VSTTE 2012 software verification competition

Problem 1: Two-Way Sort

I. Description and analysis

Problem:

We want to sort an array of Boolean values (assuming false < true) using only swaps.

Properties:

Safety: Verify that every array access is made within bounds.

Termination: Prove that function two_way_sort always terminates.

Behavior: a. Array a is sorted in increasing order (false<true).

b. Array a is a permutation of its initial contents.

Problem analysis:

Some of the properties expected are proved automatically: the safety and the termination. Proof obligations are automatically generated. The role of the engineer for the B-models is to mathematically express and prove the *'Behavior'* properties.

Thus, two functionalities are expected for this problem:

FUNC1: The array obtained has to be sorted in increasing order.

FUNC2: The array obtained is a permutation of its initial contents.

There are multiple ways to model those two properties, but the complexity and the number of POs will depend on the one chosen. The fact that the array is an array of Booleans may reduce the complexity of modeling the FUNC2 property: indeed, reasoning on the cardinality of the range of the array is easy. This way, the FUNC2 property is modeled the following:

card(array~[{FALSE}]) = card(array$0~[{FALSE}])

with array$0 the state of the array before the operation is called. 'array~' represents the reversed array BOOL --> INT (see the B Language Reference Manual for more details), and the expression array~[{FALSE}] gives all the relations similar to FALSE --> x (with x in 0..n). Taking the cardinality of this expression gives us the number of FALSE elements in the array.
Note that card(array~[{TRUE}]) = card(array$0~[{TRUE}]) is obtained since the array is a *total* function (i.e. all the elements have an image): from card(array~[{FALSE}]) we can get card(array~[{TRUE}]).

The FUNC1 property can be expressed as "there is an *index* such that for all the previous indexes the elements are false, and for all the following they are true". This property is not correct yet: what if all the elements of the array are true (or false) ? Such an 'index' is no longer in the domain of the array!
The definition of this 'index' has to be reconsidered. If all the elements are false (resp. true) this 'index' becomes index: dom(array)\/{-1} (resp. index: dom(array)\/{n+1} with n the last index of the array).
So, this property can now mathematically be written as:
For an array defined 'array: 0..n --> BOOL'

#i.(i: -1..n & !j.(j: 0..i => array(j)=FALSE) & !j.(j: i+1..n => array(j)=TRUE))

with i the number, minus one, of *FALSE* elements in the array;

or #i.(i: 0..n+1 & !j.(j: 0..i-1 => array(j)=FALSE) & !j.(j: i..n => array(j)=TRUE))

with i the number, plus one, of TRUE elements in the array.

The machine will can be refined or defined with a mix of the FUNC1 and the FUNC2 properties:

!k.(k: 0..card(tab$0~[{FALSE}])-1 => tab(k)=FALSE) &
!l.(l: card(tab$0~[{FALSE}])..NN => tabll)=TRUE)

This property will be defined in the abstraction of the operation (in the abstract machine), but it will have to be proved in the implementation: the invariant of the loop will have to be constructed.
The variant of the loop is used to prove the *Termination* of the loop.

Architecture:

We will build an abstract machine where the operation is defined. The operation has no parameter (the array is defined as variable of the machine and n as a constant).
You can define the NN constant in a context machine (i.e. a machine that contains only constants). This choice depends on the developer. The difference between both using a context machine and defining the constants in the machine is that in the POs of the implementation the constants are replaced by their values. If you modify the value of the constant after the project is proved, the demonstration of the POs that use the constants' values will not be valid.

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. Variables and constants of the machine.

The machine is made of one variable (the array), one constant (the size of the array) and one operation. The constant NN is defined on 2..MAXINT-1. Indeed, if the array's elements are all FALSE and if we have MAXINT elements then while iterating the local variable in the implementation will iterate to MAXINT+1: that will cause an overflow.
The array (named tab in the following) is defined on 0..NN --> BOOL. In the invariant you can specify that card(tab~[{FALSE}])+ card(tab~[{TRUE}]) = NN+1 (the number of TRUE elements and FALSE elements gives the number of elements in the array).

B. Machine TwoWaySort

1. Abstract machine:

This is the machine where the operation is defined.

```
Sort =
PRE
   tab: 0..NN --> BOOL &
   NN >= 2
THEN
   tab:
    (tab: 0..NN --> BOOL &
     !kk.(kk: 0..card(tab$0~[{FALSE}])-1
            => tab(kk)=FALSE) &
     !ll.(ll: card(tab$0~[{FALSE}])..NN
            => tab(ll)=TRUE)
    )
END
```

The PRE clause is used to define the preconditions when calling this operation. The THEN clause is used to specify the modification of the variables after the operation exits.

