VSTTE 2010 software verification competition

Challenge 3: Two equal elements

I. Description and analysis

Problem:

Given an integer array A of length n+2 with n>=2. It is known that at least two values stored in the array appear twice (i.e., there are at least two duplets).

Implement and verify a program finding such two values.

You may assume that the array contains values between 0 and n-1.

Problem analysis and architecture proposal:

The choice of the algorithm used to find the two duplets significantly impact on the complexity of the proofs obligations.

We will build a context machine where the array A is a constant, and the two duplets are defined.
The main machine of the program is made of one operation (the operation that finds such two values), but local operation (i.e. operations that can only be seen by the machine operations) are used so the complexity of the generated obligations proofs is reduced.

The algorithm chosen is the following:

-We first find the (lower) index of the first duplet: For every element of index *i* ($1 \leq i \leq N-3$) we run through the array from *i*+1 to N seeking for an identical element. If there is one, the index *i* is returned, incremented otherwise. There are two loops and they are imbricated.

-Then, we find the (lower) index of the second different duplet: For every element of index *j* (*i*+1 ≤ *j* ≤ N-1 with *i* the lower index of the first duplet) we run through the array from *j*+1 to N seeking for an element equal to A[j] and that is different of A[i]. If the element A[j] differs from A[i] and is a duplet, the index *j* is returned, incremented otherwise. Again, there are two loops and they are imbricated.

This operation contains two different loops that have a loop imbricated. Each loop is divided as a local operation so the errors of modeling/programing are easily detected, and the proofs are easier.

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 end of the operation.

Acronyms:

- POG: Proof Obligation Generator
- PO: Proof Obligation

A. Machine CTX

```
MACHINE
    CTX

CONCRETE_CONSTANTS
    NN, ARRAY

PROPERTIES
    NN: 4..MAXINT-1 &
    ARRAY: 0..NN --> NAT &

    //Two duplets in the array
    #(x1,x2,x3,x4).(x1: 0..NN-3 & x2: x1+1..NN &
                    x3: x1+1..NN-1 & x4: x3+1..NN &
                    not(ARRAY(x1)= ARRAY(x3)) &
                    ARRAY(x1) = ARRAY(x2) &
                    ARRAY(x3) = ARRAY(x4)
    )
END
```

The constants of this challenge are:

- NN, the number of elements in the array, and NN ≥ 4.
- The array ARRAY that contains the two duplets.

We can mathematically define a duplet as follow:

#(i,j).(i: dom(ARRAY) & j: dom(ARRAY) & not(i=j) & ARRAY(i)=ARRAY(j))

This expression can be rewrite:

#(i,j).(i: 0..NN-1 & j: i+1..NN & ARRAY(i)=ARRAY(j))

For two duplets we have:

#(i,j,k,l).(i: 0..NN-3 & j: i+1..NN &
k: i+1..NN-1 & j: k+1..NN &
ARRAY(i)=ARRAY(j) &

```
  ARRAY(k)=ARRAY(l)  &
  not(ARRAY(i)=ARRAY(k))
)
```

Explanations:

- If *i* is over NN-3, the existence of a second duplet is impossible.
- If *k* is over NN-1 (i.e. it is the last element in the array), it is impossible that *i* is the lower index of a duplet.
- Since j and k have the same domain of existence we can have j=k which means we do not have two duplets, but only one triplet. The property not(ARRAY(i)=ARRAY(k)) is added in order to provide only the existence of at least two duplets.

B. Machine Duplets

1. Abstract machine

```
CONCRETE_VARIABLES
indice
INVARIANT
indice: 0..NN-1
INITIALISATION
indice := 0

OPERATIONS
val1, val2 <-- seekForDuplets =
PRE
   val1: NAT &
   val2: NAT &
   #(x1,x2,x3,x4).(x1: 0..NN-3 & x2: x1+1..NN &
                   x3: x1+1..NN-1 & x4: x3+1..NN &
                   not(ARRAY(x1)=ARRAY(x3)) &
                   ARRAY(x1) = ARRAY(x2) &
                   ARRAY(x3) = ARRAY(x4)
    ) &
   indice : 0..NN-3
THEN
   ANY x1, x2, x3, x4 WHERE
      x1: 0..NN-3 & x2: x1+1..NN &
      x3: x1+1..NN-1 & x4: x3+1..NN &
      not(ARRAY(x1)=ARRAY(x3)) &
      ARRAY(x1) = ARRAY(x2) &
      ARRAY(x3) = ARRAY(x4)
   THEN
         val1, val2, indice:
              (val1: ran(ARRAY) & val2: ran(ARRAY) &
               val1=ARRAY(x1) & val2=ARRAY(x3) &
               indice=x3)
      END
   END
END
```

This is the only operation of the machine. In the PRE clause, we data type the input/output parameters, and specify the preconditions to have for using this operation (here, the existence of two duplets).
In the THEN clause, we specify what we have we the operation is exited: the values of the two different duplets.

The variable ‘indice’ is defined as a global variable otherwise we have to add it as a parameter of all the local operations.

2. Implementation:

The implementation of the operation is quite simple:

```
val1, val2 <-- seekForDuplets =
BEGIN
   findFirstDuplet;
   val1 := ARRAY(indice);

   findSecondDuplet;
   val2 := ARRAY(indice)
END;
```

findFirstDuplet and findSecondDuplet are both local operation, they represent the two loops and they contain also another operation (hasDuplet) that is the imbricated loop. The specification of these local operations is made in the LOCAL_OPERATION clause, and their implementation in the OPERATION clause.

- Specification of the findFirstDuplet local operation:

```
findFirstDuplet =
PRE
   indice: 0..NN-3 &
   #(x1,x2).(x1: 0..NN-3 & x2:x1+1..NN &
             ARRAY(x1)=ARRAY(x2))
THEN
   ANY x1, x2 WHERE
      x1: 0..NN-3 &
      x2: x1+1..NN &
      ARRAY(x1) = ARRAY(x2)
   THEN
      indice:=x1
   END
END
```

This operation finds the index of the first duplet in the array. The precondition for using this operation is to have at least one duplet in the array.
Since we have specified in the abstract machine that we have two duplets, we are obviously not obligated to re-specify it here (moreover, it is a local operation that can only be seen by the operation defined in the abstract machine so the duplet is already defined); but rewriting this precondition here will avoid us to prove it every time it is needed for the demonstration of some POs.

