state space A .

The extension of a problem means that new variables are introduced without any restriction on them.

The extension of a program defined on a subspace gives rise to a program which operates on the subspace in the same way as the original program does and it does not change the rest of the components of the state space.

Theorem 3.14. Let A be a state space, B a subspace of A. Let $F \subseteq B \times B$ be a problem, $S \subseteq B \times B^{**}$ a program, F' and S' the respective extensions of F and S onto the state space A. Then S solves F if and only if S' solves F'.

18. Generalisation of the definition of solution 1.

Recall: the program S is a solution of the problem F, if

1.

$$D_F \subseteq D_{p(S)}$$

2.

$$\forall a \in D_F : p(S)(a) \subseteq F(a)$$

Definition 3.15. If the extension of program S solves the problem F then we say that S solves F.

Example 1. Suppose that our state space is $\mathbb{Z} \times \mathbb{Z}$. Increase variable x by 1:

$$A = \underset{x}{\mathbb{Z}} \times \underset{y}{\mathbb{Z}}$$

$$B = \underset{x'}{\mathbb{Z}}$$

$$Q = (x = x'$$

$$R = (x = x' + 1)$$

Program $x := x + 1$ solves the problem.

19. Generalisation of the definition of solution 2.

Definition 3.16. If the projection of program S solves the problem F then we say that S solves F.

Theorem 3.17. Let A be a state space, B a subspace of A. Let $F \subseteq B \times B$ and $F' \subseteq A \times A$ be problems such that F' is the extension of F. Let $S \subseteq B \times B^{**}$ and $\bar{S} \subseteq A \times A^{**}$ a programs such that S is the projectionof $\bar{S}$. If $\bar{S}$ solves F' then S solves S.

Chapter 4. Derivation: a method for synthesising sequential programs

1. Programming theorems

- Programming theorems are problem-program pairs where the program solves the problem. They are frequently used as patterns to plan algorithms when the task to be solved is similar to the problem of the theorem.
- One of the common properties of the programming theorems is that they process a sequence of elementary values produced by an appropriate function. By expressing a programming theorem this way makes it more universal instead of processing the elements of an array: each array can be interpreted as a function over integer interval.
- In the following some programming theorem will be given (counting, summation, maximum selection, conditional maximum selection, linear search, binary search).

2. Counting 1.

Problem: Let β be a logical function defined over integers. Let us count the number of element in the interval $[m..n]$ for which β holds.

Specification of the problem:

$$A = \begin{array}{ccccc} \mathbb{Z} & \times & \mathbb{Z} & \times & \mathbb{N}_0 \\ m & & n & & d \end{array}$$

$$B = \begin{array}{ccc} \mathbb{Z} & \times & \mathbb{Z} \\ m' & & n' \end{array}$$

$$Q = (m = m' \wedge n = n')$$

$$R = (Q \wedge d = \sum_{i=m}^{n} \chi\beta(i))$$

where $\chi : \mathbb{L} \to 0,1$ and $\chi(true) = 1$ and $\chi(false) = 0$

3. Counting 2.

Algorithm:

```
$i, d := m, 0$
$i \leq n$
    $\beta(i)$
        $d := d + 1$ | SKIP
    $i := i + 1$
```

Let denote Q' the intermediate statement of the sequence, Inv the invariant and t the variant function of the loop.

$Q' = (Q \wedge d = 0 \wedge i = m)$

$Inv = (Q \wedge i \in [m..n+1] \cup \{m\} \wedge d = \sum_{k=m}^{i-1} \chi\beta(k))$

$t : n - i + 1$

$Q'' = P^{i \leftarrow i+1} \wedge t = t'$

4. Full proof of correctness of counting 1.

We prove that $Q \Rightarrow wp(S, R)$ by proving

1.

   $Q \Rightarrow wp(S, Q')$ where S denotes the initial assignment $i, d := m, 0$

2.

   $Q' \Rightarrow wp(S, R)$ where DO denotes the loop of the program

- $Q \Rightarrow wp(i, d := m, 0, Q') = Q'^{i \leftarrow m, d \leftarrow 0} = (Q \wedge 0 = 0 \wedge m = m) = Q$

  $Q \Rightarrow wp(S, Q') \qquad \checkmark$

- In the following we prove $Q' \Rightarrow wp(S, R)$ by using the derivation rule for loop. Due to the rule, it is sufficient to prove:
  - $Q' \Rightarrow Inv$
  - $Inv \wedge \neg\pi \Rightarrow R$
  - $Inv \wedge \pi \Rightarrow t > 0$
  - $Inv \wedge \pi \wedge t = t' \Rightarrow wp(S_0, P \wedge t < t')$

5. Full proof of correctness of counting 2.

- $(Q' \Rightarrow Inv) \Longleftrightarrow ((Q \wedge d = 0 \wedge i = m) \Rightarrow (Q \wedge i \in [m..n+1] \cup \{m\} \wedge s = \sum_{k=m}^{i-1} \chi\beta(k))$
  - Q since Q is contained in Q'
  - $i \in [m..n+1] \cup \{m\}$ since $i = m$ and if the interval $[m..n+1]$ is empty then $m \in \{m\}$, otherwise m is in $[m..n+1]$.
  - $s = \sum_{k=m}^{i-1} \chi\beta(k)$ since $s = 0$ and $i = m$ thus the sum is empty
- $Inv \wedge \neg\pi \Rightarrow R$

Since $i \in [m..n+1] \cup \{m\}$ and $\neg(i \le n)$, therefore we have $i = n+1$. Adding this statement to Inv we get R: $Inv \wedge \neg\pi \equiv R$.

$Inv \wedge \neg\pi \Rightarrow R \qquad \checkmark$

6. Full proof of correctness of counting 3.

- $Inv \wedge i \le n \Rightarrow n - i + 1 > 0$ since $i < n+1$

  $Inv \wedge \pi \Rightarrow t > 0 \qquad \checkmark$

- Now we wish to prove that $Inv \wedge \pi \wedge t = t' \Rightarrow wp(S_0, P \wedge t < t')$ Due to the rule of sequence it is sufficient to prove that

  1. $$Inv \wedge \pi \wedge n - i + 1 = t' \Rightarrow wp(IF, Q'')$$

  2. $$Q'' \Rightarrow wp(i := i+1, P \wedge n - i + 1 < t')$$

  - $Q'' \Rightarrow wp(i := i+1, P \wedge n - i + 1 < t') = (P^{i \leftarrow i+1} \wedge n - i < t')$ which holds since $Q'' = (P^{i \leftarrow i+1} \wedge t = t')$

  - Due to the rule of branch to prove $Inv \wedge \pi \wedge n - i + 1 = t' \Rightarrow wp(IF, Q'')$ it is sufficient to prove that

    - $Inv \wedge \pi \wedge t = t' \Rightarrow (\beta(i) \vee \neg\beta(i))$
    - $Inv \wedge \pi \wedge \beta(i) \wedge t = t' \Rightarrow wp(d := d+1, Q'')$
    - $Inv \wedge \pi \wedge \neg\beta(i) \wedge t = t' \Rightarrow wp(SKIP, Q'')$

7. Full proof of correctness of counting 4.

$Inv \wedge \pi \wedge n - i + 1 = t' \Rightarrow wp(IF, Q'')$

- $\beta(i) \vee \neg\beta(i)$ is always $true$

- $$wp(d := d+1, Q'') = (Q \wedge i+1 \in [m..n+1] \cup \{m\} \wedge d+1 = \sum_{k=m}^{i-1} \chi\beta(k)) + \chi\beta(i))$$

  which holds since

  $d = \sum_{k=m}^{i-1} \chi\beta(k))$ and

  $\beta(i)$ and

  $i \in [m..n+1] \cup \{m\}$ and $i \le n$

8. Full proof of correctness of counting 5.

- $wp(SKIP, Q'') = Q'' = (Q \wedge i+1 \in [m..n+1] \cup \{m\} \wedge d = \sum_{k=m}^{i-1} \chi\beta(k)) + \chi\beta(i))$
  which holds since

  $d = \sum_{k=m}^{i-1} \chi\beta(k))$ and

  $\neg\beta(i)$ and

  $i \in [m..n+1] \cup \{m\}$ and $i \le n$

  $Inv \wedge \pi \wedge n-i+1 = t' \Rightarrow wp(IF, Q'')$ ✓

$Inv \wedge \pi \wedge t = t' \Rightarrow wp(S_0, P \wedge t < t')$ ✓

9. Summation 1.

Problem: Let $\mathcal{H}$ be an arbitrary set where the operation of addition (+) is defined. Suppose that there exists a neutral element for the addition in H. Let the function $f : \mathbb{Z} \to \mathcal{H}$ be given. Let us calculate the sum of the values of f over the interval $[m..n]$.

Specification of the problem:

$$A = \begin{array}{ccccc} \mathbb{Z} & \times & \mathbb{Z} & \times & \mathcal{H} \\ m & & n & & s \end{array}$$

$$B = \begin{array}{ccc} \mathbb{Z} & \times & \mathbb{Z} \\ m' & & n' \end{array}$$

$$Q = (m = m' \wedge n = n')$$

$$R = (Q \wedge s = \sum_{i=m}^{n} f(i))$$

10. Summation 2.

Algorithm:

```
$i, s := m, 0$
$i \le n$
    $s := s + f(i)$
    $i := i + 1$
```

Proof outline:

$$Q' = (Q \wedge s = 0 \wedge i = m)$$

$$Inv = (Q \wedge i \in [m..n+1] \cup \{m\} \wedge s = \sum_{k=m}^{i-1} f(k))$$

$t: n - i + 1$

$Q'' = P^{i \leftarrow i+1} \wedge t = t'$

11. Maximum selection 1.

Problem: Consider a non-empty integer interval and a function $f: \mathbb{Z} \to \mathcal{H}$ where $\mathcal{H}$ is a totally ordered set. Let us seek the greates value of the function f and an argument where the function takes its maximum value.

Specification of the problem:

$$A = \begin{array}{ccccccc} \mathbb{Z} & \times & \mathbb{Z} & \times & \mathcal{H} & \times & \mathbb{Z} \\ m & & n & & max & & ind \end{array}$$

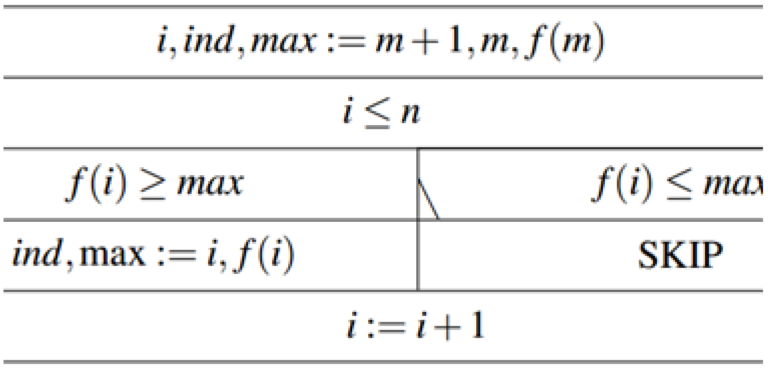
$$B = \begin{array}{ccc} \mathbb{Z} & \times & \mathbb{N} \\ m' & & n' \end{array}$$

$$Q = (m = m' \wedge n = n' \wedge m \geq n)$$

$$R = (Q \wedge ind \in [m..n] \wedge max = f(ind) \wedge \forall j \in [m..n] : f(j) \leq max)$$

12. Maximum selection 2.

Algorithm:



```
$i, ind, max := m+1, m, f(m)$
$i \leq n$
    $f(i) \geq max$                  | $f(i) \leq max$
        $ind, \max := i, f(i)$       |     SKIP
    $i := i+1$
```

Proof outline:

$$Q' = (Q \wedge max = f(m) \wedge ind = m \wedge i = m+1)$$

$$P = (Q \wedge i \in [m..n+1] \wedge max = f(ind) \wedge \forall j \in [m..i-1] : f(j) \leq max)$$

$t: n - i + 1$

$Q'' = P^{i \leftarrow i+1} \wedge t = t'$

13. Conditional maximum selection 1.

Problem: Let $f: \mathbb{Z} \to \mathcal{H}$ and $\beta: \mathbb{Z} \to \mathbb{L}$ be functions defined over integers where $\mathcal{H}$ is a totally ordered set. Let us find the maximum value of the function f over the set $[m..n] \bigcap \lceil \beta \rceil$, and if exists, an argument argument in $[m..n] \bigcap \lceil \beta \rceil$ where the function takes its maximum value.