There are no needs to sort an array of 0 or 1 elements, so the precondition NN≥2 is added.

2. Implementation:

In this machine the variable has to be concretized, and the constant has to be valuated.

```
Sort =
VAR ii, jj, auxi, auxj IN
   ii := 0;
   jj := NN;
   auxi := TRUE;
   auxj := TRUE;

   WHILE(ii <= jj) DO
      auxi := tab(ii);
      auxj := tab(jj);
      IF(auxi=FALSE) THEN
         ii := ii+1
      ELSIF(auxj=TRUE) THEN
         jj := jj-1
      ELSE
         tab(ii) := auxj;
         tab(jj) := auxi;
         ii := ii+1;
         jj := jj-1
      END
   INVARIANT
      tab: 0..NN--> BOOL &
      tab$0: 0..NN--> BOOL &
      ii: 0..card(tab$0~[{FALSE}]) &
      jj: card(tab$0~[{FALSE}])-1..NN &
      auxi: BOOL &
      auxj: BOOL &

      card(tab$0~[{FALSE}]) = card(tab~[{FALSE}]) &
      card(tab$0~[{TRUE}]) = card(tab~[{TRUE}]) &

      !kk.(kk: 0..ii-1 => tab(kk)=FALSE) &
      !ll.(ll: jj+1..NN => tab(ll)=TRUE)
   VARIANT
      2*NN-ii+jj
   END
END
```

Brief explanation of the implementation:

The invariant has to be true at three steps of the loop:

- The beginning
- After every iterations
- At the end

For the invariant to be verified at the beginning of the loop, the variables have to be initialized. There are two ways doing it:

- Doing the first iteration and 'adapting' the indexes of the loop;
- Initialize the variables manually so they verify the invariant.

In our case the variables auxi and auxj are initialized manually because once we are in the loop they are instantly modified.

Construction of the invariant:

Whether i is incremented, j is decremented or both; the property

```
!kk.(kk: 0..ii-1 => tab(kk)=FALSE) &
  !ll.(ll: jj+1..NN => tab(ll)=TRUE)
```

is always verified.

Indeed, we can explain it for the three cases:

- auxi=FALSE: i is incremented, and j is not modified
  - Since i is incremented, its previous value was i-1 thus auxi = tab(i-1) = FALSE. We know nothing about tab(i);
  - j is not modified, we know nothing about tab(j).

The property is verified for this case.

- auxi=TRUE and auxj=TRUE: i is not modified, and j is decremented
  - i is not modified, but we know that tab(i)=TRUE;
  - j is decremented, the previous j'value was j+1, and auxj = tab(j+1) = TRUE. We know nothing about tab(j).

The property is verified for this case.

- auxi=TRUE and auxj=FALSE: i is incremented and j is decremented and the tab$0(i-1) and tab$0(j+1) elements are swaped
  - Since i is incremented, its previous value was i-1 thus auxi = tab(i-1) = tab$0(j+1) = FALSE. We know nothing about tab(i);
  - j is decremented, the previous j'value was j+1, and auxj = tab(j+1) = tab$0(i-1) = TRUE. We know nothing about tab(j).

The property is verified for all the cases.

Note that the property

```
card(tab$0~[{FALSE}]) = card(tab~[{FALSE}]) &
  card(tab$0~[{TRUE}]) = card(tab~[{TRUE}])
```

proves that while iterating the loop conserves the array's elements.

Construction of the variant:

The expression in the variant has to be positive and to decrease when the loop iterates. Since j or i may not change while iterating, we know that at least one is incremented or decremented. Therefore both will figure in the variant clause. Moreover, they won't be 2N iterations but for the positivity of the expression and for the proof of the associated POs, we use the maximal values of those two values (it is easier to prove this than to prove that there are not most of N iterations).

III. Proof obligations

2 POs are generated for the TwoWay machine and 64 for the TwoWay_i machine.
A total of 51 POs are discharged by the force 0 prover (76%). The predicate prover discharges 4 POs.
11 POs remain to prove:

- 2 for the abstract machine: they are related to the invariant added.
- 9 for the implementation of the machine.

4 rules are added for the POs to be proved. Two of them are automatically proved by the rules validator, the two others are proved manually.

