VSTTE 2010 software verification competition

Problem 1: Sum and Maximum

I. Description and analysis

Problem:

Given an N-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array.

Properties:

Given that $N \geq 0$ and $a[i] \geq 0$ for $0 \leq i < N$, *prove* the post-condition that $sum \leq N * max$.

Problem analysis:

From this specification, we extract two main functionalities.

FUNC1: Create a program that computes the sum and the maximum of the elements of an array.
FUNC2: Prove the property $sum \leq N * max$.

In order to divide the complexity of FUNC1, the program will be composed of two **independent** operations: the first one will compute the maximum, the other one the sum.

Since we want the two operations to be independent, there is only one option left for the modeling of the program. There will be a first machine composed of the two operations. Then there will be a second machine (composed of one operation) that proves the property after the max and the sum are processed.
We could specify the two independent operations in the LOCAL_OPERATIONS clause of the machine Max_Sum, but if the project is consequent then those operations might be used by another machine. If we don't want to specify, implement and prove those operations again, we separate them in a machine that can be included/imported by other machines.

The project will be made of a main machine composed of three variables: maxi, sum and valid. The *valid* variable is used in order to check if an overflow occurred when the sum of the array is computed. The FUNC2 expected property is proved here.

This machine will include/import a context machine (where the N constant is defined) and another machine (called 'Array') where the array and two operations are defined: one that computes the sum (if valid=TRUE) and one the computes the max.

II. Software modeling

Reminder: for the operations in the abstract machine, we have to data type the input and the output parameters, and specify the conditions (if any) needed to execute the operation in the PRE substitution. The THEN substitution specifies the state of the variables at the exit of the operation.

Acronyms:

- POG: Proof Obligation Generator
- PO: Proof Obligation

A. Machine CTX

```
MACHINE
    CTX

CONCRETE_CONSTANTS
    NN

PROPERTIES
    NN : NAT
END
```

This machine contains the constants of the project and their properties. It is seen by the Array and the Max_Sum machines. The valuation of the constants is made in the implementation (by the clause VALUES) so that the properties are verified at the initialization, by one or more POs.

B. Machine Array

This is the machine containing the two operations that are going to compute the sum and the max of the array.

1. Abstract machine:

Operation ProcessSum:

```
sum, valid <-- ProcessSum =
PRE
   sum : NAT &
   valid : BOOL
THEN
   IF(SIGMA(xx).(xx : 0..NN | tab(xx)) : NAT)
      THEN
         valid := TRUE ||
         sum := SIGMA(xx).(xx : 0..NN | tab(xx))
   ELSE
      valid := FALSE ||
      sum :: NAT
   END
END
```

Operation ProcessMax:

```
maximum <-- ProcessMax =
PRE
   maximum : NAT
THEN
   maximum := max(ran(tab))
END;
```

2. Refinement:

This refinement is an algorithm refinement, i.e. only operations are refined. It specifies the algorithm used in the implementation.

The ProcessSum operation does not need to be refined since we won't add any information.

```
REFINEMENT
   Array_r
REFINES
   Array
SEES
   CTX
OPERATIONS
   maximum <-- ProcessMax =
   BEGIN
      maximum : (maximum : ran(tab) &
      !ii.(ii : 0..NN => tab(ii) <= maximum))
   END
END
```

The substitution 'become such as' specifies that the maximum belongs to the co-domain of the array **and** is greater or equal to all the elements. In fact, this is the extension of the maximum's notation.

```
sum, valid <-- ProcessSum =
ANY sum1 WHERE
   sum1 : NATURAL &
   sum1 = SIGMA(xx).(xx : 0..NN | tab(xx))
THEN
   sum, valid : (
      sum : NAT &
      valid : BOOL &
      (sum1 : NAT =>
          (valid = TRUE & sum = sum1)) &
      (sum1 /: NAT =>
          (valid = FALSE))
   )
END
```

The refinement of this operation is used to link the two variables sum and valid. If the sum is implementable then we have the valid variable set at TRUE and the sum variable is the sum of the array.

3. Implementation:

Loops in implementation have to be proved. The INVARIANT substitution verifies that the specification is respected. It has to be correct:

- Before entering the loop;
- After each iteration in the loop;
- At the end of the loop

The POG generates those three PO. If the invariant is incorrect (respectively incomplete), there will be wrong PO (respectively lack of hypotheses in the PO).

