

Exemples de conduite de preuve interactive avec l'*AtelierB*, version 3.5

Didier Bert, LSR-IMAG

24 février 2000

Résumé

Ce rapport décrit la validation complète d'un petit développement réalisé en **B**. On montre comment réaliser interactivement les démonstrations des obligations de preuves non prouvées automatiquement par l'*AtelierB*.

1 Introduction

Le but de ce rapport est d'initier à l'utilisation du démonstrateur interactif de l'*AtelierB*. Pour cela, on applique l'outil à la démonstration des obligations de preuve d'un exemple d'école. Cela donne une idée de la suite complète des étapes qui consistent à partir d'un modèle décrit par une machine pour aboutir finalement à une génération de code complètement validée. L'exemple choisi a été fourni par Marie-Laure Potet. Il s'agit d'une variante de celui de la "réservation" du B-Book, donné par Jean-Raymond Abrial [Abr96]. Les machines et raffinements sont donnés sans beaucoup de commentaires dans le paragraphe 2. On décrit ensuite la validation des différents modules. La machine initiale est validée par le démonstrateur automatique (paragraphe 3). Pour les autres composants (raffinements et implémentation), on détaille les obligations de preuve qui n'ont pas été démontrées automatiquement. Pour chacune, on indique qu'elle est l'idée de la preuve, c'est-à-dire la façon un peu informelle de se convaincre de la validité de la formule. On donne ensuite la réalisation de cette preuve à l'aide des commandes de l'*AtelierB* (paragraphe 4, 5 et 6).

Les scripts de preuve fournis dans ce rapport ne sont que des exemples de preuves possibles. Il est vraisemblable qu'il existe des preuves beaucoup plus courtes. De plus, le style de preuve peut être différent d'une personne à une autre. On a essayé de se laisser guider par la logique du problème, en explicitant pourquoi la preuve a été développée de telle ou telle manière. Toute suggestion pour améliorer ces preuves et les rendre plus fluides est la bienvenue. D'autre part, ces exemples ne donnent qu'un aperçu des diverses commandes de l'*AtelierB*. En conclusion (paragraphe 7), on indique quelles sont les autres commandes qui n'ont pas été utilisées, à titre d'information. On n'a pas du tout visé l'exhaustivité de l'utilisation du prouveur. De toutes façons, de nombreuses commandes et fonctionnalités de l'*AtelierB* n'apparaissent pas dans un script de preuve: ce sont les commandes de navigation, de recherche de règles ou d'hypothèses, et les commandes de création de nouvelles théories. Ces commandes ne sont pas abordées dans ce rapport. L'objectif plus modeste est simplement de montrer que faire une démonstration interactivement n'est pas une tâche insurmontable. La conclusion aborde quelques idées d'amélioration qui permettraient de faciliter l'activité de preuve.

2 Machines et raffinements de la réservation

Par rapport à l'exemple du B-Book, cette réservation permet de gérer les numéros des sièges occupés (variable *occupes* de l'état). Il a donc une substitution de choix (ANY) dans l'opération "reserver". D'autre part, la précondition de cette opération peut être testée de l'extérieur de la machine, grâce à l'opération auxiliaire "place_libre".

```

MACHINE                               /* entête de la machine et partie statique */
  RESERVATION(nb_max)
CONSTRAINTS
  nb_max ∈ 1..1000
DEFINITIONS                             /* définition auxiliaire */
  SIEGES == (1..nb_max)
VARIABLES                               /* état de la machine */
  occupes
INVARIANT
  occupes ∈ F(SIEGES)
INITIALISATION                          /* valeur initiale de l'état */
  occupes := ∅
OPERATIONS                               /* Opérations (partie dynamique) */
  nb ← place_libre =                    /* calcul du nombre de places libres */
  BEGIN
    nb := nb_max - card(occupes)
  END
;
  place ← reserver =                    /* opération de réservation */
  PRE
    nb_max - card(occupes) ≠ 0
  THEN
    ANY pp WHERE
      pp ∈ SIEGES - occupes
    THEN
      place, occupes := pp, occupes ∪ {pp}
    END
  END
;
  liberer(place) =                      /* opération de libération */
  PRE
    place ∈ SIEGES ∧ place ∈ occupes
  THEN
    occupes := occupes - {place}
  END
END

```

Le premier raffinement consiste à la fois à remplacer le calcul du nombre de places libres par une variable qui contient ce nombre (variable *nb_libre*) et à remplacer l'ensemble *occupes* par une fonction caractéristique *etat* qui associe à chaque numéro de place une valeur booléenne avec la signification :

$$etat(pp) = \text{TRUE} \Leftrightarrow pp \in occupes$$

Cette propriété est traduite dans l'invariant de collage par la formule :

$$occupes = etat^{-1}[\{TRUE\}]$$

Les modifications des opérations sont une simple reformulation qui tient compte de ces changements de variables.

```

REFINEMENT                /* entête du raffinement */
  RESERVATION1(nb_max)
REFINES                    /* machine raffinée */
  RESERVATION
DEFINITIONS                /* définition auxiliaire */
  SIEGES == (1..nb_max)
VARIABLES                  /* état de la machine */
  etat
CONCRETE_VARIABLES        /* cette variable ne sera plus raffinée */
  nb_libre
INVARIANT
  etat ∈ SIEGES → BOOL ∧
  nb_libre ∈ 0..nb_max ∧
  occupes = etat-1[\{TRUE\}] ∧
  nb_libre = nb_max - card(occupes)
INITIALISATION            /* valeur initiale de l'état */
  etat, nb_libre := SIEGES × \{FALSE\}, nb_max
OPERATIONS                 /* Opérations raffiées */
  nb ← place_libre =      /* valeur du nombre de places libres */
  BEGIN
    nb := nb_libre
  END
;
  place ← reserver =      /* opération de réservation */
  ANY pp1 WHERE
    pp1 ∈ etat-1[\{FALSE\}]
  THEN
    place, etat(pp1), nb_libre := pp1, TRUE, nb_libre - 1
  END
;
  liberer(place) =        /* opération de libération */
  BEGIN
    etat(place), nb_libre := FALSE, nb_libre + 1
  END
END

```

Dans le dernier raffinement qui est une implémentation, l'état (fonction totale) est implémenté par une machine de la bibliothèque de base *BASIC_ARRAY_VAR* dont on donne en paramètres le domaine et le codomaine. Cette machine contient les opérations *STR_ARRAY* pour ranger une valeur à un indice donné du tableau et *VAL_ARRAY* qui retourne la valeur pour un indice donné. Par rapport au raffinement précédent, l'initialisation contient une boucle pour initialiser le tableau à **FALSE** et l'opération *reserver* contient une boucle pour

trouver le premier indice qui représente une place libre. L'assertion permet de factoriser la preuve que l'indice *ind* est toujours strictement inférieur à *nb_max* à l'intérieur de la boucle.

```

IMPLEMENTATION          /* entête de l'implémentation */
  CODE(nb_max)
REFINES                 /* module raffiné */
  RESERVATION1
IMPORTS                 /* machines importées */
  BASIC_ARRAY_VAR(1..nb_max, BOOL)
INVARIANT               /* la variable d'état du tableau est arr_vrb qui est
  etat = arr_vrb        /* une fonction totale du domaine dans le codomaine */
INITIALISATION         /* initialisation du tableau */
  VAR ind IN
    nb_libre := nb_max ;
    ind := 1 ;
    WHILE ind ≤ nb_max DO
      STR_ARRAY(ind, FALSE) ;
      ind := ind + 1
    INVARIANT
      ind ∈ 1..nb_max + 1 ∧
      (1 < ind ⇒ arr_vrb[1..ind - 1] = {FALSE})
    VARIANT
      nb_max + 1 - ind
    END
  END
OPERATIONS
  nb ← place_libre =    /* valeur du nombre de places libres */
  BEGIN
    nb := nb_libre
  END
;
place ← reserver =     /* opération de réservation */
  VAR ind, bb IN
    ind := 1 ;
    bb ← VAL_ARRAY(ind) ;
    WHILE bb = TRUE DO
      ASSERT ind < nb_max THEN
        ind := ind + 1 ;
        bb ← VAL_ARRAY(ind)
      END
    INVARIANT
      ind ∈ 1..nb_max ∧ bb = arr_vrb(ind) ∧
      FALSE ∈ arr_vrb[ind..nb_max]
    VARIANT
      nb_max - ind
    END ;
    STR_ARRAY(ind, TRUE) ;
    nb_libre := nb_libre - 1 ;
    place := ind
  END
;

```

```

liberer(place) =          /* opération de libération */
  BEGIN
    STR_ARRAY(place, FALSE) ;
    nb_libre := nb_libre + 1
  END
END

```

Pour pouvoir utiliser cet exemple dans une démo, on a un programme principal qui appelle les opérations de la machine de réservation. Il donne des commentaires au fur et à mesure des appels.

```

MACHINE
  USER1
OPERATIONS
  main =
    BEGIN
      skip
    END
END

```

```

IMPLEMENTATION          /* entête de l'implémentation */
  USER1_i
REFINES                 /* module raffiné */
  USER1
IMPORTS                 /* machines importées */
  RESERVATION(2), BASIC_IO
OPERATIONS
  main =
    VAR num1, num2, num3, reste IN
      reste ← place_libre ;
    IF reste = 0 THEN
      STRING_WRITE("Plus de place\n")
    ELSE
      STRING_WRITE("Premiere reservation\n");
      num1 ← reserver ;
      reste ← place_libre ;
    IF reste = 0 THEN
      STRING_WRITE("Plus de place\n")
    ELSE
      STRING_WRITE("Seconde reservation\n");
      num2 ← reserver ;
      reste ← place_libre ;
    IF reste = 0 THEN
      STRING_WRITE("Liberation\n");
      liberer(num1)
    ELSE
      STRING_WRITE("Il reste des places\n")
    END ;
    STRING_WRITE("Troisieme reservation\n");
    num3 ← reserver ;
    reste ← place_libre ;
    IF reste ≠ 0 THEN
      STRING_WRITE("Il reste des places\n")
    END
  END
END
END
END
END

```

3 Validation de la machine RESERVATION.mch

La machine est entièrement validée par le prouveur automatique. Le tableau récapitulatif donne :

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	0	1	0	1	100
place_libre	2	0	0	0	100
reserver	1	1	0	1	100
liberer	1	1	0	1	100
RESERVATION	4	3	0	3	100

Examinons dans le détail les trois obligations de preuve non triviales fournies par l'*AtelierB*. On note une certaine redondance dans les formules engendrées, en particulier pour les intervalles où les relations avec les bornes sont rendues explicites. Les trois OP apparaissent ici avec les commentaires du démonstrateur. Pour l'initialisation, on a :

“Component constraints”
 $nb_max \in 1..1000 \wedge$
 $btrue \wedge$
 $1 \leq nb_max \wedge$
 $nb_max \leq 1000 \wedge$
 “Check that the invariant ($occupes \in \mathbb{F}(1..nb_max)$) is established by the initialisation - ref 3.3”
 \Rightarrow
 $\emptyset \in \mathbb{F}(1..nb_max)$

De même pour l'opération *reserver*, la seule obligation de preuve restante est (notons la façon dont est générée l'appartenance à une différence d'ensembles (“Local hypotheses”)) :

“Component constraints”
 $nb_max \in 1..1000 \wedge$
 $btrue \wedge$
 $1 \leq nb_max \wedge$
 $nb_max \leq 1000 \wedge$
 “Component invariant”
 $occupes \in \mathbb{F}(1..nb_max) \wedge$
 “reserver preconditions in this component”
 $\neg (nb_max - \text{card}(occupes) = 0) \wedge$
 “Local hypotheses”
 $pp \in 1..nb_max \wedge$
 $\neg (pp \in occupes)$
 “Check that the invariant ($occupes \in \mathbb{F}(1..nb_max)$) is preserved by the operation - ref 3.4”
 \Rightarrow
 $occupes \cup \{pp\} \in \mathbb{F}(1..nb_max)$

Enfin, pour l'obligation de preuve de l'opération *liberer*, on a :

<p>“Component constraints”</p> $nb_max \in 1..1000 \wedge$ $btrue \wedge$ $1 \leq nb_max \wedge$ $nb_max \leq 1000 \wedge$ <p>“Component invariant”</p> $occupes \in \mathbb{F}(1..nb_max) \wedge$ <p>“liberer preconditions in this component”</p> $place \in 1..nb_max \wedge$ $place \in occupes \wedge$ $1 \leq place \wedge$ $place \leq nb_max$ <p>“Check that the invariant ($occupes \in \mathbb{F}(1..nb_max)$) is preserved by the operation - ref 3.4”</p> \Rightarrow $occupes - \{place\} \in \mathbb{F}(1..nb_max)$
--

4 Validation du raffinement RESERVATION1.ref

L'état des obligations de preuve après le passage en automatique force 1 est :

	NbObv	NbP0	NbPRi	NbPRa	%Pr
Initialisation	1	5	0	5	100
place_libre	7	0	0	0	100
reserver	2	7	0	5	71
liberer	2	5	0	3	60
RESERVATION1	12	17	0	13	76

La liste des buts des OP générées est donnée par la commande `gs` (global situation). Elle donne :

```
PRI > State of all P0
Initialisation
  P01 Proved      ((1..nb_max)*{FALSE})~[{TRUE}] = {}
  P02 Proved      (1..nb_max)*{FALSE}: 1..nb_max +-> BOOL
  P03 Proved      dom((1..nb_max)*{FALSE}) = 1..nb_max
  P04 Proved      nb_max: 0..nb_max
  P05 Proved      nb_max = nb_max-card(((1..nb_max)*{FALSE})~[{TRUE}])
reserver
  P01 Proved      pp1: 1..nb_max
  P02 Proved      not(pp1: occupes)
  P03 Proved      (etat$1<+{pp1|->TRUE})~[{TRUE}] = occupes\/{pp1}
  P04 Proved      etat$1<+{pp1|->TRUE}: 1..nb_max +-> BOOL
  P05 Proved      dom(etat$1<+{pp1|->TRUE}) = 1..nb_max
  P06 Unproved   nb_libre$1-1: 0..nb_max
  P07 Unproved   nb_libre$1-1 = nb_max-card((etat$1<+{pp1|->TRUE})~[{TRUE}])
liberer
  P01 Unproved   (etat$1<+{place|->FALSE})~[{TRUE}] = occupes-{place}
  P02 Proved      etat$1<+{place|->FALSE}: 1..nb_max +-> BOOL
```

P03 Proved $\text{dom}(\text{etat}\$1 \leftarrow \{\text{place} \rightarrow \text{FALSE}\}) = 1..nb_max$
P04 Proved $nb_libre\$1+1: 0..nb_max$
P05 Unproved $nb_libre\$1+1 = nb_max - \text{card}((\text{etat}\$1 \leftarrow \{\text{place} \rightarrow \text{FALSE}\}) \sim \{\text{TRUE}\})$

End

Les hypothèses globales pour toutes les opérations du module `RESERVATION1.ref` sont:

“Component constraints”	
$nb_max \in 1..1000 \wedge$	(1) Mêmes conditions que pour
$btrue \wedge$	(2) la machine <code>RESERVATION</code>
$1 \leq nb_max \wedge$	(3)
$nb_max \leq 1000 \wedge$	(4)
“Previous components invariants”	
$occupes \in \mathbb{F}(1..nb_max) \wedge$	(5)
“Component invariant”	
$\text{etat}\$1 \in 1..nb_max \leftrightarrow \text{BOOL} \wedge$	(6) Les formules (6) et (7)
$\text{dom}(\text{etat}\$1) = 1..nb_max \wedge$	(7) codent la fonction totale
$nb_libre\$1 \in 0..nb_max \wedge$	(8)
$occupes = \text{etat}\$1^{-1}[\{\text{TRUE}\}] \wedge$	(9)
$nb_libre\$1 = nb_max - \text{card}(occupes) \wedge$	(10)
$0 \leq nb_libre\$1 \wedge$	(11) Explicitation des
$nb_libre\$1 \leq nb_max$	(12) relations d’intervalle

4.1 Opération reserver

Les hypothèses de l’opération `reserver` héritées de la machine `RESERVATION` sont :

“reserver preconditions in previous components”	
$\neg (nb_max - \text{card}(occupes) = 0)$	(13)

4.1.1 Preuve de reserver.6

L’obligation de preuve donnée par l’*AtelierB* est :

<p>“reserver preconditions in this component”</p> <p style="padding-left: 20px;">$\neg (nb_max - \text{card}(occupes) = 0) \wedge$</p> <p>“Local hypotheses”</p> <p style="padding-left: 20px;">$pp1 \in \text{etat}\\$1^{-1}[\{\text{FALSE}\}]$</p> <p>“Check that the invariant $(nb_libre \in 0..nb_max)$ is preserved by the operation - ref 4.4, 5.5”</p> <p style="padding-left: 20px;">\Rightarrow</p> <p style="padding-left: 20px;">$nb_libre\\$1 - 1 \in 0..nb_max$</p>

Pour cette démonstration, on va utiliser l’égalité de l’hypothèse (10) dans la précondition (13) pour construire le lemme :

$$\neg nb_libre\$1 = 0$$

et ensuite combiner avec l’hypothèse de déclaration (8) pour démontrer le but. Dans les preuves, les commandes du démonstrateur sont les lignes qui commencent par le symbole “>”. Pour la signification des commandes, se reporter au Manuel de Référence du prouveur interactif [B98] ou au “Memento” [Ber00]. Indiquons ici que “**dd**” permet de mettre les

antécédents d’une implication dans les hypothèses. La commande “**eh**” réalise une réécriture du but ou d’une hypothèse, en utilisant une égalité en hypothèse. “**ah**” est l’ajout d’une hypothèse. Si cette hypothèse est nouvelle, elle doit d’abord être démontrée, sinon (s’il s’agit d’une véritable hypothèse de l’OP), elle est mise en antécédent d’implication, comme c’est le cas ici (opération symétrique de **dd**). Enfin “**pp**” est l’appel au “predicate prover”. On lui indique en paramètre quelles sont les hypothèses à examiner. Ici “**rp.0**” signifie qu’il n’a pas besoin d’hypothèses : toute l’information pour la preuve se trouve dans le but. Le but courant, affiché par l’*AtelierB*, est indiqué après chaque commande, précédé du symbole “:” et décalé d’une tabulation constante. Les décalages du script de preuve sont faits par l’atelier pour visualiser les branches de la preuve. On indique que le but est démontré par le mot-clé “**PROVED**”, qui ne fait pas partie des messages de l’*AtelierB*. Le script de preuve et les sous-buts calculés sont :

```
> dd
:           nb_libre$1-1: 0..nb_max
>  eh(nb_max-card(occupes),nb_libre$1,Hyp(not(nb_max-card(occupes) = 0)))
:           not(nb_libre$1 = 0) => nb_libre$1-1: 0..nb_max
>  ah(nb_libre$1: 0..nb_max)
:           nb_libre$1: 0..nb_max => (not(nb_libre$1 = 0) => nb_libre$1-1: 0..nb_max)
>  pp(rp.0)
:           PROVED
```

4.1.2 Preuve de `reserver.7`

“reserver preconditions in this component”
 $\neg(nb_max - \text{card}(occupes) = 0) \wedge$
 “Local hypotheses”
 $pp1 \in \text{etat}\$1^{-1}\{\{\text{FALSE}\}\}$
 “Check that the invariant $(nb_libre = nb_max - \text{card}(occupes))$ is preserved
 by the operation - ref 4.4, 5.5”
 \Rightarrow
 $nb_libre\$1 - 1 = nb_max - \text{card}((\text{etat}\$1 \Leftarrow \{pp1 \mapsto \text{TRUE}\})^{-1}\{\{\text{TRUE}\}\})$

En examinant le but et les hypothèses, on voit qu’il faut arriver à extraire le fragment de fonction : $(\{pp1 \mapsto \text{TRUE}\})^{-1}\{\{\text{TRUE}\}\}$ qui se réduit à $\{pp1\}$. On espère ensuite arriver à une expression de la forme $nb_max - \text{card}(\text{etat}\$1 \cup \{pp1\})$ qui, avec l’hypothèse locale $pp1 \in \text{etat}\$1^{-1}\{\{\text{FALSE}\}\}$ se réduit à $nb_max - (\text{card}(\text{etat}\$1) + 1)$. En utilisant alors les hypothèses (9) et (10) et la commande de réécriture, on arrive à démontrer le but. Dans cette preuve, on utilise abondamment la commande “**ar**” qui consiste à appliquer une règle de la base de règles du démonstrateur.

La première étape est de transformer la modification de fonction par surcharge en une union de fonction. Il s’agit d’une transformation standard qui peut toujours être utilisée [Ber98]. On arrive ainsi à la forme :

$$((\{pp1\} \Leftarrow \text{etat}\$1) \cup \{pp1 \mapsto \text{TRUE}\})^{-1}\{\{\text{TRUE}\}\}$$

Une recherche dans la base de règles montre qu’on dispose de nombreuses règles de simplification qui peuvent être utilisées comme règles de réécriture de gauche à droite (symbole “**==**”).

Celles que l'on utilise dans cette démonstration sont :

SimplifyRelInvXY.13	$(r \cup s)^{-1}$	$==$	$r^{-1} \cup s^{-1}$
SimplifyRelInvXY.2	$\{a \mapsto b\}^{-1}$	$==$	$\{b \mapsto a\}$
SimplifyRelImaLongXY.5	$(r \cup s)[u]$	$==$	$r[u] \cup s[u]$
SimplifyRelImaXY.29	$(u \triangleleft r)^{-1}[v]$	$==$	$r^{-1}[v] - u$
SimplifyRelImaXY.42	$\{a \mapsto b\}[\{a\}]$	$==$	$\{b\}$

De plus, on a la règle de simplification du cardinal d'un ensemble qui est (on a supprimé les conditions de non capture de variables par filtrage) :

SimplifyEnsCard.9	$b \notin a$
	\Rightarrow
	$\text{card}(a \cup \{b\}) == \text{card}(a) + 1$

D'où le script :

```
> dd
:      nb_libre$1-1 = nb_max-card((etat$1<+{pp1|->TRUE})~[{TRUE}])
> ah(etat$1<+{pp1|->TRUE} = {pp1}<<|etat$1\/{pp1|->TRUE})
:      etat$1<+{pp1|->TRUE} = {pp1}<<|etat$1\/{pp1|->TRUE}
> pp(rp.0)
:      etat$1<+{pp1|->TRUE} = {pp1}<<|etat$1\/{pp1|->TRUE} =>
      nb_libre$1-1 = nb_max-card((etat$1<+{pp1|->TRUE})~[{TRUE}])
> dd
:      nb_libre$1-1 = nb_max-card((etat$1<+{pp1|->TRUE})~[{TRUE}])
> eh(etat$1<+{pp1|->TRUE},_h,Goal)
:      nb_libre$1-1 = nb_max-card((\{pp1}<<|etat$1\/{pp1|->TRUE})~[{TRUE}])
> ar(SimplifyRelInvXY.13,Goal)
:      nb_libre$1-1 = nb_max-card(((\{pp1}<<|etat$1)\/{pp1|->TRUE})~[{TRUE}])
> ar(SimplifyRelInvXY.2,Goal)
:      nb_libre$1-1 = nb_max-card(((\{pp1}<<|etat$1)\/{TRUE|->pp1})~[{TRUE}])
> ar(SimplifyRelImaLongXY.5,Goal)
:      nb_libre$1-1 = nb_max-card((\{pp1}<<|etat$1)~[{TRUE}]\/{TRUE|->pp1})~[{TRUE}])
> ar(SimplifyRelImaXY.29,Goal)
:      nb_libre$1-1 = nb_max-card(etat$1~[{TRUE}]-\{pp1}\/{TRUE|->pp1})~[{TRUE}])
> ar(SimplifyRelImaXY.42,Goal)
:      nb_libre$1-1 = nb_max-card(etat$1~[{TRUE}]-\{pp1}\/{pp1})
> ah(not(pp1: etat$1~[{TRUE}]))
: not(pp1: etat$1~[{TRUE}])
> ah(pp1: etat$1~[{FALSE}])
: pp1: etat$1~[{FALSE}] => not(pp1: etat$1~[{TRUE}])
> ah(etat$1: 1..nb_max +-> BOOL)
: etat$1: 1..nb_max +-> BOOL => (pp1: etat$1~[{FALSE}] => not(pp1: etat$1~[{TRUE}]))
> pp(rp.0)
: not(pp1: etat$1~[{TRUE}]) => nb_libre$1-1 = nb_max-card(etat$1~[{TRUE}]-\{pp1}\/{pp1})
> dd
:      nb_libre$1-1 = nb_max-card(etat$1~[{TRUE}]-\{pp1}\/{pp1})
> ah(etat$1~[{TRUE}]-\{pp1} = etat$1~[{TRUE}])
:      etat$1~[{TRUE}]-\{pp1} = etat$1~[{TRUE}]
> ah(not(pp1: etat$1~[{TRUE}]))
: not(pp1: etat$1~[{TRUE}]) => etat$1~[{TRUE}]-\{pp1} = etat$1~[{TRUE}]
> pp(rp.0)
:      etat$1~[{TRUE}]-\{pp1} = etat$1~[{TRUE}] =>
      nb_libre$1-1 = nb_max-card(etat$1~[{TRUE}]-\{pp1}\/{pp1})
> dd
```

```

:      nb_libre$1-1 = nb_max-card(etat$1~[{\TRUE}]-{\pp1}\/{pp1})
>      eh(etat$1~[{\TRUE}]-{\pp1},_h,Goal)
:      nb_libre$1-1 = nb_max-card(etat$1~[{\TRUE}]\/{pp1})
>      ar(SimplifyEnsCard.9,Goal)
:      nb_libre$1-1 = nb_max-(card(etat$1~[{\TRUE}])+1)
>      eh(etat$1~[{\TRUE}],occupes,Goal)
:      nb_libre$1-1 = nb_max-(card(occupes)+1)
>      ah(nb_max-(card(occupes)+1) = nb_max-card(occupes)-1)
:      nb_max-(card(occupes)+1) = nb_max-card(occupes)-1
>      pp(rp.0)
:      nb_max-(card(occupes)+1) = nb_max-card(occupes)-1 =>
nb_libre$1-1 = nb_max-(card(occupes)+1)
>      dd
:      nb_libre$1-1 = nb_max-(card(occupes)+1)
>      eh(nb_max-(card(occupes)+1),_h,Goal)
:      nb_libre$1-1 = nb_max-card(occupes)-1
>      eh(nb_max-card(occupes),nb_libre$1,Goal)
:      nb_libre$1-1 = nb_libre$1-1
>      pp(rp.0)
:      PROVED

```

4.2 Opération liberer

Les hypothèses de l'opération `liberer` héritées de la machine `RESERVATION` sont :

“liberer preconditions in previous components”

$$place \in 1..nb_max \wedge \quad (14)$$

$$place \in occupes \wedge \quad (15)$$

$$1 \leq place \wedge \quad (16)$$

$$place \leq nb_max \quad (17)$$

4.2.1 Preuve de liberer.1

“liberer preconditions in this component”

$$place \in 1..nb_max \wedge$$

$$place \in occupes$$

“Check that the invariant ($occupes = etat^{-1}[\{\TRUE\}]$) is preserved by the operation - ref 4.4, 5.5” &

“Check operation refinement - ref 4.4, 5.5”

\Rightarrow

$$(etat\$1 \Leftarrow \{place \mapsto \text{FALSE}\})^{-1}[\{\TRUE\}] = occupes - \{place\}$$

Comme pour `reserver.7`, il faut d'abord transformer la modification de fonction en union, ce qui donne l'expression :

$$(\{place\} \Leftarrow etat\$1) \cup \{place \mapsto \text{FALSE}\}$$

sur laquelle on peut appliquer les simplifications de fonction inverse et d'image par les règles données ci-dessus. On arrive alors à un sous-but de la forme :

$$etat\$1^{-1}[\{\TRUE\}] - \{place\} \cup \{\text{FALSE} \mapsto place\}[\{\TRUE\}] = occupes - \{place\}$$

La règle `SimplifyRelImaXY.42` ne s’applique pas, mais il existe d’autres règles intéressantes de la base :

$$\begin{aligned} \text{SimplifyRelImaXY.36} \quad \{ \text{FALSE} \mapsto a \} \{ \{ \text{TRUE} \} \} &== \emptyset \\ \text{SimplifyRelImaXY.37} \quad \{ \text{TRUE} \mapsto a \} \{ \{ \text{FALSE} \} \} &== \emptyset \end{aligned}$$

Un simple remplacement par l’hypothèse (9) permet d’achever la démonstration. La commande “`pr`” est l’appel du démonstrateur automatique sur le but courant.

```
> dd
:      (etat$1<+{place|->FALSE})~[{TRUE}] = occupes-{place}
> ah(etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE})
:      etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE}
>   pp(rp.0)
:      etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE} =>
:      (etat$1<+{place|->FALSE})~[{TRUE}] = occupes-{place}
>   dd
:      (etat$1<+{place|->FALSE})~[{TRUE}] = occupes-{place}
>   eh(etat$1<+{place|->FALSE},_h,Goal)
:      ({place}<<|etat$1\/{place|->FALSE})~[{TRUE}] = occupes-{place}
>   ar(SimplifyRelInvXY.13,Goal)
:      (({place}<<|etat$1)~\/{place|->FALSE})~[{TRUE}] = occupes-{place}
>   ar(SimplifyRelImaLongXY.5,Goal)
:      ({place}<<|etat$1)~[{TRUE}]\/{place|->FALSE}~[{TRUE}] = occupes-{place}
>   ar(SimplifyRelImaXY.29,Goal)
:      etat$1~[{TRUE}]-{place}\/{place|->FALSE}~[{TRUE}] = occupes-{place}
>   ar(SimplifyRelInvXY.2,Goal)
:      etat$1~[{TRUE}]-{place}\/{FALSE|->place}~[{TRUE}] = occupes-{place}
>   ar(SimplifyRelImaXY.36,Goal)
:      etat$1~[{TRUE}]-{place}\/{ } = occupes-{place}
>   eh(etat$1~[{TRUE}],occupes,Goal)
:      occupes-{place}\/{ } = occupes-{place}
>   pr
:      PROVED
```

4.2.2 Preuve de liberer.5

“liberer preconditions in this component”
 $place \in 1..nb_max \wedge$
 $place \in occupes$
 “Check that the invariant $(nb_libre = nb_max - \text{card}(occupes))$ is preserved
 by the operation - ref 4.4, 5.5”
 \Rightarrow
 $nb_libre\$1 + 1 = nb_max - \text{card}((etat\$1 \Leftarrow \{place \mapsto \text{FALSE}\})^{-1}\{ \{ \text{TRUE} \} \})$

On reprend la démarche déjà utilisée plusieurs fois pour remplacer la modification de fonction par une union et ensuite pour réécrire le but le plus possible. Par une utilisation des règles de simplification d’expressions d’ensemble :

$$\begin{aligned} \text{SimplifySetUniXY.35} \quad \emptyset \cup a &== a \\ \text{SimplifySetUniXY.36} \quad a \cup \emptyset &== a \end{aligned}$$

on arrive ainsi au sous-but :

$$nb_libre\$1 + 1 = nb_max - \text{card}(etat\$1^{-1}\{ \{ \text{TRUE} \} \} - \{place\})$$

Pour le cardinal d'une différence d'ensembles, on n'a pas une règle symétrique de celle de l'union (`SimplifyEnsCard.9`) dans la base. En revanche, on trouve :

$$\begin{array}{lcl} \text{SimplifyTryEnsCard.3} & \text{card}(A - B) & == \text{card}(A) - \text{card}(A \cap B) \\ \text{SimplifyEnsCard.16} & \text{card}(\{a\}) & == 1 \end{array}$$

Cela permet de poursuivre la démonstration, mais il faut passer par une propriété intermédiaire qui est facilement démontrée par le “predicate prouver” et qui est :

$$place \in occupes \Rightarrow occupes \cap \{place\} = \{place\}$$

Enfin, l'utilisation du “predicate prover” en regardant uniquement les hypothèses qui contiennent des symboles du but (`pp(rp.1)`) conduit à la démonstration complète de cette obligation de preuve.

```
> dd
:      nb_libre$1+1 = nb_max-card((etat$1<+{place|->FALSE})~[TRUE])
> ah(etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE})
:      etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE}
> pp(rp.0)
:      etat$1<+{place|->FALSE} = {place}<<|etat$1\/{place|->FALSE} =>
      nb_libre$1+1 = nb_max-card((etat$1<+{place|->FALSE})~[TRUE])
> dd
:      nb_libre$1+1 = nb_max-card((etat$1<+{place|->FALSE})~[TRUE])
> eh(etat$1<+{place|->FALSE},_h,Goal)
:      nb_libre$1+1 = nb_max-card(({place}<<|etat$1\/{place|->FALSE})~[TRUE])
> ar(SimplifyRelInvXY.13,Goal)
:      nb_libre$1+1 = nb_max-card((({place}<<|etat$1)~\/{place|->FALSE})~[TRUE])
> ar(SimplifyRelInvXY.2,Goal)
:      nb_libre$1+1 = nb_max-card((({place}<<|etat$1)~\/{FALSE|->place})[TRUE])
> ar(SimplifyRelImaLongXY.5,Goal)
:      nb_libre$1+1 = nb_max-card((({place}<<|etat$1)~[TRUE])\/{FALSE|->place}[TRUE])
> ar(SimplifyRelImaXY.29,Goal)
:      nb_libre$1+1 = nb_max-card(etat$1~[TRUE]-{place}\/{FALSE|->place}[TRUE])
> ar(SimplifyRelImaXY.36,Goal)
:      nb_libre$1+1 = nb_max-card(etat$1~[TRUE]-{place}\/{})
> ar(SimplifySetUniXY.36,Goal)
:      nb_libre$1+1 = nb_max-card(etat$1~[TRUE]-{place})
> ar(SimplifyTryEnsCard.3,Goal)
:      nb_libre$1+1 = nb_max-(card(etat$1~[TRUE])-card(etat$1~[TRUE]\/{place}))
> eh(etat$1~[TRUE],occupes,Goal)
:      nb_libre$1+1 = nb_max-(card(occupes)-card(occupes/\{place}))
> ah(occupes/\{place} = {place})
:      occupes/\{place} = {place}
> ah(place: occupes)
:      place: occupes => occupes/\{place} = {place}
> pp(rp.0)
:      occupes/\{place} = {place} =>
      nb_libre$1+1 = nb_max-(card(occupes)-card(occupes/\{place}))
> dd
:      nb_libre$1+1 = nb_max-(card(occupes)-card(occupes/\{place}))
> eh(occupes/\{place},_h,Goal)
:      nb_libre$1+1 = nb_max-(card(occupes)-card({place}))
> ar(SimplifyEnsCard.16,Goal)
:      nb_libre$1+1 = nb_max-(card(occupes)-1)
> pp(rp.1) &
:      PROVED
```

Une dernière remarque à propos de cette démonstration et des précédentes : on utilise assez systématiquement **pp** pour prouver un lemme auxiliaire, plutôt que **pr**. En effet, il se trouve que la première commande est souvent plus puissante (si on a restreint son champ d'exploration des hypothèses) et surtout qu'elle ne modifie pas la structure du but et donc de la preuve que l'on est en train de faire. Il est assez fréquent que l'on ne sache plus où l'on en est de la démonstration lorsqu'on utilise la commande **pr**.

4.3 Etat final du composant RESERVATION1.ref

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	1	5	0	5	100
place_libre	7	0	0	0	100
reserver	2	7	2	5	100
liberer	2	5	2	3	100
RESERVATION1	12	17	4	13	100

5 Validation de l'implémentation CODE.imp

Etat des obligations de preuve après le passage en automatique en force 1.

	NbObv	NbPO	NbPRi	NbPRa	%Pr
ValuesLemmas	1	0	0	0	100
InstanciatedConstraintsLemmas	2	2	0	2	100
Initialisation	5	17	0	15	88
place_libre	3	3	0	3	100
reserver	10	25	0	22	88
liberer	7	7	0	6	85
CODE	28	54	0	48	88

La liste des obligations de preuve pour ce module d'implémentation est :

```
PRI > State of all PO
InstanciatedConstraintsLemmas
  P01 Proved      1..nb_max: FIN(INTEGER)
  P02 Proved      not(1..nb_max = {})
Initialisation
  P01 Proved      nb_max: INTEGER
  P02 Proved      nb_max<=2147483647
  P03 Proved      -2147483647<=nb_max
  P04 Proved      1: 1..nb_max+1
  P05 Proved      nb_max+1-ind$0: INTEGER
  P06 Proved      0<=nb_max+1-ind$0
  P07 Proved      ind$0: 1..nb_max
  P08 Proved      ind$0+1: INTEGER
  P09 Proved      ind$0+1<=2147483647
  P010 Proved     -2147483647<=ind$0+1
```

```

P011 Proved      ind$0: INTEGER
P012 Proved      ind$0<=2147483647
P013 Proved      -2147483647<=ind$0
P014 Proved      nb_max+1-(ind$0+1)+1<=nb_max+1-ind$0
P015 Proved      ind$0+1: 1..nb_max+1
P016 Unproved   (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
P017 Unproved   arr_vrbz$7777 = (1..nb_max)*{FALSE}
place_libre
  P01 Proved     nb_libre$1: INTEGER
  P02 Proved     nb_libre$1<=2147483647
  P03 Proved     -2147483647<=nb_libre$1
reserver
  P01 Proved     1: 1..nb_max
  P02 Unproved   FALSE: arr_vrb$1[1..nb_max]
  P03 Proved     nb_max-ind: INTEGER
  P04 Proved     0<=nb_max-ind
  P05 Unproved   ind+1<=nb_max
  P06 Proved     ind+1: INTEGER
  P07 Proved     ind+1<=2147483647
  P08 Proved     -2147483647<=ind+1
  P09 Proved     ind: INTEGER
  P010 Proved    ind<=2147483647
  P011 Proved    -2147483647<=ind
  P012 Proved    ind+1: 1..nb_max
  P013 Proved    nb_max-(ind+1)+1<=nb_max-ind
  P014 Unproved  FALSE: arr_vrb$1[ind+1..nb_max]
  P015 Proved    nb_libre$1-1: INTEGER
  P016 Proved    nb_libre$1-1<=2147483647
  P017 Proved    -2147483647<=nb_libre$1-1
  P018 Proved    nb_libre$1: INTEGER
  P019 Proved    nb_libre$1<=2147483647
  P020 Proved    -2147483647<=nb_libre$1
  P021 Proved    ind$7777: INTEGER
  P022 Proved    ind$7777<=2147483647
  P023 Proved    -2147483647<=ind$7777
  P024 Proved    ind$7777: etat~[{FALSE}]
  P025 Proved    arr_vrb$1<+{ind$7777|->TRUE} = etat<+{ind$7777|->TRUE}
liberer
  P01 Proved     nb_libre$1+1: INTEGER
  P02 Proved     nb_libre$1+1<=2147483647
  P03 Unproved   -2147483647<=nb_libre$1+1
  P04 Proved     nb_libre$1: INTEGER
  P05 Proved     nb_libre$1<=2147483647
  P06 Proved     -2147483647<=nb_libre$1
  P07 Proved     arr_vrb$1<+{place|->FALSE} = etat<+{place|->FALSE}
End

```

5.1 Initialisation

Les hypothèses pour l'initialisation du module sont :

"Component constraints"	
$nb_max \in 1..1000 \wedge$	(1) Mêmes contraintes que pour
$btrue \wedge$	(2) la machine et le raffinement
$1 \leq nb_max \wedge$	(3)
$nb_max \leq 1000 \wedge$	(4)
"Included, imported and extended machines invariants"	
$arr_vrb\$1 \in 1..nb_max \mapsto \text{BOOL} \wedge$	(5) Ces hypothèses viennent de
$\text{dom}(arr_vrb\$1) = 1..nb_max$	(6) la machine importée

5.1.1 Preuve de Initialisation.16

"Local hypotheses"	
$arr_vrb\$0 \in 1..nb_max \mapsto \text{BOOL} \wedge$	
$\text{dom}(arr_vrb\$0) = 1..nb_max \wedge$	
$ind\$0 \in 1..nb_max + 1 \wedge$	
$2 \leq ind\$0 \Rightarrow arr_vrb\$2[1..ind\$0 - 1] = \{\text{FALSE}\} \wedge$	
$ind\$0 \leq nb_max \wedge$	
$1 \leq ind\$0$	
"Check preconditions of called operation, or While loop construction, or Assert predicates"	
\Rightarrow	
$(arr_vrb\$2 \Leftarrow \{ind\$0 \mapsto \text{FALSE}\})[1..ind\$0 + 1 - 1] = \{\text{FALSE}\}$	

Cette obligation de preuve vient de la préservation de l'invariant de la boucle :

$$1 < ind \Rightarrow (arr_vrb[1..ind - 1] = \{\text{FALSE}\})$$

Lorsqu'on remplace ind par $ind + 1$ et arr_vrb par $arr_vrb \Leftarrow \{ind \mapsto \text{FALSE}\}$, qui sont les deux substitutions de la boucle, on obtient le but indiqué, au renommage de variables près. De plus, l'antécédent de l'implication $1 < ind + 1$ est devenu $1 \leq ind$ et se retrouve en hypothèse locale.

A ce stade, il n'est pas clair de savoir pourquoi il y a une déclaration de $arr_vrb\$0$ dans les hypothèses (déclaration qui est d'ailleurs différente de celle de la machine importée $arr_vrb\$1$).

Quoiqu'il en soit, l'idée de la preuve vient de l'exploitation de l'hypothèse $2 \leq ind\$0 \Rightarrow arr_vrb\$2[1..ind\$0 - 1] = \{\text{FALSE}\}$ qui donne déjà une valeur $\{\text{FALSE}\}$ pour la partie du tableau $2..ind\$0 - 1$. Comme à l'indice $ind\$0$, on a la valeur explicite de la fonction ($\{ind\$0 \mapsto \text{FALSE}\}$), il suffit de montrer que pour $ind\$0 = 1$, on a également la valeur FALSE . On décide donc de faire une décomposition par cas, suivant les deux possibilités disjointes :

$$ind\$0 = 1 \vee ind\$0 \in 2..nb_max + 1$$

On dispose de la commande "**dcs**" (décomposition par cas spéciale) qui fait justement cette analyse. Si le but est G , cette commande va conduire à démontrer les trois sous-buts :

$$\begin{aligned} ind\$0 = 1 \vee ind\$0 \in 2..nb_max + 1 \\ ind\$0 \in 2..nb_max + 1 \wedge ind\$0 \neq 1 &\Rightarrow G \\ ind\$0 = 1 \wedge ind\$0 \notin 2..nb_max + 1 &\Rightarrow G \end{aligned}$$

Lors de la preuve du deuxième sous-but, il est facile de démontrer que l'on est bien dans le cas " $2 \leq ind\$0$ ". On peut alors appliquer le "modus ponens" en hypothèse (commande "**mh**") qui extrait le conséquent de l'hypothèse conditionnelle qui a fourni l'idée de départ :

$$arr_vrb\$2[1..ind\$0 - 1] = \{FALSE\}$$

On peut calculer le lemme intermédiaire :

$$\{ind\$0 \mapsto FALSE\}[1..ind\$0] = \{FALSE\}$$

On décompose alors le but par la transformation de la modification de fonction en union, et après simplification, on obtient :

$$(\{ind\$0\} \triangleleft arr_vrb\$2)[1..ind\$0] \cup \{ind\$0 \mapsto FALSE\}[1..ind\$0] = \{FALSE\}$$

Par preuve automatique, vu les lemmes déjà calculés, le deuxième sous-but est démontré. En ce qui concerne le troisième sous-but, il suffit de remplacer $ind\$0$ par 1 dans le but, ce qui donne une formule qui se démontre automatiquement.

```
> dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> dcs(ind$0 = 1 or ind$0: 2..nb_max+1)
:      ind$0 = 1 or ind$0: 2..nb_max+1
> ah(ind$0: 1..nb_max+1)
:      ind$0: 1..nb_max+1 => ind$0 = 1 or ind$0: 2..nb_max+1
> pp(rp.0)
:      ind$0: 2..nb_max+1 &
:      not(ind$0 = 1) &
:      =>
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> ah(2<=ind$0)
:      2<=ind$0
> ah(ind$0: 2..nb_max+1)
:      ind$0: 2..nb_max+1 => 2<=ind$0
> pp(rp.0)
:      2<=ind$0 => (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> mh(2<=ind$0 => arr_vrb$2[1..ind$0-1] = {FALSE})
:      arr_vrb$2[1..ind$0-1] = {FALSE} =>
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> ah(arr_vrb$2<+{ind$0|->FALSE} = {ind$0}<<|arr_vrb$2\/{ind$0|->FALSE})
:      arr_vrb$2<+{ind$0|->FALSE} = {ind$0}<<|arr_vrb$2\/{ind$0|->FALSE}
> pp(rp.0)
:      arr_vrb$2<+{ind$0|->FALSE} = {ind$0}<<|arr_vrb$2\/{ind$0|->FALSE} =>
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> eh(arr_vrb$2<+{ind$0|->FALSE},_h,Goal)
:      ({ind$0}<<|arr_vrb$2\/{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
> ss
```

```

:      ({ind$0}<<|arr_vrb$2|/{ind$0|->FALSE})[1..ind$0] = {FALSE}
>      ar(SimplifyRelImaLongXY.5,Goal)
:      ({ind$0}<<|arr_vrb$2)[1..ind$0]|/{ind$0|->FALSE}[1..ind$0] = {FALSE}
>      ah({ind$0|->FALSE}[1..ind$0] = {FALSE})
:      {ind$0|->FALSE}[1..ind$0] = {FALSE}
>      ah(ind$0: 1..ind$0)
:      ind$0: 1..ind$0
>      pr
:      ind$0: 1..ind$0 => {ind$0|->FALSE}[1..ind$0] = {FALSE}
>      pp(rp.0)
:      {ind$0|->FALSE}[1..ind$0] = {FALSE} =>
:      ({ind$0}<<|arr_vrb$2)[1..ind$0]|/{ind$0|->FALSE}[1..ind$0] = {FALSE}
>      pr
:      ind$0 = 1 &
:      not(ind$0: 2..nb_max+1) &
:      =>
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
>      dd
:      (arr_vrb$2<+{ind$0|->FALSE})[1..ind$0+1-1] = {FALSE}
>      eh(ind$0,_h,Goal)
:      (arr_vrb$2<+{1|->FALSE})[1..1+1-1] = {FALSE}
>      pr
:      PROVED

```

En guise d’information, la commande de décomposition normale **dc** est utilisée pour la décomposition par rapport à un prédicat h , avec les deux sous-buts : $h \Rightarrow G$ et $\neg h \Rightarrow G$, ou encore pour décomposer une variable d’un ensemble énuméré sur les différentes valeurs de cet ensemble.

5.1.2 Preuve de Initialisation.17

“Local hypotheses”

$$\begin{aligned}
& arr_vrb\$0 \in 1..nb_max \leftrightarrow \text{BOOL} \wedge \\
& \text{dom}(arr_vrb\$0) = 1..nb_max \wedge \\
& \neg(ind\$7777 \leq nb_max) \wedge \\
& ind\$7777 \in 1..nb_max + 1 \wedge \\
& 2 \leq ind\$7777 \Rightarrow arr_vrbz\$7777[1..ind\$7777 - 1] = \{\text{FALSE}\} \wedge \\
& arr_vrbz\$7777 \in 1..nb_max \leftrightarrow \text{BOOL} \wedge \\
& \text{dom}(arr_vrbz\$7777) = 1..nb_max
\end{aligned}$$

“Check that the invariant ($etat = arr_vrb\$1$) is established by the initialisation - ref 4.3, 5.4” & “Check initialisation refinement - ref 4.3, 5.4”

$$\Rightarrow \\
arr_vrbz\$7777 = (1..nb_max) \times \{\text{FALSE}\}$$

Pour démontrer cette obligation de preuve, on montre d’abord que l’on a :

$$ind\$7777 = nb_max + 1$$

Ce qui conduit par remplacement à avoir l’hypothèse :

$$2 \leq nb_max + 1 \Rightarrow arr_vrbz\$7777[1..nb_max + 1 - 1] = \{\text{FALSE}\}$$

L'antécédent de cette hypothèse est évidemment vrai puisque $1 \leq nb_max$ et par modus ponens le conséquent peut être démontré:

$$arr_vrbz\$7777[1..nb_max + 1 - 1] = \{FALSE\}$$

Finalement, après quelques simplifications, le predicate prover permet de conclure que l'implication est vraie (qui est une équivalence, puisque l'implication dans l'autre sens est aussi vraie):

$$arr_vrbz\$7777[1..nb_max] = \{FALSE\} \Rightarrow arr_vrbz\$7777 = (1..nb_max) \times \{FALSE\}$$

D'où le script de preuve :

```

> dd
:
:       arr_vrbz$7777 = (1..nb_max)*{FALSE}
> ah(ind$7777 = nb_max+1)
:
:       ind$7777 = nb_max+1
> ah(not(ind$7777<=nb_max))
:
:       not(ind$7777<=nb_max) => ind$7777 = nb_max+1
> ah(ind$7777: 1..nb_max+1)
:
:       ind$7777: 1..nb_max+1 => (not(ind$7777<=nb_max) => ind$7777 = nb_max+1)
> pp(rp.0)
:
:       ind$7777 = nb_max+1 => arr_vrbz$7777 = (1..nb_max)*{FALSE}
> dd
:
:       arr_vrbz$7777 = (1..nb_max)*{FALSE}
> eh(ind$7777,_h,Hyp(2<=ind$7777 => arr_vrbz$7777[1..ind$7777-1] = {FALSE}))
:
:       2<=nb_max+1 => arr_vrbz$7777[1..nb_max+1-1] = {FALSE} =>
:       arr_vrbz$7777 = (1..nb_max)*{FALSE}
> dd
:
:       arr_vrbz$7777 = (1..nb_max)*{FALSE}
> ah(2<=nb_max+1)
:
:       2<=nb_max+1
> ah(1<=nb_max)
:
:       1<=nb_max => 2<=nb_max+1
> pp(rp.0)
:
:       2<=nb_max+1 => arr_vrbz$7777 = (1..nb_max)*{FALSE}
> dd
:
:       arr_vrbz$7777 = (1..nb_max)*{FALSE}
> mh(2<=nb_max+1 => arr_vrbz$7777[1..nb_max+1-1] = {FALSE})
:
:       arr_vrbz$7777[1..nb_max+1-1] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE}
> ah(nb_max+1-1 = nb_max)
:
:       nb_max+1-1 = nb_max
> pp(rp.0)
:
:       nb_max+1-1 = nb_max =>
:       (arr_vrbz$7777[1..nb_max+1-1] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE})
> dd
:
:       arr_vrbz$7777[1..nb_max+1-1] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE}
> eh(nb_max+1-1,_h,Goal)
:
:       arr_vrbz$7777[1..nb_max] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE}
> ah(arr_vrbz$7777: 1..nb_max +-> BOOL)
:
:       arr_vrbz$7777: 1..nb_max +-> BOOL =>
:       (arr_vrbz$7777[1..nb_max] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE})
> ah(dom(arr_vrbz$7777) = 1..nb_max)
:
:       dom(arr_vrbz$7777) = 1..nb_max =>
:       (arr_vrbz$7777: 1..nb_max +-> BOOL =>
:       (arr_vrbz$7777[1..nb_max] = {FALSE} => arr_vrbz$7777 = (1..nb_max)*{FALSE}))
> pp(rp.0)
:
:       PROVED

```

5.2 Les opérations

Lorsque l'initialisation est faite (cas d'utilisation des opérations), les hypothèses globales du module `CODE.imp` sont:

"Component constraints"		
$nb_max \in 1..1000 \wedge$	(1)	
$btrue \wedge$	(2)	
$1 \leq nb_max \wedge$	(3)	
$nb_max \leq 1000 \wedge$	(4)	
"Included, imported and extended machines invariants"		
$arr_vrb\$1 \in 1..nb_max \leftrightarrow \mathbf{BOOL} \wedge$	(5)	Invariant de la machine
$dom(arr_vrb\$1) = 1..nb_max \wedge$	(6)	importée
"Previous components invariants"		
$nb_libre\$1 \in 0..nb_max \wedge$	(7)	Invariant du raffinement
$occupes = arr_vrb\$1^{-1}[\{\mathbf{TRUE}\}] \wedge$	(8)	où $etat\$1$ est remplacé
$nb_libre\$1 = nb_max - \mathbf{card}(occupes) \wedge$	(9)	par $arr_vrb\$1$
$occupes \in \mathbb{F}(1..nb_max) \wedge$	(10)	Invariant de la machine
"Component invariant"		
$nb_libre = nb_libre\$1 \wedge$	(11)	Invariant de l'implémentation
$etat = arr_vrb\$1 \wedge$	(12)	
$0 \leq nb_libre\$1 \wedge$	(13)	
$nb_libre\$1 \leq nb_max$	(14)	

5.3 Opération reserver

Les hypothèses de l'opération `reserver` héritées de la machine `RESERVATION` sont:

"reserver preconditions in previous components"	
$\neg (nb_max - \mathbf{card}(occupes) = 0)$	(15)

5.3.1 Preuve de reserver.2

"reserver preconditions in this component"	
$\neg (nb_max - \mathbf{card}(occupes) = 0)$	
"Check preconditions of called operation, or While loop construction, or Assert predicates"	
\Rightarrow	
$\mathbf{FALSE} \in arr_vrb\$1[1..nb_max]$	

Cette obligation de preuve vient de la vérification que l'invariant est vrai à l'entrée de la boucle. L'idée est de démontrer cette formule par contradiction avec l'hypothèse:

$$\neg (nb_max - \mathbf{card}(occupes) = 0)$$

puisqu'on a la dérivation:

$$\begin{aligned}
 \mathbf{FALSE} \notin arr_vrb\$1[1..nb_max] &\Leftrightarrow arr_vrb\$1^{-1}[\{\mathbf{FALSE}\}] = \emptyset \\
 &\Leftrightarrow arr_vrb\$1^{-1}[\{\mathbf{TRUE}\}] = 1..nb_max \\
 &\Rightarrow \mathbf{card}(arr_vrb\$1^{-1}[\{\mathbf{TRUE}\}]) = nb_max \\
 &\Leftrightarrow \mathbf{card}(occupes) = nb_max
 \end{aligned}$$

qui fournit finalement : $nb_max - \text{card}(occupes) = 0$. Pour arriver à ces calculs, on utilise les règles de simplification d'ensembles déjà vues, ainsi que :

$$\begin{array}{l} \text{SimplifyEnsCard.11} \quad 0 \leq m \\ \Rightarrow \\ \text{card}(1..m) == m \end{array}$$

La commande “**ct**” permet de démontrer un but par contradiction. Si le but est G , la commande génère le nouveau but : “**bfalse**” en mettant $\neg G$ en hypothèse. La commande “**cts**” peut aussi être utilisée. Elle construit “ $\neg G \Rightarrow \text{bfalse}$ ” comme nouveau but. Dans la démonstration qui suit, la seule difficulté est de poser suffisamment d’hypothèses pour démontrer chaque pas intermédiaire de l’esquisse de preuve donnée ci-dessus.

```
> dd
:
:      FALSE: arr_vrb$1[1..nb_max]
> ct
:
:      bfalse
> ah(arr_vrb$1~[FALSE] = {})
:
:      arr_vrb$1~[FALSE] = {}
> ah(not(FALSE: arr_vrb$1[1..nb_max]))
:
:      not(FALSE: arr_vrb$1[1..nb_max]) => arr_vrb$1~[FALSE] = {}
> ah(dom(arr_vrb$1) = 1..nb_max)
:
:      dom(arr_vrb$1) = 1..nb_max =>
:      (not(FALSE: arr_vrb$1[1..nb_max]) => arr_vrb$1~[FALSE] = {})
> ah(arr_vrb$1: 1..nb_max +-> BOOL)
:
:      arr_vrb$1: 1..nb_max +-> BOOL =>
:      (dom(arr_vrb$1) = 1..nb_max =>
:      (not(FALSE: arr_vrb$1[1..nb_max]) => arr_vrb$1~[FALSE] = {}))
> pp(rp.0)
:
:      arr_vrb$1~[FALSE] = {} => bfalse
> dd
:
:      bfalse
> ah(arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = arr_vrb$1~[FALSE,TRUE])
:
:      arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = arr_vrb$1~[FALSE,TRUE]
> pp(rp.0)
:
:      arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = arr_vrb$1~[FALSE,TRUE] => bfalse
> dd
:
:      bfalse
> ah(arr_vrb$1~[FALSE,TRUE] = 1..nb_max)
:
:      arr_vrb$1~[FALSE,TRUE] = 1..nb_max
> ah(dom(arr_vrb$1) = 1..nb_max)
:
:      dom(arr_vrb$1) = 1..nb_max => arr_vrb$1~[FALSE,TRUE] = 1..nb_max
> ah(arr_vrb$1: 1..nb_max +-> BOOL)
:
:      arr_vrb$1: 1..nb_max +-> BOOL =>
:      (dom(arr_vrb$1) = 1..nb_max => arr_vrb$1~[FALSE,TRUE] = 1..nb_max)
> pp(rp.0)
:
:      arr_vrb$1~[FALSE,TRUE] = 1..nb_max => bfalse
> dd
:
:      bfalse
> eh(arr_vrb$1~[FALSE,TRUE],_h,
:      Hyp(arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = arr_vrb$1~[FALSE,TRUE]))
:
:      arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = 1..nb_max => bfalse
> dd
:
:      bfalse
> eh(arr_vrb$1~[FALSE],{ },Hyp(arr_vrb$1~[FALSE]\arr_vrb$1~[TRUE] = 1..nb_max))
:
:      { }\arr_vrb$1~[TRUE] = 1..nb_max => bfalse
```

```

> ar(SimplifySetUniXY.35,Goal)
: arr_vrb$1~[TRUE] = 1..nb_max => bfalse
> dd
: bfalse
> ah(card(arr_vrb$1~[TRUE]) = nb_max)
: card(arr_vrb$1~[TRUE]) = nb_max
> eh(arr_vrb$1~[TRUE],_h,Goal)
: card(1..nb_max) = nb_max
> ah(0<=nb_max)
: 0<=nb_max
> ah(1<=nb_max)
: 1<=nb_max => 0<=nb_max
> pp(rp.0)
: 0<=nb_max => card(1..nb_max) = nb_max
> dd
: card(1..nb_max) = nb_max
> ar(SimplifyEnsCard.11,Goal)
: nb_max = nb_max
> pp(rp.0)
: card(arr_vrb$1~[TRUE]) = nb_max => bfalse
> eh(arr_vrb$1~[TRUE],occupes,Goal)
: card(occupes) = nb_max => bfalse
> dd
: bfalse
> ah(nb_max-card(occupes) = 0)
: nb_max-card(occupes) = 0
> eh(card(occupes),_h,Goal)
: nb_max-nb_max = 0
> pp(rp.0)
: nb_max-card(occupes) = 0 => bfalse
> ah(not(nb_max-card(occupes) = 0))
: not(nb_max-card(occupes) = 0) => (nb_max-card(occupes) = 0 => bfalse)
> pp(rp.0)
: PROVED

```

Remarque sur cette preuve : la technique de l'*AtelierB* fait que le but courant que l'on affiche ici ne donne pas la meilleure information sur le déroulement de la preuve. Il faut tenir compte des lemmes construits et éventuellement mis en hypothèse par la commande **dd**. La suite de ces lemmes significatifs est :

```

arr_vrb$1~[FALSE] = {}
arr_vrb$1~[FALSE] \ arr_vrb$1~[TRUE] = arr_vrb$1~[FALSE,TRUE]
arr_vrb$1~[FALSE,TRUE] = 1..nb_max
arr_vrb$1~[FALSE] \ arr_vrb$1~[TRUE] = 1..nb_max
arr_vrb$1~[TRUE] = 1..nb_max
card(arr_vrb$1~[TRUE]) = nb_max
card(occupes) = nb_max

```

Le traitement de l'égalité est laborieux ainsi que celui de l'utilisation des propriétés des types énumérés (booléens ici). Le passage du lemme $arr_vrb\$1^{-1}[FALSE] = \emptyset$ à son complémentaire $arr_vrb\$1^{-1}[TRUE] = 1..nb_max$ nécessite une douzaine de pas.

5.3.2 Preuve de `reserver.5`

```

“reserver preconditions in this component”
   $\neg (nb\_max - \text{card}(\text{occupes}) = 0) \wedge$ 
“Local hypotheses”
   $ind \in 1..nb\_max \wedge$ 
   $bb = \text{arr\_vrb}\$1(ind) \wedge$ 
   $\text{FALSE} \in \text{arr\_vrb}\$1[ind..nb\_max] \wedge$ 
   $bb = \text{TRUE}$ 
“Check preconditions of called operation, or While loop construction, or Assert predicates”
 $\Rightarrow$ 
   $ind + 1 \leq nb\_max$ 

```

Il s’agit de l’obligation de preuve engendrée par la substitution d’assertion à l’intérieur du corps de la boucle. Cette preuve est faite une fois pour toutes et les autres obligations de preuve de la boucle bénéficient de l’hypothèse $ind + 1 \leq nb_max$ (voir par exemple l’OP `reserver.14`).

En ce qui concerne la démonstration ci-dessous, il est clair qu’il faut relier cette assertion avec le fait qu’il existe encore des éléments du tableau à `FALSE`, par l’hypothèse locale :

$$\text{FALSE} \in \text{arr_vrb}\$1[ind..nb_max]$$

Si l’on montre qu’il est impossible d’avoir $ind = nb_max$, alors forcément, à cause de $ind \in 1..nb_max$, on a $ind < nb_max$ et le but est démontré. La pointe de la contradiction vient du fait que si $ind = nb_max$ alors :

$$\begin{aligned} \text{FALSE} \in \text{arr_vrb}\$1[ind..nb_max] &\Leftrightarrow \text{FALSE} \in \text{arr_vrb}\$1[nb_max..nb_max] \\ &\Leftrightarrow \text{FALSE} \in \text{arr_vrb}\$1\{nb_max\} \\ &\Leftrightarrow \text{FALSE} = \text{arr_vrb}\$1(nb_max) \end{aligned}$$

avec d’autre part, $bb = \text{arr_vrb}\$1(ind) \wedge bb = \text{TRUE}$, ce qui donne :

$$\text{TRUE} = \text{arr_vrb}\$1(nb_max)$$

Tout ce raisonnement est codé dans la démonstration ci-dessous :

```

> dd
:      ind+1<=nb_max
> ah(ind: 1..nb_max-1 => ind+1<=nb_max)
:      ind: 1..nb_max-1 => ind+1<=nb_max
> pp(rp.0)
:      ind: 1..nb_max-1 => ind+1<=nb_max => ind+1<=nb_max
> dd
:      ind+1<=nb_max
> ah(not(ind = nb_max))
:      not(ind = nb_max)
> ct
:      bfalse
> ah(FALSE: arr_vrb$1[ind..nb_max])
:      FALSE: arr_vrb$1[ind..nb_max] => bfalse
> ah(bb = arr_vrb$1(ind))
:      bb = arr_vrb$1(ind) => (FALSE: arr_vrb$1[ind..nb_max] => bfalse)
> eh(ind,_h,Goal)
:      bb = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[nb_max..nb_max] => bfalse)

```

```

>         eh(bb,_h,Goal)
:         TRUE = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[nb_max..nb_max] => bfalse)
>         ah(nb_max..nb_max = {nb_max})
:         nb_max..nb_max = {nb_max}
>         pp(rp.0)
:         nb_max..nb_max = {nb_max} =>
:         (TRUE = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[nb_max..nb_max] => bfalse))
>         dd
:         TRUE = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[nb_max..nb_max] => bfalse)
>         eh(nb_max..nb_max,_h,Goal)
:         TRUE = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[{nb_max}] => bfalse)
>         ah(arr_vrb$1: 1..nb_max +-> BOOL)
:         arr_vrb$1: 1..nb_max +-> BOOL =>
:         (TRUE = arr_vrb$1(nb_max) => (FALSE: arr_vrb$1[{nb_max}] => bfalse))
>         pp(rp.0)
:         not(ind = nb_max) => ind+1<=nb_max
>     dd
:         ind+1<=nb_max
>     ah(ind: 1..nb_max-1)
:         ind: 1..nb_max-1
>     ah(not(ind = nb_max))
:         not(ind = nb_max) => ind: 1..nb_max-1
>     ah(ind: 1..nb_max)
:         ind: 1..nb_max => (not(ind = nb_max) => ind: 1..nb_max-1)
>     pp(rp.0)
:         ind: 1..nb_max-1 => ind+1<=nb_max
>     pr
:         PROVED

```

5.3.3 Preuve de reserver.14

“reserver preconditions in this component”
 $\neg (nb_max - \text{card}(\text{occupes}) = 0) \wedge$
“Local hypotheses”
 $ind \in 1..nb_max \wedge$
 $bb = \text{arr_vrb}\$1(ind) \wedge$
 $\text{FALSE} \in \text{arr_vrb}\$1[ind..nb_max] \wedge$
 $bb = \text{TRUE} \wedge$
 $ind + 1 \leq nb_max$
“Check preconditions of called operation, or While loop construction, or Assert predicates”
 \Rightarrow
 $\text{FALSE} \in \text{arr_vrb}\$1[ind + 1..nb_max]$

Cette obligation de preuve est une propriété de préservation de l’invariant de boucle. L’idée de la démonstration est de poser :

$$\{ind\} \cup (ind + 1)..nb_max = ind..nb_max$$

Sachant par les hypothèses que $\text{FALSE} \in \text{arr_vrb}\$1[ind..nb_max]$ et que, d’autre part, $bb = \text{TRUE} \wedge bb = \text{arr_vrb}\$1(ind)$, c’est-à-dire $\text{TRUE} = \text{arr_vrb}\$1(ind)$, on en déduit qu’on a bien $\text{FALSE} \in \text{arr_vrb}\$1[(ind + 1)..nb_max]$. On utilisera les règles suivantes de la base de règles (pour `SimplifyRelImaXY.13`, on a supprimé les conditions de non capture de variables; de

plus, il faut que les antécédents soient dans les hypothèses pour réaliser la réécriture du but) :

$$\begin{array}{ll}
\text{SimplifyRelImaLongXY.6} & r[u \cup v] == r[u] \cup r[v] \\
\text{SimplifyRelImaXY.13} & f(a) = b \wedge a \in \text{dom}(f) \\
& \Rightarrow \\
& f[\{a\}] == \{b\}
\end{array}$$

```

> dd
:      FALSE: arr_vrb$1[ind+1..nb_max]
> ah(ind..nb_max = {ind}\ind+1..nb_max)
:      ind..nb_max = {ind}\ind+1..nb_max
> ah(ind+1<=nb_max)
:      ind+1<=nb_max => ind..nb_max = {ind}\ind+1..nb_max
> ah(ind: 1..nb_max)
:      ind: 1..nb_max => (ind+1<=nb_max => ind..nb_max = {ind}\ind+1..nb_max)
> pp(rp.0)
:      ind..nb_max = {ind}\ind+1..nb_max => FALSE: arr_vrb$1[ind+1..nb_max]
> dd
:      FALSE: arr_vrb$1[ind+1..nb_max]
> ah(FALSE: arr_vrb$1[ind..nb_max])
:      FALSE: arr_vrb$1[ind..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> eh(ind..nb_max,_h,Goal)
:      FALSE: arr_vrb$1[{ind}\ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> ar(SimplifyRelImaLongXY.6,Goal)
:      FALSE: arr_vrb$1[{ind}]\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> ah(ind: dom(arr_vrb$1))
:      ind: dom(arr_vrb$1)
> pr
:      ind: dom(arr_vrb$1) =>
      (FALSE: arr_vrb$1[{ind}]\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max])
> dd
:      FALSE: arr_vrb$1[{ind}]\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> ah(arr_vrb$1(ind) = TRUE)
:      arr_vrb$1(ind) = TRUE
> pr
:      arr_vrb$1(ind) = TRUE =>
      (FALSE: arr_vrb$1[{ind}]\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max])
> dd
:      FALSE: arr_vrb$1[{ind}]\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> ar(SimplifyRelImaXY.13,Goal)
:      FALSE: {TRUE}\arr_vrb$1[ind+1..nb_max] => FALSE: arr_vrb$1[ind+1..nb_max]
> ss
:      FALSE: arr_vrb$1[1+ind..nb_max] => FALSE: arr_vrb$1[1+ind..nb_max]
> pr
:      PROVED

```

5.4 Opération liberer

Les hypothèses de l'opération `liberer` héritées de la machine `RESERVATION` sont :

“liberer preconditions in previous components”	
$place \in 1..nb_max \wedge$	(16)
$place \in occupes \wedge$	(17)
$1 \leq place \wedge$	(18)
$place \leq nb_max$	(19)

5.4.1 Preuve de liberer.3

“liberer preconditions in this component”
 $place \in 1..nb_max \wedge$
 $place \in occupes$
 “Check preconditions of called operation, or While loop construction, or Assert predicates”
 \Rightarrow
 $-2147483647 \leq nb_libre\$1 + 1$

Il est assez étonnant que cette obligation de preuve ne soit pas démontrée automatiquement. Elle provient évidemment de la substitution $nb_libre\$1 := nb_libre\$1 + 1$. Pour cette démonstration, on va s’appuyer sur l’hypothèse: $0 \leq nb_libre\$1$ et sur les propriétés $nb_libre\$1 \leq nb_libre\$1 + 1$ et $-2147483647 \leq 0$. Par transitivité de l’ordre, on a le résultat.

```
> dd
:           -2147483647<=nb_libre$1+1
> ah(0<=nb_libre$1)
:           0<=nb_libre$1 => -2147483647<=nb_libre$1+1
> ah(nb_libre$1<=nb_libre$1+1)
:           nb_libre$1<=nb_libre$1+1
> pr
:           nb_libre$1<=nb_libre$1+1 => (0<=nb_libre$1 => -2147483647<=nb_libre$1+1)
> ah(-2147483647<=0)
:           -2147483647<=0
> pr
:           -2147483647<=0 =>
:           (nb_libre$1<=nb_libre$1+1 => (0<=nb_libre$1 => -2147483647<=nb_libre$1+1))
> pr
:           PROVED
```

5.5 Etat final du composant CODE.imp

	NbObv	NbPO	NbPRi	NbPRa	%Pr
ValuesLemmas	1	0	0	0	100
InstanciatedConstraintsLemmas	2	2	0	2	100
Initialisation	5	17	2	15	100
place_libre	3	3	0	3	100
reserver	10	25	3	22	100
liberer	7	7	1	6	100
CODE	28	54	6	48	100

6 Validation de l'implémentation USER1_i

L'implémentation du module USER1 produit une obligation de preuve non démontrée automatiquement.

<p>“Included, imported and extended machines invariants”</p> $\text{occupes}\$1 \in \mathbb{F}(1..2) \wedge$ $\text{btrue} \wedge$ <p>“Local hypotheses”</p> $\neg(2 - \text{card}(\text{occupes}\$1) = 0) \wedge$ $pp \in 1..2 \wedge$ $\neg(pp \in \text{occupes}\$1) \wedge$ $\neg(2 - \text{card}(\text{occupes}\$1 \cup \{pp\}) = 0) \wedge$ $pp\$0 \in 1..2 \wedge$ $\neg(pp\$0 \in \text{occupes}\$1) \wedge$ $\neg(pp\$0 = pp) \wedge$ $2 - \text{card}(\text{occupes}\$1 \cup \{pp\} \cup \{pp\$0\}) = 0$ <p>“Check preconditions of called operation, or While loop construction, or Assert predicates”</p> \Rightarrow $\neg(2 - \text{card}((\text{occupes}\$1 \cup \{pp\} \cup \{pp\$0\}) - \{pp\}) = 0)$
--

Il s'agit de la précondition du dernier appel à l'opération “*reserver*” lorsqu'on est passé par le cas où on a libéré la place “*num1*”. Dans les hypothèses, on trouve le fait que la machine est instanciée dans l'importation ($nb_max = 2$) et que les tests de la branche dans laquelle on est sont vrais. De plus, les conditions des choix d'éléments faits à l'intérieur de la réservation sont conservées.

L'idée de la démonstration est d'essayer de simplifier l'expression donnant le cardinal en fonction de l'union et de la différence d'ensembles. Ensuite, grâce à l'hypothèse :

$$2 - \text{card}(\text{occupes}\$1 \cup \{pp\} \cup \{pp\$0\}) = 0$$

on peut déduire que $\text{card}(\text{occupes}\$1) = 0$, ce qui permet de démontrer le but. Au passage, on utilise la règle ci-dessous qui génère deux sous-buts :

$$\text{SimplifyX.53} \quad \neg(a \in b \cup c) == \neg(a \in b) \wedge \neg(a \in c)$$

Les autres règles de simplification du cardinal d'un ensemble ont été données dans les paragraphes précédents. Le script de la preuve est :

```
> dd
:           not(2-card((occupes$1/{pp}\/{pp$0})-{pp}) = 0)
> ah(not(pp$0: occupes$1/{pp}))
:           not(pp$0: occupes$1/{pp})
> ar(SimplifyX.53,Goal)
:           not(pp$0: occupes$1)
> pr
:           not(pp$0: {pp})
> pr
:           not(pp$0: occupes$1/{pp}) => not(2-card((occupes$1/{pp}\/{pp$0})-{pp}) = 0)
> dd
:           not(2-card((occupes$1/{pp}\/{pp$0})-{pp}) = 0)
> ar(SimplifyTryEnsCard.3,Goal)
```

```

:      not(2-(card(occupes$1\/{pp}\/{pp$0})-card(occupes$1\/{pp}\/{pp$0}\/{pp})) = 0)
>
:      ss
:      not(card(occupes$1\/{pp}\/{pp$0}\/{pp}) = card(occupes$1))
>      ah(occupes$1\/{pp}\/{pp$0}\/{pp} = {pp})
:      occupes$1\/{pp}\/{pp$0}\/{pp} = {pp}
>      ss
:      btrue
>      pr
:      occupes$1\/{pp}\/{pp$0}\/{pp} = {pp} =>
:      not(card(occupes$1\/{pp}\/{pp$0}\/{pp}) = card(occupes$1))
>      dd
:      not(card(occupes$1\/{pp}\/{pp$0}\/{pp}) = card(occupes$1))
>      eh(occupes$1\/{pp}\/{pp$0}\/{pp},_h,Goal)
:      not(card({pp}) = card(occupes$1))
>      ar(SimplifyEnsCard.16,Goal)
:      not(1 = card(occupes$1))
>      ah(card(occupes$1) = 0)
:      card(occupes$1) = 0
>      ah(2-card(occupes$1\/{pp}\/{pp$0}) = 0)
:      2-card(occupes$1\/{pp}\/{pp$0}) = 0 => card(occupes$1) = 0
>      ar(SimplifyEnsCard.9,Goal)
:      2-(card(occupes$1)+1+1) = 0 => card(occupes$1) = 0
>      pp(rp.0)
:      card(occupes$1) = 0 => not(1 = card(occupes$1))
>      pp(rp.0)
:      PROVED

```

7 Conclusion et commentaires

Le premier enseignement vient de l'exemple qui a été traité. Il s'agit d'un exemple jouet, mais qui semble assez représentatif de ce que l'on rencontre au niveau de la génération des obligations de preuves. On peut reprendre les chiffres donnés au sujet du nombre d'obligations de preuves. Le nombre d'OP démontrées automatiquement par rapport au nombre total varie entre 100% et 76%, avec plutôt une moyenne proche de 90%. Ce pourcentage ne semble pas dépendre du nombre total d'OP engendrées. En ce qui concerne le rapport du nombre d'OP par rapport au nombre de lignes sources, il dépend par contre fortement de la nature du module en question. Cela va de 0 à 0.1 pour les machines à 1 (et sûrement davantage dans d'autres types de programmes avec des boucles plus complexes) pour les implémentations, la moyenne se situant à 0.5 pour notre exemple complet. L'ensemble de ces chiffres est donné dans le tableau ci-dessous¹ :

Module	lignesB	NbOP	NbPri	NbPra	%Pr	OP/Ligne
RESERVATION	34	3	0	3	100	0.1
RESERVATION1	33	17	4	13	76	0.5
CODE	52	54	6	48	88	1
USER1	8	0	0	0	100	0
USER1_i	38	13	1	12	92	0.3
Total	165	87	11	76	87	0.5

1. En ce qui concerne les lignes des programmes source, on a utilisé la présentation du paragraphe 2, aux points-virgules près.

En ce qui concerne les preuves elle-mêmes, les remarques initiales sur l’expertise (paragraphe d’introduction) du pilote du démonstrateur sont certainement primordiales. Lorsqu’une preuve a été trouvée, il est fréquent qu’on s’aperçoive qu’il en existe une autre plus courte, par une meilleure utilisation des lemmes et des commandes. Il est aussi possible que chaque fin de branche de preuve puisse être écourtée par un appel au démonstrateur automatique. Néanmoins, sur cet exemple, on a les chiffres suivants du nombre de pas nécessaires aux preuves :

OP	NbPas
reserver.6	4
reserver.7	28
liberer.1	12
liberer.5	20
initialisation.16	25
initialisation.17	20
reserver.2	36
reserver.5	22
reserver.14	18
liberer.3	7
main.10	19
Total	211

On obtient une moyenne d’une vingtaine de pas pour chaque OP. Il est amusant de voir qu’il y a plus de lignes de scripts de preuves (et quelles lignes !) que de lignes de programmes. Cela confirme le fait que la partie validation et démonstration interactive reste une tâche lourde dans le processus de construction de logiciels corrects.

Cela amène au deuxième enseignement de ce travail. Il s’agit de l’utilisation du démonstrateur interactif de l’*AtelierB*. Du point de vue des commandes utilisées pour construire la preuve, on s’aperçoit qu’un noyau de commandes représente la majorité des pas de preuve : **ah**, **dd**, **eh**, **pp**, **pr** et dans une moindre mesure **ss** et **mh**. Ensuite, on peut distinguer la commande très importante **ar** qui suppose que l’on a identifié explicitement les règles d’inférence de la base de règles susceptibles de faire progresser la preuve. Il y a enfin les commandes spécifiques au type de preuve que l’on veut réaliser (par cas, par contradiction, etc.) : **dcs**, **ct**. A ce dernier type de commande, on pourrait ajouter **dc**, **cts**, **fh**, **ph** et **se** (hypothèse contradictoire, particularisation d’hypothèse, suggestion d’un “il existe”, etc.) [Ber00], mais ces dernières n’ont pas été utilisées dans cette étude de cas, car la forme des buts et des hypothèses ne s’y prêtait pas. Parmi les commandes de construction de preuve de l’*AtelierB* dont je ne vois pas bien l’utilité, il y a **ch** (création d’hypothèses), **tp** (preuve par tentative), **mp** (miniprouveur) et assez curieusement **ap** ; appel du prouveur arithmétique. Dans le dernier cas, le prouveur **pp** semble plus efficace dans tous les cas. Il n’a pas été nécessaire, dans ce petit exemple, d’introduire des règles supplémentaires “utilisateur”. Le tableau des règles

utilisées par catégories est :

OP	NbPas	NbNoyau	Nb(ar)	NbAutres
reserver.6	4	4	0	0
reserver.7	28	22	6	0
liberer.1	12	7	5	0
liberer.5	20	13	7	0
initialisation.16	25	23	1	dcs
initialisation.17	20	20	0	0
reserver.2	36	33	2	ct
reserver.5	22	20	1	ct
reserver.14	18	16	2	0
liberer.3	7	7	0	0
main.10	19	15	4	0
Total	211	180	28	3

Parmi les avantages de ce démonstrateur interactif, il y a la grande vitesse d'exécution des commandes. La réponse est toujours quasi immédiate. Seul le "predicate prover" peut être un peu lent, essentiellement lorsqu'il n'arrive pas à démontrer le but. Il épuise alors son temps de recherche (60s. par défaut).

Parmi ses faiblesses, outre le fait que l'interface se fait uniquement par commandes sous forme "texte", il y a le manque de définition de macros. Deux exemples illustrent ce manque, avec deux niveaux possibles pour ces macros.

Le premier niveau est celui de la construction d'un texte de script qui peut être utilisé en remplacement de plusieurs lignes de commandes. Par exemple, dans la RESERVATION1, on a la transformation systématique d'un but de la forme :

$$(\{a\} \Leftarrow f \cup \{a \mapsto b\})^{-1}[\{c\}]$$

à la forme :

$$f^{-1}[\{c\}] - \{a\} \cup \{a \mapsto b\}[\{c\}]$$

par une simple application de la suite de règles :

```
ar(SimplifyRelInvXY.13,Goal) &
ar(SimplifyRelInvXY.2,Goal) &
ar(SimplifyRelImaLongXY.5,Goal) &
ar(SimplifyRelImaXY.29,Goal)
```

Il aurait été utile de pouvoir donner un nom à cette suite, pour la réutiliser plusieurs fois. Il est évident qu'un tel type de macros est limité. Un autre niveau de macros correspond à des suites de commandes qui doivent elles-mêmes être paramétrées. Si l'on considère l'autre motif qui est utilisé très souvent, on voit qu'il s'agit du passage de la forme :

$$f \Leftarrow \{a \mapsto b\}$$

à la forme :

$$\{a\} \Leftarrow f \cup \{a \mapsto b\}$$

Malheureusement, la suite de commandes de cette transformation dépend du but, puisqu'on est obligé d'utiliser des formules dans les commandes de preuve :

```
ah( f <- {a -> b} = {a} <- f ∪ {a -> b} ) &
pp(rp.0) &
dd &
eh( f <- {a -> b}, _h, Goal)
```

Pour rendre un tel schéma réutilisable, il faudrait une règle de simplification “avec jokers” dont on puisse dire qu'elle est valide par équivalence à un schéma donné. Par exemple, ici on pourrait déclarer la simplification sous forme d'une règle avec paramètres jokers (où les antécédents pourraient donner des conditions particulières sur les captures de variables ou sur des hypothèses auxiliaires). De plus, les paramètres étant instanciés par filtrage, il ne serait pas nécessaire de les fournir lors de l'appel. Il suffirait d'appeler comme d'habitude “ar(SimplifyOverriding,Goal)” :

```
SimplifyOverriding(f,a,b)          (Backward)
  f <+ {a|->b} == {a}<<|f \ / {a|->b}
```

La validation de cette règle utilisateur se ferait à partir de sa définition, qui n'est qu'une composition de règles valides du démonstrateur :

```
SimplifyOverriding(f,a,b) == ah(f <+ {a|->b} = {a}<<|f \ / {a|->b}) &
                             pp(rp.0) &
                             dd &
                             eh(f <+ {a|->b}, _h, Goal)
```

Il est facile d'imaginer plusieurs formes de mise en œuvre d'une telle macro. Enfin, un point délicat qui n'apparaît pas dans ce rapport est la recherche de règles dans la base. Il y aurait beaucoup à dire sur le sujet. Remarquons simplement que la commande de recherche avec filtres “sr” est pénible à utiliser et que le résultat est souvent difficilement exploitable (trop de règles ou pas de règles trouvées).

Références

- [Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.
- [B98] Atelier B. Prouveur Interactif, Manuel de référence, version 3.5. Technical report, Stéria, 1998.
- [Ber98] Didier Bert. Petits trucs pour s'en sortir dans l'*AtelierB*. Technical report, LSR-IMAG, décembre 1998.
- [Ber00] Didier Bert. Memento du démonstrateur interactif B, version 3.5. Technical report, LSR-IMAG, février 2000.

Table des matières

1	Introduction	1
2	Machines et raffinements de la réservation	2
3	Validation de la machine RESERVATION.mch	6
4	Validation du raffinement RESERVATION1.ref	7
4.1	Opération reserver	8
4.1.1	Preuve de reserver.6	8
4.1.2	Preuve de reserver.7	9
4.2	Opération liberer	11
4.2.1	Preuve de liberer.1	11
4.2.2	Preuve de liberer.5	12
4.3	Etat final du composant RESERVATION1.ref	14
5	Validation de l'implémentation CODE.imp	14
5.1	Initialisation	16
5.1.1	Preuve de Initialisation.16	16
5.1.2	Preuve de Initialisation.17	18
5.2	Les opérations	20
5.3	Opération reserver	20
5.3.1	Preuve de reserver.2	20
5.3.2	Preuve de reserver.5	23
5.3.3	Preuve de reserver.14	24
5.4	Opération liberer	25
5.4.1	Preuve de liberer.3	26
5.5	Etat final du composant CODE.imp	26
6	Validation de l'implémentation USER1_i	27
7	Conclusion et commentaires	28