A. Proof of the abstract machine

The two POs generated for this machine have the same goal:

card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}]) = NN+1

After all the hypotheses are added to the stack of hypotheses (**dd(0)**), add the following hypothesis:

card(tab$1~[{FALSE}]\/tab$1~[{TRUE}]) = card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}])

The interactive prover needs to prove it before adding it to the stack of hypotheses. Call the prover so the current goal becomes:

tab$1~[{FALSE}]/\tab$1~[{TRUE}] = {}

The predicate prover (**pp(rt.1)**) proves it with ease. Add now the hypothesis to the stack of hypotheses using the command **dd**.
Replace “card(tab$1~[{FALSE}])+card(tab$1~[{TRUE}])” in the current goal by the expression proved before, to do so use the command **eh**.
The current goal should be:

card(tab$1~[{FALSE}]\/tab$1~[{TRUE}]) = NN+1

We do know that

tab$1~[{FALSE}]\/tab$1~[{TRUE}] = 0..NN

because tab is a total function from 0..NN to BOOL. Add this expression as a hypothesis and call the predicate prover in order to prove it. Once proved, add it to the stack of hypotheses before using the equality on goal (**eh**).
We now have to demonstrate that card(0..NN) = NN+1; only possible if 0≤NN. Add this as an hypothesis, and call the prover twice: the first to prove the hypothesis, the second time to discharge the PO.

The demonstration of the second PO is identical to this one.

The tree of commands for this PO is the following:

```
dd(0)
    ah(card(tab$1~[{FALSE}]\/tab$1~[{TRUE}]) =
                            card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}]))
        pr
            pp(rt.1)
        dd
            eh(card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}]), _h, Goal)
                ah(tab$1~[{FALSE}]\/tab$1~[{TRUE}] = 0..NN)
                    pp(rt.1)
                    dd
                        eh(tab$1~[{FALSE}]\/tab$1~[{TRUE}], _h, Goal)
                            ah(0<=NN)
                                pr
                                pr
```

B. Proof of the implementation machine

The first PO of this component is:

```
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
NN: card(tab~[{FALSE}])-1..NN
```

This PO is demonstrated using the invariant added in the abstract machine.

Add all the hypotheses to the stack of hypotheses before calling the prover. The current goal is:

card(tab$1~[{FALSE}]) <= NN+1

The invariant added is useful for this PO because the cardinal of a set is always positive or null, thus card(tab$1~[{TRUE}])≥0.

The current goal is proved using the invariant "card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}]) = NN+1" and the property of cardinality.

Add the invariant as an hypothesis (automatically proved, this is the invariant) then add the hypothesis that 0≤card(tab$1~[{TRUE}]): a call of the prover prove it. Call the predicate prover on the current goal: the PO is discharged.

The tree of commands for this PO is the following:

```
dd(0)
    pr
        ah(card(tab$1~[{FALSE}]) + card(tab$1~[{TRUE}]) = NN+1)
            ah(0≤card(tab$1~[{TRUE}]))
                pr
                pp(rt.0)
```

```
"`Local hypotheses"' &
tab$2: 0..NN +-> BOOL &
dom(tab$2) = 0..NN &
ii$0: 0..card(tab$1~[{FALSE}]) &
jj$0: card(tab$1~[{FALSE}])-1..NN &
auxi$0: BOOL &
auxj$0: BOOL &
card(tab$1~[{FALSE}]) = card(tab$2~[{FALSE}]) &
card(tab$1~[{TRUE}]) = card(tab$2~[{TRUE}]) &
!kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE) &
!ll.(ll: jj$0+1..NN => tab$2(ll) = TRUE) &
ii$0<=jj$0 &
tab$2(ii$0) = FALSE &
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
ii$0+1: 0..card(tab$1~[{FALSE}])
```

There is another PO with the same goal: the demonstration is very similar to this one.
The demonstration of this PO is done by contradiction: negating the initial goal oblige us to find a contradiction in the hypotheses with ii$0 = card(tab$1~[{FALSE}]). The contradiction concerns the number of FALSE/TRUE elements.

Indeed, if ii$0 = card(tab$1~[{FALSE}]), using the hypothesis '!kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE' we get

card(tab$2~[{FALSE}]) >= card(tab$1~[{FALSE}])

(This is an inequality because there could be other elements).
And the hypothesis 'tab$2(ii$0) = FALSE' means that there is another element so

card(tab$2~[{FALSE}]) >= card(tab$1~[{FALSE}])+1.

Using the invariant on the cardinality gives us the contradiction.

In order to discharge this PO, a rule has to be added.

```
THEORY cardinality IS
   binhyp(!k.(k: a..b => t(k)=f)) &
   a <= b &
   binhyp(t(h)=f) &
   not(h: a..b) &
   c = b-a+1
   =>
   c+1 <= card(t~[{f}])
END
```

Demonstration

The hypothesis 'a <= b' brings the non-emptiness of the set a..b.
The first hypothesis is used to prove that

b-a+1 <= card(t~[{f}])

This is an inequality since a..b is in the domain of t (there could be other elements that are equal to ‘f’).
This is an equality if a..b is the domain of t.
Since b-a+1 is defined as ‘c’ we got

c <= card(t~[{f}])

But there is one more element t(h), with h not in a..b, thus we got the

c+1 <= card(t~[{f}]).