The VARIANT substitution proves the loop's convergence. The expression it contains has to be positive, and has to decrease for each loop's iteration.

a. Operation ProcessSum:

```
sum, valid <-- ProcessSum =
VAR pas, sum_ind, sum_valid IN
   pas := 0;
   sum_ind := tab(pas);
   sum_valid := TRUE;
   WHILE(pas < NN & sum_valid = TRUE) DO
      pas := pas+1;
         VAR delta, temp IN
            delta := MAXINT - sum_ind;
            temp := tab(pas);
            IF(delta < temp) THEN
               sum_valid := FALSE
            END
         END;

         IF(sum_valid = TRUE) THEN
            sum_ind := sum_ind + tab(pas)
         END
   INVARIANT
      pas : 0..NN &
      sum_ind : NAT &
      sum_valid : BOOL &
      (SIGMA(xx).(xx : 0..NN | tab(xx)) : NAT =>
          (sum_ind = SIGMA(xx).(xx : 0..pas | tab(xx)) &
           sum_valid = TRUE)
       ) &
      ((SIGMA(xx).(xx : 0..NN | tab(xx)) /: NAT &
        sum_valid = FALSE) =>
          (sum_valid = FALSE)
       ) &
      ((SIGMA(xx).(xx : 0..NN | tab(xx)) /: NAT &
        sum_valid = TRUE) =>
            sum_ind = SIGMA(xx).(xx: 0..pas |tab(xx))
       )
   VARIANT
      NN-pas+1
   END;
   valid := sum_valid;
   sum := sum_ind
END
```

The ‘sum_ind’ variable corresponds to the sum of the ‘pas+1’ first elements in the array.

The local vairables delta and temp are defined in the loop so they do not have to appear in the invariant of the loop. It does not prevent the compiler to reserve a register for thoses variables.

About the algorithm: before adding the value of the current element to the partial sum, we ‘look’ if there is enough space for it.

Construction of the invariant:

After the loop iterates, if we can add the current element at the index 'pas+1' to the partial sum then it is added, else an overflow is detected.

From this fact, we have three cases:

-The sum is implementable: the sum_ind variable is the sum of the 'pas+1' firsts elements and it is valid.

-The sum is not implementable and the overflow is not yet detected: the sum_ind variable is the sum of the 'pas+1' firsts elements and it is valid.

-The sum is not implementable and the overflow is detected: nothing is valid anymore.

Construction of the variant:

In order to compute the sum of all the elements in the array, we will have to read the NN elements. 'pas' is incremented for every iteration of the loop, so the expression NN-pas decrease for every iteration. 'pas' maximum's value is NN, so the expression NN-index is obviously positive.

b. Operation ProcessMax:

```
maximum <-- ProcessMax =
VAR pas, maxi_loc IN
   pas := 0;
   maxi_loc := tab(pas);
   WHILE (pas < NN) DO
      VAR loc IN
         pas:= pas + 1;
         loc := tab(pas);
         IF(maxi_loc <= loc)
            THEN maxi_loc := loc
         END
      END
   INVARIANT
      pas: 0..NN &
      maxi_loc : ran(tab) &
      !ind.(ind : 0.. pas => tab(ind) <= maxi_loc)
   VARIANT
      NN- pas
   END;
   maximum := maxi_loc
END
```

The 'loc' variable is declared **in** the loop so that it does not have to appear in the INVARIANT substitution. This declaration does not prevent the compiler to allow a register for this variable.

Construction of the invariant:

It is obtained by replacing the NN constant in the predicate of the refinement by the index of the current element read (pas). The translation of this invariant is: "Until we are in the loop, the maximum we have is a local maximum (i.e. the maximum of all the elements read).".

Construction of the variant:

This is the same variant than the ProcessSum's operation.

C. Machine Main

In this machine the FUNC2 functionality is proven.
The abstract machine is constructed according how we want to use it. If this machine is imported by another one the variables may be defined in the abstraction, else we can defined them in the refinement/implementation (and they won't be exported).
In our case we define the abstract machine with no variables, and only one operation that is defined as 'skip'.

The implementation of this machines gives the following:

```
IMPLEMENTATION
   main_i
REFINES
   Main

IMPORTS
   Array, Ctx

CONCRETE_VARIABLES
   maxi, somme, valid

INVARIANT
   maxi: NAT &
   somme: NAT &
   valid: BOOL &
   (valid = TRUE => somme <= card(dom(tab)) * maxi)

INITIALISATION
   maxi := 0;
   somme := 0;
   valid := FALSE

OPERATIONS
   Main =
   BEGIN
      maxi <-- ProcessMax;
      somme, valid <-- ProcessSum
   END
END
```

The property expected is defined in the invariant so it has to be proved at the initialization and it has to be respected by the operations. We want the property to be verified when the sum is implementable.

III. Proof obligations

A total of 79 POs are generated for the machine (abstract machine, refinement and implementation).
63 POs are automatically discharged by the force 0 prover (and 2 are discharged by the force 1).

5 POs are discharged by the predicate prover (use the command **te(pp(rp.1|10) & sw, Replace.Gen.Unproved)** or prove one using the command **p1**, then save it (**sw**) and a chose the option Try everywhere globally after a right click on the PO).

There are 9 POs left:

- 6 POs for the implementation of the machine Array.
- 3 POs for the refinement of the machine.

