TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

0101seda010100

software engineering dependability

Software Quality Assurance
Formal Verification

Contents

- Symbolic Test
- Formal verification
  - Inductive assertion method according to Floyd
  - The Hoare-Calculus
  - Total Correctness
  - Handling of Structured Data Types
  - Algebraic Specifications

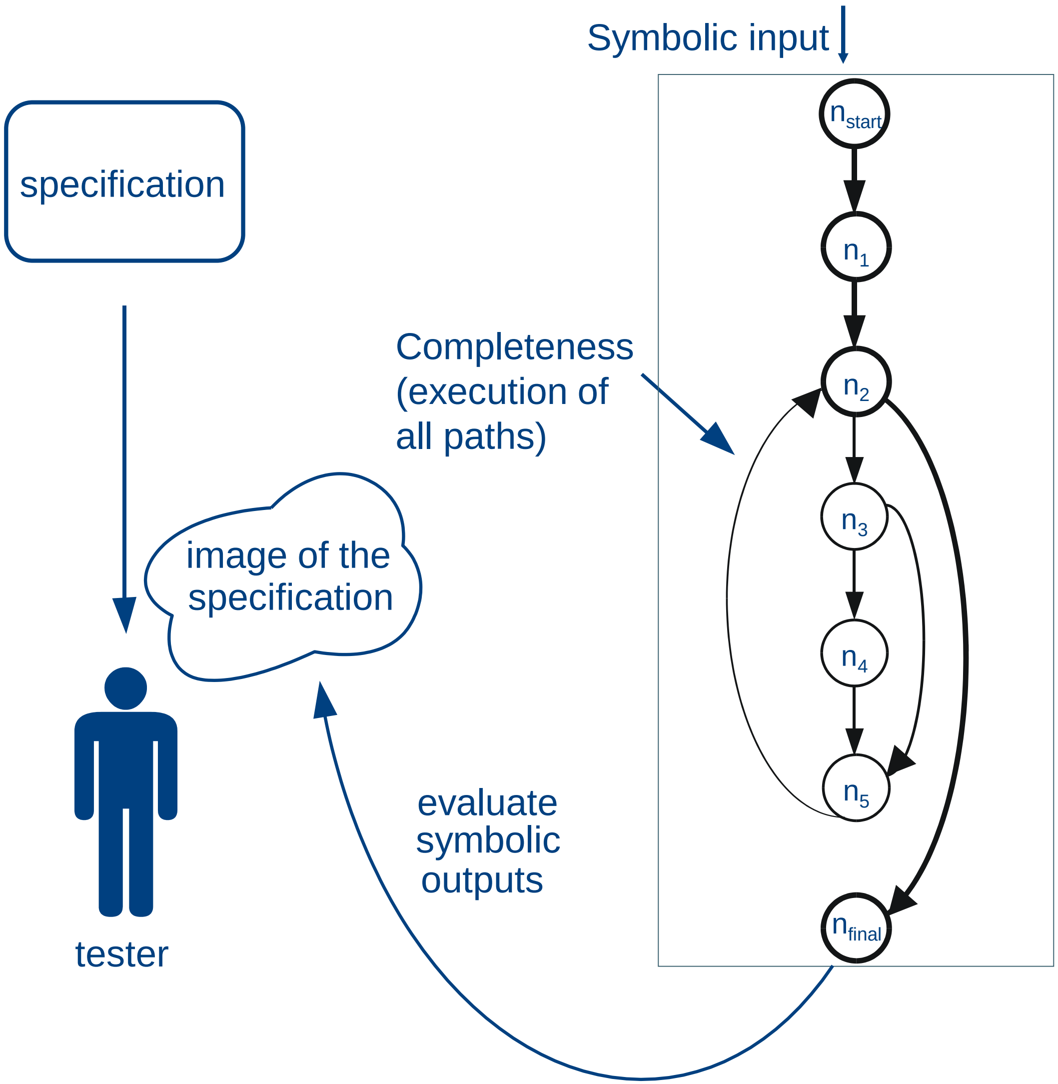
© Prof. Dr. Liggesmeyer

Symbolic Test - Properties

- The unit under test is executed with symbolic values by an interpreter
- Symbolic test runs in an artificial environment
- Symbolic test gains complete information about the correctness for whole input areas
- Symbolic test sometimes can prove the correctness
- Symbolic test takes a position between dynamic test, static analysis and verification

© Prof. Dr. Liggesmeyer

Symbolic Test



© Prof. Dr. Liggesmeyer

Symbolic Test
Symbolic Execution of *MinMax*

```
void MinMax (int &Min, int &Max)   1.
{
    int Help;                      2.
    if (Min > Max)
    {
        Help = Min;                4.
        Min = Max;                 5.
        Max = Help;                6.
    }
}                                  7.
```