How to discharge the PO:

Once all the hypotheses are added to the stack of hypotheses (command **dd**), initialize the demonstration by contradiction (command **ct**).
Add the hypothesis ‘ii$0 = card(tab$1~[{FALSE}])’ and prove it using the hypotheses ‘not(ii$0+1: 0..card(tab$1~[{FALSE}]))’ and ‘ii$0: 0..card(tab$1~[{FALSE}])’. Add the proved hypothesis in the stack of hypotheses (**dd**).

Add the hypothesis

card(tab$1~[{FALSE}])+1 <= card(tab$2~[{FALSE}])

The proof of this hypothesis is done using the added rule. Apply it with the command **ar**.
The first precondition to prove is ‘0<=ii$0-1’: replace ii$0 by its expression card(tab$1~[{FALSE}]) (command **eh**), then replace this expression by card(tab$2~[{FALSE}]) still using the command **eh**.
Call the prover twice to prove this first precondition.

The last two preconditions are proved using the prover.
A last call of the prover discharge the PO.

The tree of commands for the two similar PO is:
(the section into square brackets is for the other PO because if we want the rule to be applied the guard on the hypothesis ‘t(h)=f’ must be SUCCESS: the hypothesis must be in the stack of hypotheses)

```
ff(2)
    dd(0)
        ct
            ah(ii$0 = card(tab$1~[{FALSE}]))
                ah(not(ii$0+1: 0..card(tab$1~[{FALSE}])))
                    ah(ii$0: 0..card(tab$1~[{FALSE}]))
                        pp(rt.0)
                dd
                    ah(card(tab$1~[{FALSE}])+1<=card(tab$2~[{FALSE}]))
[           ah(tab(jj$0)=FALSE)
                ah(not(tab(jj$0)=TRUE))
                    pp(rt.0)
                dd
]
                        ar(cardinality.1, Once)
                            eh(ii$0, _h, Goal)
                                eh(card(tab$1~[{FALSE}]))
                                    pr
                                        pr
                            pr
                            pr
                                pr
                        pr
```

The next PO has a similar demonstration: We have to show that ‘jj$0-1: card(tab$1~[{FALSE}])-1..NN’.
The reasoning is similar and the rule used is the same.
There also is another PO with the same goal and very similar demonstration.

The tree of commands for those POs is:

```
dd(0)
    ct
        ah(jj$0 = card(tab$1~[{FALSE}])-1)
            ah(not(jj$0-1: card(tab$1~[{FALSE}])-1..NN))
                ah(jj$0: card(tab$1~[{FALSE}])-1..NN)
                    pr
                    pp(rt.0)
            dd
                ah(card(tab$2~[{TRUE}])>=card(tab$1~[{TRUE}])+1)
                    ah(tab$2(ii$0) = TRUE)
                        ah(not(tab$2(ii$0) = FALSE))
                            pp(rt.0)
                        dd
                            ar(overload.4,Once)
                                eh(jj$0,_h,Goal)
                                    mp
                                        eh(card(tab$1~[{FALSE}]),_h,Goal)
                                            ah(0 = 8001-card(tab$1~[{FALSE}])-card(tab$1~[{TRUE}]))
                                                pr
                                                eh(card(tab$1~[{FALSE}]),_h,Goal)
                                                    eh(card(tab$1~[{TRUE}]),_h,Goal)
                                                        ah(card(tab$2~[{TRUE}])>=1)
                                                            pr
                                                                pr
                                                            pp(rt.0)
                                        ah(ii$0<=jj$0)
                                            pp(rt.0)
                                        eh(jj$0,_h,Goal)
                                            pr
                                pr
```

```
"`Local hypotheses'" &
tab$2: 0..NN +-> BOOL &
dom(tab$2) = 0..NN &
ii$0: 0..card(tab$1~[{FALSE}]) &
jj$0: card(tab$1~[{FALSE}])-1..NN &
auxi$0: BOOL &
auxj$0: BOOL &
card(tab$1~[{FALSE}]) = card(tab$2~[{FALSE}]) &
card(tab$1~[{TRUE}]) = card(tab$2~[{TRUE}]) &
!kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE) &
!ll.(ll: jj$0+1..NN => tab$2(ll) = TRUE) &
ii$0<=jj$0 &
not(tab$2(ii$0) = FALSE) &
not(tab$2(jj$0) = TRUE) &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
card(tab$1~[{FALSE}]) = card((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|-
>tab$2(ii$0)})~[{FALSE}])
```

This PO is obvious because this is the part when the permutation of two elements is done.

