VSTTE 2012 software verification competition

Problem 3: Ring Buffer

I. Description and analysis

Problem:

We want to implement a queue data structure using a ring buffer that can be described with the following type declaration:

```
type ring_buffer = record
     data : array of int;
     size : int;
     first : int;
     len : int;
end
```

A ring buffer is an array data of fixed size 'size' where we store the 'len' queue elements starting from index first. There are two possible situations:

- If first + len ≤ size, then the queue elements are stored consecutively within data.
- If first + len > size, then the queue wraps over the end of data and continues from index 0.

A ring buffer is said to be full when len = size and empty when len = 0.
In this description, we fixed the type of queue elements to type int. However, participants are encouraged to implement and prove a generic data structure, using for example polymorphic types, type classes, generics, etc.

You are supposed to implement the following operations:

- create(n): takes a positive integer n and returns a new ring buffer with size n and length 0 (no element).
- clear(b): takes a ring buffer b as argument and empties it.
- head(b): takes a ring buffer b as argument and returns the first element in the queue.
- push(b, x): takes a ring buffer b and an element x as arguments, and adds x at the end of the queue.
- pop(b): takes a ring buffer b as argument, pops off its head (first element in the queue), and returns it.

Verification tasks:

1. *Safety*: Verify that every array access is made within bounds.
2. *Behavior*: Verify the correctness of your implementation w. r. t. the first-in first-out semantics of a queue.
3. *Harness*: The following test harness should be verified:

```
Test(x: int, y: int, z: int) :=
      b <- create(2);
      push(b, x);
      push(b, y);
      h <- pop(b); assert h = x;
      push(b, z);
      h <- pop(b); assert h = y;
      h <- pop(b); assert h = z;
```

Problem analysis and architecture proposal:

The *safety* verification task is automatically verified. Indeed, proof obligations are generated (during the proving stage) so accesses are done within bounds. Overflows are also verified by generated proof obligation.

The *behavior* verification task is done by construction. The construction of the program is based on the properties expected; this is the most important phase for modeling in B: the complexity and the number of proof obligations generated depend on this phase.
The concept of the first-in first-out semantics of a queue is a concept of order of the elements. The only way to model it in B is to use sequences: the array of integers in the most abstracted machine will be a sequence of integers. A sequence is not an implementable type, so the implementation of the ring buffer is obtained by refining progressively the abstract machine: we have to reflect about a strategy of refinement.

'Gluing invariants' are used to link the variables of two different abstractions (a refinement to its abstraction, an implementation to its refinement and maybe a refinement to its refinement).

The first refinement consists of the linking ('gluing') of the sequence to an abstract array of the same size (nb: the first index of an array that is a refined sequence is 1). The operations defined in the abstraction also have to be refined since the sequence became an array. Note that for those two machines only two variables are needed: the sequence (DATA) and its size (LEN).

The current problem is to get an implementable ring (let DATA_i be this implementable ring) defined as DATA_i: 1..SIZE --> INT that respects the notion of order: the only way to solve this problem is to introduce a variable FIRST_i that 'keeps' the order of the elements. But this solution brings a new problem: how can we link the refinement and the implementation? The link between all the variables is possible but quite complex. In order to reduce this complexity, we have the possibility to introduce a new refinement between the first one and the implementation.

Let DATA_r be the refined DATA variable. We also add the variable FIRST_r which, once refined, will give FIRST_i.
DATA_r's domain is not hard to build if we have the variable FIRST_r. Indeed, the link between the DATA array (with DATA(1) the first element of the sequence) and the DATA_i array (with DATA(FIRST_i) the first element of the sequence) is DATA_r with DATA_r: FIRST_r..FIRST_r+LEN --> INT.
For this problem all the 'gluing invariants' are given and explained in the 'Software modeling' section.

The *Harness* verification task would be done in the 'Generated code' section.

About the program architecture:

SIZE can be a formal parameter of the machine, since it is a constant and the machine's operations use it. Therefore the machine Ring can be imported by another one. The fact that SIZE is given as a formal parameter of the machine force us to add a refinement for the implementation, because DATA_i is defined on 1..SIZE and the upper bound is not known at the compilation. We will have to add in the DEFINITION clause 'max_size', a static variable (with SIZE <= max_size in the CONSTRAINTS clause), so the B0 checker allocate a bigger array than SIZE. Thus a 'gluing invariant' is added so the operations respect the bounds.
Adding this layer (of refinement) may be useful: The target language can be decided at the end of the implementation. Indeed, DATA_B0 lower bound can be 0 or 1 by slightly changing the gluing invariant. Both languages will be tested in the 'Generated code' section.

Global vision:

In this project, refinements are in fact a progressive adaption of the variables to match a final expected form. For each refinement, the variables DATA's state slowly changes to get closer to its final state: the ring.

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

In order to get through all the details of the modeling process, this section is divided similarly to it:

- Definition of all the components (machines), the variables and the gluing invariants;
- Proof the definitions are correct;
- Addition of the operations;
- Proof of the POs and the added rules.

The SIZE variable given as a formal parameter of the machine is slightly changed: Majuscules are used for sets given as formal parameter, and ‘size’ is a keyword for the atelier B. SIZE is renamed ‘ring_size’.

A. Definitions

As seen in the *problem analysis* of the problem, there are four gluing invariants for the DATA variable:

1) Implicit gluing invariant from DATA: seq(INT) to DATA: 1..LEN --> INT;
2) Explicit gluing invariant from DATA: 1..LEN --> INT to DATA_r: FIRST..FIRST+LEN-1 --> INT;
3) Explicit gluing invariant from DATA_r: FIRST..FIRST+LEN-1 --> INT to DATA_i: 1..ring_size --> INT;
4) Explicit gluing invariant from DATA_i: 1..ring_size --> INT to DATA_B0: 1..max_size --> INT;

and two for the FIRST variable:

5) Explicit gluing invariant from FIRST: NATURAL1 to FIRST_i: 1..ring_size;
6) Implicit gluing invariant from FIRST_i: 1..SIZE to FIRST_i: 1..ring_size (or FIRST_i: 0..ring_size-1).

From an array I to an array J, the elements are the same, only their indexes change following a mathematical transformation: a transposition. Therefore lambda expressions are used to describe the gluing invariants.

1) The gluing invariant is implicit. This means the variables in the abstraction and the refinement are homonyms (same name, same type). The refined variable is a concretization of the abstracted one.

2) The two arrays are identical regarding the elements, not the indexes. Do not forget that FIRST: NATURAL1 because the notion of ring is not present yet.
We want DATA(1)=DATA_r(FIRST), DATA(2)=DATA_r(FIRST+1), …
The gluing invariant is the following:

DATA = %JJ.(JJ: 1..LEN | DATA_r(FIRST+JJ-1)).

3) The notion of ring is introduced in this refinement. FIRST and DATA_r are refined here. The expression of the gluing invariant does not depend on FIRST_i, but it will be easier to understand it after the 5) gluing invariant is done.

