Initiation rapide à la méthode B

Marie-Laure Potet
Institut IMAG - Laboratoire Logiciels Systèmes Réseaux
681 rue de la Passerelle, BP 72
38402 Saint Martin d'Hères Cedex,
e-mail : Marie-Laure.Potet@imag.fr

1 B : Introduction

La méthode B est une méthode qui permet le développement de programmes corrects par construction. Elle permet en particulier :

– l'écriture de spécifications à état et la preuve de propriétés invariantes sur ces spécifications.

– le raffinement de spécification. Le raffinement permet d'écrire des spécifications plus précises et de montrer leur correction par rapport à la spécification abstraite.

– la production de code. Le dernier niveau de raffinement, appelé *implémentation*, permet de construire des programmes, qui peuvent être traduits dans différents langages.

– la construction de spécifications et de développements par assemblage.

2 Spécification

Les spécifications sont décrites en B à l'aide de *machines abstraites*. Une machine abstraite ressemble à un composant d'un langage de programmation classique (ou à un objet). Les constituants d'une machine sont :

– un état interne ;

– une initialisation ;

– des opérations ;

– des propriétés invariantes.

La méthode B est basée sur une théorie naïve des ensembles que nous ne détaillerons pas dans ce manuscrit [1]. Les ensembles permettent de modéliser les données du modèle (constantes et variables). L'initialisation et les opérations sont décrites à l'aide du langage des *substitutions généralisées*, qui peut être vu comme un langage de programmation abstrait. Ce langage sera décrit dans la suite. Enfin les propriétés sont exprimées à l'aide de la logique du premier ordre, étendue à la théorie ensembliste considérée (non détaillée ici). Le tableau ci-dessous récapitule ces différentes notions :

Modèle	Langage
données	ensembles
initialisation/opérations	substitutions généralisées
propriétés	prédicats du premier ordre

A partir d'une machine abstraite des *Obligations de preuve* sont produites. Elles permettent de vérifier si les propriétés invariantes sont validées par la dynamique, c'est à dire l'initialisation et les opérations. Ceci sera détaillé dans la suite.

2.1 Un exemple

Avant de rentrer dans les aspects plus théoriques, nous donnons un exemple de machine abstraite, simple à comprendre [1]. Cet exemple est volontairement proche d'un programme, il permettra d'illustrer un développement complet, jusqu'au code.

Le but est de décrire un service de réservation de places. Les places sont numérotées de 1 à nb_max, cette dernière valeur étant un paramètre de la spécification. Les opérations offertes permettent de tester s'il reste des places (opération `place_libre`), de réserver une place (opération `réserver`) et de libérer une place déjà réservée (opération `libérer`).

2.1.1 Les données

Le paramètre nb_max est le nombre maximum de places. Ce paramètre vérifie certaines contraintes (clause CONSTRAINTS) : il est supérieur ou égal à 1 et inférieur

1. Cet exemple est inspiré du B-Book.

ou égal à *MAXINT*. Cett dernière contrainte permettra d'implanter simplement la spécification dans un langage n'offrant que des entiers machine. D'autre part, pour des questions de lisibilité on définit l'ensemble *SIEGES* comme l'intervalle $1..nb\_max$ (clause DEFINITIONS). La clause CONSTRAINTS est :

$$nb\_max \in 1 \ .. \ MAXINT$$

L'état du système est modélisé par un ensemble, $occupes$, qui contient les places déjà allouées. On ajoute au modèle une variable, nb_libre, qui permet d'accèder directement au nombre de places libres (cette variable n'apporte pas d'information supplémentaire au modèle, elle est introduite pour illustrer la notion d'invariant). Les propriétés invariantes considérées sont les suivantes :

$$occupes \subseteq SIEGES \quad (1)$$
$$\wedge \ nb\_libre \in \mathit{0} \ .. \ nb\_max \quad (2)$$
$$\wedge \ nb\_libre = nb\_max - card(occupes) \quad (3)$$

La propriété $nb\_libre \in 0 \ .. \ nb\_max$ est une conséquence des propriétés 1 et 3, puisque nb_max est positif (hypothèse sur le paramètre de la machine `RESERVATION`). Elle permet de typer la variable nb_libre.

2.1.2 La dynamique

Initialement l'ensemble des sièges occupés est vide et le nombre de sièges libres est nb_max. Les opérations offertes ont le comportement suivant :

– l'opération `place_libre` renvoie la valeur de la variable nb_libre.