- Implementation of the findFirstDuplet local operation:

```
findFirstDuplet =
VAR hasdupl IN
   indice := 0;
   hasdupl <-- hasDuplet;
   WHILE(hasdupl=FALSE) DO
      indice := indice+1;
      hasdupl <-- hasDuplet
   INVARIANT
      indice: 0..NN-3 &
      hasdupl: BOOL &
      (hasdupl=TRUE =>
           #x2.(x2:indice+1..NN & ARRAY(indice)=ARRAY(x2))) &
      (hasdupl=FALSE =>
           !(x1,x2).(x1: 0..indice & x2: x1+1..NN
                          => not(ARRAY(x1)=ARRAY(x2))))
   VARIANT
      NN-indice
   END
END;
```

The hasDuplet operation is the imbricated loop. It returns a Boolean that indicate if the element ARRAY(indice) is a duplet.
The termination of the loop is obvious: we know that there is a duplet in the array, so the loop is exited before indice>NN-3.

The invariant is built easily:

- If the Boolean's value is FALSE then we have not yet found the duplet: all the previous elements tested aren't duplets.
- If the Boolean's value is TRUE then we just found the first duplet.

- Specification of the hasDuplet(index) local operation:

```
isDupl <-- hasDuplet =
PRE
   indice: 0..NN-1 &
   isDupl: BOOL
THEN
   isDupl: (isDupl: BOOL &
            (isDupl=TRUE
                 => #x4.(x4: indice +1..NN &
                             ARRAY(indice)=ARRAY(x4))) &
            (isDupl=FALSE
                 => !x4.(x4: indice +1..NN =>
                         not(ARRAY(indice)=ARRAY(x4))))
)
END
```

This operation returns a Boolean that indicates if there is an element in the array, with a different index, equal to the element ARRAY(indice).

- Implementation of the hasDuplet(index) local operation:

```
isDupl <-- hasDuplet =
VAR pas, temp IN
   pas := indice +1;
   temp := ARRAY(indice)-ARRAY(pas);
   IF(temp=0) THEN isDupl:=TRUE
   ELSE isDupl:=FALSE END;

   WHILE(pas<NN & isDupl=FALSE) DO
      pas := pas+1;
      temp := ARRAY(indice)-ARRAY(pas);
      IF(temp=0) THEN isDupl:=TRUE
      ELSE isDupl:=FALSE END
   INVARIANT
      pas: indice +1..NN &
      temp: INT &
      (isDupl=TRUE =>
             ARRAY(indice)=ARRAY(pas)) &
      (isDupl=FALSE =>
             !x4.(x4: ind+1..pas =>
                  not(ARRAY(indice)=ARRAY(x4))))
   VARIANT
      NN-pas
   END
END
```

In order to check the equality, we check if the subtraction of the current element tested and the element ARRAY(indice) is equal to zero. If it is not then we continue, else we exit the loop.

Construction of the invariant of the loop:

- If the Boolean's value is set at FALSE then all the previous element tested weren't equal to ARRAY(indice).
- If the Boolean's value is set at TRUE then this is the last iteration in the loop because we find an element that is equal to ARRAY(indice).

- Specification of the findSecondDuplet(index) local operation:

```
findSecondDuplet =
PRE
   indice: 0..NN-1 &
   #(x3,x4).(x3: indice+1..NN-1 & x4: x3+1..NN &
             not(ARRAY(indice)=ARRAY(x3)) &
             ARRAY(x3) = ARRAY(x4))
THEN
   ANY x3, x4 WHERE
      x3: indice +1..NN-1 &
      x4: x3+1..NN &
      not(ARRAY(x3)=ARRAY(indice)) &
      ARRAY(x3) = ARRAY(x4)
   THEN
      indice:=x3
   END
END
```

Before calling this local operation, we just found the lower index 'index' of the first duplet.
In order to find another different duplet one proceeds in the same way as for the findFirstDuplet operation, except that the duplet's value we find must be different from ARRAY(indice).
Again, in the preconditions of the operation we specify the existence of the second duplet even if we are not obligated: it will be helpful for the proof of some POs.

- Implementation of the findSecondDuplet(index) local operation:

```
findSecondDuplet =
VAR index, hasdupl, arediff IN
   index := indice;
   indice := indice +1;
   hasdupl <-- hasDuplet;
   arediff <-- differentValues(index, indice);
   WHILE(arediff=FALSE or hasdupl=FALSE) DO
      indice := indice+1;
      hasdupl <-- hasDuplet;
      arediff <-- differentValues(index, indice)
   INVARIANT
      //DATA TYPING
      indice: index+1..NN-1 &
      hasdupl: BOOL &
      arediff: BOOL &

      //LOCAL_OPERATION POSTCONDITIONS
      (hasdupl=TRUE =>
          #x4.(x4: indice+1..NN & ARRAY(indice)=ARRAY(x4))) &
      (hasdupl=FALSE =>
          !x4.(x4: indice+1..NN =>
                        not(ARRAY(indice)=ARRAY(x4)))) &
      (arediff=FALSE => ARRAY(index)=ARRAY(indice)) &
      (arediff=TRUE => not(ARRAY(index)=ARRAY(indice))) &

      //INVARIANT
      ((arediff=TRUE & hasdupl=TRUE) =>
           #x4.(x4: indice+1..NN & ARRAY(indice)=ARRAY(x4)) &
           not(ARRAY(index)=ARRAY(indice))) &
      (not(arediff=TRUE & hasdupl=TRUE) =>
           !(x3,x4).(x3: index+1..indice & x4: x3+1..NN =>
                              not(ARRAY(x3)=ARRAY(x4) &
                              not(ARRAY(index)=ARRAY(x3)))))
   VARIANT
      NN-indice
   END
END;
```

In the implementation, we define a new local operation differentValues(ind1, ind2) in order to test the difference between two element in the array (we will specify and implement it further, there are no difficulties for this operation).

The condition of the *while* is 'arediff=FALSE or hasdupl=FALSE' because when the code generator is called, only this writing is accepted.

Construction of the invariant:

In the invariant, we rewrite the post conditions of all the operations used.

The left of the invariant is obtained by reasoning on the exit of the loop:

- If the loop is exited (arediff=TRUE & hasdupl=TRUE) then we have the second duplet and its value is different from the first one.
- If the loop is not exited (not(arediff=TRUE & hasdupl=TRUE)) then each element tested either wasn't a duplet, or had the value of the first duplet.