Step	Min	Max	Help	PC	Variable name
1.	MIN	MAX		*true*	Symbolic value
2.	MIN	MAX	?	*true*	

3. (MIN > MAX)?

branching of symbolic execution

Step	Min	Max	Help	PC	Variable name
	MIN	MAX	?	MIN > MAX	Symbolic value
4.	MIN	MAX	MIN	MIN > MAX	
5.	MAX	MAX	MIN	MIN > MAX	
6.	MAX	MIN	MIN	MIN > MAX	
7.	MAX	MIN	MIN	MIN > MAX	

Step	Min	Max	Help	PC	Variable name
	MIN	MAX	?	MIN ≤ MAX	Symbolic value
8.					
9.	MIN	MAX	?	MIN ≤ MAX	

© Prof. Dr. Liggesmeyer

Symbolic Test
Symbolic Execution of *MinMax*

- At the beginning assignment of the symbolic values to Min and Max by the interpreter
  - Min = MIN $\wedge$ Max = MAX
- Specification of the procedure: At the end of the procedure the smaller value should be in Min and the greater value in Max on the side condition that the values are permuted at most, but not modified
- The symbolic results of the two program paths are
  1. MIN > MAX $\wedge$ Min = MAX $\wedge$ Max = MIN $\wedge$ Help = MIN
  2. MIN $\leq$ MAX $\wedge$ Min = MIN $\wedge$ Max = MAX
- These two terms describe the desired behavior

© Prof. Dr. Liggesmeyer

Symbolic Test - Problems

- As symbolic results are always assigned to a program path, for programs with an infinite number of paths also the number of symbolic results is infinite
- Problems caused by the data structures available in modern programming languages, as arrays or pointers
  - If, e.g., an array is accessed during the symbolic execution often the array index is a symbolic term so that the number of possibilities to be considered increases very fast

© Prof. Dr. Liggesmeyer

Symbolic Test - Problems

- If a program uses arrays during the symbolic execution normally the array index is a symbolic value so that in general it cannot be decided which concrete array element has to be accessed

```
VAR Array : ARRAY [1..10] OF CARDINAL;
  FOR i:= 1 TO 10 DO
  Array [i] := 0
END;
REPEAT
  ReadCard (i)
UNTIL ((i>=1) AND (i<=10));
Array [i] := 10;
```


© Prof. Dr. Liggesmeyer

Symbolic Test - Problems

- An array with ten elements of the type CARDINAL is initialized at first by assigning the value zero to all elements. Subsequently the variable i gets a value by keyboard entry, which is used as the index to an array element to which the value ten is assigned
- Which of the array elements gets the value ten is determined by the concrete value i. A symbolic test tool is not able to decide this on the basis of the symbolic value

© Prof. Dr. Liggesmeyer

Symbolic Test - Problems

- Floating point variables also cause problems due to the discrepancy between the discrete arithmetic of a computer and the continuous character of real numbers
- One usually requires that the symbolic execution followed by the substitution of the symbolic values by real input values leads to the same result as the choice of concrete inputs followed by a conventional program execution
- This rule is no longer valid if floating point variables are used. The symbolic execution cannot consider the discrete character floating point numbers have in the computer arithmetic and consequently treats them as value-continuous