– l'opération `réserver` a une précondition (notation `PRE`) : elle ne peut être appelée que si il reste des sièges libres. Dans ce cas elle attribue une place parmi celles disponibles. Cette opération est non-déterministe : la politique d'allocation des sièges n'est pas précisée. Le non-déterministe est décrit par la construction ANY z WHERE P THEN S END, qui permet de "paramètrer" la substitution S pour n'importe quelle valeur z vérifiant le prédicat P. L'existence d'une place libre est garantie par la précondition.

– `libérer` a une précondition qui vérifie que la place à libérer est effectivement occupée.

La figure ci-dessous décrit la machine abstraite du service de réservation. L'opérateur || permet ici de décrire des affectations en parallèle, portant sur des variables disjointes.

```
MACHINE
  RESERVATION (nb_max)

CONSTRAINTS
  nb_max ∈ 1 .. MAXINT

DEFINITIONS
  SIEGES == (1 .. nb_max)

VARIABLES
  occupes, nb_libre

INVARIANT
  occupes ⊆ SIEGES ∧ nb_libre ∈ 0 .. nb_max ∧ nb_libre = nb_max-card(occupes)

INITIALISATION
  occupes := ∅ || nb_libre :=  nb_max

OPERATIONS
  nb ← place_libre =
     BEGIN
       nb := nb_libre
     END ;

  pp ← reserver  =
     PRE card(occupes) ≠ nb_max THEN
       ANY place WHERE place ∈ SIEGES - occupes   THEN
        pp:=place || occupes:=occupes ∪ {place} || nb_libre:=nb_libre-1
       END
     END ;

   liberer (place) =
     PRE  place ∈ occupes
     THEN occupes := occupes - {place} || nb_libre:=nb_libre+1
     END

END
```

Nous allons montrer que les propriétés invariantes énoncées sont effectivement vérifiées. Pour cela nous introduisons d'abord la notion de *substitution généralisée*.

2.2 Substitutions généralisées

Le langage des substitutions généralisées est issu de la théorie des transformateurs de prédicats. On distingue un ensemble de substitutions primitives, à partir duquel d'autres constructions sont définies. Nous les présentons ci-dessous. Le B-Book introduit ces substitutions en plusieurs fois, suivant celles autorisées dans les spécifications et celles autorisées dans des implantations. Néanmoins cette classification est purement méthodologique. Nous préférons les introduire en une fois.

Généralement, dans les langages de spécifications basés sur la notion d'états (Z, VDM, TLA ...), la dynamique est décrite à l'aide de prédicat avant/après, et, dans certain cas, par des préconditions (VDM par exemple). Un lien peut être établi avec les substitutions généralisées. Ce lien ne sera pas détaillé ici (voir [1]).

2.2.1 Substitutions primitives

Les substitutions primitives sont définies dans une notation mathématique. Nous donnerons leur notation syntaxique dans la suite.

Definition 1 *Substitutions primitives.*

Notation	*Nom*
$x := e$	*substitution simple*
$x, y := e, f$	*substitution multiple simple*
skip	*substitution sans effet*
$P \mid S$	*substitution préconditionnée*
$P \Longrightarrow S$	*substitution gardée*
$S \mathbin{[\!]} T$	*substitution choix borné*
$@z \cdot S$	*substitution choix non borné*
$S \,;\, T$	*séquencement*
$\textsc{while}\ P\ \textsc{do}\ S\ \textsc{invariant}\ I\ \textsc{variant}\ V\ \textsc{end}$	*substitution itération*

L'affectation multiple est simultanée. Par exemple $x, y := y, x$ décrit la permutation des valeurs de x et y.

La substitution préconditionnée permet de donner des contraintes sur les entrées (variables d'état et paramètres). L'opération ne pourra être invoquée que dans sa précondition (ceci sera pris en compte dans les obligations de preuve).

La substitution gardée permet de donner des hypothèses sous lesquelles la substitution S sera utilisée. Contrairement à la subtitution préconditionnée la garde n'a pas à être prouvée. Cette substitution sera utilisée en particulier pour définir des substitutions conditionnelles.

La substitution choix borné permet de décrire des spécifications non déterministes. Par exemple la substitution $x := 1 \mathbin{[\!]} x := 2$ décrit l'affectation d'une des valeurs 1 ou 2 à la variable x.

La substitution choix non borné est une extension du non déterminisme. Par exemple la substitution $@z \cdot (z > x \Longrightarrow x := z)$ a pour effet d'affecter à la variable x une valeur strictement supérieure à la précédente.

L'itération contient les éléments de sa preuve, c'est à dire une expression formelle de son invariant et la donnée de l'expression qui décroit. Les itérations, après preuve, seront donc toujours finies.