- Specification of the differentValues(ind1, ind2) local operation:

```
areDiff <-- differentValues(ind1, ind2) =
PRE
   ind1: 0..NN &
   ind2: 0..NN &
   areDiff: BOOL
THEN
   areDiff: (areDiff: BOOL &
             (areDiff=FALSE => ARRAY(ind1)-ARRAY(ind2)=0) &
             (areDiff=TRUE => not(ARRAY(ind1)-ARRAY(ind2)=0))
             )
END
```

The specification and the implementation of this local operation are obvious. The POs generated are automatically discharged by the prover.

- Implementation of the differentValues(ind1, ind2) local operation:

```
areDiff <-- differentValues(ind1, ind2) =
VAR temp IN
   temp := ARRAY(ind1)-ARRAY(ind2);
   IF(temp=0) THEN areDiff:=FALSE
   ELSE areDiff:=TRUE END
END
```

III. Proof obligations

185 POs are generated. 145 are discharged with the force 0 prover, and 2 with the force 1 prover.
30 POs are discharged by the predicate prover with level 1 hypotheses (**pp(rp.1)**). In order to discharge them, use the command **te(pp(rp.1) & sw, Replace.Gen.Unproved)**. It must take a few minutes to test the 40 POs left.

10 POs remain to prove:

- 2 POs for the operation findTwoDuplets;
- 1 PO for the operation findFirstDuplet;
- 6 POs for the operation findSecondDuplet: Since we have the condition not(arediff=TRUE & hasdupl=TRUE) in the invariant, there are only 3 POs to demonstrate: indeed, there will be the case where arediff=FALSE and the case where hasdupl=FALSE;
- 1 PO for the hasDuplet operation.

One rule is added.

We first prove the two local operations, then the findTwoDuplets operation: If there are errors in the modeling/programing of the local operations, wrong POs are generated and the localization of the errors is easier. Moreover, wrong POs would also be generated for the findTwoDuplets operation and it is a loss of time trying to prove them.

- **Proof of the local operation hasDuplet:**

```
…
"`Local hypotheses"' &
not(ARRAY(indice$1)-ARRAY(indice$1+1) = 0) &
pas$0: indice$1+1..NN &
temp$0: INTEGER &
temp$0<=2147483647 &
-2147483647<=temp$0 &
isDupl$2 = TRUE => ARRAY(indice$1) = ARRAY(pas$0) &
isDupl$2 = FALSE => !x4.(x4: indice$1+1..pas$0 => not(ARRAY(indice$1) =
ARRAY(x4))) &                                                          ①
pas$0+1<=NN &
isDupl$2 = FALSE &
not(ARRAY(indice$1)-ARRAY(pas$0+1) = 0) &
x4: indice$1+1..pas$0+1 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
not(ARRAY(indice$1) = ARRAY(x4))
```

This PO concerns the conservation of the invariant while the loop iterates.
There are two cases: the case where we have not iterate yet (x4:indice$1+1..pas$0), and the case of the iteration (x4 = pas$0+1).

As usual, add all the hypotheses to the stack of hypotheses using the command **dd**, then start the demonstration by cases on the first case.

Demonstration of the first case:

Applying a modus ponens on the hypothesis ① then particularizing the hypothesis obtained on the x4 variable is enough for the prover to discharge this case.

Demonstration of the second case:

Starting with the hypothesis ‘not(x4: indice$1+1..pas$0)’ prove that ‘x4 = pas$0+1’ in adding the hypothesis defining x4 and calling the predicate prover.
Now that x4 = pas$0+1, the predicate prover is able to discharge this case.

The tree of commands for this PO is:

```
dd
    dc(x4:indice$+1..pas$0)
        dd
            mh(①)
                dd
                    ph(x4, ①_)
                        pr
        dd
            ah(x4=pas$0+1)
                ah(not(x4: indice$1+1..pas$0))
                    ah(x4: indice$1+1..pas$0+1)
                        pp(rp.0)
                pr
                    pp(rp.1)
```

With ①_ the right side of the predicate ①.

- **Proof of the POs generated for the findFirstDuplet local operation:**

The PO for this local operation is connected to the conservation of the invariant.

```
…
"`Local hypotheses'" &
isDupl$1: BOOL &
isDupl$1 = TRUE => #x4.(x4: 1..NN & ARRAY(0) = ARRAY(x4)) &
isDupl$1 = FALSE => !x4.(x4: 1..NN => not(ARRAY(0) = ARRAY(x4))) &
indice$2: 0..NN-3 &
hasdupl$0: BOOL &
hasdupl$0 = TRUE => #x2.(x2: indice$2+1..NN & ARRAY(indice$2) = ARRAY(x2)) &
hasdupl$0 = FALSE => !(x1,x2).(x1: 0..indice$2 & x2: x1+1..NN => not(ARRAY(x1) =
ARRAY(x2))) &                                                              ①
hasdupl$0 = FALSE &
isDupl$2: BOOL &
isDupl$2 = TRUE => #x4.(x4: indice$2+1+1..NN & ARRAY(indice$2+1) = ARRAY(x4)) &
isDupl$2 = FALSE => !x4.(x4: indice$2+1+1..NN => not(ARRAY(indice$2+1) = ARRAY(x4)))
&                                                                          ②
isDupl$2 = FALSE &
x1: 0..indice$2+1 &
x2: x1+1..NN &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
not(ARRAY(x1) = ARRAY(x2))
```

The proof of this PO is done by cases: when x1 belongs to 0..indice$2 (case1) and when x1 is equal to indice$2+1 (case 2).

Case 1: thanks to the hypothesis ①, this case is easily discharged by the prover.
Case 2: Using the hypothesis ②, the goal is proved quite easily.

In order to discharge this PO, after a deduction (**dd**), indicate the prover that the proof is done by cases by using the command **dc**.

Demonstration of the first case:

First apply a modus ponens on the hypothesis ① then add the hypothesis ①_ obtained in the stack of hypotheses, using the command **dd**. Particularizing the hypothesis ①_ with the variavles x1 and x2 and calling the prover discharge this first case.

Demonstration of the second case:

We first have to prove that x1=indice$2+1. To do so, call the predicate prover with the hypotheses 'not(x1: 0..indice$2)' and the definition of x1.
Now we have x1=indice$2+1, prove that x4: indice$2+1+1..NN before applying a modus ponens of the hypothesis ② and particularizing the hypothesis with the variable x2.
The PO is discharged.

The tree of commands for this PO is:

```
dd
    dc(x1: 0..indice$2)
        dd
            mh(①)
                dd
                    ph(x1, x2, ①_)
                        pr
                        pr
        dd
            ah(x1=indice$2+1)
                ah(not(x1: 0..indice$2))
                    pp(rp.1)
                mh(②)
                    dd
                        ph(x2, ②_)
                            ah(x2: x1+1..NN)
                                eh(x1,_h,Goal)
```

With ②_ the right side of the implication in the hypothesis ②.