However, we can add the following re-writing rule in the pmm file:

```
card((t<+{i|->t(j)}<+{j|->t(i)})~[{k}]) == card(t~[{k}])
```

Demonstration:

We now that i and j belongs to the array's domain: they are used in the overload relation. Therefore, we can separate two cases: i differ from j and i is equal to j.

- not(i=j) => (t<+{i|->t(j)}<+{j|->t(i)}) is a permutation of two elements in the array. The cardinality is not changed.
- i=j => (t<+{i|->t(j)}<+{j|->t(i)}) is the identity. Indeed (t<+{i|->t(j)}<+{j|->t(i)}) = (t<+{i|->t(i)}<+{i|->t(i)}) = (t<+{i|->t(i)}) = t. The cardinality is not changed.

The tree of commands for those two POs is:

```
dd(0)
    ar(cardinality.2, Goal)
        pr
```

```
"`Local hypotheses'" &
tab$2: 0..NN +-> BOOL &
dom(tab$2) = 0..NN &
ii$0: 0..card(tab$1~[{FALSE}]) &
jj$0: card(tab$1~[{FALSE}])-1..NN &
auxi$0: BOOL &
auxj$0: BOOL &
card(tab$1~[{FALSE}]) = card(tab$2~[{FALSE}]) &
card(tab$1~[{TRUE}]) = card(tab$2~[{TRUE}]) &
!kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE) &
!ll.(ll: jj$0+1..NN => tab$2(ll) = TRUE) &
ii$0<=jj$0 &
not(tab$2(ii$0) = FALSE) &
not(tab$2(jj$0) = TRUE) &
kk: 0..ii$0+1-1 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
(tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk) = FALSE
```

The two left PO have a similar goal, and the same reasoning for the demonstration: by cases.
We first prove that this property is correct for the 0..ii$0-1 indexes, then we prove that the property is conserved for the index ii$0.

First of all, as there are no rules concerning the double overload add the following ones in the pmm file:

```
THEORY overload IS
   not(i=b) &
   not(i=a)
   =>
    (f<+{a|->x}<+{b|->y})(i) = f(i);

   binhyp(not(a=b))
   =>
   (f<+{a|->x}<+{b|->y})(a) = x
END
```

Both of them are validated by the rules validator.

To beginning with, add all the hypotheses to the stack of hypotheses, using the command **dd(0)**. For an easier reading, you can simplify the expression kk: 0..ii$0 by adding this expression as a hypothesis and calling the prover.

Start the demonstration by cases with the command **dc**.

Demonstration of the first case: kk: 0..ii$0-1.

Once the demonstration by case is started, the goal should be:

kk: 0..ii$0-1 => (tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk) = FALSE

Add “kk: 0..ii$0-1” in the stack of hypotheses (**dd**).

Add the hypothesis

(tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk)=tab$2(kk)

and apply the rule: the preconditions have to be proved, call the prover twice to do so.
Add this hypothesis in the stack of hypotheses and use it to simplify the goal (command **eh**); it should become:

tab$2(kk) = FALSE

As kk: 0..ii$0-1, we can particularize the hypothesis ‘!kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE)’.
This case is proved after the prover is called.

Demonstration of the first case: kk = ii$0.

At the beginning of this case we do not have the hypothesis that kk=ii$0: we first have to prove that: add the following hypotheses ‘not(kk: 0..ii$0-1)’ and ‘kk: 0..ii$0’ then call the prover.

Apply the rule overload.4 right after the hypothesis

(tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk) = tab$2(jj$0)

is added.

The proof of the precondition is done by contradiction: indeed if ii$0=jj$0 then the hypotheses ‘not(tab$2(ii$0) = FALSE)’ and ‘not(tab$2(jj$0) = TRUE)’ are contradictory. Once it is proved, replace the expression ‘(tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk)’ in the goal a re-add the hypothesis ‘not(tab$2(jj$0) = TRUE)’ then call the predicate proved: The PO is discharged.

The last PO has a similar goal, and the demonstration is done with the same reasoning: we prove that before the step jj$0 the property is true then we proved the property at the step jj$0. The tree of commands is given further.

