TP 2 : Méthode B

SOMMAIRE

I – EXERCICE 1

1 – CREATION DE LA MACHINE ABSTRAITE

```
MACHINE
    switch
SETS
    CAPTORS = { BB , CC , NN }

OPERATIONS
    pos <-- estimate ( m1 , m2 , m3 ) =
    PRE
        m1 , m2 , m3 : CAPTORS * CAPTORS * CAPTORS
    THEN
        /* X X X */
        SELECT ( ( m1 = m2 )
                & ( m2 = m3 ) ) THEN
            pos := m1

        /* N N X */
        WHEN (    ( m1 = NN )
                & ( m2 = NN )
                & ( m3 /= NN ) ) THEN
            pos := m3

        /* X N N */
        WHEN (  ( m2 = NN )
              & ( m3 = NN )
              & ( m1 /= NN ) ) THEN
          pos := m1

        /* N X N */
        WHEN (  ( m1 = NN )
              & ( m3 = NN )
              & ( m2 /= NN ) ) THEN
          pos := m2
```

```
        /* X X N */
        WHEN (  ( m1 /= NN )
                & ( m1 = m2 )
                & ( m3 = NN ) ) THEN
            pos := m1


         /* X N X */
        WHEN (  ( m1 /= NN )
                & ( m1 = m3 )
                & ( m2 = NN ) ) THEN
            pos := m1

        /* N X X */
        WHEN (  ( m2 /= NN )
                & ( m2 = m3 )
                & ( m1 = NN ) ) THEN
            pos := m2

        ELSE
                /* B C ? */
            ANY xx , yy WHERE xx = BB & yy = CC THEN
                pos := NN
            END
        END
    END

END
```

Remarques :

Nous commençons notre machine en créant un **ensemble abstrait** `CAPTORS` sous forme d'énumération nous permettant de **représenter les états des capteurs**. Nous reconnaissons ainsi les états BB pour une voix normale, CC pour une voix renversée et NN pour la position intermédiaire.

L'utilisation d'une clause `SELECT` pour représenter un choix non borné nous oblige à **prendre tous les cas de combinaisons possibles** pour 1 ou 2 valeurs nulles (ordre des capteurs). Néanmoins, nous avons tout de même utilisé une clause ANY pour prendre en compte les cas d'oppositions Normale / Renversé pour n'importe quel capteur.

Nous pouvons également remarquer que les cas où toutes les valeurs sont Nulle et/ou toute valent la même valeur sont identiques au niveau du résultat.

Puisque cette machine ne comporte qu'une opération retournant une valeur et ne réalisant aucune substitution, **il n'y a aucun invariant à prouver**.

2 - IMPLEMENTATION

```
IMPLEMENTATION
    switch_i
REFINES
    switch

CONCRETE_VARIABLES
    somme

INVARIANT
    somme : NAT

INITIALISATION
    somme := 0

OPERATIONS

    pos <-- estimate(m1, m2, m3) =
    BEGIN
        VAR somme IN

            somme:=0;
            IF (m1 = BB) THEN
                somme := somme +5
            ELSIF (m1=CC) THEN
                somme := somme +9
            END;

            IF (m2 = BB) THEN
                somme := somme +5
            ELSIF (m2=CC) THEN
                somme := somme +9
            END;

            IF (m3 = BB) THEN
                somme := somme +5
            ELSIF (m3=CC) THEN
                somme := somme +9
            END;

            CASE somme OF
                EITHER 5 THEN
                    pos:= BB
                OR 9 THEN
                    pos:= CC
                OR 10 THEN
                    pos:= BB
                OR 15 THEN
                    pos:= BB
                OR 18 THEN
                    pos:= CC
                OR 27 THEN
                    pos:= CC
                ELSE
                    pos:= NN
                END
            END
        END
    END
END
```

Remarques :

Etant donné que l'ordre des états des capteurs n'importe pas, **l'habitude de programmation voulant que nous représentions les valeurs énumérées de `CAPTORS` sous forme de valeurs entières nous a donné l'idée d'utiliser une variable `somme`.** Ainsi, cela nous permet de réduire quelque peu le nombre de cas spécifique à gérer (ordre des valeurs nulles et non nulles) par rapport à la machine abstraite puisque les cas X N N / N N X / N X N nous donnent le même résultat (où N représente l'état nul et X un état quelconque différent de nul).