- **Proof of the POs generated for the findSecondDuplet local operation:**

The first PO generated is connected to the call of the local operation differentValues(ind1, ind2). Indeed, when we call this operation, its preconditions have to be verified.
The second PO is connected to the termination of the loop (again, the POs generated for the variant are proved thanks to the hypothesis '`indice: index+1..NN-1`', but this hypothesis remains to be proved).
Both those POs have a quite similar demonstration, made by contradiction, and we will need the precondition added in the specification of the operation.

```
...
#(x3,x4).(x3: indice$1+1..NN-1 & x4: x3+1..NN & not(ARRAY(indice$1) = ARRAY(x3)) &
ARRAY(x3) = ARRAY(x4)) &                                                    ①
"`Local hypotheses"' &
isDupl$1: BOOL &
isDupl$1 = TRUE => #x4.(x4: indice$1+1+1..NN & ARRAY(indice$1+1) = ARRAY(x4)) &
isDupl$1 = FALSE => !x4.(x4: indice$1+1+1..NN => not(ARRAY(indice$1+1) = ARRAY(x4)))
&
areDiff$1: BOOL &
areDiff$1 = FALSE => ARRAY(indice$1)-ARRAY(indice$1+1) = 0 &
areDiff$1 = TRUE => not(ARRAY(indice$1)-ARRAY(indice$1+1) = 0) &
indice$2: indice$1+1..NN-1 &
hasdupl$0: BOOL &
arediff$0: BOOL &
hasdupl$0 = TRUE => #x4.(x4: indice$2+1..NN & ARRAY(indice$2) = ARRAY(x4)) &
hasdupl$0 = FALSE => !x4.(x4: indice$2+1..NN => not(ARRAY(indice$2) = ARRAY(x4))) &
arediff$0 = FALSE => ARRAY(indice$1) = ARRAY(indice$2) &
arediff$0 = TRUE => not(ARRAY(indice$1) = ARRAY(indice$2)) &
arediff$0 = TRUE & hasdupl$0 = TRUE => #x4.(x4: indice$2+1..NN & ARRAY(indice$2) =
ARRAY(x4)) & not(ARRAY(indice$1) = ARRAY(indice$2)) &
not(arediff$0 = TRUE & hasdupl$0 = TRUE) => !(x3,x4).(x3: indice$1+1..indice$2 & x4:
x3+1..NN => not(ARRAY(x3) = ARRAY(x4) & not(ARRAY(indice$1) = ARRAY(x3)))) &  ②
arediff$0 = FALSE &                                                         ③
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
indice$2+1: 0..NN-1
```

In order to discharge this PO, add all the hypotheses to the stack of hypotheses using the command (**dd**) then start the contradiction (the goal becomes *bfalse* and the previous one is negated, and added to the stack of hypotheses).
Add the hypothesis that indice$2 is equal to NN-1, and prove it with the predicate prover after adding the initial negated goal, the hypotheses that indice$2 belongs to index+1..NN-1 and index belongs to 0..NN-3.
Add the hypothesis just proved to the stack of hypotheses using the command **dd**.

Add the hypothesis not(arediff$0 = TRUE & hasdupl$0 = TRUE) (prove it by adding the hypothesis ③ and calling the predicate prover), so we can apply a modus ponens on the hypothesis ② (it is done automatically when the property not(arediff$0 = TRUE & hasdupl$0 = TRUE) is proved).
Replace now indice$2 by its value (command **eh**) and call the predicate prover after adding the hypothesis ①. The PO is discharged.

The tree of commands for this PO is:

```
dd
    ct
        ah(indice$2 = NN-1)
            ah(not(indice$2+1: 0..NN-1))
                ah(indice$2: indice$1+1..NN-1)
                    ah(indice$1: 0..NN-3)
                        pp(rt.0)
            dd
                ah(not(arediff=TRUE & hasdupl=TRUE))
                    ah(③)
                        pp(rt.0)
                    dd
                        mh(②)
                            eh(indice$2,_h,Goal)
                                ah(①)
                                    pp(rt.0)
```

There is another PO that has the same goal, but only the hypothesis ③ changes. This tree of commands is valid for this PO.

The next PO has a quite similar Goal, and the demonstration is also very similar.
The tree of commands for the second PO of this local operation is:

```
dd
    ct
        ah(indice$2 = NN-1)
            ah(not(indice$2+1: indice$1+1..NN-1))
                ah(indice$2: indice$1+1..NN-1)
                    pp(rt.0)
            dd
                ah(not(arediff=TRUE & hasdupl=TRUE))
                    ah(③)
                        pp(rt.0)
                    dd
                        mh(②)
                            eh(indice$2,_h,Goal)
                                ah(①)
                                    pp(rt.0)
```

The other PO with the same goal has the same demonstration than this one, only the hypothesis ③ changes.

The third and last PO for this local operation is connected to the conservation of the invariant.

```
…
#(x3,x4).(x3: indice$1+1..NN-1 & x4: x3+1..NN & not(ARRAY(indice$1) = ARRAY(x3)) &
ARRAY(x3) = ARRAY(x4)) &
"`Local hypotheses'" &
isDupl$1: BOOL &
isDupl$1 = TRUE => #x4.(x4: indice$1+1+1..NN & ARRAY(indice$1+1) = ARRAY(x4)) &
isDupl$1 = FALSE => !x4.(x4: indice$1+1+1..NN => not(ARRAY(indice$1+1) = ARRAY(x4)))
&
areDiff$1: BOOL &
areDiff$1 = FALSE => ARRAY(indice$1)-ARRAY(indice$1+1) = 0 &
areDiff$1 = TRUE => not(ARRAY(indice$1)-ARRAY(indice$1+1) = 0) &
indice$2: indice$1+1..NN-1 &
hasdupl$0: BOOL &
arediff$0: BOOL &
hasdupl$0 = TRUE => #x4.(x4: indice$2+1..NN & ARRAY(indice$2) = ARRAY(x4)) &
hasdupl$0 = FALSE => !x4.(x4: indice$2+1..NN => not(ARRAY(indice$2) = ARRAY(x4))) &
arediff$0 = FALSE => ARRAY(indice$1) = ARRAY(indice$2) &
arediff$0 = TRUE => not(ARRAY(indice$1) = ARRAY(indice$2)) &
arediff$0 = TRUE & hasdupl$0 = TRUE => #x4.(x4: indice$2+1..NN & ARRAY(indice$2) =
ARRAY(x4)) & not(ARRAY(indice$1) = ARRAY(indice$2)) &
not(arediff$0 = TRUE & hasdupl$0 = TRUE) => !(x3,x4).(x3: indice$1+1..indice$2 & x4:
x3+1..NN => not(ARRAY(x3) = ARRAY(x4) & not(ARRAY(indice$1) = ARRAY(x3)))) & ②
arediff$0 = FALSE &                                                          ③
isDupl$2: BOOL &
isDupl$2 = TRUE => #x4.(x4: indice$2+1+1..NN & ARRAY(indice$2+1) = ARRAY(x4)) &①
isDupl$2 = FALSE => !x4.(x4: indice$2+1+1..NN => not(ARRAY(indice$2+1) = ARRAY(x4)))
&                                                                            ⑥
areDiff$2: BOOL &
areDiff$2 = FALSE => ARRAY(indice$1)-ARRAY(indice$2+1) = 0 &
areDiff$2 = TRUE => not(ARRAY(indice$1)-ARRAY(indice$2+1) = 0) &
not(areDiff$2 = TRUE & isDupl$2 = TRUE) &                                    ④
x3: indice$1+1..indice$2+1 &
x4: x3+1..NN &
ARRAY(x3) = ARRAY(x4) &                                                      ⑤
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
ARRAY(indice$1) = ARRAY(x3)
```