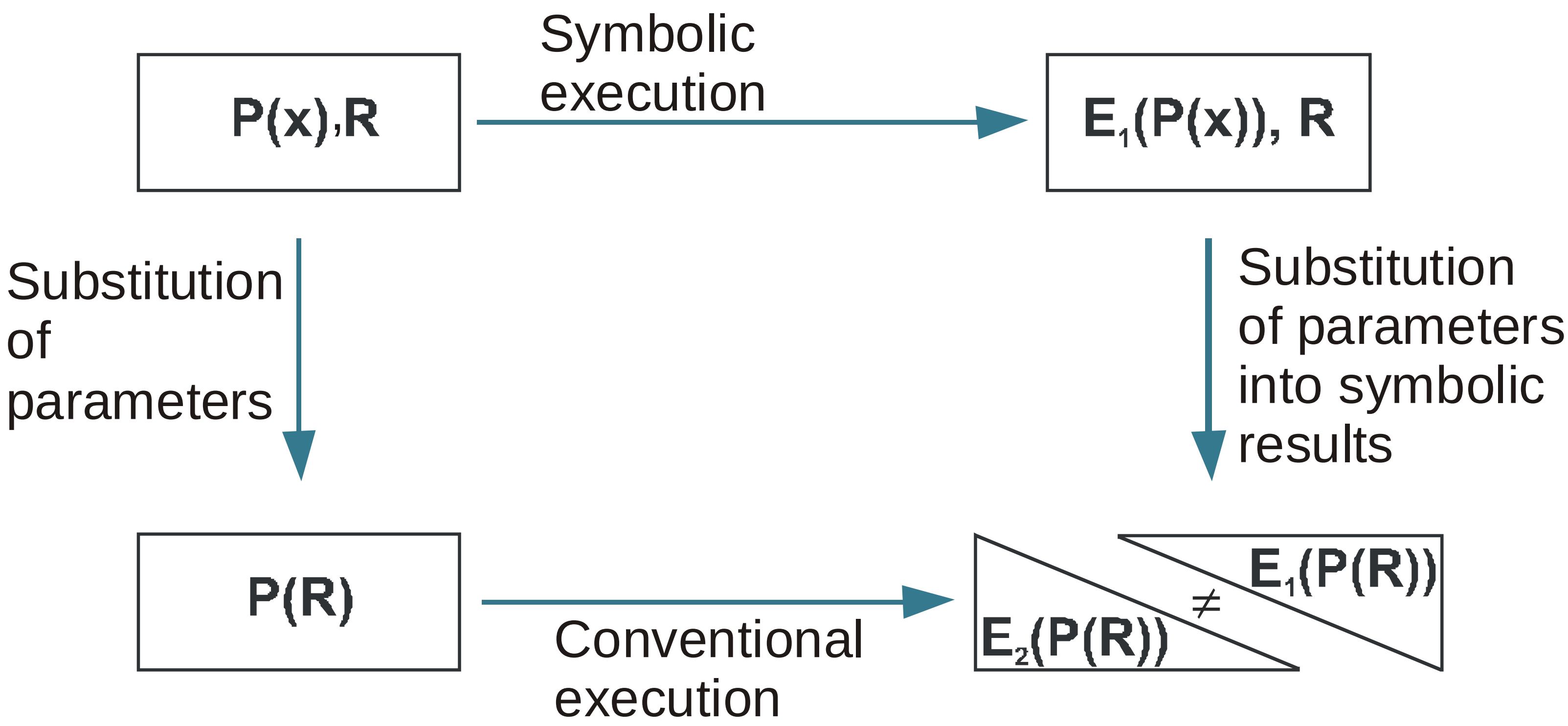
© Prof. Dr. Liggesmeyer

Symbolic Test - Problems

- In the symbolic execution no approximation errors occur. An insertion of concrete values in the symbolic results produce absolute exact values
- If a program is executed conventionally with concrete inputs approximations occur due to the computational accuracy associated with floating point values