Nous avons ensuite remplacé les clauses `SELECT` par leurs équivalents `IF` en langage de programmation et réalisé un dernier test sur la valeur de `SOMME` pour retourner le résultat sous la forme attendue d'une position.

3 – OBLIGATIONS DE PREUVES ET RESULATST

Nous étions quelque peu sceptiques sur la capacité de l'atelier B de faire le lien entre notre implémentation et la spécification réalisée dans la machine abstraite. Mais il se trouve qu'il n'y eut aucun problème à ce niveau. L'unique problème que nous avions rencontré était de bien considérer tous les cas possibles de combinaisons pour qu'il n'y ait aucun déterminisme dans la machine abstraite.

Ainsi, nous obtenons un résultat de 100%.

4 – CONVERSION EN LANGAGE C

Le code généré par l'atelier B est propre et très proche, si n'est identique à l'implémentation réalisée précédemment. Nous retrouvons ainsi l'ensemble abstrait `CAPTORS` sous la forme d'un `enum` déclaré dans switch.h, accompagné des signatures des différentes fonctions comme l'initialisation et l'opération `estimate`.

Nous pouvons tout de même remarquer que la fonction `estimate` **ne retourne en fait aucune valeur et prend simplement le paramètre pos sous forme d'un pointeur** pour modifier sa valeur.

Par contre, nous ne pouvons compiler les fichiers puisqu'aucun d'entre eux ne possède une fonction main. Pour ce faire, il suffit de créer un programme qui appellera la fonction `estimate`. Malgré ce détail, nous pouvons dire que la méthode B nous a aidé à développer une fonction d'après une spécification abstraite et que celle-ci est sûre à partir du moment où les spécifications sont bien définies.

```
#ifndef _switch_h
#define _switch_h

/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
/*------------------------------
   SETS Clause: enumerated sets
  ------------------------------*/
typedef enum {
   switch__BB,
   switch__CC,
   switch__NN
} switch__CAPTORS;


/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
extern int switch__somme;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
extern void switch__INITIALISATION(void);

/*--------------------
   OPERATIONS Clause
  --------------------*/
extern void switch__estimate(
   switch__CAPTORS switch__m1,
   switch__CAPTORS switch__m2,
   switch__CAPTORS switch__m3,
   switch__CAPTORS *switch__pos);

#endif
```

```
/*--------------------------
   Added by the Translator
  --------------------------*/
#include <stdbool.h>
#include "switch.h"

/*----------------------------
   CONCRETE_VARIABLES Clause
  ----------------------------*/
int switch__somme;

/*------------------------
   INITIALISATION Clause
  ------------------------*/
void switch__INITIALISATION(void) {
   switch__somme = 0;
}

/*--------------------
   OPERATIONS Clause
  --------------------*/
void switch__estimate(
   switch__CAPTORS switch__m1,
   switch__CAPTORS switch__m2,
   switch__CAPTORS switch__m3,
   switch__CAPTORS *switch__pos) {
   {
      int switch__somme;

      switch__somme = 0;
      if (switch__m1 ==
         switch__BB)
      {
         switch__somme = switch__somme +
            5;
      }
      else if (switch__m1 ==
         switch__CC)
      {
         switch__somme = switch__somme +
            9;
      }
      if (switch__m2 ==
         switch__BB)
      {
         switch__somme = switch__somme +
            5;
      }
```

```
else if (switch__m2 ==
         switch__CC)
      {
         switch__somme = switch__somme +
            9;
      }
      if (switch__m3 ==
         switch__BB)
      {
         switch__somme = switch__somme +
            5;
      }
      else if (switch__m3 ==
         switch__CC)
      {
         switch__somme = switch__somme +
            9;
      }
      switch (switch__somme) {
      case 5:
         *switch__pos = switch__BB;
         break;
      case 9:
         *switch__pos = switch__CC;
         break;
      case 10:
         *switch__pos = switch__BB;
         break;
      case 15:
         *switch__pos = switch__BB;
         break;
      case 18:
         *switch__pos = switch__CC;
         break;
      case 27:
         *switch__pos = switch__CC;
         break;
      default:
         *switch__pos = switch__NN;
         break;
      }
   }
}
```

II – EXERCICE 2

1 – MACHINE ABSTRAITE DU SERVICE DE RESERVATION

A – CREATION DE LA MACHINE

En suivant les indications données dans l'énoncé, nous obtenons la description machine ci-contre.

```
MACHINE
    reservation (nb_max)
CONSTRAINTS
    nb_max : NAT1
DEFINITIONS
    SIEGES == (1..nb_max)
VARIABLES
    occupes,
    nb_libre
INVARIANT
    occupes <: NAT &
    nb_libre : NAT &
    nb_libre = nb_max - card(occupes)
INITIALISATION
    occupes := {} ||
    nb_libre := nb_max
OPERATIONS
    xx <-- place_libre =
            xx := nb_libre;

    reserver =
    PRE
        nb_libre >=1
    THEN
        ANY zz WHERE (zz : SIEGES) & (zz /: occupes)
        THEN occupes := occupes \/ {zz}  ||
            nb_libre := nb_libre -1
        END
    END;
    liberer(place) =
    PRE
        place : SIEGES &
        place : occupes
    THEN
        nb_libre := nb_libre +1 ||
        occupes := occupes - {place}
    END
END
```

Nous retrouvons donc le paramètre `nb_max` qui doit être supérieur ou égal à 1 et inférieur ou égal à MAXINT, tout comme « l'ensemble » `SIEGES`, qui est en fait une définition.

L'ensemble `OCCUPES` est une variable, car il pourra être manipulé. Il est défini comme étant un sous-ensemble de `NAT` et contiendra les indices des places réservées.

`nb_libre` est une variable qui doit directement représenter le nombre de places disponibles. Ainsi, cette dernière doit vérifier en tout temps la relation suivante :

`nb_libre = nb_max - card(occupes)`

`card(occupes)` représentant le nombre de place occupées et `nb_max` le nombre de places total.

En ce qui concerne les opérations, nous pouvons observer l'utilisation du choix non déterministe d'une place lors de l'opération `reserver` qui est représenté par l'utilisation de la clause :

```
ANY zz WHERE (zz : SIEGES) & (zz /: occupes)
THEN occupes := occupes \/ {zz}  ||
     nb_libre := nb_libre -1
END
```

Avec cette instruction, nous choisissons n'importe quelle variable `zz` appartenant à l'ensemble `SIEGES` mais qui n'est pas encore occupée. Si tel est le cas, alors nous modifions les variables `occupes` et `nb_libre` en conséquence.

L'opération `liberer` réalise les opérations inverses à `reserver`, à savoir, retire un élément `place` de l'ensemble `occupes` et augmente le nombre de places disponibles.

B – Obligations de preuve

Après une **première génération des obligations de preuves**, nous obtenons les erreurs suivantes :

```
--------------------
        ″`Check that the invariant (nb_libre: NAT) is preserved by the operation - ref 3.4'″ => nb_libre+1<=2147483647
--------------------
        ″`Check that the invariant (nb_libre = nb_max-card(occupes)) is preserved by the operation - ref 3.4'″ => nb_libre+1 = nb_max-card(occupes-{place})
```

Ainsi, nous pouvons **réaliser quelques ajustements** tel que :

```
INVARIANT
    occupes <: SIEGES &
    nb_libre : (0 .. nb_max) &
    nb_libre = nb_max - card(occupes)
```

Malgré cela, le prouveur ne parvient toujours pas à montrer que l'invariant :

`nb_libre = nb_max - card(occupes)`

est préservé par l'opération :

`nb_libre := nb_libre +1`

En utilisant le prouveur interactif, nous pouvons nous rendre compte que ce dernier ne comprend pas que lorsque l'on réalise la substitution :

`occupes := occupes - {place}`

alors : `card(occupes) := card(occupes) - 1`

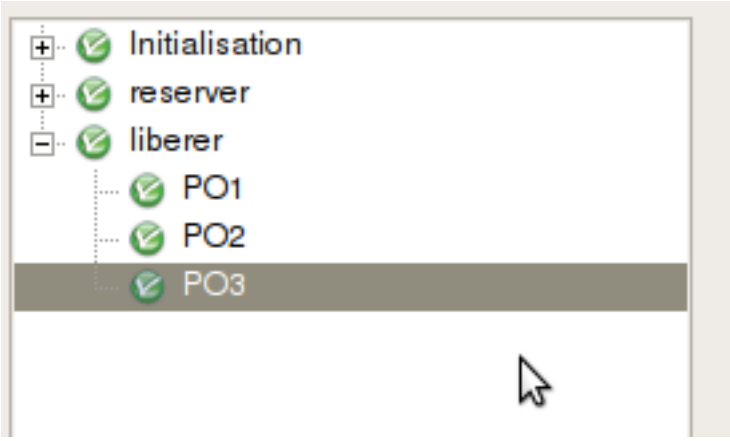
Pourtant, la variable locale `place` est bien un élément de l'ensemble abstrait `occupes` et la relation entre le retrait d'un élément et le cardinal de l'ensemble devrait fonctionner.

Pour régler ce problème, **nous « assistons » le prouveur en luis indiquant l'hypothèse suivante** :

```
Hypothesis containing "card(occupes)-card(occupes-{place}) = 1"
```

Et nous obtenons le raisonnement suivant :

```
"`Check that the invariant (nb_libre = nb_max-card(occupes)) is preserved by the operation - ref 3.4'"
=> nb_libre+1 = nb_max-card(occupes-{place})

--------------------

    0 = -1+card(occupes)-card(occupes-{place})

--------------------

    -1+nb_max-nb_libre-card(occupes-{place}) = 0

--------------------

    -1+nb_max-nb_libre-(-1+card(occupes)) = 0

--------------------

    "`Check that the invariant (nb_libre = nb_max-card(occupes)) is preserved by the operation - ref
3.4'" => nb_libre+1 = nb_max-card(occupes-{place})
```



Nous pouvons donc supposer qu'il y a un problème dans notre manière de modifier l'ensemble abstrait `occupes` lors du retrait d'un élément.

2 – RAFFINEMENT DU SERVICE DE RESERVATION

A – CREATION DU RAFFINEMENT

```
REFINEMENT
   reservation_r (nb_max)
REFINES
    reservation

VARIABLES
    nb_libre,
    occupation

INVARIANT
    occupation : SIEGES --> BOOL &
    nb_libre : (0 .. nb_max) &
    nb_libre = nb_max - card( dom(occupation |>
{TRUE}) )

DEFINITIONS
   SIEGES == ( 1 .. nb_max )

INITIALISATION
    occupation := SIEGES*{FALSE};
    nb_libre := nb_max

OPERATIONS
    xx <-- place_libre =
    BEGIN
        xx := nb_libre
    END;

    reserver =
     LET vv BE vv = min( dom(occupation |> {FALSE}) )
      IN
            occupation(vv) := TRUE     ;
            nb_libre := nb_libre - 1
    END;

   liberer ( place ) =
   BEGIN
       occupation(place):=FALSE ||
       nb_libre := nb_libre + 1

   END
END
```

Par rapport aux recommandations présentes dans l'énoncé, nous pouvons faire les remarques suivantes :

- **L'ensemble abstrait `occupes`** ne peut être représenté de manière concrète par une fonction totale dans le raffinement. En effet, dans la définition de la machine abstraite, ce dernier était un ensemble simple d'entiers alors qu'ici il s'agirait de couples de types (INT, BOOL). Ainsi, nous avons du utiliser une autre variable `occupation` pour représenter la fonction totale qui à chaque siège associe un booléen TRUE sur le siège est occupé et FALSE si il est libre.

- **Nous ajustons donc un invariant** du type : `nb_libre = nb_max - card( dom(occupation |> {TRUE}) )` afin de préserver la cohérence avec le précédent invariant..

- Afin de rendre l'opération `reserver` déterministe, nous avons utilisé une variable locale afin de sélectionner la place libre **ayant le plus petit numéro**.

- Pour finir, nous pouvons ajouter que le raffinement ne peut avoir de clause du type `CONSTRAINTS` et que nous avons remplacé toutes les pré-conditions abstraites figurant dans les opérations.

B – Obligation de preuve

Après une première génération des obligations de preuves, nous obtenons :

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
reservation	OK	OK	8	8	0	-
reservation_i	-	-	-	-	-	-
reservation_r	OK	OK	13	9	4	-
switch	OK	OK	0	0	0	-
switch_i	OK	OK	27	27	0	OK

1 - Première PO

Parmi les 4 obligations de preuves que le prouveur n'arrive pas à démontrer, malgré l'utilisation de la force 3, **dans l'opération `libérer`** :

```
--------------------

    "`liberer preconditions in this component'" &

    place: 1..nb_max &

    place: occupes &

    "`Check that the invariant (nb_libre$1: 0..nb_max) is preserved by the operation - ref 4.4, 5.5'"

=>

    nb_libre$1+1: 0..nb_max
```

Pour ce faire, nous rajoutons une condition par l'intermédiaire d'une clause SELECT tel que :

```
SELECT
        nb_libre < nb_max
    THEN
```

Cela nous permet de résoudre cette obligation.

2 – Seconde et Troisième PO

Il se trouve que la **modification de la fonction totale `occupation`** engendre des difficultés pour le prouveur. En effet, ce dernier n'arrive pas à montrer que l'ajout ou le retrait d'un élément modifie la valeur du cardinale la restriction du domaine nécessaire à la vérification de l'invariant sur `nb_libre` :

opération `reserver` :

```
----------
    "`reserver preconditions in this component'" &
    1<=nb_libre$1 &
    "`Local hypotheses'" &
    vv = min(dom(occupation$1|>{FALSE})) &
    "`Check that the invariant (nb_libre$1 = nb_max-card(dom(occupation$1|>{TRUE}))) is preserved by the operation - ref 4.4, 5.5'"
=>
    nb_libre$1-1 = nb_max-card(dom(occupation$1<+{vv|->TRUE}|>{TRUE}))
```

Dans ce cas, c'est la substitution `occupation$1 <+ {vv|->TRUE}` engendre logiquement l'augmentation de la valeur de `card(dom( occupation$1 |> {TRUE} ))`. Il s'agit du même genre d'erreur que celle résolue dans la machine abstraite.

De même, de manière contraire dans l'**opération `liberer`** :

```
----------
    "`liberer preconditions in this component'" &
    place: 1..nb_max &
    place: occupes &
    "`Local hypotheses'" &
    nb_libre$1+1<=nb_max &
    "`Check that the invariant (nb_libre$1 = nb_max-card(dom(occupation$1|>{TRUE}))) is preserved by the operation - ref 4.4, 5.5'"
=>
    nb_libre$1+1 = nb_max-card(dom(occupation$1<+{place|->FALSE}|>{TRUE}))
```

3 - Quatrième PO

```
    "`reserver preconditions in this component'" &
    1<=nb_libre$1 &
    "`Local hypotheses'" &
    vv = min(dom(occupation$1|>{FALSE})) &
    "`Check operation refinement - ref 4.4, 5.5'"
=>
    #zz.(zz: 1..nb_max & not(zz: occupes))
```

Dans cette dernière, il semblerait que le prouveur n'arrive pas à montrer que :

`vv = min(dom(occupation$1|>{FALSE}))` est équivalent à `zz`, dans le sens où il appartient à `SIEGES` et pas `à occupes`.

Ainsi, il n'arrive pas à prouver l'équivalence entre `dom(occupation$1|>{TRUE}) et occupes`.

En essayant de rajouter la gestion de la variable occupes avec un invariant de ce type, l'atelier B nous signale une erreur :

```
INVARIANT
    occupes : dom(occupation |> {TRUE})
```

D'après lui, dom ne correspond pas à un ensemble d'entiers et représente donc un conflit avec la précédente définition de occupes < : SIEGES

A ce stade, nous ne savons pas quelle action entreprendre pour régler ces obligations de preuves à l'aide de l'atelier B.

3 – IMPLEMENTATION DU SERVICE DE RESERVATION

A – CREATION DE L'IMPLEMENTATION

```
IMPLEMENTATION
   reservation_i (nb_max)
REFINES
    reservation_r

CONCRETE_VARIABLES
    occupation, nb_libre

DEFINITIONS
   SIEGES == ( 1 .. nb_max )

INITIALISATION
   occupation := ( 1 .. nb_max ) * { FALSE } ;
   nb_libre := nb_max

OPERATIONS
   xx <-- place_libre =
   BEGIN
       xx := nb_libre
   END
   ;

   reserver =
   BEGIN
       VAR vv, indice
       IN
           indice:=1;
           vv:= occupation(indice);
       WHILE (vv = TRUE & indice < nb_max) DO
           indice:=indice+1;
           vv:=occupation(indice)
       INVARIANT
           indice : SIEGES &
           vv : BOOL &
           occupation[1 .. indice-1] <: {TRUE}
       VARIANT
           nb_max -indice
       END;
           occupation( indice ) := TRUE ;
           nb_libre := nb_libre - 1
       END
   END;

   liberer ( place ) =
   BEGIN
       occupation ( place ) := FALSE ;
       nb_libre := nb_libre + 1
   END
END
```

Par rapport à au raffinement précédemment décrit, **seule l'implémentation de la recherche de l'indice minium de place libre diffère**.

Pour ce faire, parcourons le « tableau » occupation à la recherche du premier indice qui nous renverra la valeur FALSE. Nous réalisons ce **parcours à l'aide d'une boucle de type `WHILE`**.

Ainsi, nous avons dû définir deux variables locales vv pour l'état d'occupation de la place ayant l'indice `indice`.

Pour garantir l'arrêt de la boucle, nous définissons **deux conditions** :

- Vv = true, nous avons trouvé une place libre et c'est la plus petite ;
- Indice < nb_max, pour ne pas déborder et aller au-delà de la taille effective du tableau.

A noter **les invariants** de boucle qui typent les variables locales et devront vérifier que toutes les places parcourues sont occupées si nous ne sommes pas sortis de la boucle.

Le variant doit quant à lui représenter une valeur qui diminue garantissant la terminaison de la boucle. Nous n'avons pas vraiment compris l'intérêt de cette clause étant donné les conditions présentes dans la boucle WHILE, mais peut-être est-ce une double vérification ou une particularité qui permettra la génération de code C sous une autre forme de boucle.

B – Obligation de preuve

Nous obtenons 4 obligations non prouvées :

1 - Première et seconde PO

Parmi les 4 obligations de preuves que le prouveur n'arrive pas à démontrer, malgré l'utilisation de la force 3, **dans l'opération `reserver` :**, 2 concernent l'invariant : `occupation[1 .. indice-1] <: {TRUE}`

a- Encore une fois, dans une des Pos, il s'agit de la difficulté à comprendre **l'équivalence du minimum sur le domaine restreint de `occupation`** .

b- Dans le second cas, le problème intervient **au pas de boucle** :

```
"`reserver preconditions in this component'" &
        1<=nb_libre$1 &
        "`Local hypotheses'" &
        indice: 1..nb_max &
        vv: BOOL &
        not(occupation$1[1..indice-1] = {}) => occupation$1[1..indice-1] =
{TRUE} &
        vv = TRUE &
        indice+1<=nb_max &
        not(occupation$1[1..indice+1-1] = {}) &
        "`Check preconditions of called operation, or While loop
construction, or Assert predicates'"
=>
        occupation$1[1..indice+1-1] = {TRUE}
```

Malgré le fait d'avoir comme hypothèse que :

`occupation$1[1..indice-1] = {TRUE}`

`vv = TRUE`

VV étant : vv = occupation (indice) à l'entrée dans la boucle.

Si le test s'effectue à chaque instruction, nous obtenons de manière séquentielle,

`indice :=indice+1`

`et donc occupation$1[1..indice+1-1] = occupation$1[1..indice-1] \/ {vv}`

Ce qui est pourtant vrai.

2 - Troisième et quatrième PO

D**ans l'opération `libérer` :**

a- Malgré la présence de la pré-condition nb_libre < nb_max dans le raffinement, l'implémentation n'arrive pas à montrer que l'opération préserve le fait que nb_libre <= nb_max appartenant à NAT et borné par MAXINT. Pourtant, une pré-condition similaire sur nb_libre est réalisée dans l'opération reserver et n'est pas requise dans l'implémentation.

En modifiant quelque peu le code de l'implémentation de la manière suivante :

```
liberer ( place ) =
   BEGIN
        IF (nb_libre <= (nb_max - 1)) THEN
        occupation ( place ) := FALSE ;
        nb_libre := nb_libre + 1
        END
   END
```

Nous résolvons le problème, mais deux nouvelles obligations apparaissent. En effet, **malgré la condition imposée par la clause IF, le prouveur semble vouloir vérifier que les substitutions lorsque le prédicat et faux**.

b- La quatrième obligation de preuve improuvée concerne la substitution qui réalise le remplacement de la valeur booléenne affectée à l'indice de la place :

```
--------------------
        "`liberer preconditions in this component'" &
        place: 1..nb_max &
        place: occupes &
        "`Local hypotheses'" &
        nb_libre$1<=nb_max-1 &
        "`Check that the invariant (occupation = occupation$1) is
preserved by the operation - ref 4.4, 5.5'"
=>
occupation$1<+{place|->FALSE} = {place}<<|occupation$1<+{place|->FALSE}
```

Remarques :

Au niveau de l'implémentation, nous pouvons remarquer que **le B0 check n'apprécie pas l'utilisation d'un paramètre de la machine comme valeur limite d'un tableau ou d'un ensemble**. En effet, cela rend la taille de notre tableau `occupation` dynamique alors que le B, basé sur le C, préfère les éléments statiques, plus facilement démontrables. Cela se traduit par un nombre de POs à vérifier assez important : dans notre cas 41.

CONCLUSION

En conclusion de ce second TP, nous pouvons dire que nous avons perçu l'utilité de **la méthode B pour vérifier et accompagner le travail de développement lorsque les spécifications sont bien établies**. En effet, le code généré du programme de l'exercice 1 réalise exactement ce qui a été spécifié sans oublier un seul cas.

Par contre, nous avons également vu que **l'atelier B avait du mal à gérer des éléments dynamiques** qui sont pourtant très courants dans la programmation objet et **présente également quelques limites dans sa capacité à démontrer certaines preuves**. De plus avec notre niveau de familiarisation avec l'outil, nous restons quelque peu déçus, car nous n'avons pas su l'exploiter à son maximum afin de prouver à 100% toutes nos machines, nos raffinements et implémentations.

Cependant, nous pouvons tout de même dire que **cet outil est très pratique, car il permet de nous assister** et de porter notre attention sur des problèmes courants de programmation qui peuvent avoir d'importantes conséquences.