Specification of the problem:

$$A = \underset{m}{\mathbb{Z}} \times \underset{n}{\mathbb{Z}} \times \underset{max}{\mathcal{H}} \times \underset{ind}{\mathbb{Z}} \times \underset{l}{\mathbb{L}}$$

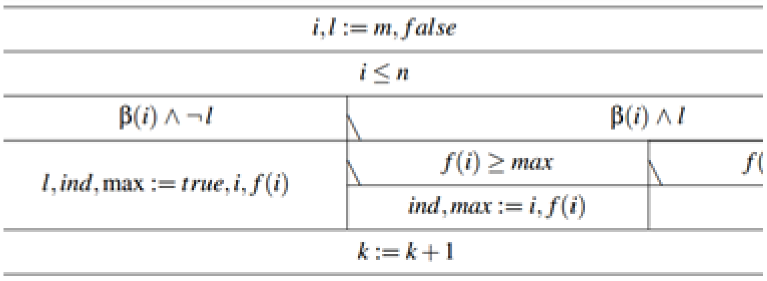
$$B = \underset{m'}{\mathbb{Z}} \times \underset{n'}{\mathbb{N}}$$

$$Q = (m = m' \land n = n')$$

$$R = (Q \land l = (\exists k \in [m..n] : \beta(k)) \land (l \Rightarrow (ind \in [m..n] \land max = f(ind) \land \beta(ind) \land \forall j \in [m..n] \cap [\beta] : f(j) \leq max))$$

14. Conditional maximum selection 2.

Algorithm:



Proof outline:

$$Q' = (Q \land l = false \land i = m)$$

$$Inv = (Q \land i \in [m..n+1] \cup \{m\} \land l = (\exists k \in [m..i-1] : \beta(k)) \land (l \Rightarrow (ind \in [m..i-1] \land max = f(ind) \land \beta(ind) \land \forall j \in [m..i-1] \cap [\beta] : f(j) \leq max))$$

$$t : n - i + 1$$

$$Q'' = P^{i \leftarrow i+1} \land t = t'$$

15. Linear search 1.

Problem: Let β be a logical function defined over integers. Let us decide whether β holds for any element of the interval $[m..n]$. Let us give the smallest element in $[m..n]$ for which β holds.

Specification of the problem:

$$A = \underset{m}{\mathbb{Z}} \times \underset{n}{\mathbb{Z}} \times \underset{l}{\mathbb{L}} \times \underset{i}{\mathbb{Z}}$$

$$B = \underset{m'}{\mathbb{Z}} \times \underset{n'}{\mathbb{Z}}$$

$$Q = (m = m' \land n = n')$$

$$R = (Q \land l = (\exists j \in [m..n] : \beta(j) \land (l \Rightarrow (i \in [m..n] \land \beta(i) \land \forall j \in [m..i-1] : \neg\beta(j))))$$

16. Linear search 2.

Algorithm:

```
$i, l := m-1, false$
$\neg l \wedge i < n$
    $i := i+1$
    $l := \beta(i)$
```

Proof outline:

$$Q' = (Q \wedge l = false \wedge i = m-1)$$

$$Inv = (Q \wedge i \in [m..n] \cup \{m-1\} \wedge \forall j \in [m..i-1] : \neg\beta(j) \wedge l = (\exists j \in [m..i] : \beta(j)))$$

$$t : n - i + 1$$

17. Binary search 1.

Problem: Let $f : \mathbb{Z} \to \mathcal{H}$ be a monotonically inceasing functiondefined over integers where $\mathcal{H}$ is a totally ordered set. Let $\beta : \mathbb{Z} \to \mathbb{L}$ be a logical function. Let us decide whether a given value $h \in \mathcal{H}$ is taken by f over the interval $[m..n]$. If h is taken by f then let us give an element of $[m..n]$ at which the value is h .

Specification of the problem:

$$A = \begin{array}{ccccccccc} \mathbb{Z} & \times & \mathbb{Z} & \times & \mathcal{H} & \times & \mathbb{Z} & \times & \mathbb{L} \\ m & & n & & h & & i & & l \end{array}$$

$$B = \begin{array}{ccc} \mathbb{Z} & \times & \mathbb{N} \\ m' & & n' \end{array}$$

$$Q = (m = m' \wedge n = n')$$

$$R = (Q \wedge l = (\exists k \in [m..n] : \beta(k)) \wedge (l \Rightarrow (ind \in [m..n] \wedge max = f(ind) \wedge \beta(ind) \wedge \forall j \in [m..n] \cap \lceil \beta \rceil : f(j) \leq max)))$$

18. Binary search 2.

Algorithm:

```
$u, v, l := m, n, false$
$\neg l \wedge u \leq v$
    $i := \lceil (u+v)/2 \rceil$
    $f(i) < h$          | $f(i) = h$    | $f(i)$
    $\iota := i+1$      | $l := true$   | $v :=$
```

Proof outline:

$$Q' = (Q \wedge l = false \wedge u = m \wedge v = n)$$

$$Inv = (Q \wedge [u..v] \subseteq [m..n] \wedge (\forall j \in [m..n] \setminus [u..v] : f(j) \neq h) \wedge \neg l \wedge i \in [u..v])$$

19. Binary search 3.

$$t = \begin{cases} v - u + 1 & \neg l \\ 0 & l \end{cases}$$

$$Q'' = (Q \wedge [u..v] \subseteq [m..n] \wedge (\forall j \in [m..n] \setminus [u..v] : f(j) \neq h) \wedge l \Rightarrow (i \in [u..v] \wedge \wedge f(i) = h))$$

20. Program derivation method 1.

Given a problem. Our task is to find a solution for the problem.

Q: precondition of the problem

R: postcondition of the problem

Due to the specification theorem it is sufficient to prove that $Q \Rightarrow wp(S, R)$ when one wants to show that program S solves problem F.

- if $Q \Rightarrow R$ then the program SKIP is a solution of the problem, since $Q \Rightarrow wp(SKIP, R)$

  Example:

  $$s = \sum_{i=m}^{k} f(i) \ \wedge \ (f(k+1) = 0) \qquad \Rightarrow \qquad wp(SKIP, s = \sum_{i=m}^{k+1} f(i))$$

21. Program derivation method 2.

- an appropriate assignment solves the problem

  Example:

  Let $Q = TRUE$ and $R = (a = 0 \wedge b = 1)$. The assignment $a, b := 0, 1$ solves the problem given by precondition Q and postcondition R since

  $$TRUE \Rightarrow wp(a, b := 0, 1, a = 0 \wedge b = 1)$$

  holds.

- Every problem can be solved by an assignment. For example, if we are looking for the gratest common divisor (let denote it by z) of two natural numbers x and y, the assigment $z := gcd(x, y)$ is a trivial solution of the problem. The question is, whether this assigment, more precisely the using of function gcd is allowed or not.

22. Program derivation method 3.

According to Böhm-Jacopini theorem, any construct of programs can be built by combining only three constructs: sequence, branch and loop. This is why we look for the solution of a problem in the form of sequence, branch or loop, when the problem cannot be solved by an assignment (or the given assignment is not allowed to use).

The postcondition of the problem usually let us deduct the structure of the result algorithm.

- Can the problem be divided into subproblems? Then the solution is a sequence. Question: what is the intermediate condition Q of the sequence?
- Are there some cases which can be handled separately? Then the solution of the problem is a branch.
- Can the problem be solved by repeating a process? Question: if so, what is the invariant of the loop?

23. Example: greatest common divisor 1.

Problem: Find the greatest common divisor of two natural numbers!

Specification:

$$A = \begin{array}{ccccc} \mathbb{N} & \times & \mathbb{N} & \times & \mathbb{N} \\ x & & y & & d \end{array}$$

$$B = \begin{array}{ccc} \mathbb{N} & \times & \mathbb{N} \\ x' & & y' \end{array}$$

$$Q = (x = x' \land y = y')$$

$$R = (Q \land d|x \land d|y \land \forall k \in \mathbb{N} : d < k \Rightarrow \neg(k|x \land k|y))$$

Since the greatest common divisor cannot be greater than the smaller number, the postcondition is equivalent to the following statement:

$$(Q \land d|x \land d|y \land \forall k \in [d+1..min(x,y)] : \neg(k|x \land k|y))$$

- Let the loop invariant Inv be the following proposition stating that all numbers greater than d may not be the greatest common divisor of x and y:

24. Example: greatest common divisor 2.

- $Inv = (Q \land d \in [1..min(x,y)] \land \forall k \in [d+1..min(x,y)] : \neg(k|x \land k|y))$
- $Q \Rightarrow P$ does not hold. Consider a sequence with intermediate condition $Q' = (Q \land d = min(x,y))$
- Now $Q' \Rightarrow Inv$ and the subproblem given by precondition Q and postcondition Q' can be solved by the assignment $d := min(x,y)$.
- If we choose $\neg(d|x \land d|y)$ as a loop condition then $Inv \land \neg\pi \Rightarrow R$ holds.
- We need to find a proper loop body which preserves Inv and decreases the value of t. Inv states that all numbers greater than d may not be the greatest common divisor of x and y.
- Since d is not a common divisor it has to be decreased. Let be $t = d$

  $Inv \land \neg(d|x \land d|y) \Rightarrow d > 0$ holds since d is a natural number.

25. Example: greatest common divisor 3.

- $Inv \land \neg(d|x \land d|y)\ d = t' \Rightarrow wp(d := d-1, P \land d < t') = (Q \land d-1 \in [1..min(x,y)] \land \forall k \in [d..min(x,y)] : \neg(k|x \land k|y)) \land d-1 \in \mathbb{N} \land d-1 < t'$
  - $d - 1 < t'$ since $d = t'$
  - $d - 1 \in \mathbb{N}$ since $\neg(d|x \land d|y)$ implies that $d \neq 1$

- $d-1 \in [1..min(x,y)]$ since $d \in [1..min(x,y)]$ and $d \neq 1$
- $\forall k \in [d..min(x,y)] : \neg(k|x \wedge k|y)$ since $\forall k \in [d+1..min(x,y)] : \neg(k|x \wedge k|y)$ and due to the loop condition d is not a common divisor.

We proved that the following program solves the problem:

$d := min(x,y)$
$\neg(d|x \wedge d|y)$
$d := d-1$

26. Example: number of digits v1 1.

Problem: determine the number of digits of a given natural number!

Specification:

$$A = \underset{x}{\mathbb{N}} \times \underset{d}{\mathbb{N}}$$

$$B = \underset{x'}{\mathbb{N}}$$

$$Q = (x = x')$$

$$R = (Q \wedge 10^{d-1} \leq x \wedge x < 10^d)$$

- In the previous example the invariant is obtained by weakening the postcondition. This can be taken as a usual advice when one wants to find a candidate for the loop invariant.

$$Inv = (Q \wedge 10^{d-1} \leq x)$$

- $Q' = (Q \wedge d = 1) \qquad Q' \Rightarrow Inv$ holds.
- $(Inv \wedge x < 10^d) \equiv R$, so it is easy to show that the loop invariant Inv together with the termination condition of the loop $x \geq 10^d$ imply R

27. Example: number of digits v1 2.

- A loop which never terminates does not solve any problem except the empty problem. Our goal is to achieve $x < 10^d$. To obtain the truth of formula $x < 10^d$ variable d should be incremented. We need a variant function for the loop that can be used to show that the loop will terminate. In this case $t : x - 10^d + 1$ is a natural choice, because it is positive at each entry to the loop and decreases with each loop iteration:

$$(Inv \wedge x \geq 10^d) \Rightarrow (x - 10^d + 1 > 0)$$

- $(Inv \wedge x \geq 10^d \wedge x - 10^d + 1 = t') \Rightarrow wp(d := d+1, Inv \wedge x - 10^d + 1 < t') = (Q \wedge 10^d \leq x \wedge \wedge x - 10^{d+1} + 1 < t')$

We proved that the following program solves the problem:

$d := 1$
$x \geq 10^d$
$d := d + 1$

28. Example: number of digits v2 1.

- Let us follow the previous line of thought but eliminate exponentiation from the loop condition. In order to get rid of using $x \geq 10^d$ as a loop condition we introduce variable h to store 10^d and we add $h = 10^d$ to the previous invariant:

  $Inv = (Q \wedge 10^{d-1} \leq x \wedge 10^d = h)$

- $Q' = (Q \wedge d = 1 \wedge h = 10)$ $Q' \Rightarrow Inv$ holds.
- The loop invariant conjoined with the negation of loop condition imply the postcondition: $(Inv \wedge x < h) \Rightarrow R$, where the loop condition is $x \geq h$.
- $t : x - h + 1$ $(Inv \wedge x \geq h) \Rightarrow (x - h + 1 > 0)$
- $(Inv \wedge x \geq h \wedge x - h + 1 = t') \Rightarrow wp(d, h := d + 1, 10h, Inv \wedge x - h + 1 < t') = (Q \wedge 10^d \leq x \wedge 10^{d+1} = 10h \wedge x - 10h + 1 < t')$.
  - $10^d \leq x$ holds since $h = 10^d$ and the loop condition states that $h \leq x$
  - $10^{d+1} = 10h$ holds since $10^d = h$
  - $x - 10h + 1 < t'$ holds since $x - h + 1 = t'$

29. Example: number of digits v2 2.

We proved that the following program also solves the problem:

$d, h := 1, 10$
$x \geq h$
$d, h := d + 1, 10h$

30. Example: number of digits v3 1.

With this example we illustrate that different specification of the same problem may lead to different solution of the problem. Using the abstract function $length$ the specification of the previous problem can be expressed in the following form:

$$A = \underset{x}{\mathbb{N}} \times \underset{d}{\mathbb{N}}$$

$$B = \underset{x'}{\mathbb{N}}$$

$$Q = (x = x')$$

$R = (Q \wedge d = length(x))$ where

$$length(x) = \begin{cases} length(x \ div \ 10) + 1 & x > 0 \\ 0 & x = 0 \end{cases}$$

31. Example: number of digits v3 2.

- $Inv = (Q \wedge length(x) = length(y) + d)$ Let y be a prefix of x Informally, the invariant states that we get the number of x by calculating the number of digits of y and adding the number of the rest digits to it.
- $Q' = (Q \wedge d = 0 \wedge y = 0)$
- It is obvious that $Inv \wedge length(y) = 0$ implies R . Since our goal is to calculate the value of the function $lenght$, using of $lenght$ in the loop condition is not allowed. The statement $y = 0$ is equivalent to statement $length(y) = 0$, so we get the loop condition $y \neq 0$.
- $t{:}\ y$
- Besides incrementing d by one, dividing y by 10 ensures the loop invariant is true after execution of the loop.