© Prof. Dr. Liggesmeyer

Symbolic Test - Problems



© Prof. Dr. Liggesmeyer

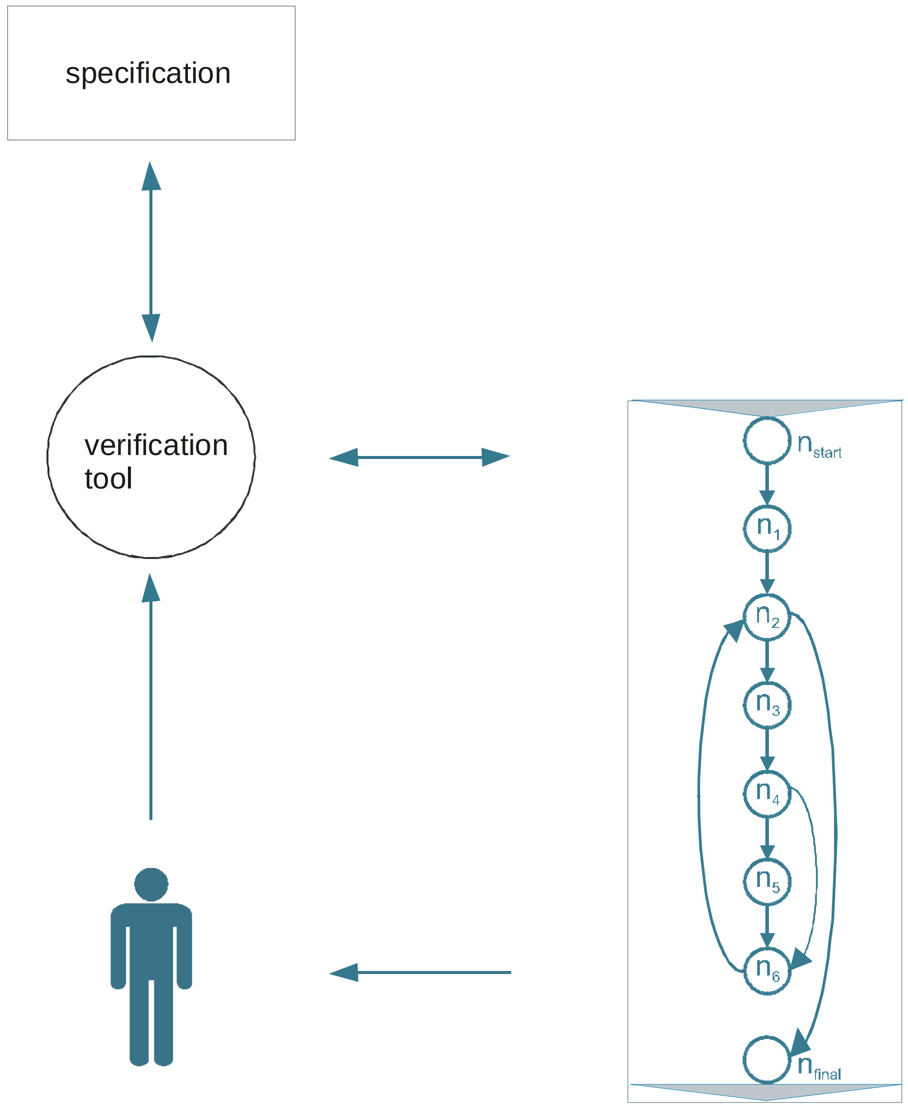
Formal Verification

- The verification demonstrates consistency between specification and implementation using proof techniques
- A formal specification is required
- The verification can prove the correctness

© Prof. Dr. Liggesmeyer

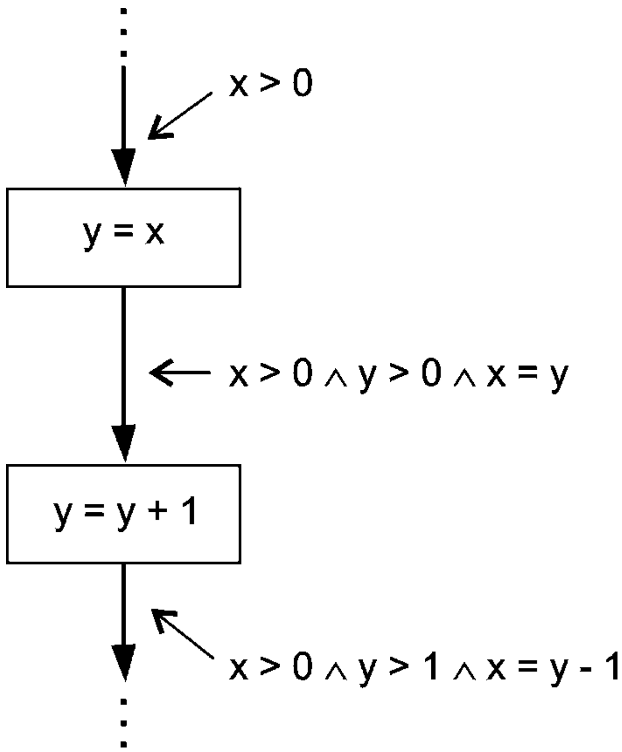
0101seda010100
software engineering dependability