4) This is the refinement concerning the B0 checker adaptation. The two arrays DATA_i and DATA_B0 are similar, except that DATA_B0 contains more elements. A mere domain's restriction is enough to define the gluing invariant:

1..ring_size <| DATA_B0 = DATA_i

5) FIRST: NATURAL1 and FIRST_i: 1..ring_size. The problem is that FIRST_i's lower bound is one. If we apply a modulo and if FIRST is a multiple of ring_size the we will get FIRST_i = 0 instead of ring_size. The trick is to deduce one to the FIRST variable, apply the modulo, then re-add one to FIRST.
The gluing invariant is:

FIRST_i = ((FIRST-1) mod ring_size)+1

6) This is an implicit gluing invariant if we want the code generated to be in Ada. If the code expected is in another language then FIRST_B0 = FIRST_i-1.

3) So, for an index JJ in the domain of DATA_r, we want the index of DATA_i to follow the same relation than for the 5) gluing invariant.
The lambda expression is:

%JJ.(JJ: FIRST..FIRST+LEN-1 | DATA_i(((JJ -1) mod ring_size)+1)) = DATA_r

The variables of the machines have to be initialized. The initialization has to be correct regarding the variables' type and the gluing invariants. This is obvious for the ring to be empty when initialized.
Before adding any operation, we prove the model is correct. The POG generates 21 POs, and they are all discharged by the force 0, 1 or predicate provers.

Now that the variables are defined and linked, we can add the operations.

B. Operations

We have to define four operations:

- Clear
- rr <-- Head
- Push(xx)
- rr <-- Pop

The first two ones are easier than the last two.

i. Machine Ring

```
OPERATIONS
   Clear =
   BEGIN
      DATA := [] ||
      LEN := 0
   END;

   rr <-- Head =
   PRE
      rr: INT &
      not(LEN=0)
   THEN
      rr := first(DATA)
   END;

   Push(xx) =
   PRE
      xx: INT &
      not(ring_size <= LEN)
   THEN
      DATA := DATA <- xx ||
      LEN := LEN+1
   END;

   rr <-- Pop =
   PRE
      rr: INT &
      not(LEN=0)
   THEN
      rr := first(DATA) ||
      DATA := tail(DATA) ||
      LEN := LEN-1
   END
```

The operations pre-conditions are obvious: we cannot add an element if the ring is full, as we cannot read/remove the first one if the ring is empty.
The post-conditions are defined with the sequences operators since DATA is a sequence.
The variable FIRST does not exist yet.

ii. Machine Ring_r

```
OPERATIONS
   Clear =
   BEGIN
      DATA := {} ||
      LEN := 0
   END;

   rr <-- Head =
   BEGIN
      rr := DATA(1)
   END;

   Push(xx) =
   BEGIN
      DATA, LEN:
      (DATA: 1..LEN$0+1 --> INT &
       DATA = DATA$0<+{LEN$0+1 |-> xx} &
       LEN=LEN$0+1)
   END;

   rr <-- Pop =
   BEGIN
      rr, DATA, LEN:
      (DATA: 1..LEN --> INT &
       rr = DATA$0(1) &
       LEN=LEN$0-1 &
       DATA = %jj.(jj: 1..LEN$0-1 | DATA$0(jj+1))
      )
   END
```

Operations are refined with the new variables.
DATA is an array so operations have to be refined with array operators/definitions. The $0 suffix is used to define the state of a variable before entering the operations. For instance, in the operations Pop, rr = DATA$0(1) means rr takes the value of the element 1 of the initial array.

iii. Machine Ring_2r

```
OPERATIONS
   Clear =
   BEGIN
      DATA_r := {} ||
      LEN := 0
   END;

   rr <-- Head =
   BEGIN
      rr := DATA_r(FIRST)
   END;

   Push(xx) =
   BEGIN
      DATA_r, LEN:
      (DATA_r: FIRST..FIRST+LEN$0 --> INT &
       DATA_r = DATA_r$0<+{FIRST+LEN$0 |-> xx} &
       LEN=LEN$0+1)
   END;

   rr <-- Pop =
   BEGIN
      rr, DATA_r, FIRST, LEN:
      (rr=DATA_r$0(FIRST$0) &
       LEN=LEN$0-1 &
       FIRST=FIRST$0+1 &
       DATA_r: FIRST$0+1..FIRST$0+LEN$0-1 --> INT &
       DATA_r = %jj.(jj: FIRST$0+1..FIRST$0+LEN$0-1
                                       | DATA_r$0(jj))
   )
   END
```

iv. Machine Ring_2r_r

```
OPERATIONS
   Clear =
   BEGIN
      LEN := 0
   END;

   rr <-- Head =
   BEGIN
      rr := DATA_i(FIRST_i)
   END;

   Push(xx) =
   BEGIN
      DATA_i, LEN :
      (DATA_i: 1..ring_size --> INT &
       ((FIRST_i+LEN$0<=ring_size)
            => DATA_i=(DATA_i$0
                     <+{FIRST_i+LEN$0 |-> xx})) &
       (not(FIRST_i+LEN$0<=ring_size)
            => DATA_i=(DATA_i$0
                     <+{FIRST_i+LEN$0-ring_size |-> xx})) &
       LEN=LEN$0+1
      )
   END;

   rr <-- Pop =
   BEGIN
      rr, DATA_i, FIRST_i, LEN:
      (DATA_i: 1..ring_size --> INT &
       DATA_i = DATA_i$0 &
       rr=DATA_i$0(FIRST_i$0) &
       FIRST_i = FIRST_i$0 mod (ring_size)+1 &
       LEN=LEN$0-1)
   END
```

The circularity of the ring is introduced here. We have now to be careful regarding the indexes: two cases appear in the Push operation: the case where FIRST_i+LEN is over than ring_size (i.e. the next element has to be placed at the beginning of the ring) and the case it is not. We also have to be careful when FIRST_i is incremented in the Pop operation.

v. Machine Ring__i

```
OPERATIONS
   Clear =
   BEGIN
      LEN := 0
   END;
   rr <-- Head =
   BEGIN
      rr := DATA_B0(FIRST_i)
   END;

   Push(xx) =
   VAR diff IN
      diff := ring_size-LEN;
      IF(FIRST_i<=diff) THEN
         DATA_B0(FIRST_i+LEN) := xx
      ELSE
         DATA_B0(FIRST_i-diff) := xx
      END;
      LEN := LEN+1
   END;

   rr <-- Pop =
   BEGIN
      rr := DATA_B0(FIRST_i);
      FIRST_i := FIRST_i mod (ring_size)+1;
      LEN := LEN-1
   END
```

This is the implementation for a code generation in Ada. A few changes are needed for this implementation if the gluing invariant for the DATA_B0 array is adapted for other languages (if arrays indexes start at the index 0).

III. Proofs obligations

162 POs are generated by the POG. 118 POs are discharged by the force 0 prover, 7 by the force 1 prover, 5 by the force 2 prover.

6 rules are added and proved: 1 by the rules validator, 5 by ‘hands’ (most of them slightly change from a refinement to another).

A. Functional proof obligations

i. Component Ring_r’s proof

There are 5 POs left for this component. 4 of them can be discharged by the predicate prover.
In order to discharge the last one, call in force 2 the predicate prover after the prover (and with all the hypotheses in the stack of hypotheses).

ii. Component Ring_2r’s proof

There are two POs left for the Push operation, and two other POs for the Pop operation.

Push operation:

The first PO needs a rule to be added.

```
"`Local hypotheses'" &
DATA_r$2: FIRST$1..FIRST$1+LEN$1 +-> INT &
dom(DATA_r$2) = FIRST$1..FIRST$1+LEN$1 &
DATA_r$2 = DATA_r$1<+{FIRST$1+LEN$1|->xx} &
LEN$2 = LEN$1+1 &
"`Check that the invariant (%jj.(jj: 1..LEN$1 | DATA_r$1(FIRST$1+jj-1)) = DATA) is
preserved by the operation - ref 4.4, 5.5'"
=>
%jj.(jj: 1..LEN$2 | DATA_r$2(FIRST$1+jj-1)) = DATA<+{LEN$1+1|->xx}
```

After adding the hypotheses to the stack of hypotheses, replace LEN$2, DATA_r$2 and DATA by theirs expressions using the command **eh**. The current goal should be:

```
%jj.(jj: 1..LEN$1+1 | (DATA_r$1<+{FIRST$1+LEN$1|->xx})(FIRST$1+jj-1))
=
%jj.(jj: 1..LEN$1 | DATA_r$1(FIRST$1+jj-1))<+{LEN$1+1|->xx}
```

And this expression is correct if for jj = LEN$1+1 we have the equality: FRIST$1+jj-1 = LEN$1+1.

In fact, the demonstration is done by the substitution of a variable.
Let f(x) be this substitution. Consider the following formula:

```
%jj.(jj: a..b | g( f(jj) )) <+{ d |->x}
```

Applying the substitution on d allows us to add the overload substitution inside the lambda expression:

```
%jj.(jj: a..b | g( f(jj) )) <+{ d |->x} = %jj.(jj: a..b\/c | (g<+{ f(d) |-> x})(f(jj)) )
```

In our case, the substitution is f(jj) = FIRST$1+jj-1 and we have d = LEN$1+1.

The equality f(LEN$1+1) = FIRST$1+LEN$1 is verified:

f(LEN$1+1) = FIRST$1+(LEN$1+1)-1 = FIRST$1+LEN$1.

In order to discharge this PO, add the following rule in the pmm associated file:

```
THEORY Lambda-Overload IS
    f+b = c
    =>
    %j.(j: 1..b+1 | (g<+{c|->x})(f+j-1))
       =
    %j.(j: 1..b | g(f+j-1))<+{b+1|->x}
END
```

Then apply this rule 'Once' and call the prover in order to prove the precondition. Note that the precondition of the rule is simplified: f(b+1) = f+(b+1)-1 = c => f+b = c.

The tree of commands for this PO is:

```
dd
    eh(DATA, _h, Goal)
        eh(LEN$2, _h, Goal)
            eh(DATA_r$2, _h, Goal)
                ar(Lambda-Overload.1, Once)
                    pr
```

The other PO of the Push operation is discharged by calling the predicate prover after the hypotheses are added to the stack of hypotheses. The commands for this PO are dd(0) & pp(rt.1).

Pop operation:

The two left POs of this operation are easily discharged. Both are different, but can have the same demonstration. The tree of commands is:

```
dd(0)
    mp
        ah(not(FIRST$1=0))
            pp(rt.1|10)
```

iii. Component Ring_2r_r's proof

This is the refinement where the circularity of the array is done.

Clear operation:

We have to prove without any hypotheses:

%jj.(jj: FIRST..FIRST+0-1 | DATA_i$1((jj-1) mod ring_size+1)) = {}

To do to, after the hypotheses are added to the stack of hypotheses (command **dd**), add the hypothesis that the set 'FIRST..FIRST+0-1' is empty, and prove it using the command **pr**. A second call of the prover will discharge the PO.

Push operation:

```
"`Local hypotheses'" &
DATA_i$2: 1..ring_size +-> INT &
dom(DATA_i$2) = 1..ring_size &
FIRST_i$1+LEN$1<=ring_size => DATA_i$2 = DATA_i$1<+{FIRST_i$1+LEN$1|->xx} &
not(FIRST_i$1+LEN$1<=ring_size) => DATA_i$2 = DATA_i$1<+{FIRST_i$1+LEN$1-
ring_size|->xx} &
LEN$2 = LEN$1+1 &
"`Check operation refinement - ref 4.4, 5.5'"
=>
dom(DATA_r<+{FIRST+LEN$1|->xx}) = FIRST..FIRST+LEN$1
```

Add the hypotheses to the stack of hypotheses.

At this step, if we call the prover, a new goal to prove is added in the tree of the demonstration: use the miniprover instead (**mp**). Then replace dom(DATA_r) by its value (command **eh**). We get the same goal a call to the prover would have, but without the added goal to prove.

The current goal should be:

FIRST..FIRST+LEN$1-1\/{FIRST+LEN$1} = FIRST..FIRST+LEN$1

Rules exist in the atelier B that can solve this kind of goal. Indeed, execute the command sr(All, a..b\/c) to find all the rules that match the same goal.

You can see that a needed hypothesis to apply such rules is to have FIRST <= FIRST+LEN$1-1. This is not the case when LEN$1 = 0. Therefore we have to do two cases: LEN$1 = 0 and not(LEN$1=0).

The case LEN$1=0 is discharged by the predicate prover (force 0).

For the other case, you first have to show that FIRST <= FIRST+LEN$1-1 before calling the predicate prover.

The tree of commands for this PO is the following:

```
dd(0)
    mp
        eh(dom(DATA_r), _h, Goal)
            dc(LEN$1=0)
                pp(rt.0)
            dd
                ah(FIRST<=FIRST+LEN$1-1)
                    pr
                        pp(rt.0)
```

```
"`Local hypotheses'" &
DATA_i$2: 1..ring_size +-> INT &
dom(DATA_i$2) = 1..ring_size &
FIRST_i$1+LEN$1<=ring_size => DATA_i$2 = DATA_i$1<+{FIRST_i$1+LEN$1|->xx} &
not(FIRST_i$1+LEN$1<=ring_size) => DATA_i$2 = DATA_i$1<+{FIRST_i$1+LEN$1-
ring_size|->xx} &
LEN$2 = LEN$1+1 &
"`Check that the invariant (%jj.(jj: FIRST..FIRST+LEN$1-1 | DATA_i$1((jj-1) mod
ring_size+1)) = DATA_r) is preserved by the operation - ref 4.4, 5.5'"
=>
%jj.(jj: FIRST..FIRST+LEN$2-1 | DATA_i$2((jj-1) mod ring_size+1)) =
DATA_r<+{FIRST+LEN$1|->xx}
```

For this PO, we will have to simplify DATA_i$2 in the goal: two cases appear; the case when FIRST_i$1 + LEN$1 <= ring_size, and the case it is not.

This PO uses the previous demonstrated rule (Lambda-Overload theory), but adapted to a new substitution of variable: in this case f(jj) = (jj-1) mod ring_size+1.

First of all, we will simplify the goal so we will not have to do it twice (the hypotheses proved doing the first case have to be reproved doing the second case)

This is important to add the hypotheses to the stack of hypotheses using the command **dd** only, because the command **dd(0)** generates derived hypotheses and the DATA_i$2 expressions will not be available (even with the modus ponens command, **mh**).

Replace now the value of the LEN$2 and DATA_r variables, using the command **eh**. You can simplify the expression FIRST+(LEN$1+1)-1 by FIRST+LEN$1 but you have to demonstrate that FIRST+(LEN$1+1)-1=FIRST+LEN$1 (add this hypothesis, call the prover and replace the expression in the goal). We can start doing the cases: start the demonstration by cases using the command **dc** (use then the command **dd** to add the hypothesis 'FIRST_i$1 + LEN$1 <= ring_size' to the stack of hypotheses)

Case FIRST_i$1 + LEN$1 <= ring_size:
Re-use the rule added for the previous refinement, do not forget to adapt the precondition with the substitution (not that the jokers are also adapted):

```
THEORY Lambda-Overload IS

    (a+b-1) mod s+1 = c
    =>
    %j.(j: a..a+b | (g<+{c|->x})((j-1) mod s+1))
       =
    %j.(j: a..a+b-1 | g((j-1) mod s+1))<+{a+b|->x};

END
```

You can apply this rule using the command **ar**. We have to prove the preconditions so this rule is applied, this is why the current goal becomes:

(FIRST+LEN$1-1) mod ring_size+1 = FIRST_i$1+LEN$1

A new rule has to be added:
Hypotheses:

f = (k*s + i) with k ≥ 0;
i + l ≤ s.

Property we want to prove:

((f+l-1) mod s)+1 = i+l

Demonstration:

((f+l-1) mod s)+1 = ((k*s+i+l-1) mod s)+1 = ((i+l-1) mod s)+4.

And i+l ≤ s implies i+l-1 < s.

=> ((i+l-1) mod s) = i+l-1+1 = i+l

The property is proved so the following rule can be added:

```
i<=s-l &
i = (f-1) mod s+1
=>
(f+l-1) mod s+1 = i+l
```

Note that f = (k*s + i) and i = (f-1) mod s+1 is the same hypothesis:

i = (f-1) mod s+1
=> i = (k*s + i -1) mod s+1
=> i = (i -1) mod s+1

We now that i, by definition, is lesser than or equal to s so (i-1) is strictly lesser than s.

=> i = i-1+1 = i.

Apply this new rule on the current goal and call the prover twice so the preconditions are proved.

Case not(FIRST_i$1 + LEN$1 <= ring_size):
Do not forget to add the hypothesis not(FIRST_i$1 + LEN$1 <= ring_size) when this case starts.
The demonstration of this case is very similar to the previous one, except that the precondition to prove after Lambda-Overload.1 is applied differs:

(FIRST+LEN$1-1) mod ring_size+1 = FIRST_i$1+LEN$1-ring_size

And another rule has to be added:
Hypotheses:

f = (k*s + i) with k ≥ 0;
not(i + l ≤ s);
i ≤ s;
l ≤ s.

Property we want to prove:

((f+l-1) mod s)+1 = i+l-s

Demonstration:

((f+l-1) mod s)+1 = ((k*s+i+l-1) mod s)+1 = ((i+l-1) mod s)+1.
i ≤ s and l ≤ s gives 2*s > i+l-1.
Then the hypothesis not(i + l ≤ s) gives i+l > s or i+l-1 >= s.
From the inequality 2*s > i+l-1 ≥ s, we get s > i+l-1-s > 0.
=> ((i+l-1) mod s)+1 = i+l-1-s +1 = i+l-s.

The property is proved so the following rule can be added:

```
not(i<=s-l) &
i = (f-1) mod s+1 &
l <= s
=>
(f+l-1) mod s+1 = i+l
```

Using the predicate prover and the prover after this rule is applied discharges the PO.

The tree of commands for this PO is:

```
dd
    eh(LEN$2, _h, Goal)
        ah(FIRST+(LEN$1+1)-1 = FIRST+LEN$1)
            pr
            dd
                eh(FIRST+(LEN$1+1)-1, _h, Goal)
                    eh(DATA_r, _h, Goal)
                        dc(FIRST_i$1+LEN$1 <= ring_size)
                            dd
                                ar(Lambda-Overload.1, Once)
                                    ar(Lambda-Overload.2, Once)
                                        pr
                                        pr
                            dd
                                ar(Lambda-Overload.1, Once)
                                    ar(Lambda-Overload.3, Once)
                                        ah(not(FIRST_i$1+LEN$1 <= ring_size))
                                            pp(rt.0)
                                        pr
                                        pr
```

The last PO to prove for this operation is to show LEN$2: 0..ring_size.
When all the hypotheses are added to the stack of hypotheses, replace LEN$2 by the expression LEN$1+1 and call the prover. The PO is discharged.

The tree of commands for this PO is:

```
dd(0)
    eh(LEN$2, _h, Goal)
        pr
```

Pop operation:

The two first POs are easy to prove, only theirs trees of commands is given.

```
"`Local hypotheses'" &
DATA_i$2: 1..ring_size +-> INT &
dom(DATA_i$2) = 1..ring_size &
DATA_i$2 = DATA_i$1 &
rr$2 = DATA_i$1(FIRST_i$1) &
FIRST_i$2 = FIRST_i$1 mod ring_size+1 &
LEN$2 = LEN$1-1 &
"`Check operation refinement - ref 4.4, 5.5'"
=>
%jj.(jj: FIRST+1..FIRST+LEN$1-1 | DATA_r(jj)): FIRST+1..FIRST+LEN$1-1 +-> INT
```

The tree of commands for this PO is:

```
dd(0)
    pr
        pr
```

```
"`Local hypotheses'" &
DATA_i$2: 1..ring_size +-> INT &
dom(DATA_i$2) = 1..ring_size &
DATA_i$2 = DATA_i$1 &
rr$2 = DATA_i$1(FIRST_i$1) &
FIRST_i$2 = FIRST_i$1 mod ring_size+1 &
LEN$2 = LEN$1-1 &
"`Check that the invariant (FIRST_i$1: 1..ring_size) is preserved by the operation - ref 4.4, 5.5'"
=>
FIRST_i$2: 1..ring_size
```

The tree of commands for this PO is:

```
dd
    mp
        pr
            eh(FIRST_i$2, _h, Goal)
                ar(b1.60, Once)
```

```
"`Local hypotheses'" &
DATA_i$2: 1..ring_size +-> INT &
dom(DATA_i$2) = 1..ring_size &
DATA_i$2 = DATA_i$1 &
rr$2 = DATA_i$1(FIRST_i$1) &
FIRST_i$2 = FIRST_i$1 mod ring_size+1 &
LEN$2 = LEN$1-1 &
"`Check that the invariant (FIRST_i$1 = (FIRST-1) mod ring_size+1) is preserved by the
operation - ref 4.4, 5.5'"
=>
FIRST_i$2 = (FIRST+1-1) mod ring_size+1
```

This PO needs a rule to be added to the pmm associated file. Indeed, this is impossible for the prover after the simplifications are done to prove the goal:

((FIRST-1) mod ring_size+1) mod ring_size = FIRST mod ring_size

Property to prove:

a mod b = (((a-1) mod b) +1) mod b.

Demonstration:
Let rewrite a = k*b + r with k: NAT and 0 ≤ r < b. The left expression becomes:

a mod b = (k*b +r) mod b = r

And the right expression becomes:

(((a-1) mod b) +1) mod b = (((k*b + r -1) mod b) +1) mod b

There are two possibilities from here: r = 0 and r ≥ 1.

Case r = 0:

(((a-1) mod b) +1) mod b = (((k*b -1) mod b) +1) mod b

Let k' be k-1, the equality becomes:

=> (((a-1) mod b) +1) mod b = (((k'*b + (b-1)) mod b) +1) mod b

Since b-1 < b, we get

=> (((a-1) mod b) +1) mod b = ((b-1)+1) mod b = b mod b = 0.

=> a mod b = 0 = (((a-1) mod b) +1) mod b.

The case is verified.

Case r ≥ 1:

(((a-1) mod b) +1) mod b = (((k*b + r-1) mod b) +1) mod b

=> (((a-1) mod b) +1) mod b = (((r-1) mod b) +1) mod b

Since r < b then r-1 < b.

=> (((a-1) mod b) +1) mod b = ((r-1)+1) mod b = r mod b = r.

=> a mod b = r = (((a-1) mod b) +1) mod b.

This case is verified.

The property is verified, you can add the following rule in the pmm associated file:

```
a mod b = (((a-1) mod b) +1) mod b
```

Using it as a rewriting rule is not wise because the prover will loop.

The left of the PO is easy: add the hypothesis 'FIRST mod ring_size = ((FIRST-1) mod ring_size+1) mod ring_size' and prove it using the just added rule (command **ar**). A call to the prover will discharge the PO.

The tree of commands for this PO is:

```
dd(0)
    eh(FIRST_i$2, _h, Goal)
        mp
            ah(FIRST mod ring_size = ((FIRST-1) mod ring_size+1) mod ring_size)
                ar(Lambda-Overload.4)
                pr
```

iv. Component Ring   i's proof

The force 3 prover discharges 3 POs on the 16 lefts, and the predicate prover discharges one PO (initialisation). Twelve PO are left:

- 8 for the Push operation (4 for each case);
- 5 for the Pop operation: they concern the FIRST_i variable's bounds.

Push operation:

There are four POs for the case 'FIRST_i$1+LEN$1<=ring_size' and four POs for the the case 'not(FIRST_i$1+LEN$1<=ring_size)'. The goals are similar, so are the demonstrations. We will demonstrate the second case: not(FIRST_i$1+LEN$1<=ring_size).

```
"`Local hypotheses'" &
not(FIRST_i$1<=ring_size-LEN$1) &
"`Check preconditions of called operation, or While loop construction, or Assert predicates'"
=>
FIRST_i$1-(ring_size-LEN$1): dom(DATA_B0$1)
```

Add all the hypotheses to the stack of hypotheses using the command **dd**.
In order to avoid the replacement of FIRST_i$1 by its value, use the command **mp**.

The goal should look like: 'aa: 1..1000' with 1000 the value of max_size defined in the definition clause. This kind of goal is demonstrate by proving '1 <= aa' and 'aa <= 1000'.

Add the hypothesis 'not(FIRST_i$1<=ring_size-LEN$1)' and call the predicate prover to discharge this sub-goal and get the other one.
The other sub-goal is prove by adding the typing definition of the variables FIRST_i$1 (: 1.. ring_size), LEN$1 and ring_size. Add the property 'not(FIRST_i$1<=ring_size-LEN$1)' and call the predicate prover: the PO is discharged.

The tree of commands for this PO is:

```
dd
   eh(dom(DATA_B0$1), _h, Goal)
      mp
         ah(not(FIRST_i$1<=ring_size-LEN$1))
            pp(rt.0)
         ah(FIRST_i$1: 1..ring_size)
            ah(LEN$1: 1..ring_size)
               ah(ring_size: 1..1000)
                  pr
                     ah(not(FIRST_i$1<=ring_size-LEN$1))
                        pp(rt.0)
```

```
"`Local hypotheses'" &
not(FIRST_i$1<=ring_size-LEN$1) &
"`Check that the invariant (FIRST_i = FIRST_i$1) is preserved by the operation - ref 4.4, 5.5'"
=>
1..ring_size<|(DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}): 1..ring_size +-> INT
```

Before starting this PO, we search rules that can help thanks to the command **sr**. The goal of this PO contains an expression similar to a <| f, so searching a rule that contains this expression could help. The rule InFunctionXY.32 and the rule InFunctionXY.8 have a goal totally similar, but only one will be useful. Indeed, InFunctionXY.32 will not be used since the precondition is not true; we do not have:

DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}: 1..1000

Because dom(DATA_B0$1) is not included in 1..ring_size\/{FIRST_i$1-(ring_size-LEN$1)}.

So the rule InFunctionXY.8 seems good: we have three easy preconditions to prove.
First of all, add the hypotheses to the stack of hypotheses.
Then add the hypothesis 'DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}: 1..1000 +-> INT' that has to be in the stack before the rule is applied.
The proof of this hypothesis is done using the hypothesis 'not(FIRST_i$1<=ring_size-LEN$1', the data typing of the variables and the predicate prover (this is a like the demonstration of the previous PO).

Now the hypothesis 'DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}: 1..1000 +-> INT' is proved, add it in the stack of hypotheses using the command **dd** then apply the rule InFunctionXY.8.
The prover is able to prove the preconditions of the rule and the PO is discharged.

The tree of commands for this PO is the following:

```
dd
    ah(DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}: 1..1000 +-> INT)
        pr
            ah(not(FIRST_i$1<=ring_size-LEN$1))
                pp(rt.0)
            ah(FIRST_i$1: 1..ring_size)
                ah(LEN$1: 0..ring_size)
                    ah(ring_size: 2..1000)
                        ah(not(FIRST_i$1<=ring_size-LEN$1))
                            pp(rt.0)
                pr
            dd
                ar(InFunction.8, Once)
                    pr
                    pr
```

```
"`Local hypotheses'" &
not(FIRST_i$1<=ring_size-LEN$1) &
not(FIRST_i$1+LEN$1<=ring_size) &
"`Check that the invariant (FIRST_i = FIRST_i$1) is preserved by the operation - ref 4.4, 5.5'"
=>
1..ring_size<|(DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}) =
DATA_i<+{FIRST_i$1+LEN$1-ring_size|->xx}
```

This PO needs a rule to be added. Add the following rule to the pmm associated file:

```
b: 1..a
=>
1..a<|(f<+{b|->x}) = 1..a<|f<+{b|->x}
```

This rule is validated by the rules validator (project > Validate Rules).

In order to discharge the PO, add all the hypotheses to the stack of hypotheses then replace DATA_i by its expression.
Prove the equality ‘FIRST_i$1+LEN$1-ring_size=FIRST_i$1-(ring_size-LEN$1)’ and replace one of the two expressions.
Apply the rule and prove the precondition (that is similar to the first PO).

The tree of commands for this PO is the following:

```
dd
    eh(DATA_i,_h,Goal)
        ah(FIRST_i$1+LEN$1-ring_size = FIRST_i$1-(ring_size-LEN$1))
            pr
            dd
                eh(FIRST_i$1-(ring_size-LEN$1),_h,Goal)
                    ar(Lambda-Overload.1,Once)
                        mp
                            ah(not(FIRST_i$1+LEN$1<=ring_size))
                                pp(rt.0)
                            ah(LEN$1: 0..ring_size)
                                ah(FIRST_i$1: 1..ring_size)
                                    ah(ring_size: 2..1000)
                                        ah(not(FIRST_i$1+LEN$1<=ring_size))
                                            pp(rt.0)
```

```
"`Local hypotheses'" &
not(FIRST_i$1<=ring_size-LEN$1) &
"`Check that the invariant (DATA_B0$1: 1..1000 --> INT) is preserved by the operation - ref 4.4,
5.5'"
=>
dom(DATA_B0$1<+{FIRST_i$1-(ring_size-LEN$1)|->xx}) = 1..1000
```

This PO is easily demonstrated: we only have to prove that 'FIRST_i$1-(ring_size-LEN$1): 1..1000'.
This is similar to the first PO proved.

Pop operation:

The demonstrations of all the POs are done by using manually three rules: b1.59, b1.60 and b1.46.
You can find these rules using the command **sr**(All, a mod b).
In order to use these rules, prove theirs precondition(s) before using the command **ar**(b1.XX, Once).

B. Well-definedness proof obligations

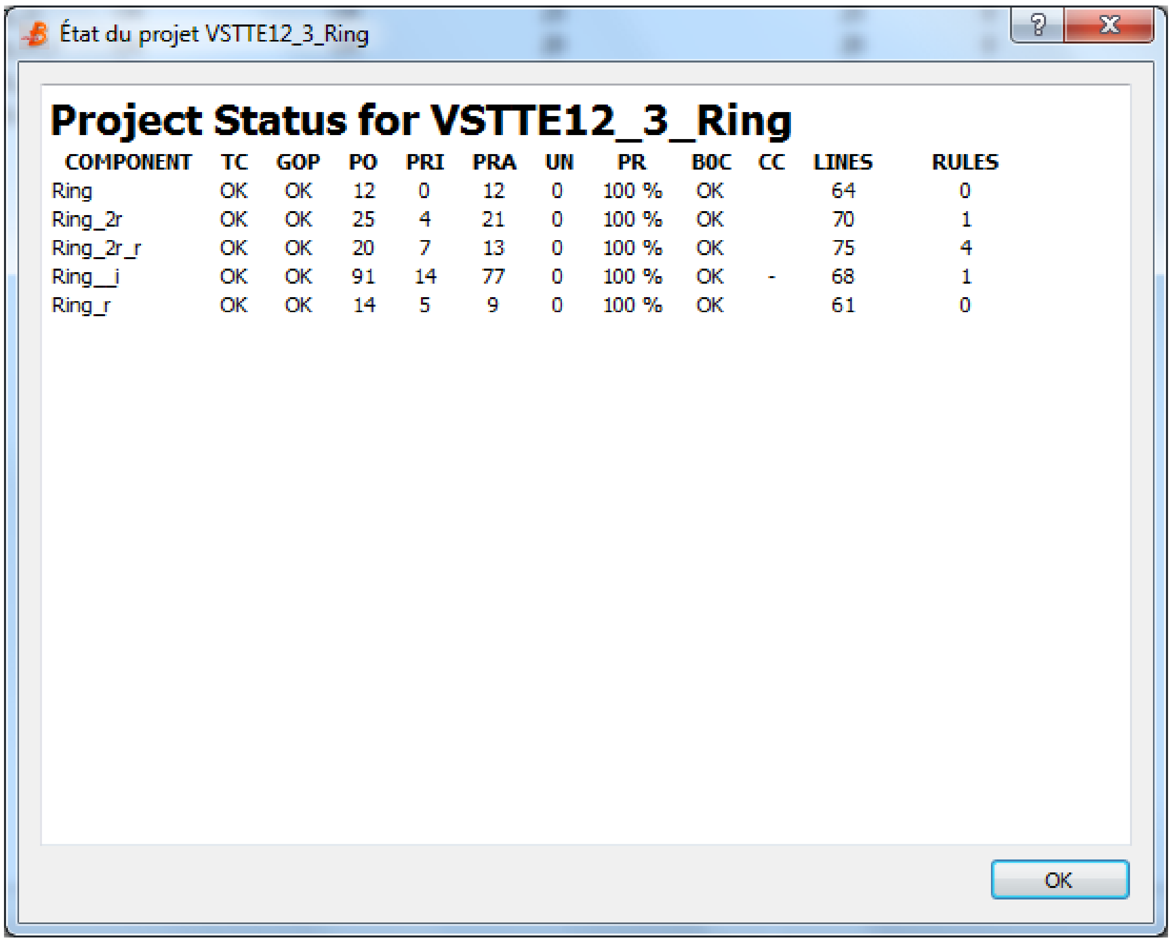
27 POs of well-definedness are generated. 24 are discharged by the force 0 prover, 2 are discharged by the force 2 prover.

The only left PO concerns the Ring_2r_r component.

```
"`Local hypotheses'" &
FIRST_i: 1..ring_size &
FIRST_i = (FIRST-1) mod ring_size+1 &
DATA_i: 1..ring_size +-> INT &
dom(DATA_i) = 1..ring_size &
jj: FIRST..FIRST+LEN-1
=>
(jj-1) mod ring_size+1: dom(DATA_i)
```

The expression ‘(jj-1) mod ring_size+1’ has to be encapsulated by an abstract variable (command **ae**: abstract expression).
Replace ‘dom(DATA_i)’ by its value then call the miniprover so the current goal becomes ‘1 <= aa’.
Replace ‘aa’ by its expression (command **eh**) and call the prover to discharge this sub-goal. The new sub-goal is ‘aa <= 1000’. Replace ‘aa’ by its value and apply the rule b1.60: the PO is discharged.

C. Project status



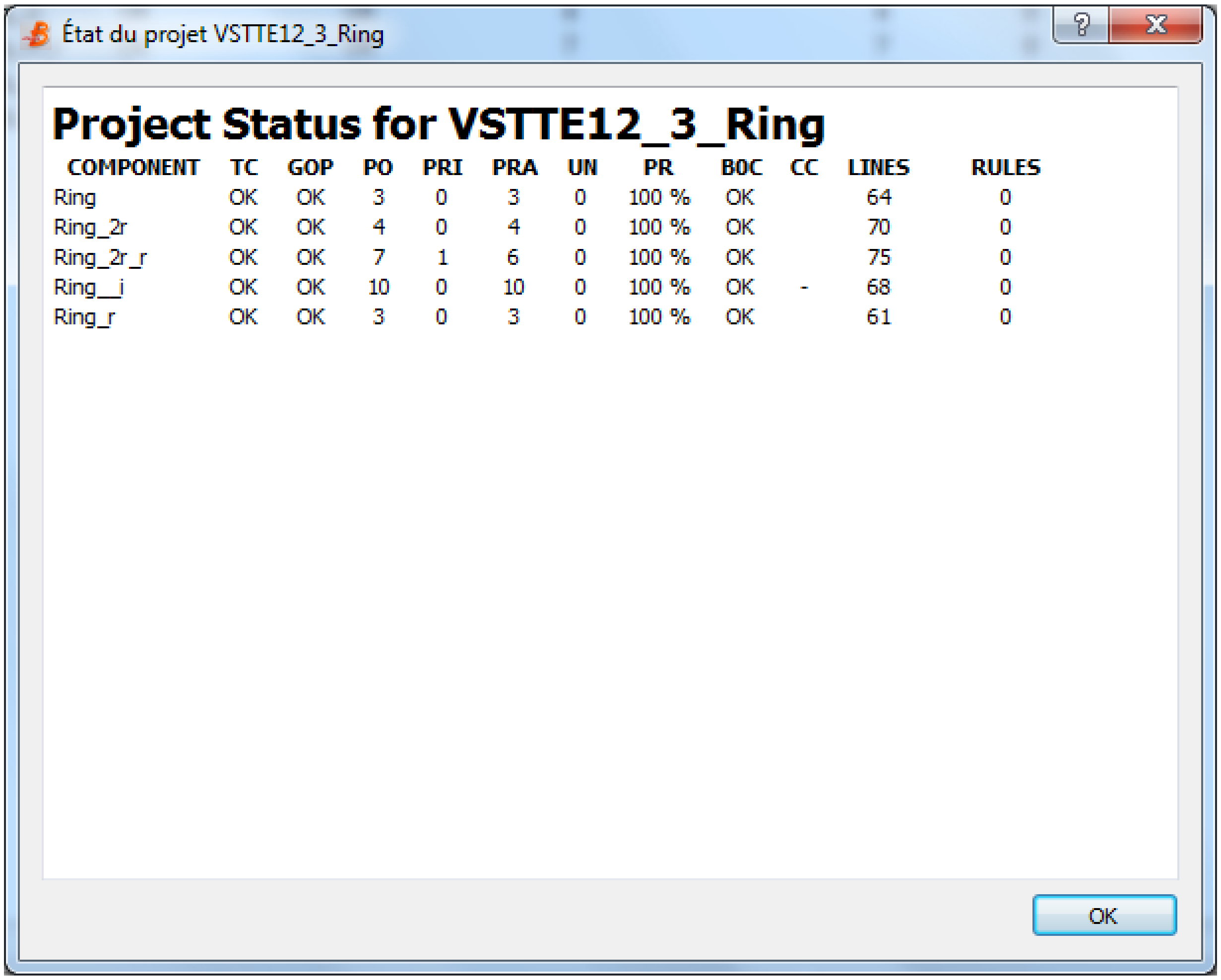
État du projet VSTTE12_3_Ring

Project Status for VSTTE12_3_Ring

COMPONENT	TC	GOP	PO	PRI	PRA	UN	PR	B0C	CC	LINES	RULES
Ring	OK	OK	12	0	12	0	100 %	OK		64	0
Ring_2r	OK	OK	25	4	21	0	100 %	OK		70	1
Ring_2r_r	OK	OK	20	7	13	0	100 %	OK		75	4
Ring__i	OK	OK	91	14	77	0	100 %	OK	-	68	1
Ring_r	OK	OK	14	5	9	0	100 %	OK		61	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 VSTTE12_3_Ring

COMPONENT	TC	GOP	PO	PRI	PRA	UN	PR	BOC	CC	LINES	RULES
Ring	OK	OK	3	0	3	0	100 %	OK		64	0
Ring_2r	OK	OK	4	0	4	0	100 %	OK		70	0
Ring_2r_r	OK	OK	7	1	6	0	100 %	OK		75	0
Ring__i	OK	OK	10	0	10	0	100 %	OK	-	68	0
Ring_r	OK	OK	3	0	3	0	100 %	OK		61	0

Well-definedness status.

IV. Generated Code

The harness verification task is done in this section, before generating the code.
The ASSERT substitution used in the test function will not appear in the generated code, but POs will be generated.

A. Ada translation

For the implementation to be translated in Ada, we must have an entry operation: this will be the test function.

Add an abstract machine Main declared the following:

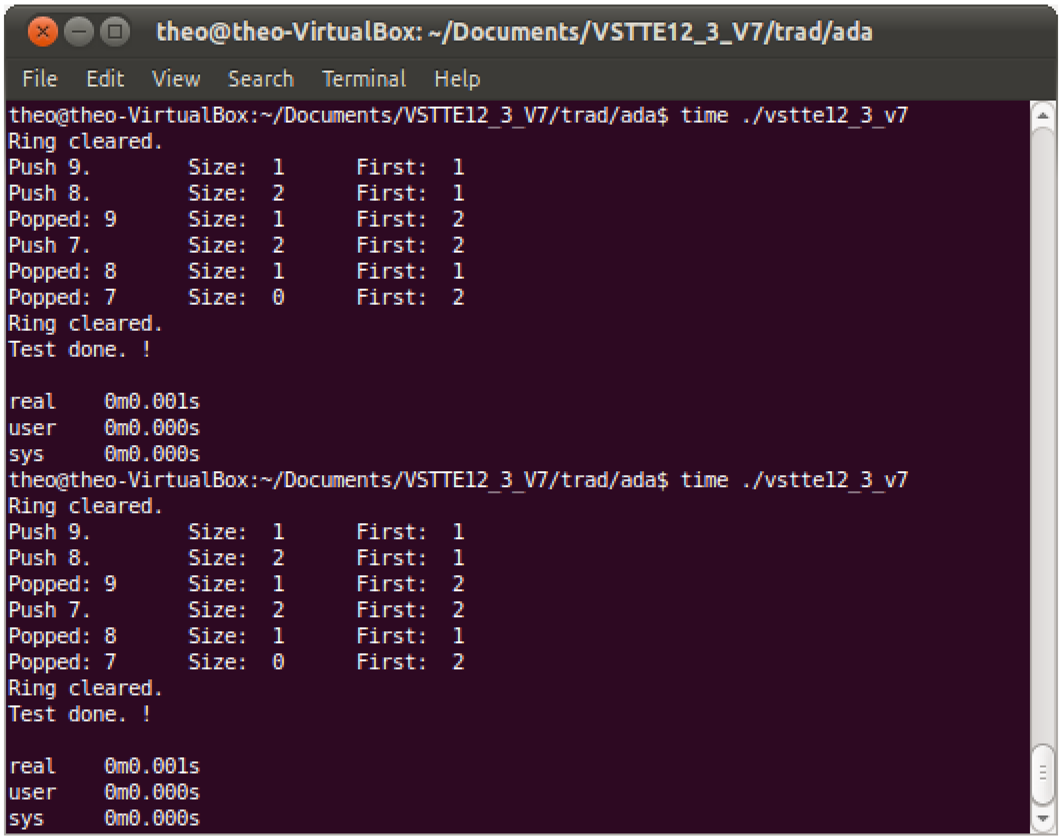
```
MACHINE
   Main

ABSTRACT_VARIABLES
   fifo,
   len

INVARIANT
   fifo : seq(INT) &
   len = size(fifo)

INITIALISATION
   fifo := [] ||
   len := 0

OPERATIONS
   test =
   BEGIN
      fifo := [] ||
      len := 0
   END
END
```

And create Main_i, the implementation of this machine:

```
IMPLEMENTATION
   Main_i
REFINES
   Main

IMPORTS
   m1.Ring(2)

CONCRETE_VARIABLES
   rr

INVARIANT
   rr: INT &
   fifo = m1.DATA &
   len = m1.LEN

INITIALISATION
   rr := 0

OPERATIONS
   test =
   VAR xx, yy, zz IN
      xx:=9;
      yy:=8;
      zz:=7;
      m1.Clear;
      m1.Push(xx);
      m1.Push(yy);
      rr <-- m1.Pop;
      ASSERT rr = xx THEN
         m1.Push(zz);
         rr <-- m1.Pop;
         ASSERT rr = yy THEN
            rr <-- m1.Pop;
            ASSERT rr = zz THEN
               m1.Clear
            END
         END
      END
   END
END
```

An Assert substitution generates a PO.
17 POs are generated for the two machines. They are all proved by the force 0 prover, which means the test function is good.
The code can be generated (in Ada). Remember that in Ada, arrays indexes start from 1.