  $Inv \wedge y \neq 0 \wedge y = t' \Rightarrow wp(y, d := y \ div \ 10, d + 1, Inv \wedge y < t')$

32. Example: number of digits v3 3.

We proved that the following program solves the problem:

$d, h := 1, 10$
$x \geq h$
$d, h := d + 1, 10h$

33. Example: Binomial coefficient 1.

Problem: calculate the binomial coefficient $\binom{n}{k}$ of natural numbers n and k !

Specification:

$$A = \underset{n}{\mathbb{N}} \times \underset{k}{\mathbb{N}_0} \times \underset{s}{\mathbb{N}}$$

$$B = \underset{n'}{\mathbb{N}} \times \underset{k'}{\mathbb{N}_0}$$

$$Q = (x = x' \wedge k = k')$$

$$R = (Q \wedge s = \binom{n}{k})$$

Let us assume that $n - k > k$. In this case an efficient method to compute the binomial coefficients $\binom{n}{k}$ is given by the formula:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{(n-k+1) \cdot \ldots \cdot n}{1 \cdot \ldots \cdot k} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

34. Example: Binomial coefficient 2.

- We need to choose a loop invariant. If there is an interval in the problem, often it is a good hueristic for choosing a loop invariant by modifying the postcondition of the loop to make it a proposition over a subinterval.

  $Inv = (Q \wedge x \in [1..k+1] \wedge s = \prod_{i=1}^{x-1} \frac{n+1-i}{i})$

- $Q' = (Q \wedge s = 1 \wedge x = 1) \qquad Q' \Rightarrow Invholds.$
- $Inv \wedge x - 1 = k \Rightarrow R$. $x \leq k$ is a proper loop condition.
- In order to achieve $x - 1 = k$ we have to increment variable x. On the other hand it means that the difference $k - (x-1)$ should be decreased. We get a proper variant function by choosing $t: \ : k - x + 1$

  $Inv \wedge x \leq k \Rightarrow k - x + 1 > 0$

35. Example: Binomial coefficient 3.

- $Inv \wedge x \leq k \wedge k - x + 1 = t' \Rightarrow wp(s, x := s \cdot \frac{n+1-x}{x}, x+1, Inv \wedge t < t') = (Q \wedge x + 1 \in [1..k+1] \wedge s \cdot \frac{n+1-x}{x} = \prod_{i=1}^{x} \frac{n+1-i}{i} \wedge \frac{n+1-i}{i} \in \mathbb{N} \wedge k - x < t')$

We get the following program:

$s, x := 1, 1$
$x \leq k$
$x, s := x + 1, s \cdot \frac{n+1-x}{x}$

In fact, we did not solve the original problem. We solved the problem where the precondition is $Q \wedge n - k > k$ and postcondition is R. By repeating the same reasoning with some modifications we get the program which solves the problem given by its precondition $Q \wedge n - k \leq k$ and postcondition R. Then due to the branch derivation rule, the branch constructed from the two mentioned program solves the original problem.

36. Example: Number represented by an array v1 1.

Problem: Given an array of digits. Calculate the number represented by the array. Specification:

$$A = \underset{x}{\mathbb{Z}^n} \times \underset{s}{\mathbb{N}_0}$$

$$B = \underset{x'}{\mathbb{Z}^n}$$

$$Q = (x = x' \wedge \forall j \in [1..n] : 0 \leq x_i \leq 9)$$

$$R = (Q \wedge s = \sum i = 0n - 1x_{n-i} \cdot 10^i)$$

- $Inv = (Q \wedge s = \sum i = 0k - 1 x_{n-i} \cdot 10^i \wedge k \in [0..n] \wedge h = 10^k)$ We introduced variable k to avoid using exponentiation.
- $Q' = (Q \wedge s = 0 \wedge k = 0 \wedge h = 1)$
- $Inv \wedge k \neq n \Rightarrow R$ Loop condition: $k \neq n$
- $Inv \wedge k \neq n \Rightarrow n - k > 0$
- By increasing k the value of the variant function is decreasing. To preserve the truth of loop invariant h has to be multiplied by 10 and s has to be increased by $s + x_{n-k} \cdot a$.

37. Example: Number represented by an array v1 2.

We get the following program:

$k, s, a := 0, 0, 1$
$k \neq n$
$k, s, a := k + 1, s + x_{n-k} \cdot a, 10a$

38. Example: Number represented by an array v2 1.

A more obvious algorithm for computing the value of the number represented by an array is the following:

1. Let assume variable s stores the value of a number represented by the first $k - 1$ elements of array x .
2. Multiply s by 10 and then add x_k to the product.
3. …

It is an iteration. Question: what is the invariant of the loop? The rationale behind the following invariant is, that it expresses that s consists the value of the number represented by the first $k - 1$ elements of array x :

$$Inv = (s = value(k - 1) \wedge k \in [1..n + 1])$$

39. Example: Number represented by an array v2 2.

We provide a new specification of the problem by introducing function $value$:

$$A = \begin{array}{ccc} \mathbb{Z}^n & \times & \mathbb{N}_0 \\ x & & s \end{array}$$

$$B = \begin{array}{c} \mathbb{Z}^n \\ x' \end{array}$$

$$Q = (x = x' \wedge \forall j \in [1..n] : 0 \leq x_i \leq 9)$$

$R = (Q \wedge s = value(n))$

- $Q' = (Q \wedge s = 0 \wedge k = 1)$
- Since $Inv \wedge k = n + 1 \equiv R$, we get the loop condition $k \leq n$.
- $t : n + 1 - k$
- We look for the loop body in the form of sequence where the second program of the sequence is the assignment $k := k + 1$. Let the intermediate statement be $Q'' = (Inv^{k \leftarrow k+1})$. We will prove the folowing:

  1.

  $$Inv \wedge k \leq n \wedge n + 1 - k = t' \Rightarrow wp(s := s + x_k, Q'' \wedge n + 1 - k = t')$$

  2.

  $$Q'' \wedge n + 1 - k = t' \Rightarrow wp(k := k + 1, Inv \wedge n + 1 - k < t')$$

40. Example: Number represented by an array v2 3.

The second statement obviously holds with the choice of $Q'' = (Inv^{k \leftarrow k+1})$. Now we prove that $Inv \wedge k \leq n \wedge n + 1 - k = t' \Rightarrow wp(s := s + x_k, Q'' \wedge n + 1 - k = t')$

$wp(s := s + x_k, Q'' \wedge n + 1 - k = t') = (Q \wedge s + x_k = value(k) \wedge k + 1 \in [1..n + 1]) \wedge k \in [1..n]$

- Q holds since it contained in Inv
- $s + x_k = value(k)$ holds since $s = value(k - 1)$ and $value(k) = value(k - 1) + x_k$
- $k + 1 \in [1..n + 1]$ and $k \in [1..n]$ hold since $k \in [1..n + 1]$ and $k \leq n$

We proved that the following program solves the problem:

$k, s := 1, 0$
$k \leq n$
$s := 10s + x_k$
$k := k + 1$

41. Example: Reversing an array 1.

Problem: Reverse the order of the elements in a given array of integers! Specification:

$$A = \underset{x}{\mathbb{Z}^n}$$

$$B = \underset{x'}{\mathbb{Z}^n}$$

$$Q = (x = x')$$

$R = (\forall j \in [1..n] : x_i = x_{n+1-i})$

We try to solve the problem with a loop. We need to choose a loop invariant. Let formula Inv informally mean that the first $k-1$ elements of the array and the corresponding last $k-1$ elements are swapped whereas $n-2k+2$ elements in the middle of the array remained unchanged:

$$Inv = (k \in [1..n\ div\ 2+1] \wedge \forall j \in [1..k-1] : (x_i = x'_{n+1-i} \wedge x_{n+1-i} = x'_i) \wedge \forall j \in [1..n-2k+2] : x_{k-1+j} = x'_{k-1+j})$$

42. Example: Reversing an array 2.

Since the middle element of the array equals to itself, the reverse is completed if $k-1 = n\ div\ 2$.

- $Q \not\Rightarrow P$. Let $Q' = (Q \wedge k = 1)$. $Q \Rightarrow wp(k := 1, Q')$ and $Q' \Rightarrow Inv$.
- $Inv \wedge (n\ div\ 2 = k-1) \Rightarrow R$ implies that $k \neq n\ div\ 2+1$ is a proper loop condition.
- $Inv \wedge k-1 = n\ div\ 2 \Rightarrow ((n\ div\ 2 - k + 1) > 0)$
- Let look for the loop body in the form of a sequence dividing the problem given by the following precondition and postcondition, respectively: $Inv \wedge k \neq n\ div\ 2+1 \wedge n\ div\ 2-k+1 = t'$ and

  $Inv \wedge n\ div\ 2-k+1 < t'$

43. Example: Reversing an array 3.

Let $Q'' = (Inv^{k \leftarrow k+1})$ be the intermediate statement of the sequence.

- $Q'' \Rightarrow wp(k := k+1, Inv \wedge n\ div\ 2-k+1 < t') = Inv^{k \leftarrow k+1} \wedge n\ div\ 2-k < t'$ holds.
- We need a program which takes from $Inv \wedge k \neq n\ div\ 2+1$ to Q'' while the value of the variant function does not change.

  $Q'' = (k+1 \in [1..n\ div\ 2+1] \wedge \forall j \in [1..k] : (x_i = x'_{n+1-i} \wedge x_{n+1-i} = x'_i) \wedge \forall j \in [1..n-2k] : x_{k+j} = x'_{k+j})$ To satisfy Q'' we need to swap elements x_k and $x_n + 1 - k$.

  $(Inv \wedge (k \neq n\ div\ 2+1) \wedge (n\ div\ 2-k+1 = t')) \Rightarrow wp(x_k, x_{n+1-k} := x_{n+1-k}, x_k,\ Q'' \wedge (n\ div\ 2-k+1 = t'))$.

  Let us calculate the given weakest precondition:

  $$(k+1 \in [1..n\ div\ 2+1] \wedge \forall j \in [1..k-1] : (x_i = x'_{n+1-i} \wedge x_{n+1-i} = x'_i) \wedge (x_{n+1-k} = x'_{n+1-k} \wedge x_k = x'_k) \wedge \forall j \in [1..n-2k] : x_{k+j} = x'_{k+j}) \wedge k, n+1-k \in [1..n]$$

44. Example: Reversing an array 4.

- $(k+1 \in [1..n\ div\ 2+1]$ since $(k \in [1..n\ div\ 2+1]$ and k is not equal to the endpoint of the interval
- the statement $\forall j \in [1..k-1] : (x_i = x'_{n+1-i} \wedge x_{n+1-i} = x'_i)$ is included in Inv
- $(x_{n+1-k} = x'_{n+1-k} \wedge x_k = x'_k)$ is included in Inv
- $\forall j \in [1..n-2k] : x_{k+j} = x'_{k+j})$ is also assumed in Inv

- $k, n+1-k \in [1..n]$ are legal indexes of the array of x due to the loop condition and the statement $k \in [1..n\ div\ 2+1]$ contained in Inv

We proved that the following program solves the problem:

$k := 1$
$k \neq n\ div\ 2+1$
$x_k, x_{n-k+1} := x_{n-k+1}, x_k$
$k := k+1$

Chapter 5. Temporal logic of concurrent programs

1. Introduction 1.

In classical mathematics the truth of the proposition $x = x_0$ implies the falsity of $x = x_0 + 1$. Investigating the two propositions at different time points, both of them may be true. For example at time points before and after the assigment $x := x + 1$.