The tree of commands for this PO is:

```
dd(0)
    ah(kk: 0..ii$0)
        pr
        dd
            dc(kk: 0..ii$0-1)
                dd
                    ah((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk)=tab$2(kk))
                        ar(overload.1, Once)
                            pr
                            pr
                        dd
                            eh((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk))
                                ph(kk, !kk.(kk: 0..ii$0-1 => tab$2(kk) = FALSE))
                                    pr
                dd
                    ah(kk=ii$0)
                        ah(not(kk: 0..ii$0-1))
                            ah(kk: 0..ii$0)
                                pp(rt.0)
                        ah((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk)=tab$2(jj$0))
                            ar(overload.2, Once)
                                pr
                                    ct
                                        ah(not(tab$2(ii$0)=FALSE))
                                            ah(not(tab$2(jj$0)=TRUE))
                                                eh(jj$0, _h, Goal)
                                                    pp(rt.0)
                                pr
                                dd
                                    eh((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(kk), _h, Goal)
                                        ah(not(tab$2(jj$0)=TRUE))
                                            pp(rt.0)
```

The tree of commands for the last PO is:

```
dd
    ah(ll: jj$0..NN)
        ah(ll: jj$0-1+1..NN)
            pp(rt.0)
    dd
        dc(ll: jj$0+1..NN)
            dd
                ah((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(ll) = tab$2(ll))
                    ar(overload.1, Once)
                        ah(ll: jj$0+1..NN)
                            pp(rt.0)
                        ah(ll: jj$0+1..NN)
                            ah(ii$0<=jj$0)
                                pp(rt.0)
                    dd
                        eh((tab$2<+{ii$0|->tab$2(jj$0)}<+{jj$0|->tab$2(ii$0)})(ll), _h, Goal)
                            ph(ll, !ll.(ll: jj$0+1..NN => tab$2(ll)=TRUE))
                                pr
            dd
                ah(ll=jj$0)
                    ah(not(ll: jj$0+1..NN))
                        ah(ll: jj$0..NN)
                            pp(rt.0)
                    pr
                        ah(not(tab$2(ii$0)=FALSE))
                            pp(rt.0)
```

C. Well-definedness proof obligations

14 POs of well-definedness are generated: 6 for the abstract machine, 8 for the implementation machine.
The force 0 prover discharges 2 POs and the force 3 prover discharges 1 PO.

10 POs have a similar goal, only the variable tab differs from a PO to another: tab, tab$0, tab$1.

```
"`Local hypotheses"' &
tab$1: 0..NN +-> BOOL &
dom(tab$1) = 0..NN
=>
tab~[{FALSE}]: FIN(tab~[{FALSE}])
```

The only way to discharge this PO is to apply rules. We first search a rule that can be applied thanks to the command **sr**. The theory that has rules about the 'FIN' keyword is InFINXY. Try to find a rule that can be used when the command **sr(InFINXY, r[u])** is send to the interactive prover.
The rule we are going to use is the 111th one.

Before the rule can be applied, the preconditions have to be in the stack of hypotheses.
The preconditions needed are that 'tab~: BOOL <-> 0..NN' and '0..NN: FIN(x)', with x a set. Since 0..NN is a set of naturals we can prove that 0..NN: FIN(NATURAL) but there are not a lot of rules regarding FIN(NATURAL): try the command sr(All, FIN(d)). Since NATURAL <: INTEGER, it would be easier to show that 0..NN: FIN(INTEGER).

Those two preconditions are proved by the prover when they are added as hypotheses (do not forget to add all the hypotheses in the stack of hypotheses at the beginning of the demonstration). Apply the InFINXY.111 rule: the prover will discharge the PO.

The tree of commands is:

```
dd(0)
    ah(tab~: BOOL <-> 0..NN)
        pr
        dd
            ah(0..NN: FIN(INTEGER))
                pr
                dd
                    ar(InFINXY.111,Once)
                        pr
```

You must adapt the first precondition for the other POs: 'tab$0~: BOOL <-> 0..NN' or 'tab$1~: BOOL <-> 0..NN', regarding the initial goal.

```
"`Local hypotheses'" &
tab$1: 0..NN +-> BOOL &
dom(tab$1) = 0..NN &
kk: 0..card(tab~[{FALSE}])-1
=>
kk: dom(tab$1)
```

After all the hypotheses are added to the stack of hypotheses and the dom(tab$1) expression is replaced, the goal is:

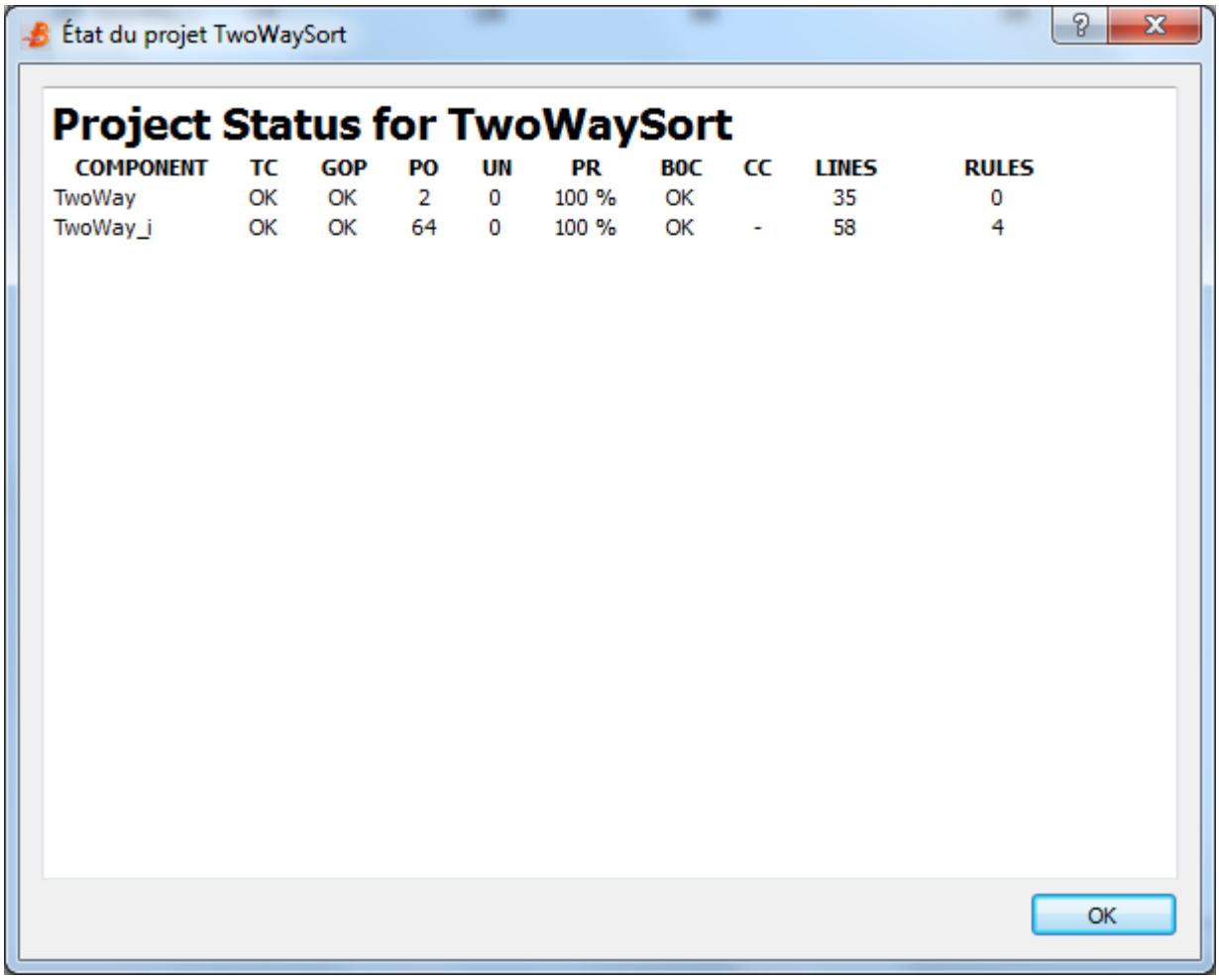
kk: 0..NN

This is easy to demonstrate because we have to property card(tab~[{FALSE}])<=NN+1.
This property is proved using the invariant: 'card(tab~[{FALSE}])+card(tab~[{TRUE}]) = NN+1', because card(tab~[{TRUE}]) >= 0.

The tree of commands for this PO is:

```
dd(0)
    eh(dom(tab$1),_h,Goal)
        ah(card(tab~[{FALSE}])<=NN+1)
            ah(card(tab~[{FALSE}])+card(tab~[{TRUE}]) = NN+1)
                ah(0<=card(tab~[{TRUE}]))
                    pr
                    pp(rt.0)
                pr
```