Formal Verification



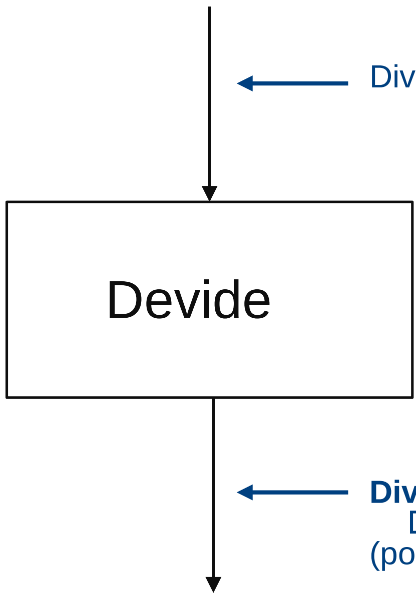
© Prof. Dr. Liggesmeyer

Assertions



© Prof. Dr. Liggesmeyer

Precondition and Postcondition



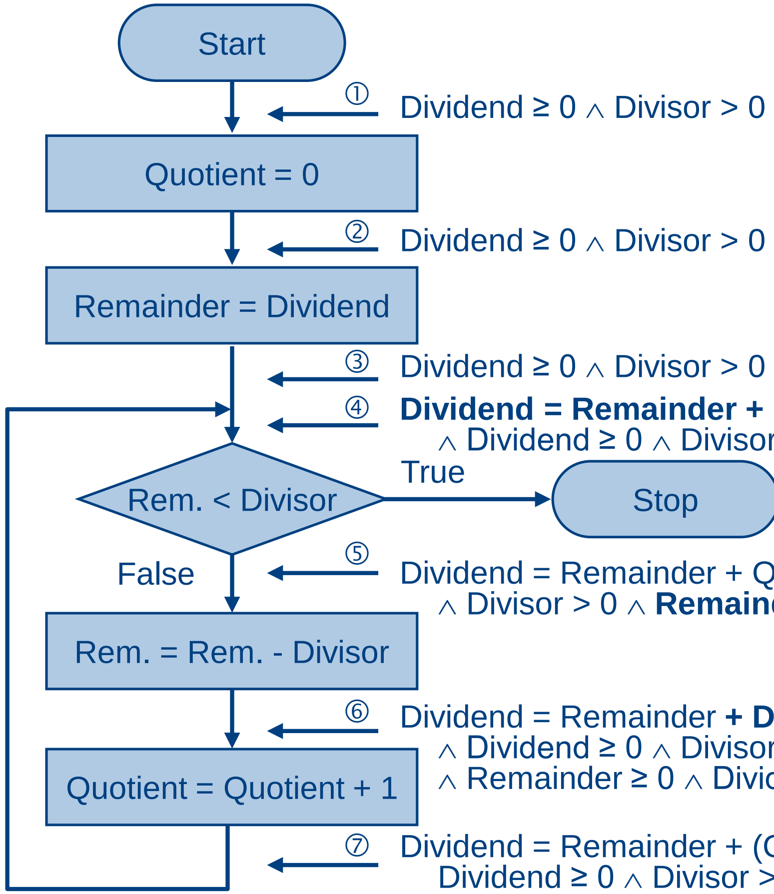
Dividend ≥ 0 ∧ Divisor > 0 (precondition)

Dividend = Remainder + Quotient * Divisor ∧ Remainder ≥ 0 ∧ Remainder < Divisor ∧
Dividend ≥ 0 ∧ Divisor > 0
(postcondition)

© Prof. Dr. Liggesmeyer

Formal Verification - Example

- Verification of a program flow chart with the inductive assertion method according to Floyd



① Dividend ≥ 0 ∧ Divisor > 0 (precondition)

② Dividend ≥ 0 ∧ Divisor > 0 ∧ **Quotient = 0**

③ Dividend ≥ 0 ∧ Divisor > 0 ∧ Quotient = 0 ∧ **Remainder = Dividend** ∧ **Remainder ≥ 0**

④ **Dividend = Remainder + Quotient * Divisor** ∧ **Remainder ≥ 0** (loop invariant) ∧ Dividend ≥ 0 ∧ Divisor > 0

⑧ **Dividend = Remainder + Quotient * Divisor ∧ Remainder ≥ 0 ∧ Remainder < Divisor** ∧ ∧ Dividend > 0 ∧ Divisor > 0 (postcondition)

⑤ Dividend = Remainder + Quotient * Divisor ∧ Remainder ≥ 0 ∧ Dividend ≥ 0 ∧ Divisor > 0 ∧ **Remainder ≥ Divisor**

⑥ Dividend = Remainder **+ Divisor** + Quotient * Divisor ∧ Remainder + **Divisor** ≥ Divisor ∧ Dividend ≥ 0 ∧ Divisor > 0 ⇔ Dividend = Remainder + (Quotient + 1) * Divisor ∧ Remainder ≥ 0 ∧ Dividend ≥ 0 ∧ Divisor > 0

⑦ Dividend = Remainder + (Quotient – 1 + 1) * Divisor ∧ Remainder ≥ 0 ∧ Dividend ≥ 0 ∧ Divisor > 0 ⇔ **Dividend = Remainder + Quotient * Divisor ∧ Remainder ≥ 0** ∧ Dividend ≥ 0 ∧ Divisor > 0

© Prof. Dr. Liggesmeyer

- Basis
  - A program is a formal description of behavior. It contains all the information necessary to determine the program properties and the effects of a program execution
  - It is possible to determine the behavior of a program by application of inference rules
  - A formal description of the semantics is necessary
- If S is a program or a part of a program and P is the precondition before the execution of S and if after the execution of S the postcondition Q is valid on condition that S terminates, one writes
  - P { S } Q

© Prof. Dr. Liggesmeyer

- P is the precondition of S w.r.t. Q
  - If S is a complete program, P is also referred to as the entry assertion and Q as the exit assertion
  - If no precondition exists one writes TRUE { S } Q
- The effect of an assignment x := f might be described as follows
  - P_f^x { x := f } P
  - P_f^x emerges from P by substitution of all occurrences of x with f. The assertion P which should be true after the execution of the assignment had to be fulfilled before the execution for the variable on the right hand side of the assertion

© Prof. Dr. Liggesmeyer

Formal Verification
The Hoare-Calculus: Example

P_f^x { x := f } x > 0

- If after the execution of the assignment x > 0 is to be valid before the execution of the assignment f > 0 had to be fulfilled. This is the precondition P_f^x which is generated by simple substitution of all variables x of the postcondition by variables f
  - f > 0 { x: = f} x > 0
- In general the semantics of the assignment can be described as axiom by
  - A0: P_f^x { x := f} P
- Hoare gives many additional rules in order to be able to deal with all constructs of a programming language

© Prof. Dr. Liggesmeyer

The Hoare-Calculus: Inference Rules

- A1: $\dfrac{P\ \{S\}\ Q,\ Q \supset R}{P\ \{S\}\ R}$ A postcondition Q can be replaced by a weaker Postcondition R
- A2: $\dfrac{Q\ \{S\}\ R,\ P \supset Q}{P\ \{S\}\ R}$ A precondition Q can be replaced by a stronger Precondition P
- A3: $\dfrac{P\ \{S_1\}\ Q,\ Q\ \{S_2\}\ R}{P\ \{S_1;\ S_2\}\ R}$ If P is precondition of S_1 wrt. Q and Q is precondition of S_2 wrt. R, than P is precondition of the Sequence S_1; S_2 wrt. R
- A4: $\dfrac{P \wedge B\ \{S\}\ P}{P\ \{\text{WHILE B DO S END}\}\ P \wedge \neg B}$ If $P \wedge B$ is precondition of S wrt. P then P can be used as loop invariant for the loop with S as the body of the loop and B as the loop condition. P is precondition of the loop *WHILE B DO S END* wrt. $P \wedge \neg B$

© Prof. Dr. Liggesmeyer

- Let us assume – in contrast to the previous example - that we would require no specific precondition, i.e., instead of the precondition $Dividend \geq 0 \wedge Divisor > 0$ we would use the boolean constant *TRUE*
- We would also be satisfied if we could prove a weaker postcondition in comparison to the previous example. We do not require $Remainder \geq 0$. We will use the precondition:

$$Dividend = Remainder + Quotient * Divisor \wedge \neg (Divisor \leq Remainder)$$

- The left column numbers the steps of the proof. The right column contains the used rule and if necessary the numbers of the used steps of the proof

© Prof. Dr. Liggesmeyer

No.	precondition	statement	postcondition	axiom
1	TRUE $\Rightarrow$ Dividend = Dividend + 0 * Divisor			lemma
2	Dividend = Dividend + 0 * Divisor	{Remainder = Dividend}	Dividend = Remainder + 0 * Divisor	A0
3	Dividend = Remainder + 0 * Divisor	{Quotient = 0}	Dividend = Remainder + Quotient * Divisor	A0
4	TRUE	{Remainder = Dividend}	Dividend = Remainder + 0 * Divisor	A2 (1,2)
5	TRUE	{Remainder = Dividend, Quotient = 0}	Dividend = Remainder+ Quotient * Divisor	A3 (4,3)
6	Dividend = Rem.+Quotient*Divisor $\wedge$ Divisor $\leq$ Rem. $\Rightarrow$ Dividend = (Rem.-Divisor) +(Quotient+1)*Divisor			lemma
7	Dividend = (Remainder - Divisor) + (Quotient + 1) * Divisor	{Rem. := Rem. - Divisor}	Dividend = Rem. + (Quotient + 1) * Divisor	A0
8	Dividend = Remainder + (Quotient + 1) * Divisor	{Quotient := Quotient + 1}	Dividend = Remainder+ Quotient * Divisor	A0
9	Dividend = (Remainder - Divisor) + (Quotient + 1) * Divisor	{Remainder := Rem. - Divisor; Quotient := Quotient + 1}	Dividend = Remainder+ Quotient * Divisor	A3 (7,8)
10	Dividend = Remainder + Quotient * Divisor $\wedge$ Divisor $\leq$ Remainder	{Remainder := Rem. - Divisor; Quotient := Quotient + 1}	Dividend = Remainder+ Quotient * Divisor	A2 (6,9)
11	Dividend = Remainder + Quotient * Divisor	{WHILE Divisor $\leq$ Rem. DO Rem. := Rem. - Divisor; Quotient := Quotient + 1 END}	$\neg$ (Divisor $\leq$ Remainder) $\wedge$ Dividend = Remainder + Quotient * Divisor	A4 (10)
12	TRUE	{Remainder = Dividend; Quotient = 0; WHILE Divisor $\leq$ Rem. DO Rem. := Rem. - Divisor; Quotient := Quotient + 1 END}	$\neg$ (Divisor $\leq$ Remainder) $\wedge$ Dividend = Remainder + Quotient * Divisor	A3 (5,11)

© Prof. Dr. Liggesmeyer

Total Correctness

- Termination of an arbitrary algorithm is not decidable. However, it might be proven for many programs
- A usual method is the use of well-sorted sets. Every not empty subset of a well-sorted set has a smallest element. Thus no infinitely decreasing sequences are possible
- A termination function is assigned to loops, which maps loop traversals into a well-sorted set W
- If it can be shown that the W-function after every loop iteration delivers a lower value than before, the values of the W-function form a strictly monotonic decreasing sequence. As in a well-sorted set a smallest element exists, on certain conditions no infinitely decreasing sequences are possible. From this it follows that the program terminates

© Prof. Dr. Liggesmeyer

- A special case of the required W-function is the so-called termination function t which maps the values of the program variables to the set of nonnegative integers
- Example
  - In the division program all involved variables are Integers
  - Divisor > 0 => Divisor ≥ 1
  - Dividend ≥ 0 => Remainder ≥ 0
  - In every loop execution Remainder is reduced by Divisor, but at the same time Remainder ≥ 0; from this follows: the loop terminates; the termination function is t = Remainder

© Prof. Dr. Liggesmeyer

- The loop rule of the Hoare-calculus might be adapted accordingly

$$\text{A4*.}\ \frac{P \wedge B\ \{S\}\ P,\ P \wedge B \wedge t = z\ \{S\}\ t < z,\ P \Rightarrow t \geq 0}{P\ \{\text{WHILE B DO S END}\}\ P \wedge \neg B}$$

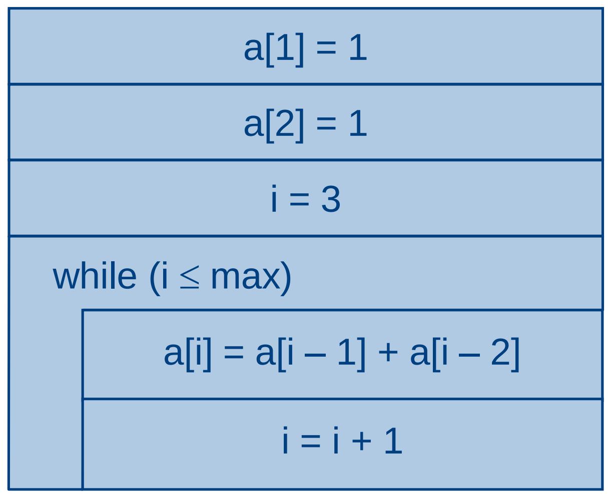
- z has to be constant for the considered program section (the loop): Before the execution of the loop body S, the value of t is z (t = z), and after the execution t< z, i.e. the value of the termination function becomes lower
- From the validity of the loop invariant P it has to follow that also $t \geq 0$ is valid

- Example division routine: z = Dividend, t = Remainder

© Prof. Dr. Liggesmeyer

Handling of Structured Data Types: Quantifiers

- Until now, it has been possible to write all assertions in simple logic. The cause is the exclusive use of simple data types
- For this example this does not work anymore



© Prof. Dr. Liggesmeyer

- The routine should assign the following values to an array a with the index area 1 to max, with max >= 2
  - a[1] and a[2] get the value 1. All array elements the index of which is greater than two are assigned the sum of the values of the two preceding array elements (i.e. a[3] = 2, a[4] = 3, a[5] = 5, etc.)
- In order to describe this, we need quantifiers
  - For all array elements ...
  - There is at least one array element ...
- A boolean algebra enhanced by quantifiers is called first order predicate calculus

© Prof. Dr. Liggesmeyer

- An array should be sorted in ascending order between the indices 0 and n
  - All j: 0 ≤ j < n: a[j] ≤ a[j+1]
  - $\forall_{j \mid 0 \leq j < n} a[j] \leq a[j+1]$
  - $\bigwedge_{j \mid 0 \leq j < n} a[j] \leq a[j+1]$

© Prof. Dr. Liggesmeyer

The Existence Quantifier

- An array has at least one positive element between the indices 0 and n
  - Ex j: 0 ≤ j ≤ n: a[j] > 0
  - $\underset{j|0\leq j\leq n}{\exists} a[j] > 0$
  - $\bigvee_{j|0\leq j\leq n} a[j] < 0$

© Prof. Dr. Liggesmeyer

Formal Verification
Handling of Structured Data Types: Quantifiers

$\max \geq 2$

$a[1] = 1$

$a[1] = 1, \max \geq 2$

$a[2] = 1$

$a[1] = 1, a[2] = 1, \max \geq 2$

$i = 3$

$i = 3, \max \geq 2$
Invariant P: $a[1] = 1, a[2] = 1,$
$(\text{All } j: 3 \leq j < i: a[j] = a[j-1] + a[j-2])$

while ($i \leq \max$) loop

- P and $i \leq \max$
- $a[i] = a[i-1] + a[i-2]$
- $(\text{All } j: 3 \leq j \leq i: a[j] = a[j-1] + a[j-2])$
  $a[1] = 1, a[2] = 1, i \leq \max$
- $i = i + 1$
- $(\text{All } j: 3 \leq j < i: a[j] = a[j-1] + a[j-2])$
  $a[1]=1, a[2]=1,$
  $i \leq \max + 1$

$(\text{All } j: 3 \leq j < i: a[j] = a[j-1] + a[j-2])$
$i > \max, i \leq \max + 1, a[1] = 1, a[2] = 1$
$\Rightarrow a[1] = 1, a[2] = 1, i = \max+1$
$(\text{All } j: 3 \leq j \leq \max: a[j] = a[j-1] + a[j-2])$

© Prof. Dr. Liggesmeyer

- An appropriate form of specification for data abstractions is the algebraic specification
- Data abstraction
  - data structure (internal memory)
  - access operations to this memory
- The only access to the memory is provided by the access operations belonging to the data abstraction
- The specification describes the data objects and the effects of the operations

© Prof. Dr. Liggesmeyer

Algebraic Specifications: Example

- Q is a variable of the type queue for integers and i is an integer
  - Push (Q, i) adds i to the end of the queue of Q; return type queue
  - Pop (Q) deletes an element from the head of the queue; return type queue
  - Initial (Q) determines if the queue is empty. The result is a boolean value
  - Length (Q) provides the actual number of elements in queue Q; return type integer, nonnegative

© Prof. Dr. Liggesmeyer

Algebraic Specifications: Example

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

- Possible axioms in the specification are:
  - pop (push (Q, i)) = IF Initial (Q) THEN Q ELSE push (pop (Q), i) END;
  - IF Initial(Q) THEN Length (Q) = 0 ELSE Length (Q) > 0
  - Length (push (Q, i)) = Length (Q) + 1
  - Length (pop (Q)) = IF Initial(Q) THEN error ELSE Length (Q) – 1
  - Initial(Q) = (Length(Q) = 0)
  - ...

© Prof. Dr. Liggesmeyer

1. pop (push (Q, i)) = IF Initial (Q) THEN Q ELSE push (pop (Q), i)
2. IF Initial(Q) THEN Length (Q) = 0 ELSE Length (Q) > 0
3. Length (push (Q, i)) = Length (Q) + 1
4. Length (pop (**Q**)) = IF Initial(**Q**) THEN error ELSE Length (**Q**) – 1
5. Initial(Q) = (Length(Q) = 0)

Substitution of Q by *push (Q,i)* in 4

- Length (pop (**push (Q, i))**) = IF **Initial(push (Q, i)**) THEN error ELSE Length (**push (Q, i)**) – 1

Applications of 5

- Length (pop (push (Q, i))) = **IF (Length (push (Q, i)) = 0)** THEN error ELSE Length (push (Q, i)) – 1

© Prof. Dr. Liggesmeyer

Algebraic Specifications: Example

logically false

Application of 3

- Length (pop (push (Q, i))) = IF (**(Length (Q) +1)** = 0) THEN error ELSE
  Length (push (Q, i)) – 1 = <u>Length (push (Q,i))</u> – 1

Application of 3

- Length (pop (push (Q, i))) = **Length (Q) + 1** – 1 = Length (Q)

Application of 1

- Length (pop (push (Q, i))) = **IF Initial (Q) THEN Length (Q) ELSE <u>Length (push (pop (Q), i))</u>**
  = Length (Q)

Application of 3

- IF Initial (Q) THEN Length (Q) ELSE **<u>Length (pop (Q))</u>** + 1 = Length (Q)

© Prof. Dr. Liggesmeyer

Algebraic Specifications: Example

logically false

Application of 4

- IF Initial (Q) THEN Length (Q) ELSE **(IF Initial(Q) THEN error ELSE (Length (Q) – 1)** + 1 = Length (Q)

⇒

- IF Initial (Q) THEN Length (Q) ELSE **Length (Q) – 1)** + 1 = Length (Q)

⇒

- IF Initial (Q) THEN Length (Q) ELSE **Length (Q)** = Length (Q)

⇒

- **Length (Q)** = Length (Q) (true assertion/statement)

© Prof. Dr. Liggesmeyer

Literature

- Burch J.R., Clarke E.M., Long D.E., McMillan K.L., Dill D.L., Symbolic Model Checking for Sequential Circuit Verification, in: IEEE Transactions on Computers, Vol. 13, No. 4, April 1994, pp. 401-424
- Clarke E.M., Emerson E.A., Sistla A.P., Automatic Verification of Finite state Concurrent Systems using Temporal Logic Specifications, in: ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 244-263
- Floyd R.W., Assigning meanings to Programs, in: Proceedings of the American Mathematical Society Symposium in Applied Mathematics, Vol. 19, 1967, pp. 19-32
- Hoare C.A.R., Proof of a Program: FIND, in: Communications of the ACM, Vol. 14, No. 1, January 1971, pp. 39-45
- Howden W.E., An evaluation of the effectiveness of symbolic testing, in: Software-Practice and Experience, Vol. 8, No. 4, July/August 1978, pp. 381-397
- Logrippo L., Melanchuk T., Du Wors R.J., The Algebraic Specification Language LOTOS: An Industrial Experience, in: Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, May 1990, Software Engineering Notes, Vol. 15, No. 4, September 1990, pp. 59-66

© Prof. Dr. Liggesmeyer