Lecture 3: Verification of algorithms correctness
- Basic steps in algorithms correctness verification
- Formal methods in correctness verification
- Exercises

When an algorithm is designed it should be analyzed at least from the following points of view:

- *Correctness.* This means to verify if the algorithm leads to the solution of the problem (hopefully after a finite number of processing steps).
- *Efficiency.* This means to establish the amount of resources (memory space and processing time) needed to execute the algorithm on a machine (a formal one or a physical one).

## 1 Basic steps in algorithms correctness verification

To verify if an algorithms really solves the problem for which it is designed we can use one of the following strategies:

- *Experimental analysis (testing).* We test the algorithm for different instances of the problem (for different input data). The main advantage of this approach is its simplicity while the main disadvantage is the fact that testing cannot cover always all possible instances of input data (it is difficult to know how much testing is enough). However the experimental analysis allows sometimes to identify situations when the algorithm doesn't work.
- *Formal analysis (proving).* The aim of the formal analysis is to prove that the algorithm works for any instance of data input. The main advantage of this approach is that if it is rigourously applied it guarantee the correctness of the algorithm. The main disadvantage is the difficulty of finding a proof, mainly for complex algorithms. In this case the algorithm is decomposed in subalgorithms and the analysis is focused on these (simpler) subalgorithms. On the other hand the formal approach could lead to a better understanding of the algorithms. This approach is called formal due to the use of formal rules of logic to show that an algorithm meets its specification.

The main steps in the formal analysis of the correctness of an algorithm are:

- Identification of the properties of input data (the so-called *problem's preconditions*).
- Identification of the properties which must be satisfied by the output data (the so-called *problem's postconditions*).
- Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the postconditions.

When we analyze the correctness of an algorithm a useful concept is that of *state*.

*The algorithm's state is the set of the values corresponding to all variables used in the algorithm.*

The state of the algorithm changes (usually by variables assignments) from one processing step to another processing step. The basic idea of correctness verification is to establish which should be the state corresponding to each processing step such that at the end of the algorithm the postconditions are satisfied. Once we established these intermediate states is is sufficient to verify that each processing step ensures the transformation of the current state into the next state.

When the processing structure is a sequential one (for example a sequence of assignments) then the verification process is a simple one (we must only analyze the effect of each assignment on the algorithm's state).

Difficulties may arise in analyzing loops because there are many sources of errors: the initializations may be wrong, the processing steps inside the loop may be wrong or the stopping condition may be wrong. A formal method to prove that a loop statement works correctly is the mathematical induction method.

*Example.* Let us consider the following algorithm whose aim is to find the minimal value in a finite sequence of real numbers:

```
minimum($a[1..n]$)
$min \leftarrow a[1]$
for $i \leftarrow \overline{2,n}$ do
    if $min > a[i]$ then $min \leftarrow a[i]$ endif
endfor
return $min$
```

The input array can be arbitrary, thus the preconditions set consists only of $\{n \geq 1\}$ (meaning that the array is not empty).

The postcondition is represented by the property of the minimal value: $min \leq a[i]$ for all $i = \overline{1,n}$. We only have to prove that this postcondition is satisfied after the algorithm's execution.

Thus, we will prove by using the mathematical induction that $min \leq a[i]$, $i = \overline{1,n}$. Since $min = a[1]$ and the value of $min$ is replaced only with a smaller one it follows that $min \leq a[1]$.

Let us suppose that $min \leq a[k]$ for all $k = \overline{1, i-1}$. Now, we only have to prove that $min \leq a[i]$. In the **for** loop, the value of $min$ is modified as follows:

- If $min \leq a[i]$ then $min$ keeps its old value.
- If $min > a[i]$ then the value of $min$ is replaced with $a[i]$.

Thus in both situations $min$ will satisfy $min \leq a[i]$. According to the mathematical induction method it follows that the postcondition is satisfied, thus the algorithm is correct.

Let us consider now the following algorithm:

```
minim($a[1..n]$)
for $i \leftarrow \overline{1,n-1}$ do
    if $a[i] < a[i+1]$ then $min \leftarrow a[i]$
        else $min \leftarrow a[i+1]$
endfor
return $min$
```

In this case the processing step of the loop body will lead to $min = \min\{a[i], a[i+1]\}$, thus one cannot anymore prove that $min = \min\{a[1..i]\}$ implies $min = \min\{a[1..i+1]\}$. On the other hand it is easy to find a counterexample (for instance $(2,1,3,5,4)$) for which the above algorithm doesn't work. However it can be proved that it computes $\min\{a[n-1], a[n]\}$.

## 2 Formal methods in correctness verification

To prove the algorithms correctness one can apply the following strategy:

- based on problem's preconditions and postconditions some *assertions* concerning the algorithm state are constructed;
- for each processing state one proves that from the previous assertion one arrive to the next assertion.

The assertions are statements about the program's state which are "asserted" to be true. The assertions are used to annotate the algorithm as comments. They are useful not only for formal analysis but also as a documentation tool which can improve the understanding of the algorithm.

Let us consider the annotated version of the algorithm used in the previous lecture to find the minimum among three distinct values:

```
if $a < b$ then    //$\{a < b\}$
        if $a < c$
        then    //$\{a < b, a < c\}$
           $min \leftarrow a$    //$\{min = a, a < b, a < c\}$
              else    //$\{a < b, c < a\}$
                 $min \leftarrow c$    //$\{min = c, c < a, c < b\}$
        endif
else    //$\{b < a\}$
        if $b < c$
        then    //$\{b < a, b < c\}$
           $min \leftarrow b$    //$\{min = b, b < a, b < c\}$
        else    //$\{b < a, c < b\}$
           $min \leftarrow c$    //$\{min = c, c < b, c < a\}$
        endif
endif
```

Let us denote by $P$ the problem's preconditions, by $A$ the algorithm and by $Q$ the problem's postconditions. The triple $\langle P, A, Q\rangle$ (called Hoare triple) corresponds to a correct algorithm (in this situation we shall use the notation $P \xrightarrow{A} Q$) if the following statement holds:

*For input data which satisfies the preconditions $P$ the algorithm will: (i) lead a result which satisfies the problem's postconditions; (ii) stop after a finite number of processing steps.*

To verify the correctness of an algorithm we can analyze the correctness of each processing structure of the algorithm. For each processing structure (sequential, conditional and looping) there exist rules which make the verification easy.

**Sequential statement rule.** Let $A$ be the algorithm consisting of the sequence of processing steps: $(A_1, A_2, \dots, A_n)$. Let us denote by $P_{i-1}$ and $P_i$ the algorithm state before and after executing the action $A_i$. The sequential structure's rule can be stated as follows:

If $P \Rightarrow P_0$, $P_{i-1} \xrightarrow{A_i} P_i$ for $i = \overline{1,n}$ and $P_n \Rightarrow Q$ then $P \xrightarrow{A} Q$

*Example.* Let $a$ and $b$ be two distinct integers and $x$ and $y$ two variables which contain the values $a$ and $b$, respectively. Swap the values of the two variables.

The problem's preconditions are $\{x = a, y = b\}$ while the postconditions are $\{x = b, y = a\}$. Let us consider the following processing steps:

|  |  |
|---|---|
| $A_1$ : | $aux \leftarrow x$ |
| $A_2$ : | $x \leftarrow y$ |
| $A_3$ : | $y \leftarrow aux$ |

The assertions concerning the algorithm state are the following: $P_0 = \{x = a, y = b\}$ (before $A_1$), $P_1 = \{aux = a, x = a, y = b\}$ (after $A_1$), $P_2 = \{aux = a, x = b, y = b\}$ (after $A_2$), $P_3 = \{aux = a, x = b, y = a\}$ (after $A_3$). It is easy to see that $P = P_0$, $P_3 \Rightarrow Q$ and $P_{i-1} \xrightarrow{A_i} P_i$ for $i = \overline{1,3}$. According to the rule of sequential structure it follows that this algorithm is correct.

Let us consider now the following sequence of processing steps:

|  |  |
|---|---|
| $A_1$ : | $x \leftarrow y$ |
| $A_2$ : | $y \leftarrow x$ |

In this case the corresponding assertions are: $\{x = b, y = b\}$ (both after $A_1$ and after $A_2$). Thus, in this case the steps $A_1$ and $A_2$ cannot lead to postconditions.

The same problem of swapping two integer variables can be solved without using that auxiliary variable:

|  |  |
|---|---|
| $A_1$ : | $x \leftarrow x - y$ |
| $A_2$ : | $y \leftarrow x + y$ |
| $A_3$ : | $x \leftarrow y - x$ |

In this case the assertions corresponding to the algorithm's state are: $P_0 = \{x = a, y = b\}$, $P_1 = \{x = a - b, y = b\}$, $P_2 = \{x = a - b, y = a\}$, $P_3 = \{x = b, y = a\}$. It is easy to see that $P = P_0$, $P_3 = Q$ and $P_{i-1} \xrightarrow{A_i} P_i$ for $i = \overline{1,3}$.

Sometimes the following rules are also useful:

*Precondition's strengthening.* If $R \Rightarrow P$ and $P \xrightarrow{A} Q$ then $R \xrightarrow{A} Q$ (the algorithm is still correct if the precondition is stronger - the input data have additional properties).

*Postcondition's weakening.* If $P \xrightarrow{A} Q$ and $Q \Rightarrow R$ then $P \xrightarrow{A} R$ (the algorithm is still correct if the problem's demands are simplified).

*"Baggage lemma".* If $P_1 \xrightarrow{A} Q_1$ and $P_2 \xrightarrow{A} Q_2$ then $P_1 \wedge P_2 \xrightarrow{A} Q_1 \wedge Q_2$ (it can be used when we intend to prove that after applying an algorithm an assertion stays unaltered - this particular case correspond to $P_1 = Q_1$).

**Conditional statement rule.** Let us consider the following conditional structure:
`S:` **if** $c$ **then** $A_1$ **else** $A_2$ **endif**
If $P$ and $Q$ are the problem's preconditions and postconditions respectively then the rule can be stated as follows:

*If $c$ is well-defined (meaning that it can be evaluated), $P \wedge c \xrightarrow{A_1} Q$ and $P \wedge \overline{c} \xrightarrow{A_2} Q$ then $P \xrightarrow{A} Q$.*

This rule suggest the fact that we have to verify the correctness of each branch (both when $c$ is true and when $c$ is false).
*Example.* Let us consider the problem of finding the minimum between two distinct real values:

```
if $a < b$ then $A_1$ :   $m \leftarrow a$    $\{ a < b, m = a\}$
           else $A_2$ :   $m \leftarrow b$    $\{ a \geq b, m = b\}$
endif
```

The algorithm preconditions are $a \in R$, $b \in R$, $a \neq b$ and the postcondition is $m = \min\{a, b\}$. The condition $c$ is $a < b$. If $a < b$ then $m = a < b$ thus $m = \min\{a, b\}$. In the other case, $a > b$ holds and the processing step $A_2$ leads to $m = b < a$ thus we obtain again that $m = \min\{a, b\}$.

**Loop statement rule.** Loops are a very common source of programming errors because it is often difficult to determine that a loop body executes exactly the right number of times or that loop execution meets its specification. Since all loop statements can be rewritten by using a **while** loop we shall analyze only this loop statement. Let us consider the following structure:

$S$: **while** $c$ **do** $A$ **endwhile**,

a precondition $P$ and a postcondition $Q$.

The loop statement is correct if it exists an assertion $I$ (called *loop invariant*) concerning the algorithm's state and a function $t : N \to N$ (called *termination function*) for which the following properties hold:

(a) The assertion $I$ is true at the beginning (this means that the preconditions imply it, $P \Rightarrow I$).

(b) $I$ is an *invariant property*: if $I$ is true before executing $A$ and $c$ is also true then $I$ remains true after the execution of $A$ ($I \wedge c \xrightarrow{A} I$).

(c) At the end of the loop (when $c$ becomes false) the postcondition $Q$ can be inferred from $I$ ($I \wedge \overline{c} \Rightarrow Q$).

(d) After each execution of $A$ the value of $t$ decreases ($c \wedge (t(p) = k) \xrightarrow{A} t(p + 1) < k$, where $p$ can be interpreted as a counting variable associated to each loop execution).

(e) If $c$ is true then $t(p) \geq 1$ (at least one iteration remains to be executed) and when $t$ becomes 0 then the condition $c$ becomes false (meaning that the algorithm stops after a finite number of iterations).

The key element in proving the correctness of a loop statement is the loop invariant. This is an assertion concerning the algorithm's state which is true before entering the loop, remains true after each iteration and implies the postconditions at the end of the loop. Finding an invariant is equivalent with proving the correctness by mathematical induction.
*Example 1.* Let us consider the problem of finding the minimum in a finite sequence. We shall analyze the following algorithm:

```
minim($a[1..n]$)
$A_1$ :  $min \leftarrow a[1]$                                  $\{ min = a[1] \}$
$A_2$ :  $i \leftarrow 2$                                       $\{ min = a[1], i = 2 \}$
         while $i \leq n$ do
$A_3$ :      if $min > a[i]$ then $min \leftarrow a[i]$ endif   $\{ min \leq a[j], j = \overline{1,i} \}$
$A_4$ :      $i \leftarrow i + 1$                               $\{ min \leq a[j], j = \overline{1,i-1} \}$
         endwhile
         return $min$                                 $\{ i = n+1, min \leq a[j], j = \overline{1,i-1} \}$
```

The precondition is $n \geq 1$ and the postcondition is $min \leq x[i]$, $i = \overline{1,n}$. The assertions corresponding to each processing step can be easily found. The loop invariant is $min \leq a[j]$, $j = \overline{1,i-1}$. Indeed, after the execution of $A_2$ the property is satisfied. From the assertion specified after $A_4$ it follows that the property remains true after each iteration.

The stopping condition is $i = n+1$ and one can see that if it is satisfied then the final assertion implies the postcondition. The termination function is in this case $t(p) = n + 1 - i_p$ where $i_p$ is the value of index $i$ corresponding to iteration $p$. Since $i_{p+1} = i_p + 1$ it follows that $t(p+1) < t(p)$ and $t(p) = 0$ implies $i = n+1$ which implies the stopping condition.
*Example 2.* We shall analyze the Euclid's algorithm to compute the greatest common divisor of two strictly positive integers:

```
gcd (a,b)
$d \leftarrow a$
$i \leftarrow b$
$r \leftarrow d$ MOD $i$
while $r \neq 0$ do
    $d \leftarrow i$
    $i \leftarrow r$
    $r \leftarrow d$ MOD $i$
endwhile
return $i$
```

The preconditions are: $a, b \in N^*$ and the postcondition is: $i = \gcd(a,b)$. We will consider as loop invariant, $I$, the property $\gcd(a,b) = \gcd(d,i)$ and as termination function $t(p) = r_p$ where $r_p$ is the remainder obtained during the $p$th iteration. To prove that $I$ is indeed a loop invariant we shall prove that if $r \neq 0$ then $\gcd(d,i) = \gcd(i,r)$. Let us denote $d_1 = \gcd(d,i)$ and $d_2 = \gcd(i,r)$. On the other hand $d = i \cdot q + r$. The following implications can be easily verified:

$$d_1|d \text{ and } d_1|i \Rightarrow d_1|d - i \cdot q = r, d_1|i \Rightarrow d_1|i \text{ and } d_1|r \Rightarrow d_1 \leq d_2,$$

$$d_2|i \text{ and } d_2|r \Rightarrow d_2|i \cdot q + r, d_2|r \Rightarrow d_2|d \text{ and } d_2|r \Rightarrow d_2 \leq d_1.$$

It follows that $d_1 = d_2$, thus $I$ satisfies the properties (a) and (b). When the condition $c$ becomes false ($r = 0$) one obtains $\gcd(i,r) = i$ thus the third property of a loop invariant is also satisfied. Since the termination function is determined by the remainder's value and since the sequence of remainders is strictly decreasing it follows that the properties (d) and (e) are also satisfied.
*Remark.* Loop invariants can be used not only to prove the correctness of a loop but also as a design tool. Thus identifying a loop invariant can be seen as a preliminary step before designing

the loop. In this case the invariant serves as a specification for the loop and can be used as a guide to help determine loop initialization, loop body and the stopping condition.
**Example 3.** Computation of the sum, $S$, of the first $n$ natural numbers ($n \geq 1$).

The precondition is $P = \{n \geq 1\}$ and the postcondition is $Q = \{S = 1 + 2 + ... + n\}$. Since the sum will be computed by successively adding the current term, $i$, an adequate loop invariant could be $S = 1 + 2 + \ldots + i$. To assure the validity of the loop invariant we have to prepare the term just before it is added to the sum. On the other hand to ensure that at the end of the loop the postcondition is satisfied it follows that the value of $i$ should be $n$. Thus the continuation condition of the WHILE loop should be $i < n$.

This means that the algorithm could be written as follows:

```
sum (n)
$i \leftarrow 1$
$S \leftarrow 1$
**while** $i < n$ **do**
    $i \leftarrow i + 1$
    $S \leftarrow S + i$
**endwhile**
**return** $S$
```

It is obvious that there exist other correct versions of this algorithm. One of these are:

```
sum (n)
$S \leftarrow 0$
$i \leftarrow 1$
**while** $i \leq n$ **do**
    $S \leftarrow S + i$
    $i \leftarrow i + 1$
**endwhile**
return $S$
```

In this case the loop invariant is $S = 1 + 2 + ... + (i - 1)$. It is easy to see that it is satisfied at the beginning of the loop. After the first statement of the loop body the following assertion holds: $S = 1 + 2 + \ldots + i$ but after incrementing $i$ we obtain again $S = 1 + 2 + \ldots + (i - 1)$. When the stopping condition is satisfied, $i = n + 1$ and the postcondition is satisfied.