The code is compiled using the gnat compiler (gnatmake *.adb).
Outputs are added so the execution gives the following:



theo@theo-VirtualBox: ~/Documents/VSTTE12_3_V7/trad/ada

```
theo@theo-VirtualBox:~/Documents/VSTTE12_3_V7/trad/ada$ time ./vstte12_3_v7
Ring cleared.
Push 9.        Size:  1      First:  1
Push 8.        Size:  2      First:  1
Popped: 9      Size:  1      First:  2
Push 7.        Size:  2      First:  2
Popped: 8      Size:  1      First:  1
Popped: 7      Size:  0      First:  2
Ring cleared.
Test done. !

real    0m0.001s
user    0m0.000s
sys     0m0.000s
theo@theo-VirtualBox:~/Documents/VSTTE12_3_V7/trad/ada$ time ./vstte12_3_v7
Ring cleared.
Push 9.        Size:  1      First:  1
Push 8.        Size:  2      First:  1
Popped: 9      Size:  1      First:  2
Push 7.        Size:  2      First:  2
Popped: 8      Size:  1      First:  1
Popped: 7      Size:  0      First:  2
Ring cleared.
Test done. !

real    0m0.001s
user    0m0.000s
sys     0m0.000s
```

B. C++ translation

The C++ translation needs a modification of the machine ring's implementation, regarding the gluing invariant and the operations.

The variable FIRST_i is not up to date, it should be defined on 0..ring_size-1.
We thus refine it as FIRST_B0, that is defined on 0..ring_size-1.
The gluing invariant is FIRST_B0 = FRIST_i-1.

DATA_B0 is also not up to date. Indeed, it is still defined on 1..max_size --> INT and should be defined on 0..max_size-1 --> INT. Re-define it, and modify the gluing invariant the following:

%jj.(jj: 1..ring_size | DATA_B0(jj-1)) = DATA_i

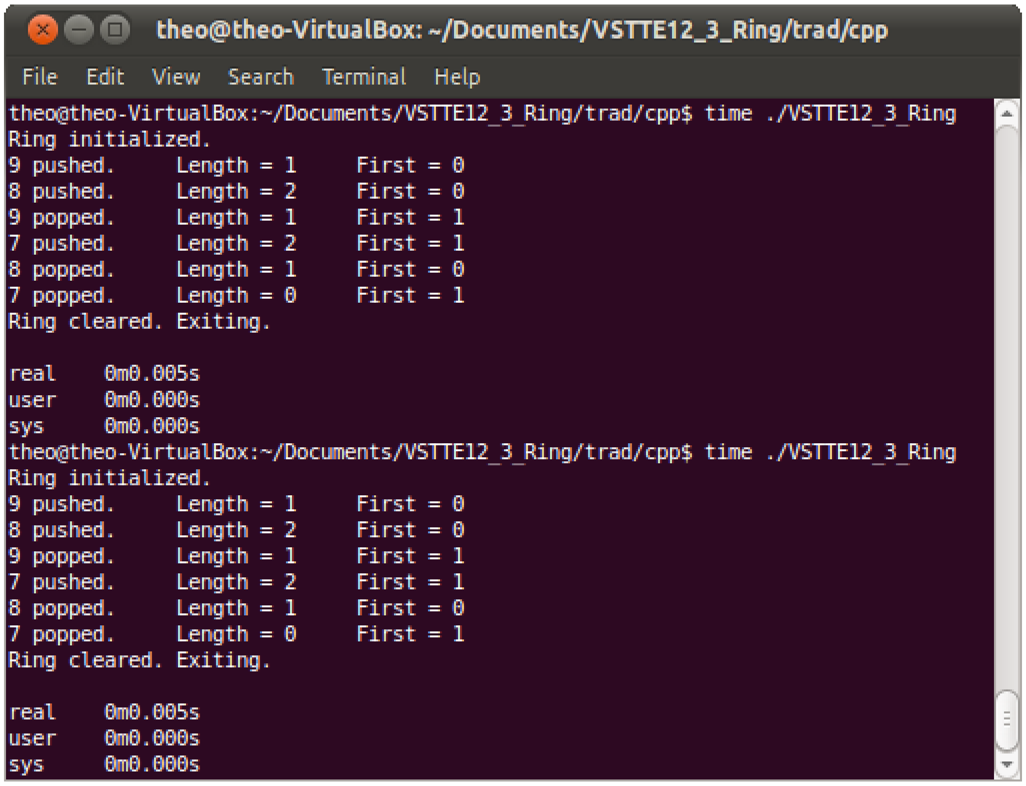
The push and the pop functions are also not up to date. Indeed, the modifications on FIRST_B0 and on DATA_B0 do not respect the new gluing invariants. Modify the operations:

```
Push(xx) =
VAR diff IN
   diff := ring_size-LEN;
   IF(FIRST_B0<diff) THEN
      DATA_B0(FIRST_B0+LEN) := xx
   ELSE
      DATA_B0(FIRST_B0-diff) := xx
   END;
   LEN := LEN+1
END;

rr <-- Pop =
BEGIN
   rr := DATA_B0(FIRST_B0);
   FIRST_B0 := (FIRST_B0+1) mod (ring_size);
   LEN := LEN-1
END
```

You have to re-prove approximately the same amount of POs, but mostly similar to the POs for the Ada code generation. The rule has to be slightly modify: this is the same rule that is modified from the refinement Ring_r to the refinement Ring_2r, and from the refinement Ring_2r to the refinement Ring_2r_r: only the substitution changes.

The same Main and Main_i machine (in the Ada code generation section) are used. The POs still are automatically proved by the force 0 prover.

The code is compiled using the make command. Outputs are added in the main function before the compilation is done so the execution gives the following:


theo@theo-VirtualBox: ~/Documents/VSTTE12_3_Ring/trad/cpp
File Edit View Search Terminal Help

```
theo@theo-VirtualBox:~/Documents/VSTTE12_3_Ring/trad/cpp$ time ./VSTTE12_3_Ring
Ring initialized.
9 pushed.     Length = 1     First = 0
8 pushed.     Length = 2     First = 0
9 popped.     Length = 1     First = 1
7 pushed.     Length = 2     First = 1
8 popped.     Length = 1     First = 0
7 popped.     Length = 0     First = 1
Ring cleared. Exiting.

real    0m0.005s
user    0m0.000s
sys     0m0.000s
theo@theo-VirtualBox:~/Documents/VSTTE12_3_Ring/trad/cpp$ time ./VSTTE12_3_Ring
Ring initialized.
9 pushed.     Length = 1     First = 0
8 pushed.     Length = 2     First = 0
9 popped.     Length = 1     First = 1
7 pushed.     Length = 2     First = 1
8 popped.     Length = 1     First = 0
7 popped.     Length = 0     First = 1
Ring cleared. Exiting.

real    0m0.005s
user    0m0.000s
sys     0m0.000s
```