CLEARSY
SYSTEM ENGINEERING

VSTTE 2010 software verification competition

Problem 2: Inverting an Injection

I. Description and analysis

Problem:

Invert an injective array A on N elements in the subrange from 0 to N-1, i.e. the output array B must be such that B[A[i]] = i for $0 \le i < N$.
You can assume that A is surjective.

Properties:

Show that the resulting array B is also injective. For bonus points, you can demonstrate other properties, e.g., that A and B are inverses.

Problem analysis:

A is a permutation, i.e. A is a bijective homomorphism. Since B[A[i]] = i, for $0 \le i < N$ is a bijection (this is the identical permutation that maps every element of the set to itself) we can deduce that the array B is surjective. But its injectivity remains to prove. Proving that B is injective is equivalent to prove that B is bijective.
Since the composition of B with A is the identical permutation, it is easy to prove that B = A~ **IF** B is injective.

From the specification of the problem, we extract two functionalities:

FUNC1: Create a function that inverts the array A.
FUNC2: Prove the array B is injective or/and arrays A and B are inverses.

Architecture:

In this case, the two functionalities are merged in only one machine and its refinements. Indeed, the injectivity of the array B is proved in the refinement (FUNC2 is an abstract functionality) of the inverting function as the implementation bring useful hypotheses for the demonstration.

As usual, constants like the array A and N the number of elements in the array are defined in the context machine. It is seen by the machine that inverts the Array.

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, tab

PROPERTIES
   NN : NAT &
   tab : 0..NN >->> 0..NN

END
```

This machine contains the constants of the project and their properties. It is seen by the main machine InvArray. 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.
Warning: In the array tab: 0..NN >->> 0..NN, there are NN+1 elements.

B. Machine InvArray

1. Abstract machine:

This machine is the machine containing the two functionalities of the project. It is made of only one operation.

```
tabB <-- InvertArray =
PRE
   tabB : 0..NN --> 0..NN
THEN
   tabB := tabA~
END
```

In the PRE clause, the properties and the type of the constant are not rewrite. Only the typing of the array B is made, and at this step the property is not verified.

In the case the property to prove is that the array B is the inverse of the array A, we add the property wanted in the THEN clause. In the case the property to prove is that the array B is injective, instead of “tabB := tabA~” write “skip”: the property to prove is not a substitution, it will be specified in the refinement.

2. Refinement:

```
tabB <-- InvertArray =
BEGIN
   tabB : (tabB = tabA~ &
        !ii.(ii : 0..NN => tabB(tabA(ii)) = ii))
END
```

This refinement is an algorithmic refinement.
In the case this is the property we want to prove, it is rewritten in order to have to prove it in a PO of the implementation. In the case we want to prove the injectivity of the array B: replace “tabB = tabA~” by “tabB: 0..NN >-> 0..NN”.

The algorithm for the inversion is also described. The invariant of the loop is obtained by replacing the NN constant by the index of the current element in the loop of the implementation.

3. Implementation:

```
tabB <-- InvertArray =
VAR index IN
   index:= 0;
   tabB(tabA(index)) := index;
   WHILE(index < NN) DO
      index:= index + 1;
      tabB(tabA(index)) := index
   INVARIANT
      index: 0..NN &
      tabB : 0..NN --> 0..NN &
      !ii.(ii : 0.. index => tabB(tabA(ii)) = ii)
   VARIANT
      NN-index
   END
END
```

Construction of the invariant:

After the loop iterates, the relation tabA(index)|->index is added in the tabB array. So, at the index+1’th iteration, there are ‘index+1’ elements treated.

Construction of the variant:

The ‘index’ variable is incremented at the beginning of a loop’s iteration. It maximal value is NN, so the expression NN-index is decreasing and positive.

III. Proof obligations

23 POs are generated for the InvTab machine. 19 POs are automatically discharged by the force 0 prover. 2 POs are discharged by the predicate prover (**p1**).

One rule is added. It concerns the property wanted (FUNC2). Both the injectivity and the fact that tabB is the inverse of tabA are demonstrated.

A. Functional proof obligations

- The first PO:

```
…
"`Local hypotheses'" &
pas$0: 0..NN &
tabB$2: 0..NN +-> 0..NN &
dom(tabB$2) = 0..NN &
!ii.(ii: 0..pas$0 => tabB$2(tabA(ii)) = ii) &                                    ①
pas$0+1<=NN &
ii: 0..pas$0+1 &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
(tabB$2<+{tabA(pas$0+1)|->pas$0+1})(tabA(ii)) = ii
```

This PO concerns the conservation of the invariant. When the loop iterates, we have to prove that the relation added to the array preserves the invariant.

We can distingue two cases:
First case: ii=pas$0+1
By definition of the overloading operator: we well have pas$0+1 = ii.

Second case: ii: 0..pas$0
This case is more delicate because the overloading's definition gives us nothing, except that since ii is different from pas$0+1, we have to demonstrate tabB$2(ii) = ii. Fortunately, we have the hypothesis ①.

After adding all the hypotheses to the stack of hypotheses (**dd**), initialize the demonstration by cases (**dc**) with the first case.
A call to the prover discharge this first case, and we now have to do the second case (add the hypothesis *not(i=pas$0+1)* to the stack of hypotheses).

First add the hypothesis that now ii: 0..pas$0, and prove it adding the previous hypothesis not(ii=pas$0+1) and the hypothesis ii: 0..pas$0+1 (do forget to call the predicate prover).

Particularize now the hypothesis ① on the ii variable, so the current goal becomes:

tabB$2(tabA(ii)) = ii => (tabB$2<+{tabA(pas$0+1)|->pas$0+1})(tabA(ii)) = ii

A call to the miniprover (**mp**) in force 2 discharges the PO (it takes time however).

The tree of commands for this PO is:

```
ff(2)
    dd(0)
        dc(ii = pas$0+1)
            pr
            dd
                ah(ii: 0..pas$0)
                    ah(not(ii = pas$0+1))
                        ah(ii: 0..pas$0+1)
                            pp(rt.0)
                     dd
                        ph(①)
                            pr
                            mp
```

- Second PO : proof of the property(ies)

```
…
"`Local hypotheses'" &
not(pas$7777+1<=NN) &
pas$7777: 0..NN &
tabBz$7777: 0..NN +-> 0..NN &
dom(tabBz$7777) = 0..NN &
!ii.(ii: 0..pas$7777 => tabBz$7777(tabA(ii)) = ii) &                ①
"`Check operation refinement - ref 4.4, 5.5'"
=>
tabBz$7777 = tabA~
```

According to the property specified in the abstract machine, the rule to add differs. However the demonstrations of both are done.

Case the specification is tabB: 0..NN >->> 0..NN (tabB is injective):

Hypotheses:

Let E be the set of natural numbers from 0 to N: E = 0..NN.

Let tabA be the bijonction from E to E, and tabB be a function from E to E.

Let a and b elements of E.

Let the equation (1) be: tabB(a) = tabB(b).

Let the equation (2) be: tabB(tabA(i)) = i for i in E.

Demonstration:

Since tabA is surjective, and a and b are elements of the co-domain of tabA, there are a' and b' variables belonging to the domain of tabA such that

a = tabA(a') and b = tabA(b').

Replacing the values of a and b in the equation (1) gives the equation (3):

tabB(tabA(a')) = tabB(tabA(b'))

Applying the equation (2) on the members of equation (3) gives:

a' = b'

As tabA is injective, we get tabA(a') = tabA(b').

From the previous definition of tabA(a') and tabA(b'), we obtain the equality a = b.

In conclusion, we have tabB(a) = tabB(b) ⇔ a = b.

This is the definition of the injectivity for the tabB function.

From this demonstration, we can add a rule in the pmm:

```
THEORY Bijectivity IS
    binhyp(t: 0..n >->> 0..n) &
    binhyp(!i.(i: dom(t) => h(t(i)) = i)
    =>
    h : 0..n >->> 0..n
END
```

The implication in this rule is stronger: we have the bijectivity because tabB is also surjective.

Case the specification is tabB = tabA~

Hypotheses:

tabB(tabA(i)) = i for 0 ≤ i ≤ N equality (1)

tabA is bijective.

Using the previous demonstration, we can assume that tabB is also bijective (previous proof).

Demonstration:

Since tabB is bijective, its inverse tabB~ exist.

Composing the equality (1) by tabB~ (it is possible in view that tabB is injective) gives:

tabB~(tabB(tabA(i))) = tabB~(i) for 0 ≤ i ≤ N

⇔ tabA(i) = tabB~(i) for 0 ≤ i ≤ N

tabB is then the inverse of tabA.

The rule that can be added is the following:

```
dom(h) = dom(t) &
binhyp(!i.(i: dom(t) => h(t(i)) = i))
=>
h = t~
```

Demonstration of the PO (case Func2 is 'tabB = tabA~'):

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

The first of the demonstration is to prove that pas$7777 = NN. To do so, add the hypothesis not(pas$7777<=NN) (a call of the prover is need to prove its existence) then add the hypothesis pas$7777: 0..NN. The predicate prover demonstrates the equality.

Now, add the following hypothesis :

ah(!ii.(ii: dom(tabA) => tabBz$7777(tabA(ii)) = ii))

And the proof of thius hypothesis is made by adding the hypothesis ①, replacing pas$7777 by its value, and replacing 0..NN by dom(tabBz$7777), then by replacing dom(tabBz$7777) by dom(tabA) after the equality is proved.

Apply now the rule and prove the preconditions. The PO is discharged.

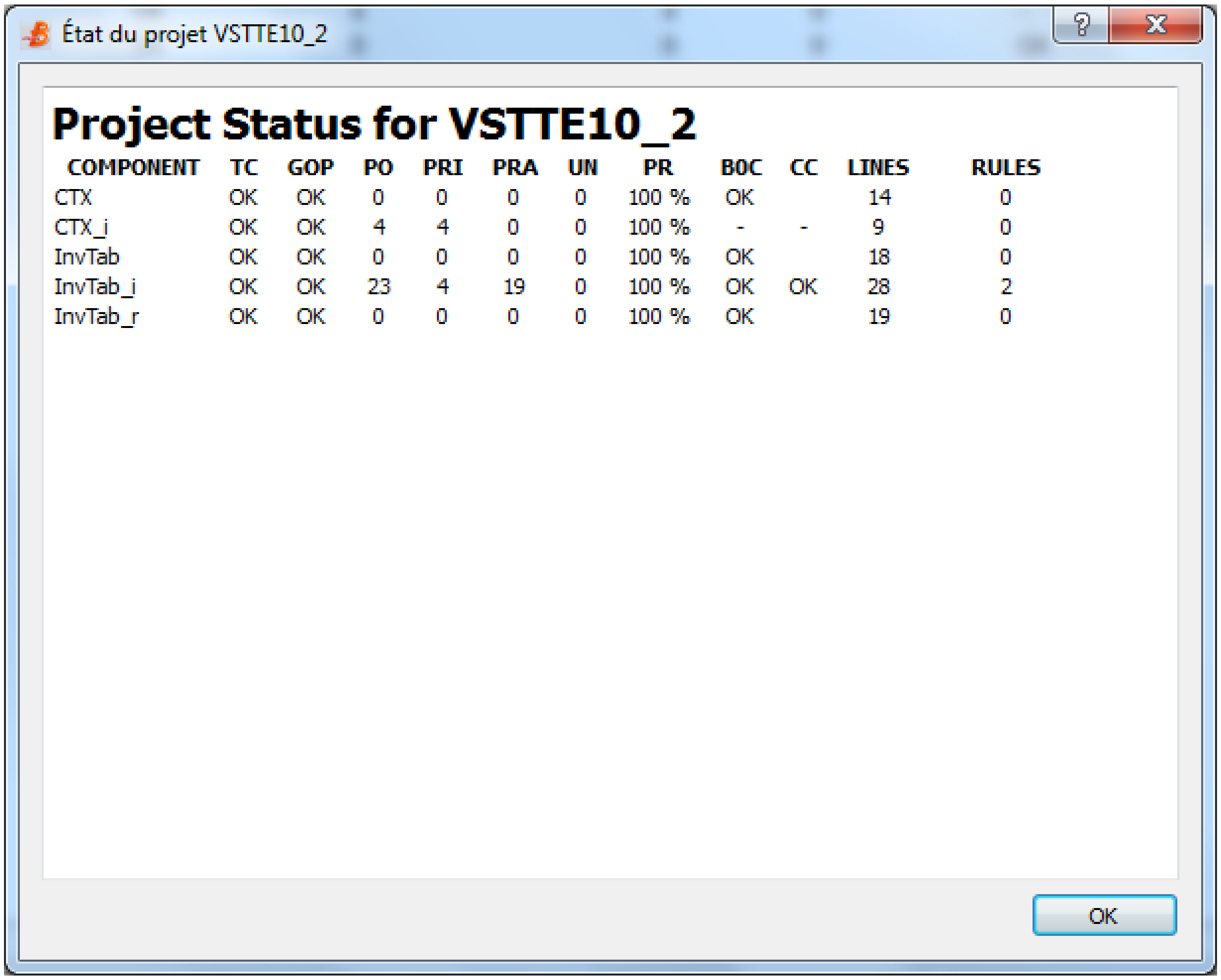
The tree of commands for this PO is:

```
dd(0)
    ah(pas$7777 = NN)
         ah(not(pas$7777+1<=NN))
             pr
             ah(pas$7777: 0..NN)
                 pp(rt.0)
        dd
             ah(!ii.(ii: dom(tabA) => tabBz$7777(tabA(ii)) = ii))
                 ah(①)
                     eh(pas$7777,_h,Goal)
                         eh(0..NN,_h,Goal)
                             ah(dom(tabBz$7777) = dom(tabA))
                                 pr
                                 dd
                                     eh(dom(tabBz$7777),_h,Goal)
                 dd
                     ar(Surcharge.2,Once)
                         pr
```

B. Well-definedness proof obligations

6 POs of well-definedness are generated, and they are all automatically discharged by the prover.

C. Status

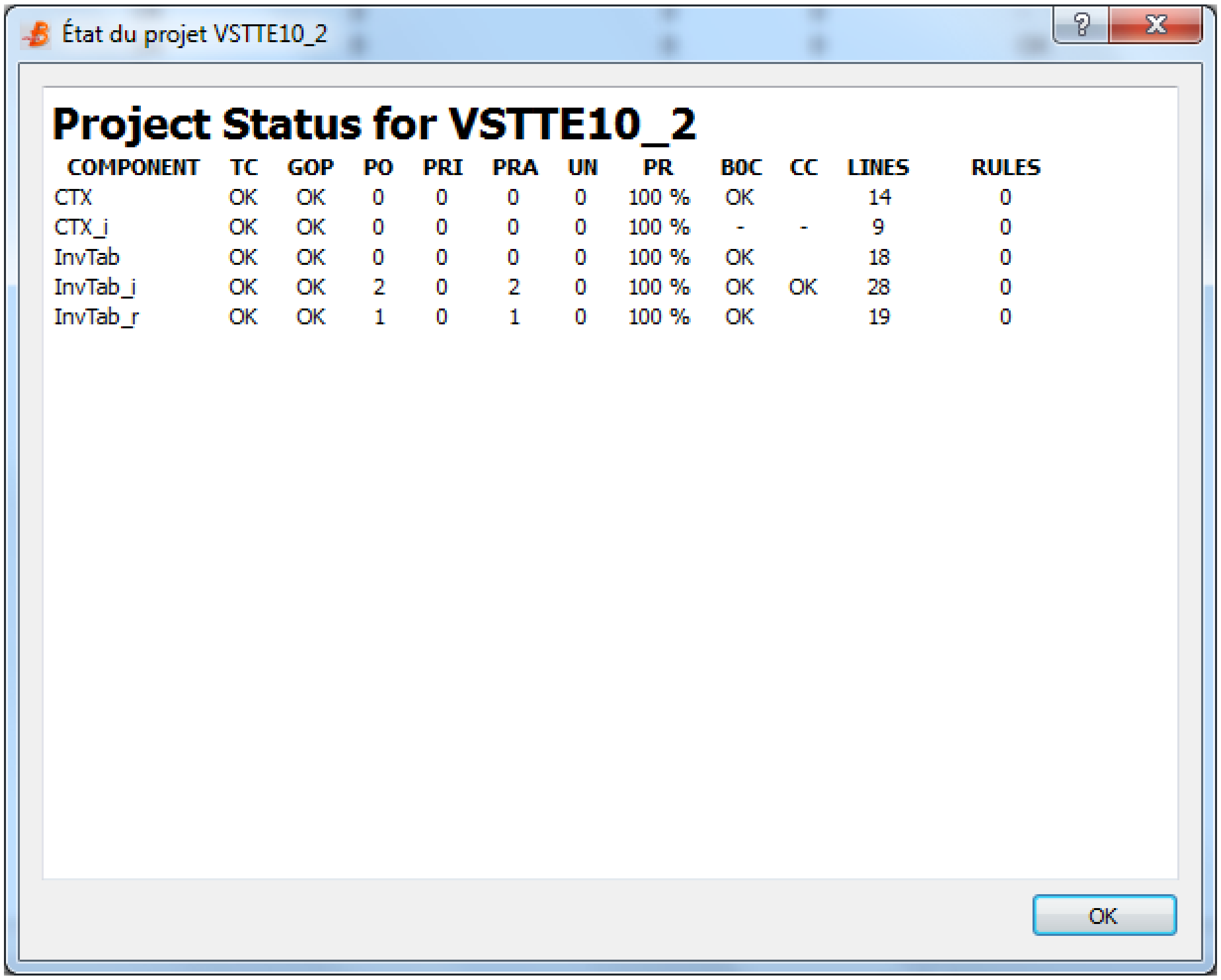


Project Status for VSTTE10_2

COMPONENT	TC	GOP	PO	PRI	PRA	UN	PR	BOC	CC	LINES	RULES
CTX	OK	OK	0	0	0	0	100 %	OK		14	0
CTX_i	OK	OK	4	4	0	0	100 %	-	-	9	0
InvTab	OK	OK	0	0	0	0	100 %	OK		18	0
InvTab_i	OK	OK	23	4	19	0	100 %	OK	OK	28	2
InvTab_r	OK	OK	0	0	0	0	100 %	OK		19	0

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.



Project Status for VSTTE10_2

COMPONENT	TC	GOP	PO	PRI	PRA	UN	PR	BOC	CC	LINES	RULES
CTX	OK	OK	0	0	0	0	100 %	OK		14	0
CTX_i	OK	OK	0	0	0	0	100 %	-	-	9	0
InvTab	OK	OK	0	0	0	0	100 %	OK		18	0
InvTab_i	OK	OK	2	0	2	0	100 %	OK	OK	28	0
InvTab_r	OK	OK	1	0	1	0	100 %	OK		19	0

Project’s status of well-definedness.

IV. Generated Code

The generation of the code is made after the components are B0 checked.
A problem with the B0 checker prevents us either to prove the POs generated for the CTX implementation, either to B0 check it.
We generate the code of the CTX_i separately with CTX_i:

```
VALUES
NN = 5;
tabA = (0..NN) * {0}
```

And with tabA defined as tabA: 0..NN --> 0..NN. The POs generated are automatically discharged and both the components can be B0 checked.
Right click on the CTX component, select Code generator then choose c. Do the same for the CTX_i component.

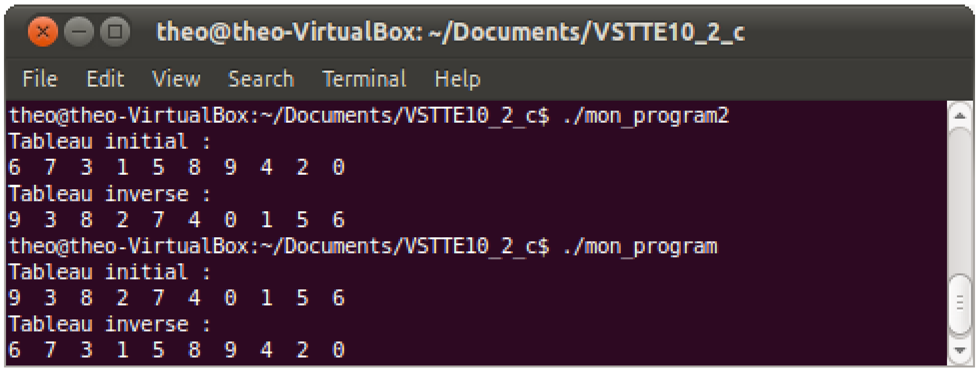
Once this is done, rewrite tabA: 0..NN >->> 0..NN in the CTX machine, and delete the CTX_i machine.

Replay the proof obligations in order to have the full project proved, and generate the code for the project (the CTX_i machine will generate an error). Select the checkbox that propose the generation of the main() function.

Edit the main.c file:

```
#include "stdio.h"
#include "CTX.h"
#include "InvTab.h"

static int32_t Main__tab[CTX__NN+1];

int main(void)
{
   unsigned int i = 0;
   for(i = 0; i <= CTX__NN; i++)
   {
      Main__tab[i] = CTX__tabA[i];
   }

   printf("Tableau initial :\n");
   for(i = 0; i <= CTX__NN; i++)
   {
      printf("%d\n", Main__tab[i]);
   }

   InvTab__InverserTab(Main__tab);

   printf("Tableau inverse :\n");
   for(i = 0; i <= CTX__NN; i++)
   {
      printf("%d\n", Main__tab[i]);
   }
}
```

Edit also the ctx.c and ctx.h files so we well have a bijective array.


theo@theo-VirtualBox: ~/Documents/VSTTE10_2_c
File Edit View Search Terminal Help

```
theo@theo-VirtualBox:~/Documents/VSTTE10_2_c$ ./mon_program2
Tableau initial :
6  7  3  1  5  8  9  4  2  0
Tableau inverse :
9  3  8  2  7  4  0  1  5  6
theo@theo-VirtualBox:~/Documents/VSTTE10_2_c$ ./mon_program
Tableau initial :
9  3  8  2  7  4  0  1  5  6
Tableau inverse :
6  7  3  1  5  8  9  4  2  0
```