The demonstration is made by cases. The case where x3: indice$1+1..indice$2 and the case where x3=indice$2+1.

As usual, add all the hypotheses in the stack of hypotheses (**dd**). Initialize then the demonstration by cases (**dc**).

Discharging the first case:

After a deduction, add the hypothesis not(arediff$0=TRUE & hasdupl$0=TRUE) and prove it by adding the hypothesis ③ and calling the predicate prover. The modus ponens is automatically done by the prover. After adding the right side of ② to the stack of hypotheses, particularize it with x3 and x4. Call the prover twice in order to prove the well-definedness of the variables x3 and x4, then add the hypothesis ⑤ and call the predicate prover. This case is proved.

Discharging the second case:

After a deduction, add the hypothesis 'x3 = indice$2+1'. Prove it with the predicate prover after adding the hypotheses 'not(x3: indice$1+1..indice$2)' and 'x3: indice$1+1..indice$2+1'.
The goal becomes:

ARRAY(indice$1) = ARRAY(indice$2+1).

Add the hypothesis 'x4: indice$2+1+1..NN' than prove it by adding the hypothesis 'x4: x3+1..NN' and replacing x3 by its value. Apply a deduction in order to have the previous goal:

ARRAY(indice$1) = ARRAY(indice$2+1).

Add the hypothesis #x4.(x4: indice$2+1+1..NN & ARRAY(indice$2+1) = ARRAY(x4)). Suggest the existence of such a value (**se(x4)**) and call the prover so this hypothesis is proved. Add it to the stack of hypotheses (**dd**).

Indicate the prover that the isDupl$2's value is TRUE. The proof of this hypothesis is made by contradiction: if isDupl$2=FALSE then the previous hypothesis we proved is in contradiction with the hypothesis ⑥_, generated after a modus ponens on hypothesis ⑥ (you have to demonstrate isDupl$2=FALSE with the hypothesis not(isDupl$2=TRUE) before doing the modus ponens).

Since isDupl$2=TRUE, thanks the hypothesis ④ we can prove that the variable areDiff$2's value is FALSE. With this value of areDiff$2, we have ARRAY(indice$1)-ARRAY(indice$2+1) = 0 (modus ponens). Calling the predicate prover on this goal discharge this PO.

The tree of commands for this PO is:

```
dd
    dc(x3: indice$1+1..indice$2)
        dd
            ah(not(arediff=TRUE & hasdupl=TRUE))
                ah(③)
                    pp(rt.0)
                dd
                    ph(x3,x4,②_)
                        pr
                        pr
                        ah(⑤)
                            pp(rt.0)
        dd
```

```
ah(x3 = indice$2+1)
    ah(not(x3: indice$1+1..indice$2))
        ah(x3: indice$1+1..indice$2+1)
            pp(rp.0)
ah(x4: indice$2+1+1..NN)
    ah(x4: x3+1..NN)

        eh(x3,_h,Goal)
    dd
        ah(①)
            se(x4)
                pr
            dd
                ah(isDupl$2=TRUE)
                    ct
                        ah(isDupl$2=FALSE)
                            ah(not(isDupl$2=TRUE))
                                pp(rt.0)
                            dd
                                mh(⑥)
                                    ah(①)
                                        pp(rt.0)
                    dd
                        ah(areDiff$2 = FALSE)
                            ah(④)
                                eh(isDupl$2,_h,Goal)
                                    pr
                            pp(rt.0)
```

With ① the hypothesis '#x4.(x4: indice$2+1+1..NN & ARRAY(indice$2+1) = ARRAY(x4))'.
Again, the other PO with the same goal has the same demonstration, only the hypothesis ③ changes.

Proof of the POs generated for the findTwoDuplets operations:

The first PO concerns the verification of the preconditions for the findSecondDuplet(ind) operation.

```
…
"`Local hypotheses"' &
x1: 0..NN-3 &
x2: x1+1..NN &
ARRAY(x1) = ARRAY(x2) &
indice$1: 0..NN-3 &
indice$1 = x1 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates"'
=>
#(x3,x4).(x3: indice$1+1..NN-1 & x4: x3+1..NN & not(ARRAY(indice$1) = ARRAY(x3)) &
ARRAY(x3) = ARRAY(x4))
```

For the demonstration of this PO, a rule is added.

Since we have the existence of two duplets, if we have found the first one (the one that have the lower index in the array), it is trivial that the second one still exists.

The rule added is the following:

```
THEORY ExistDuplet IS

   t: 0..n --> NAT &
   #(i,j,k,l).(i: 0..n-3 & j: i+1..n &
               k: i+1..n-1 & l: k+1..n &
               not(t(i)=t(k)) &
               t(i)=t(j) &
               t(k)=t(l)) &
   i: 0..n-3 &
   binhyp(j: i+1..n) &
   t(i)=t(j)
   =>
   #(k,l).(k: i+1..n-1 & l: k+1..n &
           not(t(i)=t(k)) &
           t(k)=t(l))
END
```

All the jokers in the goal are instanced when the rule is applied. The jokers in the goal are i, k, l, n and t: the joker j is not instanced, this is why there is a guard binhyp(`j: i+1..n`).

In order to discharge this PO, as usual add all the hypotheses to the stack of hypotheses (**dd**).
Replace the variable indice$1 by its value (x1) using the command **eh**.
Apply the rule previously added to the PMM associated file. And call the prover to prove the preconditions. The PO is discharged.

The tree of commands for this PO is:

```
dd
    eh(indice$2,_h,Goal)
        ar(ExistDuplet.1, Once)
            pr
            pr
            pr
            pr
```

The second PO of this operation is related to the specification: we have to prove we well have the values expected at the exit of the operation (the specification is respected).

```
…
"`Local hypotheses'" &
x3: indice$2+1..NN-1 &
x4: x3+1..NN &
not(ARRAY(x3) = ARRAY(indice$2)) &
ARRAY(x3) = ARRAY(x4) &
x1: 0..NN-3 &
x2: x1+1..NN &
ARRAY(x1) = ARRAY(x2) &
indice$1: indice$2+1..NN-1 &
indice$2: 0..NN-3 &
indice$1 = x3 &
indice$2 = x1 &
"`Check that the invariant (val1$1 = val1 & val2$1 = val2) is preserved by the operation - ref 4.4, 5.5'"
=>
#(x1,x2,x3,x4).(x1: 0..NN-3 & x2: x1+1..NN & x3: x1+1..NN-1 & x4: x3+1..NN &
not(ARRAY(x1) = ARRAY(x3)) & ARRAY(x1) = ARRAY(x2) & ARRAY(x3) = ARRAY(x4)
& #(val1$1,val2$1).(val1$1: ran(ARRAY) & val2$1: ran(ARRAY) & val1$1 = ARRAY(x1) &
val2$1 = ARRAY(x3) & (ARRAY(indice$2) = val1$1 & ARRAY(indice$1) = val2$1)))
```

In order to discharge this PO, add all the hypotheses to the stack of hypotheses (**dd**).

Then, suggest the existence of the four variables x1, x2, x3 and x4. We now have to prove the validity of the conditions and calling the predicate prover or the prover is enough to discharge this PO.

The tree of commands for this PO is:

```
dd
    se(x1,x2,x3,x4)
        pr
            pp(rp.1)
            pp(rp.1)
            pp(rp.1)
            eh(x1, _h, Goal)
                pr
            pr
            pr
            pr
                pr
```

C. Well-definedness of the B-components

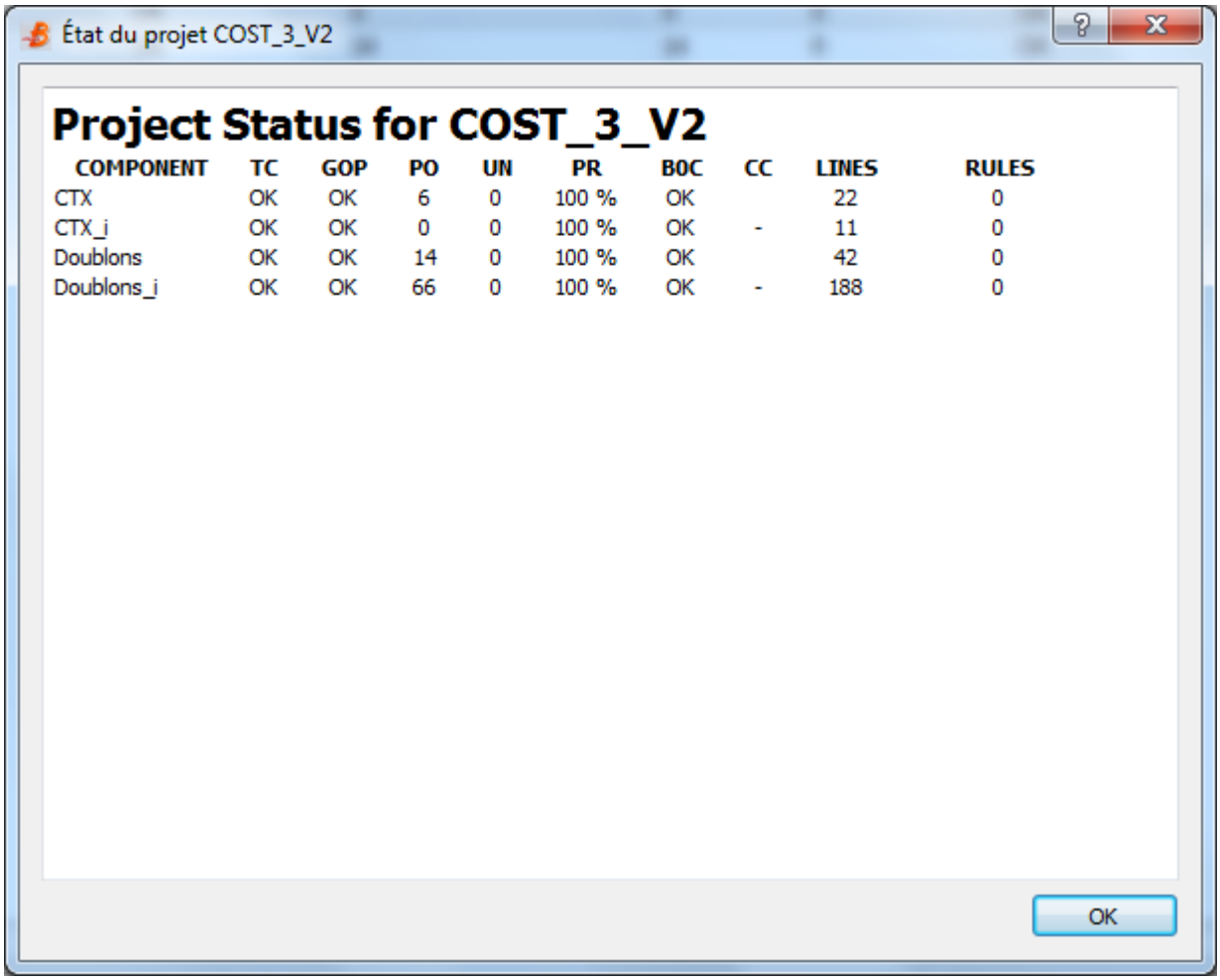
A total of 86 POs of well-definedness are generated. 76 are discharged by the force 0 prover, 7 by the force 1 prover.
The three POs left have the same goal:

x4: dom(ARRAY).

After a deduction, use the command **eh** on dom(ARRAY) so the goal becomes:

x4: 0..NN.

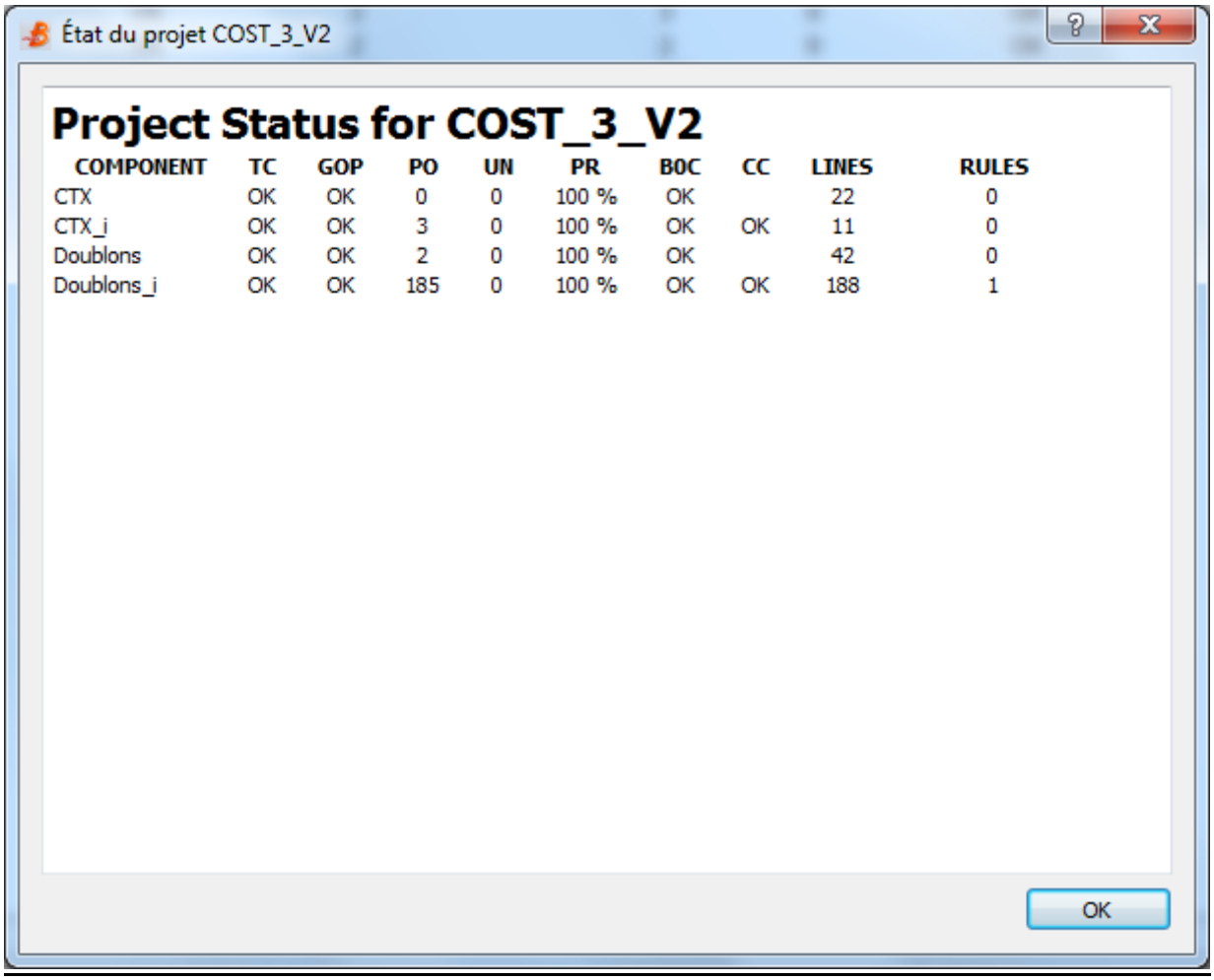
Add the hypothesis where x4 is defined. Then add the hypotheses in relation with the previous added hypothesis (for instance, if x4: x3+1..NN then add the hypothesis where x3 is defined. If x3 is defined on index+1..indice$0 then add the hypotheses index+1: 0..NN and indice$0: index+1..NN) until all the variables are defined. Call the predicate prover (**pp(rp.0)**) so the POs are discharged.



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.

D. Project status



IV. Generated code

The constants in the CTX machine have to be valuated in the clause VALUES of the implementation (so we have the properties verified by the generated POs).

To be able to translate the project, it has to be b0 checked. For the CTX implementation to be b0 checked, I set NN to 4 and the array to 0. Doing that does not provide the property of the abstraction. In fact the project has to be translated in two times. First the CTX_i component then the rest. Before generating the POs of the CTX_i component, comment the property in the abstraction. Prove the component, b0 check it and generate the code separately.

Delete the CTX_i machine, uncomment the property in the CTX component, replay the proofs. Now the project is B0 checked without any errors. We can now generate the code of the hole project (Project > Code Generator). Do not forget to indicate the generator that you want it to create a main function (The main component is the component Doublons).

Edit the main.c file a follow:

```
#include "stdio.h"
#include "b_init.h"
#include "Doublons.h"

int main(void)
{
      int32_t val1, val2;
     val1=0;
     val2=0;

     b_initialisation();
     Doublons__seekForDuplets(&val1, &val2);

     printf("doublon 1: %d\ndoublon 2: %d\n", (int)val1, (int)val2);
}
```

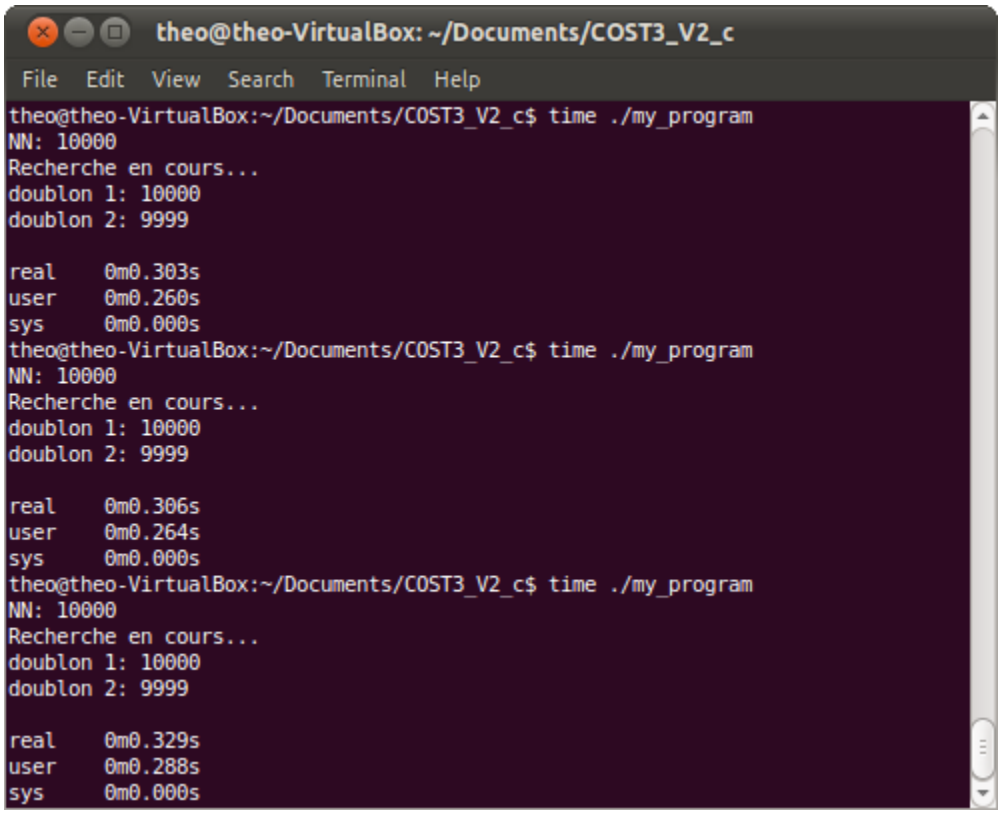
Edit also the ctx.h and ctx.c files so the array verifies the property of the CTX component.
I edit the CTX_INITIALISATION called by the b_initialisation() function as follow:

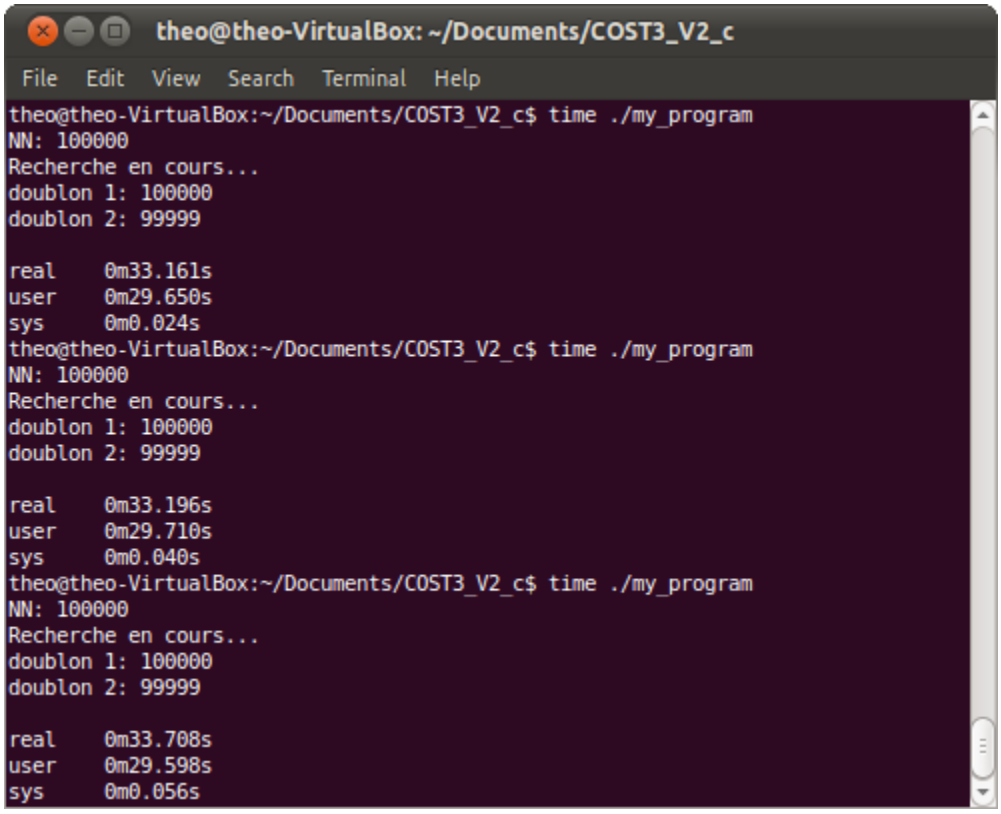
```
void CTX__INITIALISATION(void)
{
    int i;
    i = 0;
    for(i=0; i <= CTX__NN; i++)
    {
        CTX__ARRAY[i]=i;
    }

    //Creation of the two duplets
    CTX__ARRAY[CTX__NN-3]=CTX__NN;
    CTX__ARRAY[CTX__NN-2]=CTX__NN-1;

    printf("NN: %d\n", CTX__NN);
}
```

Of course this initialization generates the worst case for this algorithm. There will be n(n+1)/2 comparisons.

You can now generate the executable code using the commands:
gcc –c * (create the .o files)
gcc –o my_program *.o (link the .o files and create the executable)

The algorithm used has a n^2 time complexity, so for big array it takes a time (including the initialization).
The time is calculate using the command *time* under linux (and this is not very accurate).


theo@theo-VirtualBox: ~/Documents/COST3_V2_c
File Edit View Search Terminal Help
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 10000
Recherche en cours...
doublon 1: 10000
doublon 2: 9999
real 0m0.303s
user 0m0.260s
sys 0m0.000s
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 10000
Recherche en cours...
doublon 1: 10000
doublon 2: 9999
real 0m0.306s
user 0m0.264s
sys 0m0.000s
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 10000
Recherche en cours...
doublon 1: 10000
doublon 2: 9999
real 0m0.329s
user 0m0.288s
sys 0m0.000s


theo@theo-VirtualBox: ~/Documents/COST3_V2_c
File Edit View Search Terminal Help
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 100000
Recherche en cours...
doublon 1: 100000
doublon 2: 99999
real 0m33.161s
user 0m29.650s
sys 0m0.024s
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 100000
Recherche en cours...
doublon 1: 100000
doublon 2: 99999
real 0m33.196s
user 0m29.710s
sys 0m0.040s
theo@theo-VirtualBox:~/Documents/COST3_V2_c$ time ./my_program
NN: 100000
Recherche en cours...
doublon 1: 100000
doublon 2: 99999
real 0m33.708s
user 0m29.598s
sys 0m0.056s