0 rules are added

A. Machine Array's functional proof obligations

Array's refinement demonstration:

```
"`Local hypotheses'" &
sum1: INTEGER &
0<=sum1 &
sum1 = SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
sum$2: INTEGER &
0<=sum$2 &
sum$2<=2147483647 &
valid$2: BOOL &
sum1: INTEGER & 0<=sum1 & sum1<=2147483647 => valid$2 = TRUE & sum$2 = sum1 &  ①
not(sum1: INTEGER & 0<=sum1 & sum1<=2147483647) => valid$2 = FALSE &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)) &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))<=2147483647 &
"`Check that the invariant (sum$1 = sum) is preserved by the operation - ref 4.4, 5.5'"
=>
sum$2 = SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))
```

The demonstration of this PO is obvious: the sum is implementable.

Add all the hypotheses to the stack of hypotheses using the command **dd**.

Add then the following hypothesis:

sum1: INTEGER & 0<=sum1 & sum1<=2147483647

Before it is added to the stack of hypotheses it has to be proved. Call the proved twice in order to discharge the two firsts assertions. The third one is discharged be replacing sum1 by its expression, and by then replacing the 0..NN by dom(tab$1): to do so use the command **eh** (use equality) on the goal. Add the hypothesis to the stack of hypothesis (**dd**).

You can now apply a modus ponens on the hypothesis ① using the command **mh** (if not accepted, try adding the full hypothesis to the stack of hypotheses and it will be done automatically).

Call the prover for the proof obligation to be discharged.

This demonstration can be used for another PO, except that the PO is discharged using the pp(rt.1) command.

The tree of commands for this PO is:

```
dd
    ah(sum1: INTEGER & 0<=sum1 & sum1<=2147483647)
        pr
        pr
        eh(sum1,_h,Goal)
            eh(0..NN,_h,Goal)
                ah(①)
        dd
            pr/pp(rt.1)
```

```
"`Local hypotheses"' &
sum1: INTEGER &
0<=sum1 &
sum1 = SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
sum$2: INTEGER &
0<=sum$2 &
sum$2<=2147483647 &
valid$2: BOOL &
sum1: INTEGER & 0<=sum1 & sum1<=2147483647 => valid$2 = TRUE & sum$2 = sum1 &
not(sum1: INTEGER & 0<=sum1 & sum1<=2147483647) => valid$2 = FALSE &                    ①
not(SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: dom(tab$1) |
tab$1(xx)) & SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))<=2147483647) &                      ②
"`Check that the invariant (valid$1 = valid) is preserved by the operation - ref 4.4, 5.5"'
=>
valid$2 = FALSE
```

The demonstration of this PO is very similar to the previous one.

When the hypotheses are in the stack of hypotheses (**dd**), add the precondition of the ① hypothesis as a hypothesis. As usual, before it is added to the stack of hypotheses, it has to be proved: Add the hypothesis ② and replace the 'dom(tab$1)' expression by '0..NN' (command **eh**) and then replace the expression 'SIGMA(xx).(xx: 0..NN | tab$1(xx))' by sum1, still using the command **eh**: the PO is automatically discharged.

The tree of command for this PO is:

```
dd
    ah(_①)
        ah(②)
            eh(dom(tab$1),_h,Goal)
                eh(SIGMA(xx).(xx: 0..NN | tab$1(xx)),_h,Goal)
```

With _① the left side of the ① hypothesis.

Array's implementation demonstration:

```
…
"`Local hypotheses"' &
pas: 0..NN &
sum_ind: INTEGER &
0<=sum_ind &
sum_ind<=2147483647 &
sum_valid: BOOL &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 => sum_ind = SIGMA(xx).(xx: 0..pas |
tab$1(xx)) & sum_valid = TRUE &                                                    ①
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN |
tab$1(xx)) & SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid = FALSE =>
sum_valid = FALSE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN |
tab$1(xx)) & SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid = TRUE =>
sum_ind = SIGMA(xx).(xx: 0..pas | tab$1(xx)) &                                     ②
pas+1<=NN &
sum_valid = TRUE &
not(2147483647-sum_ind+1<=tab$1(pas+1)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
sum_ind+tab$1(pas+1) = SIGMA(xx).(xx: 0..pas+1 | tab$1(xx))
```

The first step of the demonstration is to find the value of sum_ind. To do so, apply a modus ponens on the hypothesis ① right after the all the hypotheses are added to the stack of hypotheses.
The goal becomes:

SIGMA(xx).(xx: 0..pas | tab$1(xx))+tab$1(pas+1) = SIGMA(xx).(xx: 0..pas+1 | tab$1(xx))

The only way to prove this goal without adding any rules is to use one in the set of rules. In order to search for a rule, use the command **sr**: for instance, in this case, you can use the command

sr(All, SIGMA(x).(E|F)) that will give all the rules that use the SIGMA operator.

A rule that can be used is the rule s1.3: we can divide the SIGMA(xx).(xx: 0..pas+1 |tab$1(xx)) into SIGMA(xx).(xx: 0..pas |tab$1(xx))+ SIGMA(xx).(xx=pas+1 |tab$1(xx)).
First of all, add the following hypothesis:

0..pas+1= 0..pas\/{pas+1}

and prove it using the predicate prover (and do not forget to add it to the stack of hypotheses using the command **dd**). You may now replace 0..pas+1 by this new expression (**eh**).
A precondition to use this rule is that 0..pas/\{pas+1} = {}. Again, add this as an hypothesis and call the prover to prove it. Add it in the stack of hypotheses.

We have now all the preconditions for the rule to be applied: use the command **ar(s1.3, Goal)**.
Call the prover for the last time: the PO is discharged.

This demonstration is applicable to the following PO.

The tree of commands for those PO is:

```
dd
    mh(①)
        dd
            ah(0..pas+1 = 0..pas\/{pas+1})
                pp(rt.1)
                dd
                    eh(0..pas+1,_h,Goal)
                        ah(0..pas/\{pas+1} = {})
                            pr
                            dd
                                ar(s1.3,Goal)
                                    pr
```

```
"`Local hypotheses"' &
pas: 0..NN &
sum_ind: INTEGER &
0<=sum_ind &
sum_ind<=2147483647 &
sum_valid: BOOL &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 => sum_ind = SIGMA(xx).(xx: 0..pas |
tab$1(xx)) & sum_valid = TRUE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx))
& SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid = FALSE => sum_valid =
FALSE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx))
& SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid = TRUE => sum_ind =
SIGMA(xx).(xx: 0..pas | tab$1(xx)) &
pas+1<=NN &
sum_valid = TRUE &
2147483647-sum_ind+1<=tab$1(pas+1) &                                    ①
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
not(SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647)
```

The demonstration of this PO is done by contradiction: indeed, a contradiction consists of proving *bfalse* with the negated initial goal in the stack of hypotheses. Here, the negated goal means that the sum is implementable; however the hypothesis ① means that we cannot add the current element because the sum won't be implementable anymore.

Begin the demonstration by adding all the hypotheses in the stack of hypotheses.
Initialize the demonstration by contradiction using the command **ct**.

Add the previous negated goal so the goal becomes:

SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 => bfalse

Our goal is to find the sigma expression of the hypothesis ① in the current goal: we will divide the sigma expression in the goal into two sigma expressions: the first defined on 0..pas+1 and the other one on pas+1+1..NN. Note that even if pas+1+1 is over NN (i.e. pas+1 = NN) this is correct since SIGMA(x).(bfalse|p) is equal to 0.

Add the hypothesis 0..NN = 0..pas+1\/pas+1+1..NN and use the predicate prover in order to prove this hypothesis. Replace the '0..NN' expression in the by this new expression.
Prove then that 0..pas+1/\pas+1+1..NN = {} using the predicate prover too (do not forget to add this hypothesis in the stack of hypotheses): You can now apply the rule s1.3.

The current goal becomes:

SIGMA(xx).(xx: 0..pas+1 | tab$1(xx))+SIGMA(xx).(xx: pas+1+1..NN | tab$1(xx))<=2147483647 => bfalse

Using the same demonstration as the previous POs, prove that

2147483648 <= SIGMA(xx).(xx: 0..pas+1 | tab$1(xx))

The PO cannot be discharged until we have not proved that

0 <= SIGMA(xx).(xx: pas+1+1..NN | tab$1(xx)).

Search for a rule to be applied: the b1.47 seems nice, but we have to differentiate two cases here for the precondition that (a <= b) to be true:

- the case where pas+1 = NN and the precondition is not true (but we thus already have the contradiction;
- the case where not(pas+1 = NN) i.e. pas+1+1 <= NN.

Start the demonstration by cases using the **dc**.

The case where pas+1 = NN gives us immediately the contradiction: replace the 'pas+1' expression by NN in the SIGMA expression.

The case where not(pas+1 = NN): the hypothesis not(pas+1 = NN) is enough to prove one of the precondition of the rule b1.47:

pas+1+1 <= NN.

The other preconditions needed are:

- not(NAT = {}): the prover proves it with ease;
- NAT: FIN(NAT): Use the prover to prove the hypothesis NAT: FIN(INTEGER) and then apply the rule InFINXY.117 (another call to the prover is needed to prove this hypothesis);
- 0<=(NN-(pas+1+1)+1)*min(NAT): add the hypothesis that min(NAT) = min(NATURAL), prove it using the rule EqualityXY.61. Replace min(NAT) by min(NATURAL) and call the prover on the goal.

Once the rule is apply and this property proved, a call to the MiniProver or the prover will discharge the PO.

The tree of command for this PO is too huge to write it here, but you are invited to replay the demonstration (with the interactive prover) recorded in the archive file.

The demonstration of the next PO is included in the previous PO's demonstration: the rule b1.47 is applied.

```
"`Local hypotheses'" &
not(pas$7777+1<=NN & sum_valid$7777 = TRUE) &
pas$7777: 0..NN &
sum_ind$7777: INTEGER &
0<=sum_ind$7777 &
sum_ind$7777<=2147483647 &
sum_valid$7777: BOOL &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 => sum_ind$7777 = SIGMA(xx).(xx:
0..pas$7777 | tab$1(xx)) & sum_valid$7777 = TRUE &                                  ①
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid$7777 = FALSE =>
sum_valid$7777 = FALSE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid$7777 = TRUE =>
sum_ind$7777 = SIGMA(xx).(xx: 0..pas$7777 | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 &
"`Check operation refinement - ref 4.4, 5.5'"
=>
sum_ind$7777 = SIGMA(xx).(xx: 0..NN | tab$1(xx))
```

The prefix $7777 on a variables means the value of concerned the variable at the exit of the loop. Since the sum is implementable, we know that the value of pas at the exit of the loop is NN.

After all the hypotheses are added to the stack of hypotheses, apply a modus ponens on the hypothesis ①. Now that we have sum_valid$7777=TRUE, add as hypothesis that pas$7777=NN.
Before it is added to the stack of hypotheses, it has to be proved:
Add the hypothesis 'not(pas$7777+1<=NN & sum_valid$7777 = TRUE)', replace the variable sum_valid$7777 by its value (**eh**) and also add the hypothesis 'pas$7777: 0..NN'. A call to the predicate prover will prove this hypothesis and discharge the PO.

The tree of commands for this PO is:

```
dd
    mh(①)
        dd
            ah(pas$7777 = NN)
                ah(not(pas$7777+1<=NN & sum_valid$7777 = TRUE))
                    eh(sum_valid$7777,_h,Goal)
                        ah(pas$7777: 0..NN)
                            pp(rt.0)
```

```
"`Local hypotheses'" &
not(pas$7777+1<=NN & sum_valid$7777 = TRUE) &                                    ①
pas$7777: 0..NN &
sum_ind$7777: INTEGER &
0<=sum_ind$7777 &
sum_ind$7777<=2147483647 &
sum_valid$7777: BOOL &
SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647 => sum_ind$7777 = SIGMA(xx).(xx:
0..pas$7777 | tab$1(xx)) & sum_valid$7777 = TRUE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid$7777 = FALSE =>
sum_valid$7777 = FALSE &
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) & sum_valid$7777 = TRUE =>
sum_ind$7777 = SIGMA(xx).(xx: 0..pas$7777 | tab$1(xx)) &                         ②
not(SIGMA(xx).(xx: 0..NN | tab$1(xx)): INTEGER & 0<=SIGMA(xx).(xx: 0..NN | tab$1(xx)) &
SIGMA(xx).(xx: 0..NN | tab$1(xx))<=2147483647) &                                 ③
"`Check operation refinement - ref 4.4, 5.5'"
=>
sum_valid$7777 = FALSE
```

This PO is easily discharged by contradiction: indeed, reasoning by contradiction will give us

sum_valid$7777=TRUE

Note that the contradiction is visible in the hypotheses since we cannot have

- sum_ind$7777<=2147483647;
- sum_valid$7777=FALSE.

Moreover, have sum_valid$7777=TRUE will gives us pas$7777=TRUE and the sigma expression of sum_ind$7777 (thanks to the hypotheses ① and ②).

As usual, add all the hypotheses to the stack of hypotheses (**dd**).

Start the demonstration by contradiction and add then prove the following hypotheses:

- pas$7777= NN;
- sum_ind$7777= SIGMA(xx).(xx: 0..pas$7777 | tab$1(xx));

the same way they are demonstrated in the previous POs.

Add the hypothesis ③ so the goal becomes:

③ => bfalse

Replace NN by pas$7777 in the current goal (command **eh**) and then replace 'SIGMA(xx).(xx: 0..pas$7777 | tab$1(xx))' by sum_ind$7777: the goal should be the same as:

not(sum_ind$7777: INTEGER & 0<=sum_ind$7777 & sum_ind$7777<=2147483647) => bfalse

Add the hypotheses concerning sum_ind$7777: 'sum_ind$7777: INTEGER', '0<=sum_ind$7777' and 'sum_ind$7777<=2147483647'. Call the predicate prover for the PO to be discharged.

B. Machine Main's functional proof obligations

Three PO have to be proved in this machine.

```
"`Local hypotheses'" &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)) &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))<=2147483647 &
"`Check that the invariant (maxi$1: NAT) is preserved by the operation - ref 4.4, 5.5'"
=>
0<=max(ran(tab$1))
```

After all the hypotheses are added to the stack of hypotheses, call the prover in order to simplify the goal:

not(ran(tab$1)/\NATURAL = {}).

Since tab$1: 0..NN --> NAT, this is obvious. Add this hypothesis and call the prover so it is proved.
Add the hypothesis in the stack of hypotheses and call the predicate prover: the PO is discharged.

The three of commands for this PO is:

```
dd(0)
    ah(tab$1: 0..NN --> NAT)
        pr
        pp(rt.1)
```

This demonstration also discharges another PO.

```
"`Local hypotheses"' &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)): INTEGER &
0<=SIGMA(xx).(xx: dom(tab$1) | tab$1(xx)) &
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))<=2147483647 &
"`Check that the invariant (valid$1 = TRUE => somme$1<=card(dom(tab$1))*maxi$1) is
preserved by the operation - ref 4.4, 5.5"'
=>
SIGMA(xx).(xx: dom(tab$1) | tab$1(xx))<=card(dom(tab$1))*max(ran(tab$1))
```

The demonstration of this PO needs a rule to be applied.
Search in the set of rules for a rule that corresponds to the goal: the b1.48 one looks good.
A problem exists though: our array is defined on 0..NN --> NAT, so the right side of the inequation will be card(dom(tab$1)) * max(NAT). The trick is to redefine at the beginning of the demonstration the array as:

tab$1: 0..NN --> ran(tab$1).

Start the demonstration adding all the hypotheses to the stack of hypotheses (**dd**).
We first prove the preconditions of rule with tab$1: 0..NNN --> ran(tab$1):

- 0 <= NN
- not(ran(tab$1) = {}): since tab$1 is a total function its range cannot be empty. The predicate prover proves it with ease.
- ran(tab$1): FIN(ran(tab$1)): this precondition is proved using the rules InFINXY.72 and InFINXY.117.
- (NN-0+1)*max(ran(tab$1))<=card(dom(tab$1))*max(ran(tab$1)): the MiniProver discharges this with ease.

Once those preconditions are proved, apply the rule and the goal is discharged.

B. Well-definedness proof obligations

20 POs are generated for the well-definedness of the components.
16 are automatically discharged by the force 0 prover, and 2 by the force 1 prover.

The two left POs are similar and have the same demonstration.

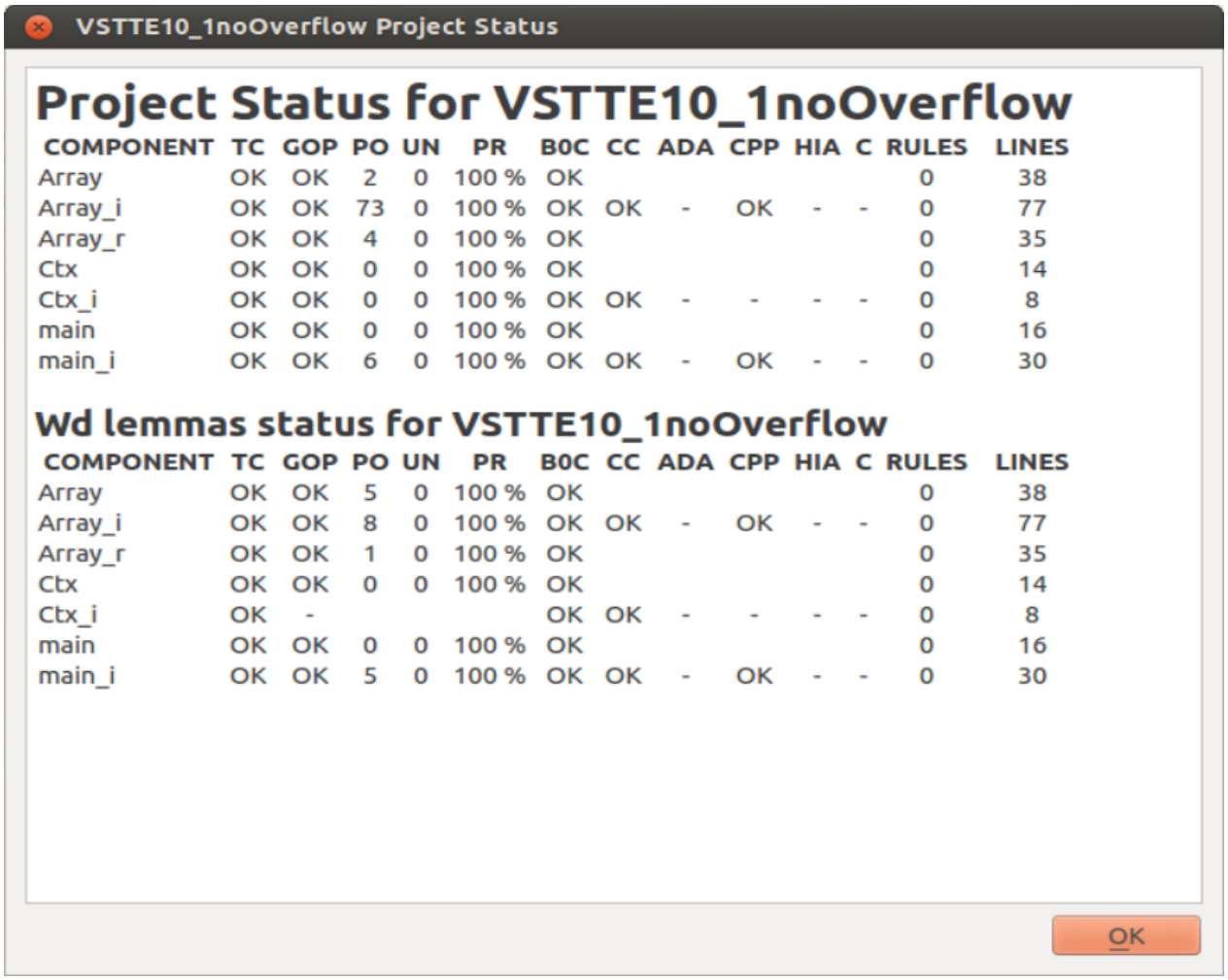
```
btrue
=>
ran(tab)/\NATURAL: FIN(NATURAL)
```

Add all the hypotheses in the stack of hypotheses then simplify the 'ran(tab)/\NATURAL' by 'ran(tab)': Add as hypothesis that 'ran(tab)/\NATURAL = ran(tab)' and simplify it using the miniprover so we will have to show that ran(tab) <: NATURAL. The predicate prover discharges this sub-goal when you show the 'ran(tab) <: NAT'. Replace the value in the goal using the eh command for the goal to become:

ran(tab): FIN(NATURAL)

A rule that seams perfect for this goal is the InFINXY.72 one. A precondition to prove before applying this rule is that 0..NN: FIN(D), with D a set; and we know that dom(tab): FIN(INTEGER). Once proved, apply the rule for the PO to be discharged.

C. Project status



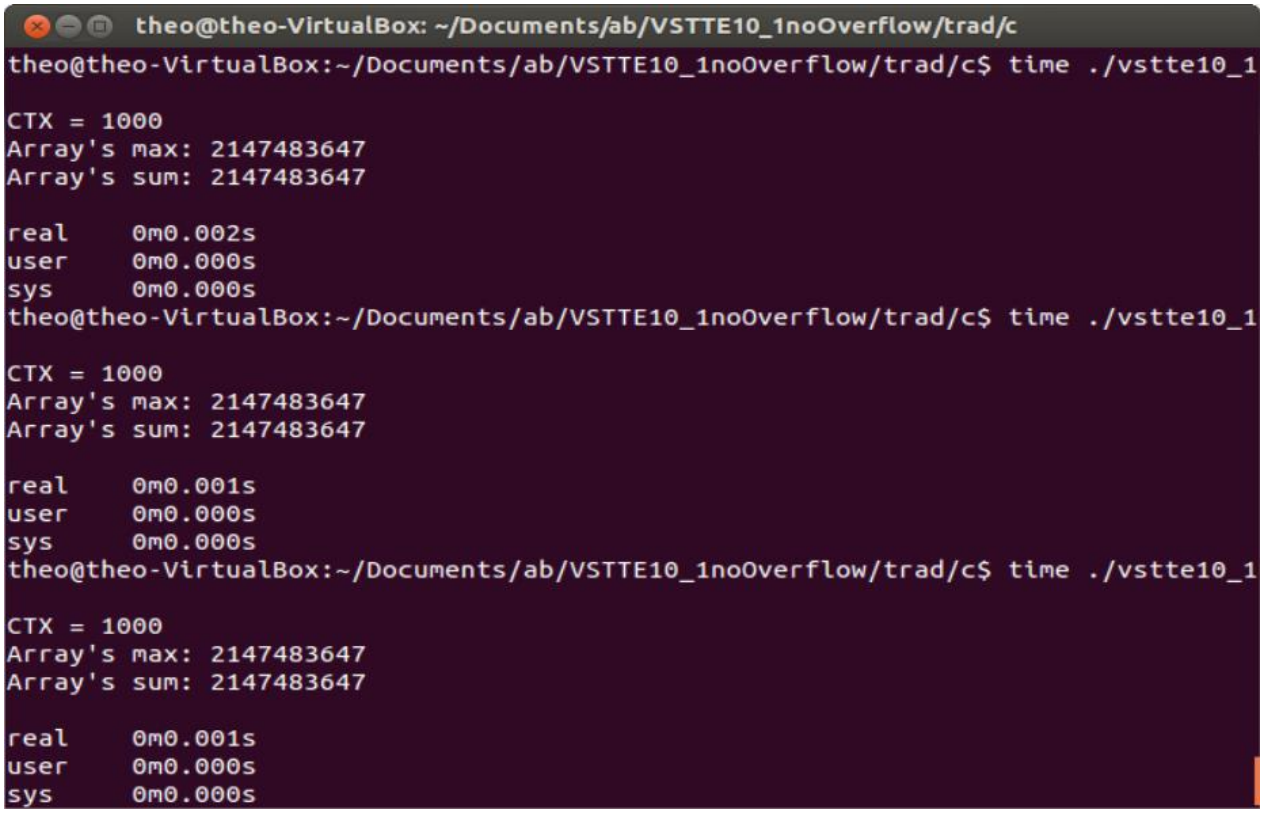
COMPONENT: the components (machines) of the project.
TC: Type Check.
GOP: Générateur d'Obligations de Preuve (the POG).
PO : Proof Obligation.
UN : The number of unproved PO in the component

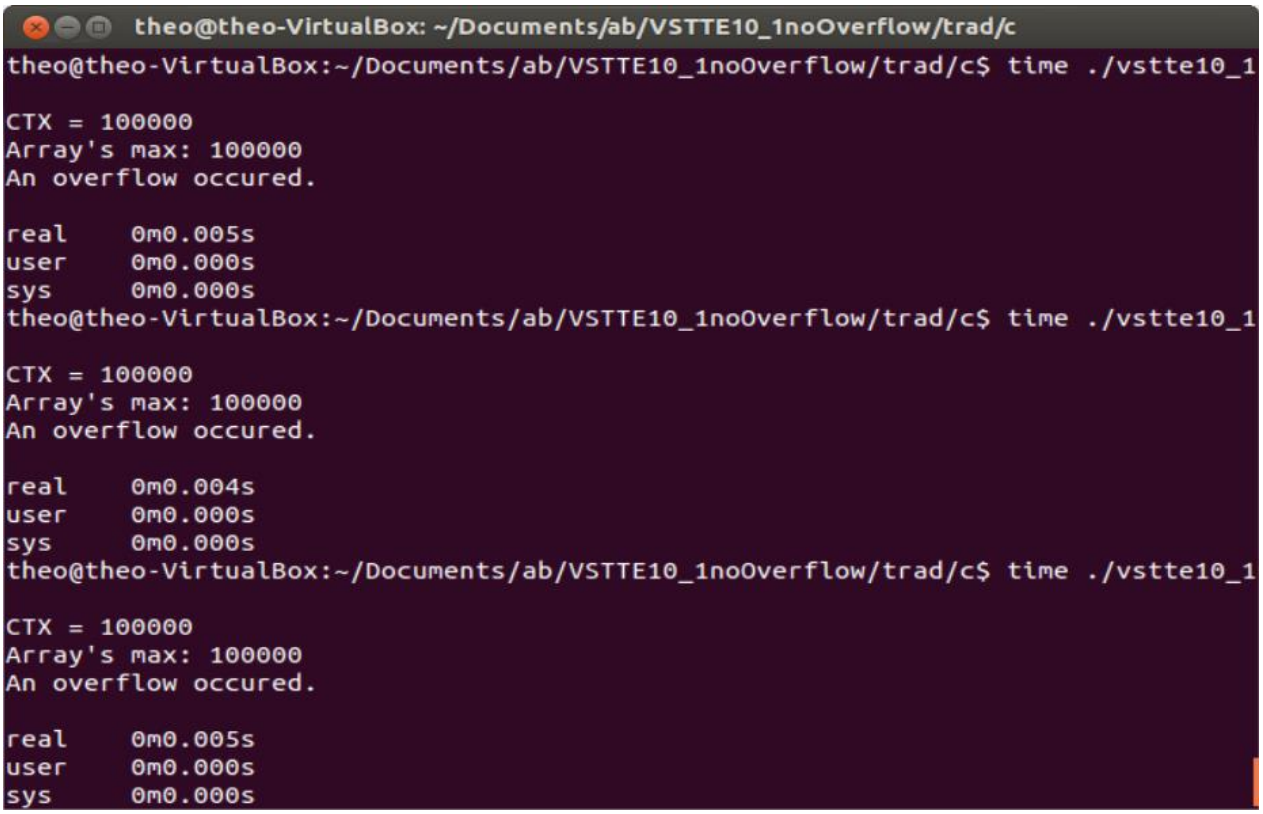
PR: The rate of proved PO of the component.
B0C: B0 Check.
CC:
LINES: The number of lines in the component.
RULES: The number of added rules for the component.

IV. Generated Code

Now that all the POs are discharged, call the B0 checker on the components.
Once all the elements are b0 checked, go to the Project tab and click on Code generator .

Once the code is generated compile it (some outputs are added, and the initialisation of the array is changed for tests).




theo@theo-VirtualBox: ~/Documents/ab/VSTTE10_1noOverflow/trad/c
theo@theo-VirtualBox:~/Documents/ab/VSTTE10_1noOverflow/trad/c$ time ./vstte10_1
CTX = 100000
Array's max: 100000
An overflow occured.
real 0m0.005s
user 0m0.000s
sys 0m0.000s
theo@theo-VirtualBox:~/Documents/ab/VSTTE10_1noOverflow/trad/c$ time ./vstte10_1
CTX = 100000
Array's max: 100000
An overflow occured.
real 0m0.004s
user 0m0.000s
sys 0m0.000s
theo@theo-VirtualBox:~/Documents/ab/VSTTE10_1noOverflow/trad/c$ time ./vstte10_1
CTX = 100000
Array's max: 100000
An overflow occured.
real 0m0.005s
user 0m0.000s
sys 0m0.000s