Consider the following fragment of a program: $c := b; b := b - a$ Let A denote the proposition $(a + b = c \wedge a > 0) \Rightarrow b > 0$ and assume that the variables a, b have the values 3,-3,0 respectively before the execution of the program fragment.

1.

   With these values A is false before the executon of $c := b$

2.

   With these values A is true after the execution of $c := b$

3.

   With these values A is false after the execution of the program fragment.

Temporal logic is a logic of propositions whose truth and falsity may depend on time. Temporal logic is useful for the formal description and analysis of dynamic properties in particular in the field of parallel programs.

2. Syntax of $\mathcal{L}_{TA}$ language 1.

- Alphabet
  - a denumerable set $\mathcal{V}$ of atomic formulas
  - the symbols $\neg$, $\rightarrow$, (,), $\underline{atnext}$, $\Box$, $\bigcirc$
- Formulas
  - every atomic formula is formula
  - if A is formula then $\neg A$, $\bigcirc A$, $\Box A$ are formulas
  - if A and B are formulas then $(A \rightarrow B)$, $A \ \mathbf{atnext} \ B$ are formulas
- Further operators

  $\wedge, \vee, \leftrightarrow$

  $\Diamond A$ is $\neg\Box\neg A$

operator $\mathbf{atnext}^n$ defined as follows:

$(A \ \mathbf{atnext}^{n+1} \ B)$ is $((A \ \mathbf{atnext}^n \ B) \ \mathbf{atnext} \ B)$

3. Syntax of $\mathcal{L}_{TA}$ language 2.

Priority order (descending):

- $\neg, \bigcirc, \Box, \Diamond$
- $\mathbf{atnext}$
- $\wedge, \vee$
- $\leftrightarrow, \rightarrow$

4. Semantics of $\mathcal{L}_{TA}$ 1.

- We extend the concept of valuation of classical propositional logic
- A Kripke structure for $\mathcal{L}_{TA}$ consists of an infinite sequence $\{\eta_0, \eta_1, \ldots\}$ of mappings, where $\eta_i : \mathcal{V} \rightarrow \{true, false\}$ are called states and η_0 is the initial state.
- The truth value $K_i(F) \in \{true, false\}$ is defined for every formula F, every Kripke structure K and every $i \in \mathbb{N}_0$ in the following inductive way:

  1. $K_i(v) = \eta_i(v)$ for $vin\mathcal{V}$
  2. $K_i(\neg A) = true$ iff $K_i(A) = false$
  3. $K_i(A \rightarrow B) = true$ iff $K_i(A) = false$ or $K_i(B) = true$
  4. $K_i(\bigcirc A) = true$ iff $K_{i+1}(A) = true$
  5. $K_i(\Box A) = true$ iff $K_j(A) = true$ for every $j \geq i$
  6. $K_i(A \ \mathbf{atnext}\ B) = true$ iff $K_j(B) = false$ for every $j > i$ or $K_k(A) = true$ for the smallest $k > i$ with $K_k(B) = true$

5. Valuation of formulas with other operators of $\mathcal{L}_{TA}$ 1.

- $K_i(A \wedge B) = true$ iff $K_i(A) = true$ and $K_i(B) = true$
- $K_i(A \vee B) = true$ iff $K_i(A) = true$ or $K_i(B) = true$
- $K_i(A \leftrightarrow B) = true$ iff $K_i(A) = K_i(B)$
- $K_i(true) = true$

- $K_i(false) = false$
- $K_i(\Diamond A) = true$ iff $K_i(A) = true$ for some $j \geq j$
- $K_i(A \ \mathbf{atnext}^2 \ B) = true$ iff $K_j(B) = true$ for at most one $j > i$ or $K_k(A) = true$ for the second smallest $k > i$ with $K_k(B) = true$

6. Valuation of formulas with other operators of $\mathcal{L}_{TA}$ 2.

- The rules above for operators $\wedge, \vee \leftrightarrow, \Diamond, \mathbf{atnext}^2$ have to be proved based on the definitions given before for the operators ($\neg, \leftarrow, \bigcirc, \Box, \mathbf{atnext}$)

  Example 2. $K_i(\Diamond A) = true \iff K_i(\neg\Box\neg A) = true$

  $\iff K_i(\Box\neg A) = false$

  $\iff K_j(\neg A) = false$ for some $j \geq i$

  $\iff K_j(A) = false$ for some $j \geq i$

- It is not necessary to introduce $\bigcirc$ and $\Box$ as basic operators because both can be expressed by $\mathbf{atnext}$ in the following way:

  $K_i(\bigcirc A) = K_i(A \ \mathbf{atnext} \ true)$

  $K_i(\Box A) = K_i(A \wedge false \ \mathbf{atnext} \ \neg A)$

7. Definitions and theorems 1.

Definition 5.1. A formula A of $\mathcal{L}_{TP}$ is called valid in the temporal structure K ($\models_K A$) if $K_i(A) = true$ for every $i \in \mathbb{N}_0$. A is called valid ($\models A$) if $\models_K A$ for every K.

Definition 5.2. A follows from a set $\mathcal{F}$ of closed formulas if $\models_K A$ for every K with $\forall B \in \mathcal{F}$: $\models_K B$.

Theorem 5.3. If $\mathcal{F} \models A$ and $\models B$ for every $B \in \mathcal{F}$ then $\models A$

In classical logic the following holds: $A_1, \ldots, A_n \models B$ iff $\models A_1 \wedge \ldots \wedge A_n \to B$

Note that this classical fact no longer holds in $\mathcal{L}_{TA}$. Counterexample: $A \models \Box A$, since this holds but formula $A \to \Box A$ is not valid. In $\mathcal{L}_{TA}$ the following analogon of this classical fact holds:

8. Definitions and theorems 2.

Theorem 5.4. If $A_1, \ldots, A_n \models B$ iff $\models \Box A_1 \wedge \ldots \wedge A_n \to B$

Theorem 5.5. If $\mathcal{F} \models A$ and $\mathcal{F} \models A \to B$ $\mathcal{F} \models B$.

Theorem 5.6. If $\mathcal{F} \models A$ then $\mathcal{F} \models \bigcirc A$ and $\mathcal{F} \models \Box A$.

9. Notion of satisfiability 1.

Definition 5.7. A set $\mathcal{F}$ of formulas is called satisfiable if there is some Kripke structure K and $i \in \mathbb{N}_0$ such that $K_i(A) = true$ for every $A \in \mathcal{F}$. A formula A is called satisfiable if $\{A\}$ is satisfiable.

Theorem 5.8. $\models A$ iff $\neg A$ is not satisfiable.

10. Notion of satisfiability 2.

Example 3. Consider the formulas

$A = (v_1 \wedge \Box v_2 \wedge v_3 \ \mathbf{atnext} \ \neg v_1)$ and

$B = ((v_2 \rightarrow \bigcirc v_1) \vee \bigcirc \neg v_2)$

and the set $\mathcal{F} = \{A, B\}$. We prove that $\mathcal{F}$ is satisfiable by showing that there are a Kripke structure K and $i \in \mathbb{N}_0$ such that $K_i(A) = K_i(B) = true$.

	η_0	η_1	η_2	η_3	…
v_1	true	true	false		…
v_2	true	true	true	…	forever true
v_3			true		…

Notice that the truth value of formulas A and B are true in state η_0.

11. Temporal logical laws 1.

Consider de Morgan's law from classical logic: $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$. Such tautologies remain valid in temporal logic if we substitute formulas of $\mathcal{L}_{TA}$ for A and B.

Example: $\neg(\bigcirc C \wedge \Box D) \leftrightarrow \neg \bigcirc C \vee \neg \Box D$ is a valid formula.

Definition 5.9. A formula of $\mathcal{L}_{TA}$ is tautologically valid if it derives from a tautology A (of classical propositional logic) by consistently replacing the atomic formulas of A by formulas of $\mathcal{L}_{TA}$.

Theorem 5.10. Every tautologically valid formula is valid.

12. Temporal logical laws 1.

Definition 5.11. Let $A_1, \dots, A_n, B \quad (n \geq 1)$ be formulas of $\mathcal{L}_{TA}$. B is called a tautological consequence of $A_1, \dots, A_n$ if the formula $A_1 \wedge \dots \wedge A_n$ is tautologically valid.

Theorem 5.12. If B is a tautological consequence of $A_1, \dots, A_n$ then $A_1, \dots, A_n \models B$

So far we have logical laws results from the classical part of temporallogic.

13. “Proper” temporal logical laws 1.

- Duality laws

(T1) $\neg \bigcirc A \leftrightarrow \bigcirc \neg A$

(T2) $\neg \Box A \leftrightarrow \Diamond \neg A$

(T3) $\neg \Diamond A \leftrightarrow \Box \neg A$

- Reflexivity laws

(T4) $\Box A \leftrightarrow A$

(T5) $A \leftrightarrow \Diamond A$

- Laws about the “strength” of the operators

(T6) $\Box A \leftrightarrow \bigcirc A$

(T7) $\bigcirc A \leftrightarrow \Box A$

(T8) $\Box A \leftrightarrow \Diamond A$

(T9) $\Box A \leftrightarrow A \ atnext \ B$

(T10) $\Diamond \Box A \leftrightarrow \Box \Diamond A$

14. “Proper” temporal logical laws 2.

- Expressibility (by atnext) laws

(T11) $\bigcirc A \leftrightarrow A \ atnext \ true$

(T12) $\Box A \leftrightarrow A \wedge false \ atnext \ \neg A$

(T13) $\Diamond A \leftrightarrow A \vee \neg (false \ atnext \ A)$

- Idempotency laws (T14) $\Box \Box A \leftrightarrow \Box A$

(T15) $\Diamond \Diamond A \leftrightarrow \Diamond A$

- Commutativity laws (T16) $\Box \bigcirc A \leftrightarrow \bigcirc \Box A$

(T17) $\Diamond \bigcirc A \leftrightarrow \bigcirc \Diamond A$

15. “Proper” temporal logical laws 3.

- Distributivity laws

(T18) $\bigcirc (A \rightarrow B) \leftrightarrow \bigcirc A \rightarrow \bigcirc B$

(T19) $\bigcirc (A \wedge B) \leftrightarrow \bigcirc A \wedge \bigcirc B$

(T20) $\bigcirc (A \vee B) \leftrightarrow \bigcirc A \vee \bigcirc B$

(T21) $\bigcirc (A \ atnext \ B) \leftrightarrow \bigcirc A \ atnext \ \bigcirc B$

(T22) $\Box(A \wedge B) \leftrightarrow \Box A \wedge \Box B$

(T23) $\Diamond(A \vee B) \leftrightarrow \Diamond A \vee \Diamond B$

(T24) $(A \wedge B)\ atnext\ C \leftrightarrow A\ atnext\ C \wedge B\ atnext\ C$

(T25) $(A \vee B)\ atnext\ C \leftrightarrow A\ atnext\ C \vee B\ atnext\ C$

16. “Proper” temporal logical laws 4.

- Weak distributivity laws

(T26) $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$

(T27) $\Box A \wedge \Box B \rightarrow \Box(A \vee B)$

(T28) $(\Diamond A \rightarrow \Diamond B) \rightarrow \Diamond(A \rightarrow B)$

(T29) $\Diamond(A \wedge B) \rightarrow \Diamond A \wedge \Diamond B)$

(T30) $A\ atnext\ (B \wedge C) \leftrightarrow A\ atnext\ B \wedge A\ atnext\ C$

17. “Proper” temporal logical laws 5.

- Recursion equivalences

(T31) $\Box A \leftrightarrow A \wedge \bigcirc \Box A$

(T32) $\Diamond A \leftrightarrow A \vee \bigcirc \Diamond A$

(T33) $A\ atnext\ B \leftrightarrow \bigcirc(B \rightarrow A) \wedge \bigcirc(\neg B \rightarrow A\ atnext\ B)$

18. Some further temporal operators

1.

$K_i(A\ until\ B) = true$ iff $K_j(B) = true$ for some $j > i$ and $K_k(A) = true$ for every k, where $i < k < j$

2.

$K_i(A\ unless\ B) = true$ iff $K_j(B) = true$ for some $j > i$ and $K_k(A) = true$ for every k where $i < k < j$ or $K_k(A) = true$ for every $i < k$

3.

$K_i(A\ while\ B) = true$ iff $K_j(B) = false$ for some $j > i$ and $K_k(A) K_k(B) = true$ for every k where $i < k < j$ or $K_k(A) = true$ for every $i < k$

4.

$K_i(A\ before\ B) = true$ iff for every $j > i$ with $K_j(B) = true$ there is some k , $i < k < j$ with $K_k(A) = true$

19. Further laws for new operators

(T43) A until $B \leftrightarrow B$ atnext $(A \to B) \land \bigcirc \diamond B$

(T44) A unless $B \leftrightarrow B$ atnext $(A \to B)$

(T45) A while $B \leftrightarrow \neg B$ atnext $(A \to \neg B)$

(T46) A before $B \leftrightarrow \neg B$ atnext $(A \lor B)$

(T47) A atnext $B \leftrightarrow \neg B$ until $(A \land B) \lor \bigcirc\Box\neg B$

(T48) A atnext $B \leftrightarrow \neg B$ unless $(A \land B)$

(T49) A atnext $B \leftrightarrow \neg B$ while $(A \to \neg B)$

(T50) A atnext $B \leftrightarrow B$ before $(\neg A \land B)$

20. The formal system Σ_{TA}

Axioms

- all tautologically valid formulas
- (ax1) $\neg \bigcirc A \leftrightarrow \bigcirc\neg A$
- (ax2) $\bigcirc (A \to B) \to (\bigcirc A \to \bigcirc B)$
- (ax3) $\Box A \to A \land \bigcirc\Box A$
- (ax4) $\bigcirc \Box\neg B \to A$ atnext B
- (ax5) A atnext $B \leftrightarrow \bigcirc(B \to A) \land \bigcirc$ $(\neg B \to A$ atnext $B)$

Rules

- (mp) A , $A \to B \Vdash B$
- (nex) $A \Vdash \bigcirc A$
- (ind) $A \to B$, $A \to \bigcirc A \Vdash A \to \Box B$

21. Theorems 1.

Theorem 5.13. Soundness theorem for Σ_{TA} :

Let A be a formula and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \vdash A$ then $\mathcal{F} \models A$. In particular: If $\vdash A$ then $\models A$.

Theorem 5.14. If B is a tautological consequence of $A_1, \ldots A_n$ then $A_1, \ldots, A_n \Vdash B$.

Theorem 5.15. Deduction theorem:

Let $\vdash \Box A$, $\to B$ B be formulas and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \cup \{A\} \vdash B$ then .

22. Theorems 2.

Theorem 5.16. Let A, B be formulas and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \vdash \Box A \to B$ then $\mathcal{F} \cup \{A\} \vdash B$.

Theorem 5.17. Let A, B be formulas and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \vdash \Box A \to B$ then $\mathcal{F} \cup \{A\} \vdash B$.

Theorem 5.18. Completness theorem

For every formula A, if $\models A$ then $\vdash A$.

23. Syntax of $\mathcal{L}_{TP}$ language 1.

- Alphabet
  - denumerably many variables
  - for every $n \in \mathbb{N}_0$, at most denumerably many n-ary function symbols
  - for every $n \in \mathbb{N}_0$ $(n \geq 1)$, at most denumerably many n-ary predicate symbols
  - the predicate symbol $=$
  - the symbols $\neg, \to, \forall, (,), \underline{atnext}, \Box, \bigcirc$
- Terms
  - every variable is term
  - if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots t_n)$ is also term

24. Syntax of $\mathcal{L}_{TP}$ language 2.

- Formulas
  - if p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is called an atomic formula
  - every atomic formula is formula
  - if A and B are formulas then $\neg A$, $(A \to B)$, $\bigcirc A$, $\Box A$, $A \ \underline{atnext} \ B$ are formulas
  - if A is a formula and x is a variable then $\forall x A$ is a formula

There are two kinds of variable:

- global variable: its value does not depend on the state
- local variable: its value may change during state transition

The occurence of a variable x in some formula A is called free if it does not appear in some part $\forall x B$ of A. Otherwise it is called bound. A formula of $\mathcal{L}_{TP}$ is called closed if it contains no free global variables. If

$x_1, \dots, x_n$ are all free global variables of some formula A then the formula $\forall x_1 \dots \forall x_n A$ is called the universal closure of A .

25. The semantics of $\mathcal{L}_{TP}$ 1.

The semantics of $\mathcal{L}_{TP}$ is defined by the help of first-order temporal structure $K = (S, \xi, W)$, where

- a structure S of classical logic consisting of
  - a set $|S| \neq \emptyset$, called universe
  - an n -ary function $S(f)\colon |S|^n \to |S|$ for every n -ary function symbol f
  - an n -ary relation $S(p) \subset |S|^n$ for every n -ary predicate symbol p other than $=$
- a global variable valuation ξ with respect to S
- $W = \{\eta_0, \eta_1, \dots\}$ an infinite sequence of states where each η_i assigns an element of $|S|$ to every local variable

26. The semantics of $\mathcal{L}_{TP}$ 2.

In any K , S and ξ together with η_i assign a value $S^{(\xi,\eta_i)}(t) \in |S|$ to every term t and a value $S^{(\xi,\eta_i)}(A) \in \{false, true\}$ for every atomic formula such that

- $S^{(\xi,\eta_i)}(x) = \xi(x)$ for every global variable x
- $S^{(\xi,\eta_i)}(a) = \eta_i(a)$ for every local variable a
- $S^{(\xi,\eta_i)}(f(t_1, \dots, t_n)) = S(f)(S^{(\xi,\eta_i)}(t_1), \dots, S^{(\xi,\eta_i)}(t_n))$
- $S^{(\xi,\eta_i)}(p(t_1, \dots, t_n)) = true$ iff $(S^{(\xi,\eta_i)}(t_1), \dots, S^{(\xi,\eta_i)}(t_n)) \in S(p)$ for p other than $=$
- $S^{(\xi,\eta_i)}(t_1 = t_2) = true$ iff $S^{(\xi,\eta_i)}(t_1) =_{|S|} S^{(\xi,\eta_i)}(t_2)$

27. The semantics of $\mathcal{L}_{TP}$ 3.

For every formula F and $i \in \mathbb{N}_0$ we define the truth value of the formula inductively in state $K_i(S, \xi, \eta_i)$:

- $K_i(A) = S^{(\xi,\eta_i)}(A)$ for every atomic formula A
- $K_i(\neg A) = true$ iff $K_i(A) = false$
- $K_i(A \to B) = true$ iff $K_i(A) = false \vee K_i(B) = true$
- $K_i(\bigcirc A) = true$ iff $K_{i+1}(A) = true$
- $K_i(\Box A) = true$ iff $K_j(A) = true$ for every $j \geq i$
- $K_i(A atnext B) = true$ iff $K_j(B) = false$ for every $j > i$ or $K_k(A) = true$ for the smallest $k > i with K_k(B) = true$

- $K_i(\forall x A) = true$ iff $K'_i(A) = true$ for every structure $K' = (S, \xi', W)$ where $S\xi'(y) =_{|S|} \xi(y)$ for every y other than x

28. Definitions

Definition 5.19. A formula A of $\mathcal{L}_{TP}$ is called valid in the temporal structure K ($\models_K A$) if $K_i(A) = true$ for every $i \in \mathbb{N}_0$. A is called valid ($\models A$) if $\models_K A$ for every K.

Definition 5.20. A follows from a set $\mathcal{F}$ of closed formulas if $\models_K A$ for every K with $\forall B \in \mathcal{F}\colon\ \models_K B$.

29. The formal system Σ_{TP} 1.

Axioms

- all axioms of Σ_{TA}
- (ax6) $\forall x A \to A_x(t)$ if t is substitutable for x in A
- (ax7) $\forall x \bigcirc A \to \bigcirc \forall x A$
- (ax8) $A \to \bigcirc A$ if A does not contain local variables
- (eq1) $x = x$
- (eq2) $(x = y) \to (A \to A_x(y))$ if A does not contain temporal operator

30. The formal system Σ_{TP} 2.

Rules

- (mp) A, $A \to B \Vdash B$
- (nex) $A \Vdash \bigcirc A$
- (ind) $A \to B$, $A \to \bigcirc A \Vdash A \to \Box B$
- (gen) $A \to B \Vdash A \to \forall x B$, if there is no free occurence of x in A

31. Theorems

Theorem 5.21. Soundness theorem for Σ_{TP}:

Let A be a formula and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \vdash A$ then $\mathcal{F} \models A$

Theorem 5.22. Deduction theorem:

Let A, B be formulas, A closed and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \cup \{A\} \vdash B$ then $\mathcal{F} \vdash \Box A \to B$.

Theorem 5.23. Let A, B be formulas and $\mathcal{F}$ a set of formulas. If $\mathcal{F} \vdash \Box A \to B$ then $\mathcal{F} \cup \{A\} \vdash B$.

32. Introduction

- We restrict programs to the following syntactic form:

  initial Pre;

  cobegin $\Pi_1 \| \ldots \| \Pi_2$ coend

  where every Π_i is either a cyclic or a non-cyclic while program. The Π_i components are thought to be executed in parallel.

- Let $\mathcal{A} = \mathcal{E} \cup \mathcal{T}$ where $\mathcal{E}$ is the set of elementary statements and $\mathcal{T}$ is the set of synchronization statements.

- Synchronization statements:

  await B then α

  await B

  where $\alpha \in \mathcal{E}$

- Every statement (except under an await) is labelled by a unique label.

33. Notations

- $\mathcal{M}_{\Pi_i}$: set of labels occuring in program Π_i , mostly $\{\alpha_0, \alpha_1, \ldots\}$
- $\alpha_0^{(i)}$: start label
- if $\mathcal{M}_{\Pi_i} = \{\alpha_0^i, \alpha_1^i, \ldots \alpha_e^i\}$ then α_e^i denotes the stop “statement” in the non-cyclic Π_i component
- $\bar{\mathcal{M}}_\Pi = \mathcal{M}_\Pi \setminus \{\alpha_e^1, \ldots, \alpha_e^1\}$
- Program state: $\eta = (\mu; \lambda_1, \ldots, \lambda_p, \kappa)$
- For every label we introduce the following propositional variables:
  - λ for every $\lambda \in \bar{\mathcal{M}}_\Pi$
  - atλ for every $\lambda \in \mathcal{M}_\Pi$

  with the informal meaning

  λ : the action λ is executed next

  atλ : λ is ready to execute

34. Example 1. - execution

Consider the following program:

initial $a = 0 \wedge b = 0$;

cobegin

loop

```
$\alpha_0$: a:=a+1;
$\alpha_1$: await $b \neq 0$;
$\alpha_2$: a:=a+3
end
$\|$ loop
$\beta_0$: a:=2*a;
$\beta_1$: await $a \neq 1$;
$\beta_2$: b:=b+1
end
coend
```

35. Example 2. - execution

step	action	a	b
		0	0
1.	β_0: a:=2*a	0	0
2.	α_0: a:=a+1	1	0
	deadlock		

step	action	a	b
		0	0
1.	α_0: a:=a+1	1	0
2.	β_0: a:=2*a	2	0
3.	β_1: await $a \neq 1$	2	0
4.	β_2: b:=b+1	2	1
5.	α_1: await $b \neq 0$	2	1
	$\vdots$ no deadlock		

36. Operational semantics of programs

Let $\mathcal{F}(\mathcal{L}_P)$ be the set of all formulas of $\mathcal{L}_P$. Every S statement sequence has the following three entities:

- $entry(S) \in \mathcal{M}_S$
- $trans(S) \in \mathcal{M}_S \times \mathcal{F}(\mathcal{L}_P) \times \mathcal{M}_S$
- $exits(S) \in \mathcal{M}_S \times \mathcal{F}(\mathcal{L}_P)$

Now let Π_i be some parallel component of Π. We define the set $trans(\Pi_i)$, where S is a statement sequence

- $\Pi_i \equiv S; \alpha$: stop

  $entry(\Pi_i) = entry(S)$

  $trans(\Pi_i) = trans(S) \cup \{(\beta, C, \alpha) \mid (\beta, C) \in exits(S)\}$

  $exits(\Pi_i) = \{\alpha, true\}$

- $\Pi_i \equiv$ loop S end

  $entry(\Pi_i) = entry(S)$

  $trans(\Pi_i) = trans(S) \cup \{(\beta, C, entry(S)) \mid (\beta, C) \in exits(S)\}$

  $exits(\Pi_i) = \emptyset$

37. Operational semantics of statement sequence 1.

- $S \equiv \alpha$: a, where $a \in \mathcal{A}$

  $entry(S) = \alpha$

  $trans(S) = \emptyset$

  $exits(S) = \{\alpha, true\}$

- $S \equiv \alpha$: await B then a, or α : await B

  $entry(S) = \alpha$

  $trans(S) = \emptyset$

  $exits(S) = \{\alpha, B\}$

- $S \equiv \alpha$: if B then S_1 else S_2 fi

  $entry(S) = \alpha$

  $trans(S) = trans(S_1) \cup trans(S_2) \cup \{(\alpha, B, entry(S_1)), (\alpha, \neg B, entry(S_2))\}$

  $exits(S) = exits(S_1) \cup exits(S_2)$

Informally, the triple α, C, β is in set $trans(S)$ if in S the execution can proceed in one step from α to β if C holds.

38. Operational semantics of statement sequence 2.

- $S \equiv \alpha$: if B then S_1 fi

  $entry(S) = \alpha$

  $trans(S) = trans(S_1) \cup \{(\alpha, B, entry(S_1))\}$

  $exits(S) = exits(S_1) \cup \{\alpha, \neg B\}$

- $S \equiv \alpha$: while B do S_0 fi

  $entry(S) = \alpha$

  $trans(S) = trans(S_0) \cup \{(\alpha, B, entry(S_0))\} \cup \{(\beta, C, \alpha) \mid (\beta, C) \in exits(S_0)\}$

  $exits(S) = \{\alpha, \neg B\}$

- $S \equiv \alpha$: $S_1; S_2$, where S_1 is an unlabelled statement, S_2 is a statement sequence

  $entry(S) = \alpha$

  $trans(S) = trans(S_1) \cup trans(S_2) \cup \cup\{(\beta, C, entry(S_2)) \mid (\beta, C) \in exits(S_1)\}$

  $exits(S) = exits(S_2)$

39. Program axioms

We divide the program axioms into two classes:

- structural axioms describe general properties hold for every program
- specification axioms specify the execution sequences of some given programs

40. Structural axioms

Basic axioms:

- (B1) $(start_\Pi \to \Box A) \vdash A$
- (B2) $nil_\Pi \wedge A \to \bigcirc(nil_\Pi \wedge A)$

Additional axioms:

- ($\Pi 1$) $\lambda \to \neg\lambda'$ if $\lambda \neq \lambda'$
- ($\Pi 2$) $\lambda \to at\lambda$
- ($\Pi 3$) $at\alpha_j^{(i)} \to \neg\, at\alpha_k^{(i)}$ if $j \neq k$
- ($\Pi 4$) $at\lambda \wedge E_\lambda \to \neg nil_\Pi$
- ($\Pi 5$) $at\lambda \wedge \neg\lambda \to \bigcirc at\lambda$
- ($\Pi 6$) $at\alpha_e^{(i)} \to \bigcirc at\alpha_e^{(i)}$

- ($\Pi 7$) $\lambda \wedge P \rightarrow \bigcirc P$ if λ is the label of a statement not included in the set A

41. Specification axioms

The specification of a program Π contains three parts

- specification of possible sequences

  (CS) $\lambda \rightarrow (C_1 \wedge \bigcirc at\lambda_1) \wedge \cdots \wedge (C_q \wedge \bigcirc at\lambda_q)$ where $\exists i \in [1..p] : (\lambda, C_1, \lambda_1), \ldots (\lambda, C_q, \lambda_q) \in trans(\Pi_i)$ and $trans(\Pi_i)$ contains no other element beginning with $(\lambda, \ldots)$

- specification of the data structure
- specification of effects of the statements included in A of Π is given by formulas of the form $\lambda \wedge Q \rightarrow \bigcirc R$ where Q and R are the precondition and postcondition of the statement considered, respectively
- in case of assignment $\lambda : a := t$ the effect can be described by $\lambda \wedge P^{a \leftarrow t}$
- example: if $\lambda : a := 2 * a$ then $\lambda \wedge b = 2 * a \rightarrow \bigcirc(b = a)$

42. Examples

Example 4. Let $\alpha : a := 2 * a$. Prove that $\alpha \wedge b = 0 \wedge a = k \rightarrow \bigcirc(b = 0 \wedge a = 2 * k)$ is derivable.

1. $\alpha \wedge b = 0 \wedge 2 * a = 2 * k \rightarrow \bigcirc(b = 0 \wedge a = 2 * k)$ (effect)

2. $a = k \rightarrow 2 * a = 2 * k$ (data)

3. $\alpha \wedge b = 0 \wedge a = k \rightarrow \bigcirc(b = 0 \wedge a = 2 * k)$ (1. and 2.)

Example 5. Let α_1 : if $a \neq 0$ then α_2 :~ else α_3 :~ fi. Prove that $at\alpha_1 \rightarrow a > 0 \vdash \alpha_1 \rightarrow \bigcirc\, at\alpha_2$.

1. $at\alpha_1 \rightarrow a > 0$ (assumption)

2. $\alpha_1 \rightarrow (a \neq 0 \wedge \bigcirc at\alpha_2) \vee (a = 0 \wedge \bigcirc at\alpha_3)$ (CS ax.)

3. $\alpha_1 \rightarrow at\alpha_1$ ($\Pi 2$)

4.

$\alpha_1 \to a \neq 0$ (data, 1. and 3.)

5.

$\alpha_1 \to \bigcirc at\alpha_2$ (2. and 4.)

43. Form of program properties

- Safety properties are expressed by formulas of the form: $A \to \Box B$. If $A = TRUE$ the formula is reduced to $\Box B$.
- Liveness properties are expressed by formulas of the form: $A \to \Diamond B$
- The simplest form of precedence properties is

  $A \to B$ atnext C or

  $A \to B$ unless C.

44. Safety properties 1.

- Partial correctness

  Let Π be a non-cyclic parallel program. If Q holds upon the start of a computation of Π and the computation terminates then R holds upon termination.

  $start_\Pi \wedge Q \to \Box(at\alpha_e^{(1)} \wedge \ldots at\alpha_e^{(p)} \to R)$

- Global and generalized invariants

  Invariant is a predicate that always holds in some states. There are two aspects that can be considered:

  - I is true in every state

    $start_\Pi \wedge Q \to \Box R$

  - I is true in certain states

    $start_\Pi \wedge Q \to \Box(at\alpha_1 \vee \cdots \vee at\alpha_m \to I)$

45. Safety properties 2.

- Mutual exclusion

  Consider the parallel program Π and its two components Π_1 and Π_2 with the precondition Q. Suppose Π_1 contains a section beginning with label α_i and ending with label α_j, whereas Π_2 contains a section beginning with label β_k and ending with label β_l in such a way that $(\alpha_i, \ldots \alpha_j)$ and $(\beta_k, \ldots \beta_l)$ are critical sections. It means that the parallel components Π_1 and Π_2 must not be in theses sections in the same time. The mutual exclusion is expessed by

  $start_\Pi \wedge Q \to \Box\neg((at\alpha_i \vee \cdots \vee at\alpha_j) \wedge (at\beta_k \vee \cdots \vee at\beta_l))$

- Deadlock freedom

  A deadlock of Π occurs if its components are at locations α_i and β_k, respectively, and both B_1 and B_2 are false. The property that excludes deadlock is expressed by

$$start_\Pi \wedge Q \rightarrow \Box at\alpha_i \wedge at\beta_k \rightarrow B_1 \vee B_2$$

46. Liveness properties

- Total correctness and termination

  Let Π be a non-cyclic parallel program. If Q holds upon the start of a computation of Π then the computation terminates and R holds upon termination.

  $$start_\Pi \wedge Q \rightarrow \Diamond(at\alpha_e^{(1)} \wedge \ldots at\alpha_e^{(p)} \wedge R)$$

- Termination

  $$start_\Pi \wedge Q \rightarrow \Diamond(at\alpha_e^{(1)} \wedge \ldots at\alpha_e^{(p)})$$

- More general accessibility properties

  The properties above can be generalized in the following forms:

  $$at\alpha \rightarrow \Diamond at\alpha'$$

  $$at\alpha \wedge Q \rightarrow \Diamond(at\alpha' \wedge R)$$

47. Precedence properties

$$start_\Pi \wedge Q \rightarrow \Box(A \rightarrow R)$$

This property states that every time when A holds then R holds. More generally we can express a similar property but with some sequence $P_0, P_1, \ldots$ of assertions holding at all points with A :

$$start_\Pi \wedge Q \rightarrow P_0 \ atnext \ A$$

$$A \wedge P_i \rightarrow P_{i+1} \ atnext \ A$$

48. Example 1. - Reader/Writer problem

Reader/Writer problem requirements:

- at most one writer may be in its write section
- writers and readers may not be in their write and read sections at the same time
- However, arbitrary many readers may be in their critical section at the same time

A possible solution for achieving the goals given above is the program Π consisting of $n+m$ reader and writer parallel components.

Π: initial $ex = true \wedge s = 0 \wedge num = 0$;

cobigin $\Pi_1^r || \ldots || \Pi_n^r || \Pi_1^w || \ldots || \Pi_m^w$ coend

49. Example 2. - Reader/Writer problem

Reader component

Π_i^r :

```
loop
$\alpha_0^{(i)}$ : await ex=true then ex=false;
$\alpha_1^{(i)}$ : num=num+1;
$\alpha_2^{(i)}$ : if num=1 then;
$\alpha_3^{(i)}$ : await s=true then s:=false fi;
$\alpha_4^{(i)}$ : ex:=true;
⋮ {read section}
$\alpha_5^{(i)}$ : await ex=true then ex:=false;
$\alpha_6^{(i)}$ : num:=num-1;
$\alpha_7^{(i)}$ : if num=0 then $\alpha_8^{(i)}$ : s:=true fi;
$\alpha_9^{(i)}$ : ex:=true;
end
```

50. Example 3. -Reader/Writer problem

Writer component

Π_i^w :

```
loop
$\beta_0^{(j)}$ : await s=true then s=false;
⋮ {write section}
$\beta_1^{(i)}$ : s:=true
end
```

Let L_i^r and L_j^w denote the set of labels in the component Π_i^r and Π_j^w, respectively. Furthermore, let define operator $exor_i$ in such a way, that $exor_i(A_1, \ldots, A_k)$ means that exactly i formulas out of $A_1, \ldots A_k$ are true. Now we can express the mutual exclusion by the formula

$$start_\Pi \rightarrow \Box exor_1((atL_1^r, \ldots, atL_k^r), atL_1^w, \ldots, atL_m^w)$$

51. Model checking 1.

- The model checker tools provide an algorithmic mean determining whether the defined abstract model satisfies the specification.

- In order to establish the model checking process, two task should be solved. First, the finite model of the system should be defined in the language of a model checker tool. Second, the specification of the system should be expressed. The specification of the system is the set of properties we are interested in.
- Model checker tools usually support temporal languages like Linear Temporal Logic and Computational Tree Logic for expressing the properties.

52. Model checking 2.

- During the verification procedure the model checker tool investigates every possible behaviour of the modelled system. Then the tool informs the user which property proved to be true. If the model fails to satisfy the specification, most tools provide the user with a counterexample. A counterexample is a possible execution of the system, which violates the specification.

53. LTL Model checking

LTL formulas are evaluated on linear paths, and a formula is considered true in a given state if it is true for all the paths starting in that state. LTL specifications are introduced by the keyword LTLSPEC. Operators:

- Xp : p holds at the next state (Next)
- Gp : p holds on the entire subsequent path starting from the current state (Globally)
- Fp : p holds eventually, somewhere on the subsequent path starting form the current state (Finally)
- $[pUq]$: p is true up to a state in which condition q holds

54. CTL Model checking 1.

- In SMV a CTL specification is given as CTL formula introduced by the keyword “SPEC”.
- In CTL properties can be expressed that hols for all the paths that start in a state, as well as properties that hold for some of the paths start in a given state.
- Path quantifiers
  - $A\Phi$: Φ holds on all paths starting from the current state
  - $E\Phi$: there exists at least one path starting from the current state where Φ holds

55. CTL Model checking 2.

- AFp : for all the paths stating from a state, eventually in the future condition p must hold
- EFp : there exists some path that eventually in the future satisfies p
- AGp : condition p is always true, in all the states of all the possible paths
- EGp : there is some path along which condition p is continuously true
- AXp : condition p is true in all next states reachable from the current state
- EXp : condition p is true in some next states reachable from the current state

56. SMV language 1.

- NuSMV is a reimplemantation and extension of SMV, Symbolic Model Verifier. The input language of NuSMV model checker called SMV.
- The SMV language allows the desciption of finite state machines. Finite state machines consist of a set of variables and predicates on these variables.
- All assignments are made concurrently, i.e. all variables change value at the same time. Two concurrent assignment to the same variable are forbidden.

57. SMV language 2.

The model specification in SMV consists of three parts.

- The possible values of variables determine the space of states. A state is an assigment of values to a set of variables. These variables can be of type boolean or can be enumerative, and are declared using the VAR keyword. Constant 1 denotes true whereas 0 denotes false.
- The initial values of the variables and the transition relation should be defined as well. Predicates defining the initial state are proceded by the INIT keyword.
- There are predicates defining the transition relation, relating the current values of some variables with their possible next values. These predicates are proceded by the TRANS keyword.

58. A sample SMV model

```
  MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
    case
        state=ready & request: busy;
        1: {ready, busy};
    esac;
    -- Specification part:
SPEC AG(request -> AF (state = busy))
```

59. Moduls and hierarchy 1.

- Each SMV program has a module main.
- The modules are independent from each other and the main module. They communicate with each other by a clearly defined set of variables. Variables declared outside a module can be passed as parameters. Parameters are passed by reference.
- Modules can be instantiated.
- Internal variables of a module can be used in enclosing modules.

60. Moduls and hierarchy 2.

```
  MODULE main
VAR bit0 : counter_cell(1);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
SPEC
    AG AF bit2.carry_out
```

```
MODULE counter_cell(carry_in)
VAR value : boolean;
ASSIGN
   init(value) := 0;
   next(value) := value + carry_in mod 2;
DEFINE carry_out := value & carry_in;
```

61. Modelling interleaving

The program executes a step by non-deterministically choosing a process, then executing all of its assignment statements in parallel.

```
  MODULE main
VAR
   gate1 : process inverter(gate3.output);
   gate2 : process inverter(gate1.output);
   gate3 : process inverter(gate2.output);
SPEC
    (AG AF gate1.output) & (AG AF !gate1.output)
MODULE inverter(input)
VAR
   output : boolean;
ASSIGN
   init(output) := 0;
   next(output) := !input;
```

62. Mutual exclusion example 1.

```
  MODULE user(turn, id, other)
VAR state: {n, t, c};
ASSIGN init(state) := n;
   next(state) :=
   case
      state = n : {n, t};
      state = t & other = n: c;
      state = t & other = t & turn = id: c;
      state = c: n;
      1: state;
   esac;
SPEC AG(state = t -> AF (state = c))
```

63. Mutual exclusion example 2.

```
  MODULE main
VAR turn: {1, 2};
   user1: user(turn, 1, user2.state);
   user2: user(turn, 2, user1.state);
ASSIGN  init(turn) := 1;
   next(turn) :=
   case
      user1.state = n & user2.state = t: 2;
      user2.state = n & user1.state = t: 1;
      1: turn;
   esac;
SPEC AG !(user1.state = c & user2.state = c)
```

Chapter 6. Owiczki-Gries method: a proof technique for parallel programs

1. Extension of the sequential language 1.

cobigin statement:

$\textbf{cobegin } S_1 \| \ldots \| S_n \textbf{ coend}$

where $S_1, \ldots, S_n$ are statements.

- The execution of the $\textbf{cobegin}$ statement causes the statements S_i to be executed in parallel. Execution of the $\textbf{cobegin}$ statement terminates when execution of all processes S_i have terminated. There are no restrictions on the way in which parallel execution is implemented; nothing is assumed about the relative speeds of the processes.
- We do require that each assignment statement and each expression be executed or evaluated as an individual, indivisible action. However this restriction can be lifted if programs adhere to the following simple convention:

  Each expression E may refer at most one variable y which can be changed by another process while E is being evaluated, and E may refer to y at most once. A similar restriction holds for assignment statements x:=E.

2. Extension of the sequential language 2.

With this convention, the only indivisible action is the memory reference. That is, suppose process S_i refers variable c while a different process S_j is changing c. We require that the value received by S_i for c be the value of c either before or after the assignment to c, but it may not be some spurious value caused by the fluctuation of the value of c during assignment.

3. Extension of the sequential language 3.

await statement: $\textbf{await } B \textbf{ then } S$

where B is a boolean expression and S is a statement not containing a $\textbf{cobegin}$ or another $\textbf{await}$ statement.

- When a process attempts to execute an $\textbf{await}$, it is delayed until the condition B is true. Then the statement S is executed as an indivisible action. Upon termination of S, parallel processing continues. If two or more processes are waiting for the same condition B, any of them may be allowed to proceed when B becomes true, while the others continue waiting. The waiting processes can be scheduled by any scheduling rule.

  Note that evaluation of B is part of the indivisible action of the $\textbf{await}$ statement; another process is not allowed to change variables so as to make B false after B has been evalueated but before S begins execution.

4. Extension of the sequential language 4.

- The $\textbf{await}$ statement can be used to turn any statement S into an indivisible action:

  $\textbf{await } true \textbf{ then } S$

- or it can be used purely as a means of synchronization:

  $\textbf{await}$ “some condition” $\textbf{then}$ $skip$

5. Extension of the proof rules 1.

- $$\frac{\{P \land B\}\, S\, \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\, \{Q\}}$$
- $$\frac{\{P_1\}\, S_1\, \{Q_1\}, \ldots, \{P_n\}\, S_n\, \{Q_n\}\ \text{are interference-free}}{\{P_1 \land \ldots \land P_n\}\, \textbf{cobegin}\ S_1 \| \ldots \| S_n\ \textbf{coend}\, \{Q_1 \land \ldots \land Q_n\}}$$

6. Extension of the proof rules 2.

Definition 6.1. Given a proof $\{P\}\, S\, \{Q\}$ and a statement T with precondition $pre(T)$, we say that T does not interfere with $\{P\}\, S\, \{Q\}$ if the following two conditions hold:

1. $$\{Q \land pre(T)\}\, T\, \{Q\}$$

2. Let S' be any statement within S but not within an $\textbf{await}$. Then $\{pre(S) \land pre(T)\}\, T\, \{pre(S)\}$

Definition 6.2. $\{P_1\}\, S_1\, \{Q_1\}, \ldots, \{P_n\}\, S_n\, \{Q_n\}$ are interference-free if the following holds. Let T be an $\textbf{await}$ or an assignment statement (which does not appear in an $\textbf{await}$ of process S_i). Then for all j (where $j \neq i$), T does not interfere with $\{P_j\}\, S_j\, \{Q_j\}$

7. Extension of the proof rules 3.

To proove that a parallel program is correct with respect to a given specification, so called auxiliary variables are needed. Typically, they record the history of execution or inducate which part of a program is currently executing.

Definition 6.3. Let AV be a set of variables which appear in S only in assignments $x := E$, where x is in AV. Then AV is an auxiliary variable set for S.

Theorem 6.4. Auxiliary variable tansformation: Let AV be an auxiliary variable set for S' and P and Q assertions which do not contain free variables from AV. Let S be obtained from S' by deleting all assignments to the variables in AV. Then

$$\frac{\{P\}\, S'\, \{Q\}}{\{P\}\, S\, \{Q\}}$$

8. Blocking and deadlock 1.

Because of the $\textbf{await}$ statements, a process may be delayed or blocked at an $\textbf{await}$, until its condition B is true.

Definition 6.5. Suppose a statement S is being executed. S is blocked if it has not terminated, but no progress in its execution is possible, because it (or all of its subprocesses that have not yet terminated) are delayed at an $\mathbf{await}$.

Blocking by itself is harmless; processes may become blocked and unblocked many times during execution. However, if the whole program becomes blocked, this is serious because it can never be unblocked and thus the program can not terminate.

Definition 6.6. Execution of a program ends in deadlock if it is blocked.

9. Blocking and deadlock 2.

Definition 6.7. A program S with proof $\{P\}\,S\,\{Q\}$ is free from deadlock if no execution of S which begins with $Ptrue$ ends in deadlock.

We wish to derive sufficient conditions under which a program is free from deadlock.

Theorem 6.8. Suppose program S' is free from deadlock, and suppose S is derived from S' by application of Auxiliary variable transformation rule. Then S is also free from deadlock.

10. Blocking and deadlock 3.

Theorem 6.9. Let S be a statement with proof $\{P\}\,S\,\{Q\}$. Let the $\mathbf{await}$s of S which do not occur within $\mathbf{cobegin}$ of S denoted by A_j.

A_j: $\mathbf{await}\ B\ \mathbf{then}\ S$

Let the $\mathbf{cobegin}$s of S which do not occur whithin other $\mathbf{cobegin}$s of S be

T_k: $\mathbf{cobegin}\ S_1^k \| \ldots \| S_{n_k}^k\ \mathbf{coend}$

Define statements $D(S)$ and $D1(T_k)$ (see the following slide)

Then $D(S) = false$ implies that in no execution of S can be blocked. Hence, program S is free from deadlock.

11. Blocking and deadlock 4.

- $$D(S) = \left[\bigvee_{j=1}^{m} (pre(A_j) \wedge \neg B_j)\right] \vee \left[\bigvee_{k=1}^{n} D1(T_k)\right]$$
- $$D1(T_k) = \left[\bigwedge_{j=1}^{n} (post(S_i^k) \vee D(S_i^k))\right] \wedge \left[\bigvee_{i=1}^{n} D(S_i^k)\right]$$

12. Dining philosophers program

To prove freedom from deadlock for the dining philosophers program we use its proof outline given before. We have

$$D(diningphilosophers) = D1(DP)$$

$D1(DP) = F1 \wedge F2$ where

$F1 = [(post(DP0) \vee D(DP0)) \wedge \ldots \wedge (post(DP4) \vee D(DP4))]$
$F1 = [D(DP0) \wedge \ldots \wedge \vee D(DP4)]$ where

$D(DPi) = (eating[i] = 0) \wedge (ji < leqNi) \wedge I(forks) \wedge (af[i] \neq 2)$

$post(DPi) = (eating[i] = 0) \wedge I(forks) \wedge (ji = Ni + 1)$

$F1 \Rightarrow \forall i(af[i] = 2)$ $F2 \Rightarrow \exists i(af[i] \neq 2)$

$F1 \wedge F2 \Rightarrow false$, therefore our program is free from deadlock.

Chapter 7. Synthesis of Synchronization code

1. Introduction

For synthesis of the parallel programs we have many different mathematical tool like:

- classical prime logic
- temporal logic
- different kind of algebra

2. Synchronization synthesis of concurrent programs with prime logic

The concurrent programs are special parallel programs. They have non-deterministic sequential programs (ie. process) which work together to make a common goal. The processes communicate by

- shared variables
- message sending. In this case the processes have own local cache.

In most cases the concurrent program functionality and the synchronisation problems are separable.

3. The correct synthesis of synchronisation code

1.

   Define the problem

   - Assign that P_1, ..., P_n processes which work in the solution
   - Introduce that shared variables which needed to make the solution
   - Define an invariant, which describes the problem. The processes need to observe this statement.

1.

   Skeleton of the solution

   - We take assignment statements to the shared variables on processes. We do this that way the statements are transact in itself, it takes correct result.
   - We take the initial value of shared variables. The invariant fulfil with these shares variables
   - The initialization statement take in the atomic block, which execute with mutual exclusion.

4. The correct synthesis of synchronisation code

1.

   Generate abstract program

   - For every atomic statements we define the weakest prediction. It will guarantee that the invariant true before and after the execution of atomic statements.

- Where it needs, we assign guard for the atomic activity. These guards guarantee, that the atomic activity execute when the invariant do not damage.

1.

   Implements the atomic activities Transform the atomic activities to executable code. Use for it the semaphores.

5. How to define the invariant

$\{P\}\ S\ \{Q\}$ theorem is the usual form where S is a program P and Q partially correct in point of specification. The correctness of this theorem demonstrate with Qwicki and Gries method. In parallel environment there are two way to find the invariant rule:

- Mark with BAD predicate that states which are bad in our case. In that case the $\neg BAD$ is a good invariant.
- The states are marked with $GOOD$ predicate which immediately good for us.

Let $< S >$ is that the S statement is executed by atomic. Let define the $\{K\}\ S\ \{L\}$ theorem where K L predicates refer to the S local variables only. Let $< S >$ is that the S statement is executed by atomic.

6. How to define the invariant

Let define the $\{K\}\ S\ \{L\}$ theorem where K L predicates refer to the S local variables only. Let define the $\{K\}\ S\ \{L\}$ theorem where K L predicates refer to the S local variables only. Let I is invariant. It refers to sheared variable, and it is true before the execution of S atomic statements and after it. Let $< \mathbf{awaitB} \rightarrow \mathbf{S} >$ is the guarded atomic statements with the familiar semantics. Let $wp(S, Q)$ is a function which is defining the weakest prediction by Dijsktra algorithm.

7. How to define the invariant

Let S is a statement Q a predicate and $P = wp(S, Q)$ is a weakest predicate. If P is true, when execution of S is started, then P guarantee that when the execution of S terminates the Q is true. For example:

- In case $S = y := e$ assignment statement, $wp(y := e, Q) = Q[y|e]$, where $Q[y|e]$ is a predicate, which is calculated by the place of all of y free incidences of Q we substitute this expression in Q.
- $wp(S1; S2, Q) = wp(S_1, wp(S_2, Q))$

In $< \mathbf{awaitB} \rightarrow \mathbf{S} >$ atomic statement case the B guard can be define like:

$$K \wedge I \wedge B \Rightarrow wp(S, L \wedge I)$$

where K L and I already known predicates.

8. Example 1: Critical section

1.

   step **Define the problem**

We have processes $P_1, \ldots, P_n$ Each P_i process, where $i \in [1, \ldots, n]$ use the following algorithm:

- execute a critical section, where it uses some kind of resource exclusively

- execute a non-critical section, where it uses local data only

P_i executes cyclical the algorithm. Let define an array $in[1,...,n]$ where $in[i]=1$, if P_i is in the critical section and $in[i]=0$ otherwise. The invariant is:

$CS: in[1]+...+in[n]1$

9. Example 1: Critical section

1.

step **The skeleton of solution**

The processes use the $in[1,...,n]$ array commonly. P_i sets the $in[i]$ value to 1, when it starts its own critical section. When the critical section is ended, the value of $in[i]$ is 0. Initially $\forall i \in 1,...,n, in[i]=0$, so CS is true.

```
  var in [1..N]:integer := ([N]0) #Invariant CS
P[i:1..N]::do true -> {in[i] = 0}
                       <in[i] := 1>
                       {in[i] = 1}
                      //Critical section
                        <in[i] := 0>
                       {in[i] = 0}
                         //Non-critical section
               od
```

10. Example 1: Critical section

1.

step **Deduce the guards to protect the invariant**

We deduce the guard to protect the variant CS . Before and after the execution of every atomic statement the invariant is true, if we have a right guard. Let see the atomic statement of $< in[i] := 0 >$ P_i process. The weakest prediction is: $wp(in[i]:=1, in[i]=1 \wedge CS) = (1=1 \wedge in[1]+...+in[i-1]+1+in[i+1]+...+in[n] \leq 1)$ Because of the elements of array are 0 or 1 $in[1]+...+in[n]=0$, which is guard of the atomic statement. Let see the second assignment: $wp(in[i]:=0, in[i]=0 \wedge CS) = (0=0 \wedge in[1]+...+in[i-1]+0+in[i+1]+...+in[n] \leq 1)$ It is implicate the first $in[i]=1 \wedge CS$ guard. So the atomic statement do not have guard.

11. Example 1: Critical section

- Our solution after the 3rd step:

```
  var in [1..N]:integer := ([N]0)
P[i:1..N]::do true -> {in[i] = 0 \wedge CS}  #Invariant CS
                      <await in[1]+ ... + in[n] = 0 -> in[i] := 1>
                      {in[i] = 1 \wedge CS}
                      Critical section
                      <in[i] := 0>
                      {in[i] = 0 \wedge CS}
                      Non-critical section
           od
```

Because of the program is fulfils the CS invariant, so also the verification is true.

12. Example 1: Critical section

1.

step **Implement the atomic statements with semaphores**

Let we take in a mutex semaphore variable:

$$mutex = 1 - (in[1] + ... + in[n])$$

CS demonstrate that the value of mutex is non-negative. So the atomic statements can be change to this: $< await mutex > 0 \rightarrow mutex := mutex - 1; in[i] := 1 >$ and $< mutex := mutex - 1; in[i] := 0 >$ So in this case the array in should became a auxiliary variable. It is effaceable from the solution. The solution:

```
    var mutex:semaphore := 1
  P[i:1..N]::do true -> P(mutex)
                         Critical section
                         V(mutex)
                         Non-critical section
             od
```

13. Discussion

The program synthetic method is applicable, when the following conditions are true:

- Semantically different guards refer to the different set of variables, and the atomic statements use these variables.
- Every guard seems like $expr > 0$, where $expr$ is a hole expression.
- Every guarded atomic statement contains an assignment statement, which decreases the value of transformed guard expression
- Every non-guarded atomic statement increases the value of transformed guard expression.

14. Example 2: Producer-consumer problem

Let see in that special case of the problem, when the buffer has one element. The buffer has two operation

- **deposit** : it takes an element out of the buffer
- **fetch** : it takes an element into the buffer

The classical constraints of problem are:

- We can not fetch an element, if the buffer is full
- We can not deposit an element, if the buffer is empty
- An element is no overwriting, until it is not deposited
- An element do not deposit two or more times

15. Example 2: Producer-consumer problem

1.

step **Define the problem**

- Processes:
  - Producer $[i : 1..M]$
  - Consumer $[j : 1..N]$
- Variables
  - $\mathbf{inD}$: a counter, which is count how many times try to execute the `deposit` operation by the processes since the system start.
  - $\mathbf{afterD}$: a counter, which is count how may times finished the execution of `deposit` operation by the processes since the system start.
  - $\mathbf{inF}$: a counter, which is count how many times try to execute the `fetch` operation by processes since the system start.
  - $\mathbf{afterF}$: a counter, which is count how may times finished the execution of `fetch` operation by the processes since the system start.

16. Example 2: Producer-consumer problem

The invariant is

$$PC : inD \leq afterF + 1 \wedge inF \wedge afterD$$

The invariant is informally:

- The `deposit` operation with at most one repeatedly can be started, than as much `fetch` operation ended till then.
- The `fetch` operation with at most one repeatedly can be started, than as much `deposit` an operation ended.

17. Example 2: Producer-consumer problem

1.

step **The skeleton of solution**

```
  var buf: T                 # for some type T
var inD, afterD, inF, afterF: integer := 0,0,0,0
Producer[i: 1..M]:: do true->
        produce message m
        deposit: <await inD := inD +1>
                 buf := m
                 <afterD := afterD +1>
    od
Consumer[j: 1..N]:: do true ->
        fetch: <await inF := inF+1>
               m := buf
               <afterF := afterF+1>
        consume message m
        od
```

18. Example 2: Producer-consumer problem

1.

step **Generate the abstract program**

```
  var buf: T # for some type T
var inD, afterD, inF, afterF: integer := 0,0,0,0
Producer[i: 1..M]:: do true->
                produce message m
                deposit: <await inD \leq afterF -> inD := inD + 1)
                         buf := m
                         <afterD := afterD +1>
             od
Consumer[j: 1..N]:: do true->
            fetch: <await inF < afterD -> inF := inF+1>
                   m := buf
                   <afterF := afterF +1>
            consume message m
         od
```

19. Example 2: Producer-consumer problem

1.

The solution

In this case the variable renaming method is applicable with method of binary semaphore

```
  empty = afterD - inD + 1
full = afterD -inF
```

Let b[1], ..., b[n] are binary semaphores, which fulfil the following invariant:

$$SPLIT: 0 \leq b[1] + ... + b[n] \leq 1$$

20. Example 2: Producer-consumer problem

```
  var buf: T # for some type T
var empty,full:semaphore := 1,0
Producer[i: 1..M]::
    do true-> produce message m
              deposit: P(empty)
                       buf := m
                       V(full)
    od
Consumer[j: 1..N]::
    do true->
        fetch: P(full)
               m := buf
               V (empty)
        consume message m
    od
```

Chapter 8. Synthesis of Synchronization code

1. Reminder

For synthesis of the parallel programs we have many different mathematical tool like:

- classical prime logic
- temporal logic
- different kind of algebra

2. Synchronization synthesis of concurrent programs with prime logic

The concurrent programs are special parallel programs. They have non-deterministic sequential programs (ie. process) which work together to make a common goal.

The processes communicate by

- shared variables
- message sending. In this case the processes have own local cache.

In most cases the concurrent program functionality and the synchronisation problems are separable.

3. The correct synthesis of synchronisation code

1.

   Define the problem

   - Assign that P_1, ..., P_n processes which work in the solution
   - Introduce that shared variables which needed to make the solution
   - Define an invariant, which describes the problem. The processes need to observe this statement.

1.

   Skeleton of the solution

   - We take assignment statements to the shared variables on processes. We do this that way the statements are transact in itself, it takes correct result.
   - We take the initial value of shared variables. The invariant fulfil with these shares variables
   - The initialization statement take in the atomic block, which execute with mutual exclusion.

4. The correct synthesis of synchronisation code

1.

   Generate abstract program

- For every atomic statements we define the weakest prediction. It will guarantee that the invariant true before and after the execution of atomic statements.
- Where it needs, we assign guard for the atomic activity. These guards guarantee, that the atomic activity execute when the invariant do not damage.

1.

Implements the atomic activities Transform the atomic activities to executable code. Use for it the semaphores.

5. Example 3: Reader-writer problem

1.

step **Define the problem**

- Processes:
  - Reader $[i:1..M]$ readers
  - Writer $[j:1..N]$ writers
- Variables:
  - $\mathbf{nr}$: the number of processes, which are just read a database.
  - $\mathbf{nw}$: the number if processes, which are just write a database.
- Invariant:

$$RW:(nr=0 \vee nw=0) \wedge nw \leq 1$$

6. Example 3: Reader-writer problem

1.

step **The skeleton of solution**

```
  var nr, nw: integer := 0,0
Reader[i: 1..M]::
    do true->
        <nr := nr + 1>
        read the database
        <nr := nr - 1>
    od
Writer[j: 1..N]::
    do true ->
        <nw := nw + 1>
        write the database
        <nw := nw + 1>
    od
```

7. Example 3: Reader-writer problem

1.

step **Generate the abstract program**

```
  var nr, nw: integer := 0,0 # Invariant RW
Reader[i: 1..M]::
    do true->
        <await nw = 0 -> nr := nr + 1>
        read the database
        <nr := nr - 1>
    od
Writer[j: 1..N]::
    do true ->
        <await nr = 0 and nw = 0 -> nw := nw + 1>
        write the database
        <nw := nw + 1>
    od
```

8. Example 3: Reader-writer problem

1.

The solution

Two kind of atomic statments:

- $F_1 :< S_i >$

or

- $F_2 :< await B_j \rightarrow S_j >$

```
 F1: P(e) {I}
    Si: {I}
    SIGNAL
```

F2:

P(e) {I}

if B_j $\longrightarrow$ **skip** □**not** B_j $\longrightarrow$ d_j := d_j + 1; **V**(e); **P**(c_j) **fi** {I $\wedge$ B_j }

S_j : {I}

SIGNAL

9. Example 3: Reader-writer problem

SIGNAL:

if B_1 and d_1 > 0 $\longrightarrow$ {I $\wedge$ B_1} d_1:= d_1-1; **V**(c_1)

□

□ B_n and d_n > 0 $\longrightarrow$ {I $\wedge$ B_n } d_n:= d_n -1; **V**(c_n)

□**else** $\longrightarrow$ {I} **V**(e)

fi

Let define two semaphores:

- `r`: it suspends the readers, if $nw = 0$ is false
- `dr` is a counter for `r` semaphore
- `w`: it suspends the writes, if $nr = 0 \vee nw = 0$ is false
- `dw` is a counter for `w` semaphore

10. Example 3: Reader-writer problem

```
var nr, nw: integer := 0,0 # Invariant RW
var e, r, w: semaphore := 1,0,0
# Invariant 0 $\leq$ (e + r + w) $\leq$ 1
var dr, dw: integer := 0,0
# Invariant dr $\geq$ 0 $\wedge$ dw $\geq$ 0
Reader [i:1..m] :: do true $\longrightarrow$ P(e)
if nw = 0 $\longrightarrow$ skip
□ nw > 0 $\longrightarrow$ dr:= dr+1; V(e); P(r)
fi
nr := nr + 1
$SIGNAL_1$
read the database
P(e)
nr := nr - 1
$SIGNAL_2$
od
```

11. Example 3: Reader-writer problem

```
Writer [j:1..n] :: do true $\longrightarrow$ P(e)
if nr = 0 and nw = 0 $\longrightarrow$ skip
□ nr > 0 or nw > 0 $\longrightarrow$ dw:= dw + 1; V(e); P(w)
fi
nw := nw + 1
$SIGNAL_3$
write the database
P(e)
nw := nw - 1
```

$SIGNAL_4$

od

12. Example 3: Reader-writer problem

$SIGNAL_i$:

if nw = 0 and dr > 0 $\longrightarrow$ dr := dr - 1; **V**(r)

☐ nr = 0 and nw = 0 and dw > 0 $\longrightarrow$ dw := dw - 1; **V**(w)

☐ (nr > 0 or dr = 0) and (nr > 0 or nw > 0 or dw = 0) $\longrightarrow$ **V**(e)

fi

13. Example 3: Reader-writer problem

var nr, nw, dr, dw:**integer** := 0,0,0,0 # Invariant RW

var e, r, w:**semaphore** := 1,0,0

# Invariant 0 $\leq$ (e+r+w) $\leq$ 1, dr $\geq$ 0 $\wedge$ dw $\geq$ 0

Reader [i:1..m] :: do true $\longrightarrow$ **P**(e)

if nw = 0 $\longrightarrow$ **skip**

☐ nw > 0 $\longrightarrow$ dr:= dr+1; **V**(e); **P**(r)

fi

nr := nr + 1

if dr > 0 $\longrightarrow$ dr := dr - 1; **V**(r)

☐ dr = 0 $\longrightarrow$ **V**(r)

fi

read the database

P(e)

nr := nr - 1

if nr = 0 and dw > 0 $\longrightarrow$ dw := dw - 1; **V**(w)

☐ (nr > 0 or dw = 0) and dw = 0 $\longrightarrow$ **V**(e)

fi

od

14. Example 3: Reader-writer problem

Writer [j:1..n] :: do true $\longrightarrow$ **P**(e)

if nr = 0 and nw = 0 $\longrightarrow$ **skip**

☐ nr > 0 or nw > 0 $\longrightarrow$ dw:= dw + 1; **V**(e); **P**(w)

```
fi
nw := nw + 1
V(e);
write the database P(e)
nw := nw - 1
if dr > 0 $\longrightarrow$ dr := dr - 1; V(r)
□ dw > 0 $\longrightarrow$ dw := dw - 1;V(w)
□ dw = 0 and dw = 0 $\longrightarrow$ V(e)
fi
od
```