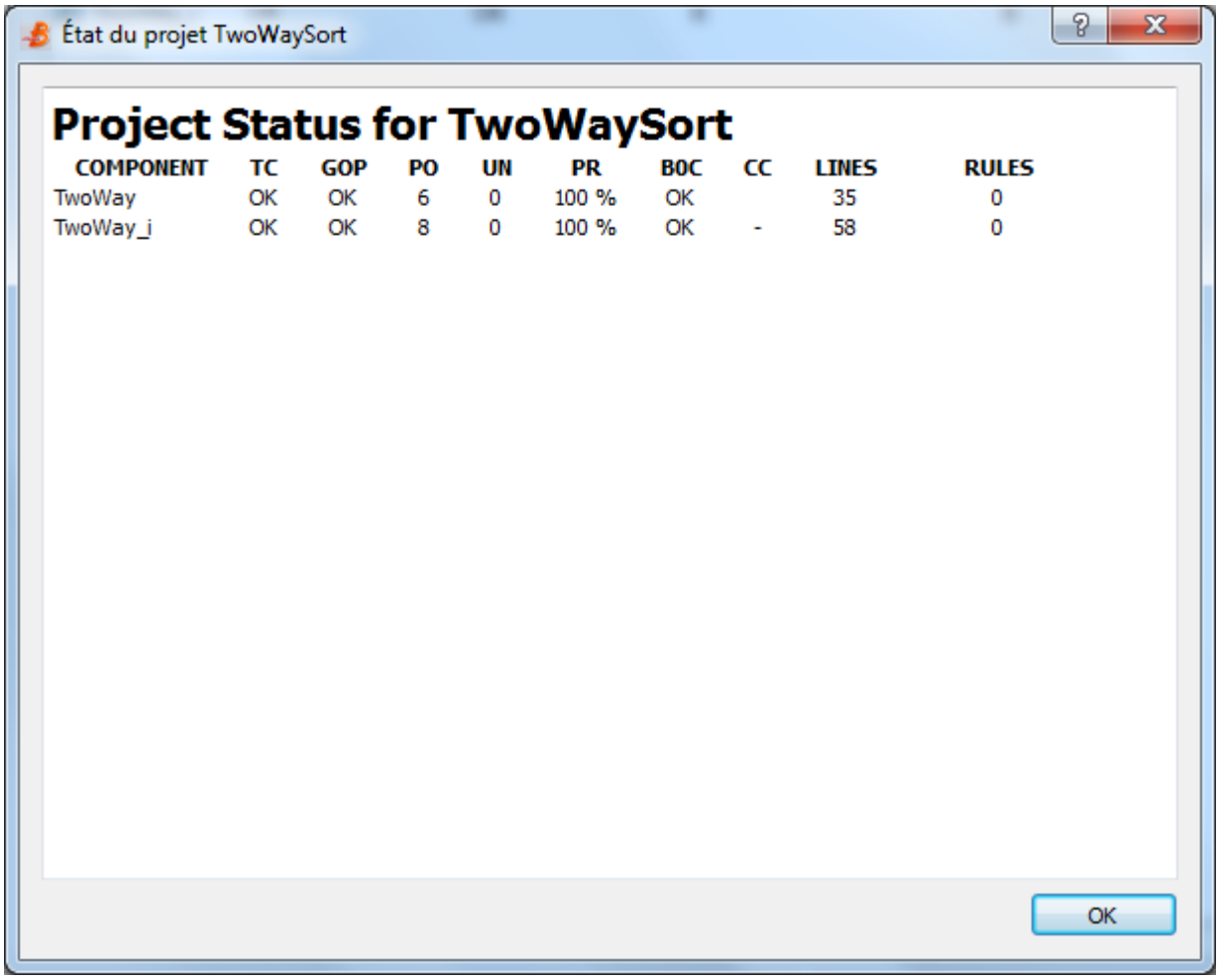
D. Project's status



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.

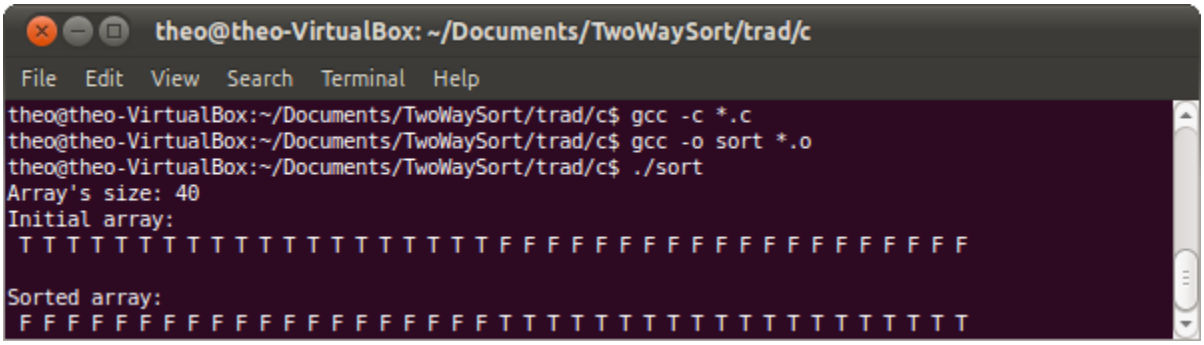
Well-definedness project's status:



IV. Generated code

In order to generate the code go to Project > Generate Code.
Select the “Generate a main file” box and click valid: the code is generated.

Once ouputs are added and the code compilated, this gives:



Note that the array in the implementation is not initialized this way. A test is manually added in the initialization (indexes are not modified) but we normally add an operation ‘initialization’ in the machine if we want an initialization that the B0 checker refuses and that respects the invariants.

The code generated is tested on an array of 80Millions elements; the command time gives us the time of execution: