Formal Verification

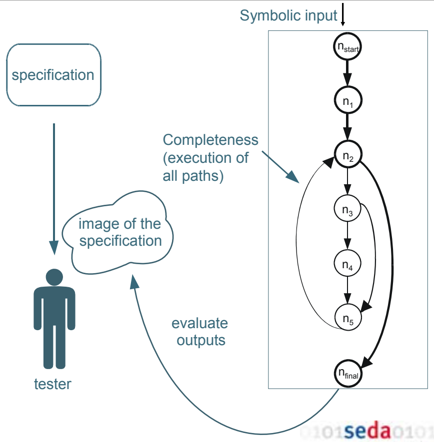
Contents

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

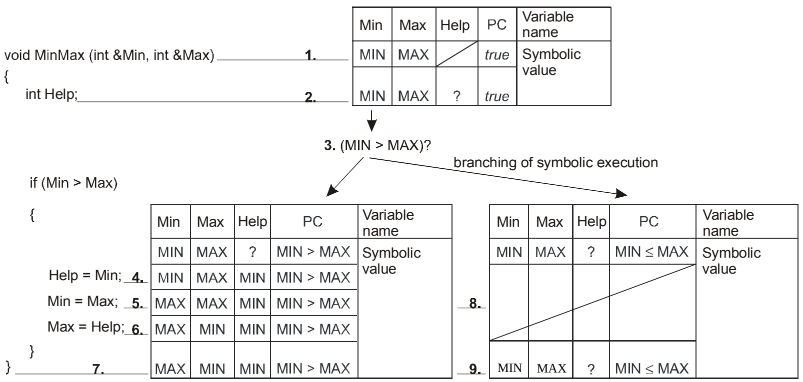
Symbolic Test
Properties

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

Symbolic Test



Symbolic Test
Symbolic Execution of *MinMax*



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

  From this follows
  - Max > Min $\wedge$ Min = MAX $\wedge$ Max = MIN

  2. MIN ≤ MAX $\wedge$ Min = MIN $\wedge$ Max = MAX

  Substitution results in
  - Min ≤ Max $\wedge$ Min = MIN $\wedge$ Max = MAX
- These two terms describe the desired behavior

