

# Rapport de TP – Méthode B

---

## Introduction à l'atelier B

Cette première partie du TP sur la méthode B permet de prendre en main l'atelier B, sa syntaxe assez stricte et la gestion de ses prouveurs automatiques et interactif.

### Recherche d'un élément dans un tableau

On souhaite réaliser une fonction qui cherche un élément  $v$  dans un tableau  $t$  d'entiers de taille  $n$  : cette fonction renvoie `true` ou `false` selon que l'élément est respectivement dans le tableau ou non.

La machine abstraite correspondante :

```
MACHINE
  search
CONSTANTS
  tt, nn
PROPERTIES
  nn : NAT &
  tt : 0..nn-1-->NAT
OPERATIONS
  ok<--searching(xx) =
PRE
  xx : NAT
THEN
  ok := bool (xx: ran(tt))
END
END
```

On observe que la machine utilise deux constantes : le nombre d'entiers du tableau et le tableau qui est une fonction totale des indices (0 à  $n-1$ ) dans les entiers naturels. L'opération `searching` renvoie `true` si l'élément recherché appartient à l'ensemble d'arrivée de la fonction qui définit le tableau, donc si l'élément appartient au tableau, et `false` sinon.

La génération des obligations de preuves pour cette machine nous renvoie 0 obligations de preuves : on ne substitue aucune variable dans les opérations et aucun invariant à vérifier n'est indiqué dans la machine abstraite.

Sous l'atelier B, on trouve les termes suivants :

- TC : TypeChecked (vérifie le typage du composant)
- POG : Proof Obligations generated (obligations de preuves générées)
- nPO : number of Proof Obligations (nombre d'obligations de preuves)
- nUn : number of unproved (nombre d'obligations de preuves non prouvées par le prouveur automatique de l'atelier B)
- Pr : proved (nombre d'obligations de preuves prouvées par le prouveur automatique de l'atelier B)
- BOC : B0 Checked (vérifie la syntaxe du composant : le B0 est le langage utilisé dans l'atelier B)

### Fonctions minimum et maximum d'entiers naturels

On souhaite désormais calculer le minimum de deux entiers naturels et le maximum de trois entiers naturels. Ce qui nous amène à écrire la machine abstraite suivante :

```
MACHINE
```

```

utils
OPERATIONS
  res<--mini (xx,yy) =
  PRE
    xx : NAT &
    yy : NAT
  THEN
    res := min ({xx,yy})
  END
;
res<--maxi (xx,yy,zz) =
  PRE
    xx : NAT &
    yy : NAT &
    zz : NAT
  THEN
    res := max ({xx,yy,zz})
  END
END
END

```

Les pré-conditions des opérations consistent uniquement à vérifier que les éléments dont on cherche le minimum ou le maximum sont bien des entiers naturels. Les fonctions « min » et « max » ensemblistes autorisées dans l'atelier B sont des fonctions génériques dont on ne se soucie pas au niveau de la machine abstraite. Dans le processus de raffinement ou d'implémentation, on détaillera l'algorithme qui sert à calculer ces minimum et maximum. A signaler qu'il n'y a toujours aucune obligation de preuve générée par l'atelier B car aucune substitution de variable n'a lieu dans les opérations et il n'y a aucun invariant à vérifier.

On donne ensuite une implémentation de cette machine abstraite :

```

IMPLEMENTATION
  utils_i
REFINES
  utils
OPERATIONS
  res <-- mini ( xx , yy ) =
  IF
    xx >= yy
  THEN
    res := yy
  ELSE
    res := xx
  END
;
res <-- maxi ( xx , yy , zz ) =
BEGIN
  IF
    xx >= yy
  THEN
    res := xx
  END;
  IF
    xx <= yy
  THEN
    res := yy
  END;
  IF
    res <= zz
  THEN

```

```

    res := zz
  END
  END
END

```

Dans l'implémentation, on détaille l'algorithme permettant de trouver le minimum de deux entiers et le maximum de trois entiers. On perd donc un niveau d'abstraction et on se rapproche du code.

19 obligations de preuves sont générées : 4 pour l'opération « mini » et 15 pour l'opération « maxi ». Pour l'opération mini, le prouveur doit prouver que  $xx$  et  $yy$  sont bien des entiers naturels (inférieurs à  $MAXINT$ ) et que quand on fait la substitution de  $res$  par  $xx$  ou  $yy$ , le résultat reste bien un entier naturel. Idem pour l'opération « maxi » mais il y a plus de cas à vérifier (tous les cas engendrés par les IF...ELSE...).

Ces 19 obligations de preuve sont prouvées grâce au prouveur automatique. Lors de la première rédaction de cette implémentation, nous n'avions pas pu tout prouver automatiquement car dans l'opération « maxi », le premier IF s'appliquant pour  $xx \geq yy$  nous n'avions mis que  $xx < yy$  dans le second IF, sachant que le cas  $xx=yy$  avait déjà été traité dans le premier cas. Or comme le prouveur tente de prouver les obligations de preuve dans tous les cas, dans le cas où  $xx=yy$  pour le second IF, il n'arrivait pas à prouver la PO générée. Il a donc fallu préciser  $xx \leq yy$  dans le second IF...

Afin de tester les possibilités de l'atelier B, nous avons poussé jusqu'à la génération du code C du composant « utils », code généré dans le répertoire « lang » du projet de l'atelier B. Il génère le squelette des fonctions dans un fichier d'entête (.h) et le code des fonctions dans un fichier source (.c). Ne reste qu'à créer un programme source pour utiliser ces fonctions :

```

#include <stdio.h>
#include <stdlib.h>
// fichier d'entete généré qui contient le squelette des fonctions
#include "utils.h"

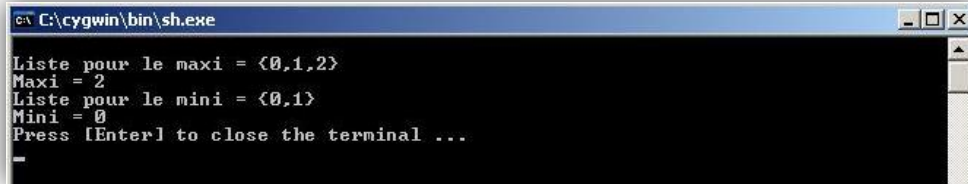
int main(int argc, char** argv) {
    int x,y,z;
    x = 0; // 0
    y = x+1; // 1
    z = y+1; // 2
    int res;
    int res2;

    // utilisation de la fonction max de 3 entiers générée par l'atelier B
    utils__maxi(x,y,z,&res);
    // utilisation de la fonction min de 2 entiers générée par l'atelier B
    utils__mini(x,y,&res2);

    printf("Liste pour le maxi = {%d,%d,%d}\n",x,y,z);
    printf("Maxi = %d\n",res);
    printf("Liste pour le mini = {%d,%d}\n",x,y);
    printf("Mini = %d\n",res2);
    return (EXIT_SUCCESS);
}

```

On crée donc 3 entiers : 0, 1 et 2. Le résultat de l'exécution après compilation de ces sources nous renvoie bien le minimum des deux premiers entiers (c'est-à-dire 0) et le maximum des 3, c'est-à-dire 2 :



```

C:\cygwin\bin\sh.exe
Liste pour le maxi = <0,1,2>
Maxi = 2
Liste pour le mini = <0,1>
Mini = 0
Press [Enter] to close the terminal ...

```

## Introduction au prouveur interactif

Afin de voir les obligations de preuves générées par l'atelier B, de les prouver interactivement, de guider le prouveur à partir de l'axiome pour démontrer le théorème, il y a une interface de « prouveur interactif ». On utilise une machine abstraite très simple pour tester cette interface :

```

MACHINE
  Machine1
VARIABLES
  few, many
INVARIANT
  few <: NATURAL &
  many <: NATURAL &
  few <: many
INITIALISATION
  few,many := {1,2,3},{2,3,4}
END

```

Lors de l'initialisation (substitution) des variables few et many, 3 POs sont générées pour respecter l'invariant, c'est-à-dire que few est un sous-ensemble d'entiers naturels, many idem et que few est un sous ensemble de many. Les deux premières sont prouvées via le prouveur automatique dès la force 0 mais la dernière résiste au prouveur automatique même en force 3. Il est alors nécessaire d'ouvrir le prouveur interactif afin de voir pourquoi la PO n'est pas prouvée automatiquement. On peut voir à partir de cet écran que l'obligation de preuve récalcitrante est la suivante :

```

"Check that the invariant (few <: many) is established by the
initialisation - ref 3.3'" =>
1: {2,3,4}

```

On voit bien ici que le théorème est faux car le prouveur est sensé prouver que l'élément « 1 » appartient à l'ensemble {2,3,4}, ce qui est impossible à prouver. Il est à noter que ce n'est pas parce que le prouveur n'arrive pas à prouver quelque chose que c'est faux, c'est peut être parce qu'il ne sait pas le prouver à partir de ses lemmes de bases et ceux indiqués dans la machine abstraite. Si on veut corriger la machine pour prouver les POs générées, il faut corriger l'initialisation en :

```

few,many := {1,2,3},{1,2,3,4}

```

La machine abstraite suivante contient 3 constantes aa, bb et cc qui appartiennent respectivement à l'intervalle [1-10], [2-10] et [3-10] des entiers naturels. On cherche à prouver interactivement que le maximum de ces 3 constantes appartient forcément à l'intervalle [1-10].

```

MACHINE
  Machine2
VARIABLES
  maxi
INVARIANT
  maxi : 1..10
CONSTANTS
  aa,bb,cc
PROPERTIES

```

```

aa : 1..10 &
bb : 2..10 &
cc : 3..10
INITIALISATION
  maxi := max({aa,bb,cc})
END

```

La PO générée est la suivante (la substitution à l'initialisation doit respecter l'invariant) :

```

"Check that the invariant (maxi: 1..10) is established by the
initialisation - ref 3.3'" =>
max({aa,bb,cc}): 1..10

```

On ouvre le prouveur interactif, on liste les hypothèses que l'on sait sur aa, bb et cc puis on lance la commande « pr » qui réussit à prouver l'obligation de preuve à partir des hypothèses.

The screenshot shows the Coq proof assistant interface. The main window displays a proof goal: "Check that the invariant (maxi: 1..10) is established by the initialisation - ref 3.3" => max({aa,bb,cc}): 1..10. The left sidebar shows the proof tree with a goal named 'pr'. Below it, the 'Situation' panel shows the current state: 'Initialisation' and 'PO1'. The bottom panel, 'Search hypothesis result', lists hypotheses for variables 'aa', 'bb', and 'cc'. The status bar at the bottom indicates 'Starting prover call' and 'Initialisation.1 100% (0 unproved)'.

## Réservation de places dans un avion

On souhaite maintenant réaliser une machine abstraite qui permet de réserver et annuler des places dans un avion. Cette machine est assez basique puisqu'elle ne gère que le nombre de places libres.

```

MACHINE
  Reservation
VARIABLES
  nbPlaceLibre
INVARIANT
  nbPlaceLibre : NAT
INITIALISATION
  nbPlaceLibre := 10
OPERATIONS
  reserver =
  PRE
    nbPlaceLibre >=1
  THEN

```

```

    nbPlaceLibre := nbPlaceLibre - 1
  END;
  annuler =
  PRE
    nbPlaceLibre <= 9
  THEN
    nbPlaceLibre := nbPlaceLibre + 1
  END
END

```

Les pré-conditions servent à prévoir les cas de débordement, car on ne peut annuler de place si toutes les places sont libres et inversement, on ne peut pas réserver de place si aucune place n'est libre.

Il y a 6 obligations de preuve générées suite à cette machine abstraite, qui consistent à vérifier l'invariant, c'est-à-dire que nbPlaceLibre est bien un entier naturel ( $0 \leq \text{nbPlaceLibre} \leq \text{MAXINT}$ ) lors de sa substitution dans les deux opérations « réserver » et « annuler ».

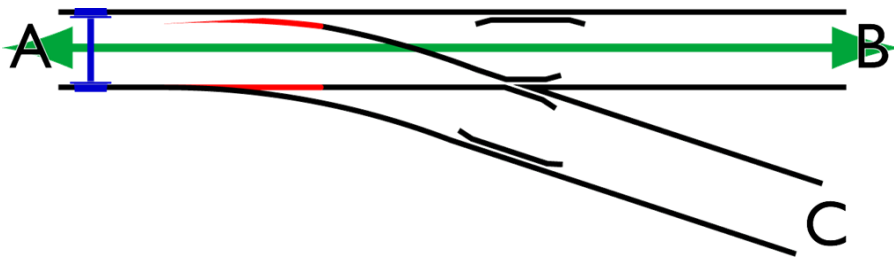
## Spécifier un logiciel et le mettre en œuvre en utilisant l'atelier B

Cette seconde partie du TP consiste à réaliser des machines abstraites plus élaborées, avec des opérations plus complexes et réalistes. On ira même jusqu'à la génération du code et son exécution.

### Aiguillage

On souhaite spécifier une fonction critique qui permet d'estimer la position d'un aiguillage à partir de 3 capteurs dont les valeurs  $\in \{\text{Normale}, \text{Renversée}, \text{Nulle}\}$ .

- Si au moins une valeur normale et une valeur renversée => résultat nul
- Si toutes les valeurs sont nulles => résultat nul
- Dans tous les autres cas => résultat normal ou inversé



La machine abstraite représentant la situation, en tenant compte des contraintes du sujet, est la suivante :

```

MACHINE
  switch
SETS
  POS = {NULL, NOR, RENVERSE}
OPERATIONS
  pos<--estimate(m1, m2, m3) =
  PRE
    m1 : POS &
    m2 : POS &
    m3 : POS
  THEN
  SELECT

```

```

    (m1 = NOR) & (m2 = NOR) & (m3 = NOR)
  THEN
    pos := NOR
  WHEN
    m1 = RENVERSE & m2 = RENVERSE & m3 = RENVERSE
  THEN
    pos := RENVERSE
  ELSE
    pos := NULL
  END
END
END

```

L'opération « estimate » prend les valeurs des 3 capteurs en paramètres et renvoie la position estimée de l'aiguillage. Le non-déterminisme dans la substitution est garanti par la construction SELECT... THEN... WHEN..., qui permet de représenter un choix borné gardé.

On passe maintenant à l'implémentation de cette machine abstraite d'aiguillage :

```

IMPLEMENTATION
  switch_i
REFINES
  switch
OPERATIONS
  pos<--estimate(m1,m2,m3) =
  IF (m1 = NOR) & (m2 = NOR) & (m3 = NOR)
  THEN
    pos := NOR
  ELSIF m1 = RENVERSE & m2 = RENVERSE & m3 = RENVERSE
  THEN
    pos := RENVERSE
  ELSE
    pos := NULL
  END
END
END

```

On redéfinit l'opération « estimate » par raffinement, en utilisant des instructions classiques des langages de programmation telles que IF, CASE, etc... (on se rapproche du code). L'opération devient donc déterministe car on a remplacé le SELECT par des IF (on aurait pu le faire avec un CASE aussi).

3 obligations de preuve sont générées à partir de l'implémentation (la machine abstraite ne génère aucune PO). Elles consistent à vérifier que pour chacun des cas du IF (m1=NOR et m2=NOR et m3=NOR ; m1=RENVERSE et m2=RENVERSE et m3=RENVERSE ; tous les autres cas => else), la substitution de pos respecte l'invariant c'est-à-dire qu'il appartient bien à l'ensemble abstrait POS (défini dans SETS). Ces POs sont prouvées automatiquement par le prouveur de l'atelier B.

On passe maintenant à la génération du code, ce qui nous amène à créer un programme de test :

```

#include <stdlib.h>
#include "../switch.h"
int main(int argc, char** argv) {

  switch__POS m1,m2,m3;
  m1= switch__RENVERSE; m2= switch__RENVERSE; m3= switch__RENVERSE;
  switch__POS res;
  switch__estimate(m1,m2,m3,&res);
  printf("Les 3 capteurs sont en position RENVERSE\n");
  switch (res){
    case switch__NULL :
      printf("Position de l'aiguillage : NULL\n"); break;

```

```

    case switch__NOR :
        printf("Position de l'aiguillage : NORMAL\n"); break;
    case switch__RENVERSE :
        printf("Position de l'aiguillage : RENVERSE\n"); break;
}

switch__POS m7,m8,m9;
m7= switch__NOR; m8= switch__NOR; m9= switch__NOR;
switch__POS res3;
switch__estimate(m7,m8,m9,&res3);
printf("-----\n3 capteurs en NORMAL\n");
switch (res3){
    case switch__NULL :
        printf("Position de l'aiguillage : NULL\n"); break;
    case switch__NOR :
        printf("Position de l'aiguillage : NORMAL\n"); break;
    case switch__RENVERSE :
        printf("Position de l'aiguillage : RENVERSE\n"); break;
}

switch__POS m4,m5,m6;
m4= switch__NOR; m5= switch__RENVERSE; m6= switch__RENVERSE;
switch__POS res2;
switch__estimate(m4,m5,m6,&res2);
printf("-----\n1 capteur en NORMAL et 2 autres en
RENVERSE\n");
switch (res2){
    case switch__NULL :
        printf("Position de l'aiguillage : NULL\n"); break;
    case switch__NOR :
        printf("Position de l'aiguillage : NORMAL\n"); break;
    case switch__RENVERSE :
        printf("Position de l'aiguillage : RENVERSE\n"); break;
}

switch__POS m10,m11,m12;
m10= switch__NULL; m11= switch__NULL; m12= switch__NULL;
switch__POS res4;
switch__estimate(m10,m11,m12,&res4);
printf("-----\n3 capteurs en NULL\n");
switch (res4){
    case switch__NULL :
        printf("Position de l'aiguillage : NULL\n"); break;
    case switch__NOR :
        printf("Position de l'aiguillage : NORMAL\n"); break;
    case switch__RENVERSE :
        printf("Position de l'aiguillage : RENVERSE\n"); break;
}
return (EXIT_SUCCESS);
}

```

On test donc les différents cas pour voir si tout fonctionne comme on l'a défini dans la machine abstraite puis décrit dans l'implémentation : à l'exécution, on s'aperçoit que tout est correct :

```

C:\cygwin\bin\sh.exe
Les 3 capteurs sont en position RENVERSE
Position de l'aiguillage : RENVERSE
-----
3 capteurs en NORMAL
Position de l'aiguillage : NORMAL
-----
1 capteur en NORMAL et 2 autres en RENVERSE
Position de l'aiguillage : NULL
-----
3 capteurs en NULL
Position de l'aiguillage : NULL
Press [Enter] to close the terminal ...

```

## Réservation de places dans un cinéma

Cette fois-ci, nous souhaitons décrire un service de réservation de places de cinéma par une machine abstraite. Le nombre de places du cinéma est passé en paramètre de la machine, et les places sont indicées. On souhaite créer des opérations pour tester s'il reste des places, réserver une place et libérer une place.

```

MACHINE
  Reservation(nb_max)
CONSTRAINTS
  nb_max : NAT1
DEFINITIONS
  SIEGE == 1..nb_max
VARIABLES
  occupes, nb_libres
INVARIANT
  occupes : seq(SIEGE) &
  nb_libres : 0..nb_max
INITIALISATION
  nb_libres := nb_max ||
  occupes := <>
OPERATIONS
  res<--place_libre =
  res := nb_libres;

  reserver =
  PRE
    nb_libres >= 1
  THEN
    ANY ss
    WHERE ss:SIEGE & ss /: ran(occupes)
    THEN
      occupes := occupes<-ss ||
      nb_libres := nb_libres - 1
    END
  END;

  liberer(place) =
  PRE
    place : SIEGE &
    place : ran(occupes) &
    nb_libres <= nb_max-1
  THEN
    occupes := occupes |>> {place} ||
    nb_libres := nb_libres + 1
  END
END

```

La variable « occupes » représente les sièges occupés, c'est-à-dire déjà réservés et la variable « nb\_libres » recense le nombre de places encore disponibles. L'opération « reserver » ne peut être réalisée que si au moins une place est disponible (pré-condition) et cette opération est rendue non-déterministe par l'utilisation de la structure ANY... WHERE... THEN... qui permet de réserver une des places disponibles, quelle qu'elle soit. L'opération « liberer » quant à elle prend en paramètre une place qui doit donc être préalablement réservée (pré-condition) et se charge de rendre cette place à nouveau disponible.

6 obligations de preuves sont générées par l'atelier B : 2 pour vérifier que les substitutions dans l'initialisation respectent bien l'invariant, 2 pour l'opération « reserver » lors de la substitution dans

le bloc THEN (en tenant compte des hypothèses décrites dans les blocs PRE et WHERE). Les deux dernières PO concernent l'opération « libérer », afin de garantir que les 2 substitutions respectent bien l'invariant. Toutes ces obligations de preuve ont été prouvées par le prouveur automatique de l'atelier B.

## Conclusion

Ces deux TP nous ont permis d'assimiler la syntaxe de l'atelier B et de mettre en pratique les éléments théoriques vus en cours et en TD, sur des exercices concrets.

Tout d'abord, le processus machine abstraite, raffinement, implémentation puis génération du code est un processus très intéressant car il permet de spécifier les besoins de façon très abstraite en premier lieu (ce qui est parfois déroutant au début) et de ne se soucier des aspects de programmation qu'à partir de la phase d'implémentation.

Les obligations de preuves générées et prouvées par l'atelier B permettent d'assurer que le programme va fonctionner tel qu'il a été décrit, grâce au fait que le B est une méthode formelle, basée sur la théorie des ensembles et des prédicats du premier ordre.

La seule faille du processus réside en la traduction de la spécification en langage naturel en machine abstraite, d'où la nécessité d'être très vigilant lors de cette phase.