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

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;
```

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

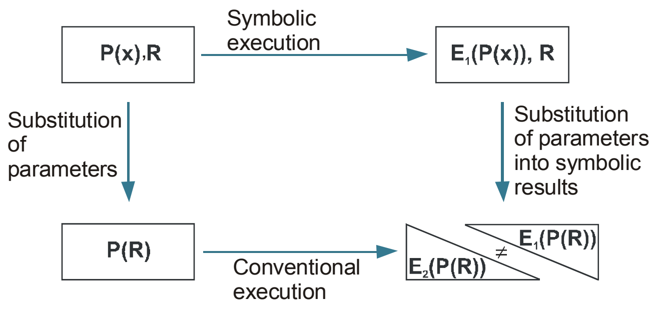
Symbolic Test
Problems

- 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

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

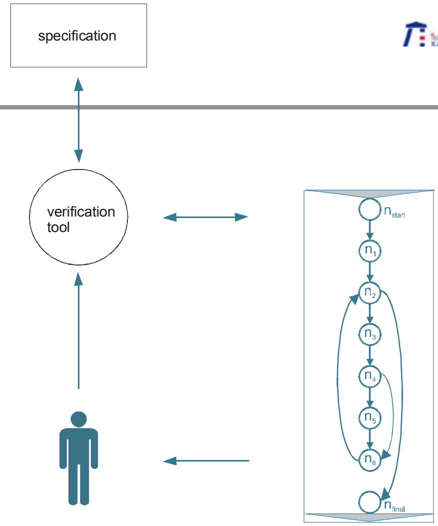
Symbolic Test
Problems



Formal Verification

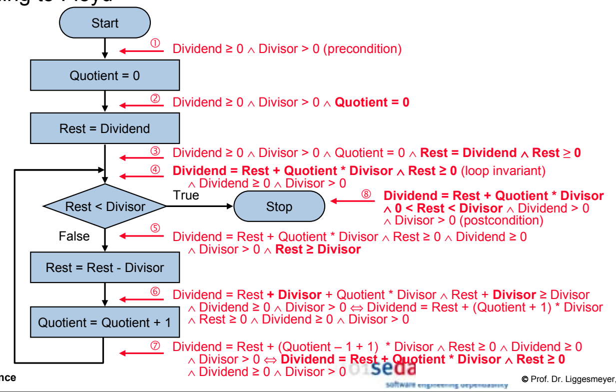
- ☐ The verification demonstrates consistency between specification and implementation using proof techniques
- ☐ A formal specification is required
- ☐ The verification can prove the correctness

Formal Verification



Formal Verification
Example

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



Formal Verification
The Hoare-Calculus

- ☐ 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

Formal Verification
The Hoare-Calculus

- 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 assertion 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 left hand side of the assertion

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

Formal Verification
The Hoare-Calculus: Inference Rules

- A1: $\dfrac{P\ \{S\}\ Q,\ Q \supset R}{P\ \{S\}\ R}$
- A2: $\dfrac{Q\ \{S\}\ R,\ P \supset Q}{P\ \{S\}\ R}$
- A3: $\dfrac{P\ \{S_1\}\ Q,\ Q\ \{S_2\}\ R}{P\ \{S_1;\ S_2\}\ R}$
- A4: $\dfrac{P \wedge B\ \{S\}\ P}{P\ \{\text{WHILE B DO S END}\}\ P \wedge \neg B}$

Formal Verification
The Hoare-Calculus: Inference Rules

- 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
- The specification of the program consists of an entry assertion and an exit assertion
- Since there is no specific precondition, the boolean constant TRUE is used
- The postcondition is Dividend = Rest + Quotient * Divisor ∧ ¬ (Divisor ≤ Rest)

Formal Verification
The Hoare-Calculus

No.	precondition	statement	postcondition	axiom
1	TRUE ⇒ Dividend = Dividend + 0 * Divisor			lemma
2	Dividend = Dividend + 0 * Divisor	{Rest = Dividend}	Dividend = Rest + 0 * Divisor	A0
3	Dividend = Rest + 0 * Divisor	{Quotient = 0}	Dividend = Rest + Quotient * Divisor	A0
4	TRUE	{Rest = Dividend}	Dividend = Rest + 0 * Divisor	A2 (1,2)
5	TRUE	{Rest = Dividend, Quotient = 0}	Dividend = Rest + Quotient * Divisor	A3 (4,3)
6	Dividend = Rest+Quotient*Divisor ∧ Divisor ≤ Rest ⇒ Dividend = (Rest-Divisor) +(Quotient+1)*Divisor			lemma
7	Dividend = (Rest - Divisor) + (Quotient + 1) * Divisor	{Rest := Rest - Divisor}	Dividend = Rest + (Quotient + 1) * Divisor	A0
8	Dividend = Rest + (Quotient + 1) * Divisor	{Quotient := Quotient + 1}	Dividend = Rest + Quotient * Divisor	A0
9	Dividend = (Rest - Divisor) + (Quotient + 1) * Divisor	{Rest := Rest - Divisor; Quotient := Quotient + 1}	Dividend = Rest + Quotient * Divisor	A3 (7,8)
10	Dividend = Rest + Quotient * Divisor ∧ Divisor ≤ Rest	{Rest := Rest - Divisor; Quotient := Quotient + 1}	Dividend = Rest + Quotient * Divisor	A2 (6,9)
11	Dividend = Rest + Quotient * Divisor	{WHILE Divisor ≤ Rest DO Rest := Rest - Divisor; Quotient := Quotient + 1 END}	¬ (Divisor ≤ Rest) ∧ Dividend = Rest + Quotient * Divisor	A4 (10)
12	TRUE	{Rest = Dividend; Quotient = 0; WHILE Divisor ≤ Rest DO Rest := Rest - Divisor; Quotient := Quotient + 1 END}	¬ (Divisor ≤ Rest) ∧ Dividend = Rest + Quotient * Divisor	A3 (5,11)

Formal Verification
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

Formal Verification
Total Correctness

- 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 => Rest ≥ 0
  - In every loop execution Rest is reduced by Divisor, but at the same time Rest ≥ 0; from this follows: the loop terminates; the termination function is t = Rest

Formal Verification
Total Correctness

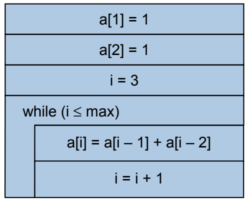
- The loop rule of the Hoare-calculus might be adapted accordingly

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

- 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 ≥ 0 is valid
- Example division routine: z = Dividend, t = Rest

Formal Verification
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



Formal Verification
Handling of Structured Data Types: Quantifiers

- ☐ 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

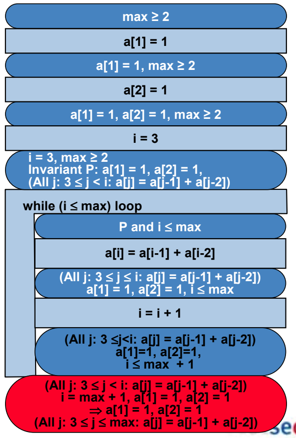
Formal Verification
The All-Quantifier

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

Formal Verification
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
  - $\exists_{j|0\le j\le n} a[j] > 0$
  - $\bigvee_{j|0\le j\le n} a[j] < 0$

Formal Verification
Handling of Structured Data Types: Quantifiers



Formal Verification
Algebraic Specifications

- 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

Formal Verification
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

Formal Verification
Algebraic Specifications: Example

- 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)
  - ...

Formal Verification
Algebraic Specifications: Example

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

Formal Verification
Algebraic Specifications: Example

Application of 3 — *logically false*

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

Application of 3

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

Formal Verification
Algebraic Specifications: Example

Application of 4 — *logically false*

- 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)

Formal Verification
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