Les autres notations ont leur signification habituelle. Le séquencement peut ne pas être introduit comme une substitution primitive puisqu'il existe des règles permettant de l'éliminer. Nea'nmoins il est plus simple, pour les preuves, de ne pas utiliser ces règles.

La première partie du tableau décrit les substitutions admises dans les machines abstraites. Le séquencement n'est autorisé que dans les raffinements et les implémentations. L'itération n'est possible que dans les implémentations [2].

2.2.2 Notations syntaxiques

Des notations verbeuses permettent de faciliter la lecture et d'introduire des abréviations. Nous présentons ici les plus usuelles et celles que nous utilisons dans l'exemple. Par la suite nous préciserons toute nouvelle construction.

Notation	Définition
BEGIN S END	S
PRE P THEN S END	$P \mid S$
$x := e \mid\mid y := f$	$x, y := e, f$
ANY $x_1, \ldots, x_n$ WHERE P THEN S END	$@x_1 \ldots @x_n\ (P \Longrightarrow S)$

Il est aussi possible de définir d'autres constructions, comme les conditions :

Notation	Définition
IF P THEN S ELSE T END	$(P \Longrightarrow S) \mathbin{[]} (\neg P \Longrightarrow T)$
IF P THEN S END	$(P \Longrightarrow S) \mathbin{[]} (\neg P \Longrightarrow \mathsf{skip})$

Nous utiliserons aussi la construction :

$$\boxed{\text{LET } x_1, ..., x_n \text{ BE } \ x_1 = e_1 \wedge \ldots x_n = e_n \text{ IN } \ S \text{ END}}$$

qui est une forme particulière de ANY:

$$\boxed{\text{ANY } x_1, \ldots x_n \text{ WHERE } \ x_1 = e_1 \wedge \ldots x_n = e_n \ \text{ THEN } S \text{ END}}$$

2.3 Substitutions et transformateurs de prédicats

2.3.1 Axiomes des substitutions primitives

Comme dit précédemment, les substitutions généralisées peuvent être vues comme des transformateurs de prédicats. Nous donnons les axiomes pour les substitutions primitives. Ils permettront par la suite de construire les obligations de preuve.

2. Ces restrictions sont principalement d'ordre méthodologique.

Le cas de base $[x := e]R$ est la substitution uniforme de x par e dans R, décrite ci-après (définition 4).

Definition 2 *Axiomes*

Cas de substitution	*Réduction*	*Condition*
$[x, y := e, f]\, R$	$[z := f][x := e][y := z]\, R$	$z \backslash e, f, r$
$[\mathsf{skip}]\ R$	R	
$[P \mid S]\, R$	$P\ \wedge\ [S]\, R$	
$[S \parallel T]\, R$	$[S]R\ \wedge\ [T]\, R$	
$[P \Longrightarrow S]\, R$	$P \Rightarrow [S]\, R$	
$[@z \cdot S]\, R$	$\forall z \cdot [S]\, R$	$z \backslash R$
$[S\ ;T]\, R$	$[S]([T]\, R)$	

La notation $x\backslash P_1, \ldots P_n$ signifie que la variable x n'est pas libre dans les formules P_i. Si tel n'est pas le cas, il est toujours possible de renommer.

Pour une boucle, il n'est pas possible de déterminer exactement la plus faible précondition, comme dans le cas des autres substitutions. En effet, il s'agit d'un point fixe qui peut ne pas avoir d'expression finie. Par contre, on est capable d'exhiber des conditions suffisantes qui, si elles sont vérifiées, assurent qu'une propriété R est vraie à la sortie de la boucle. Ce sont ces conditions qui sont utilisées dans les preuves. Dans la définition ci-dessous x désigne l'espace de variables sur lequel est définie la substitution while.

Definition 3 *Preuve pour l'itération*

*[*WHILE P DO S INVARIANT I VARIANT V END*]* R

$\Leftrightarrow$

	I	*l'invariant est vrai avant la boucle*
$\wedge$	$\forall x\ .\ (I\ \wedge\ P \Rightarrow [S]\, I)$	*l'invariant est conservé dans la boucle*
$\wedge$	$\forall x\ .\ (I \Rightarrow V \in N)$	*Le variant est un entier naturel*
$\wedge$	$\forall x\ .\ (I\ \wedge\ P \Rightarrow [n := V][S](V < n))$	*le variant décroît à chaque pas*
$\wedge$	$\forall x\ .\ (I\ \wedge\ \neg P \Rightarrow R)$	*la sortie de boucle implique R*

En pratique, il y a souvent une initialisation avant une boucle. En terme de preuve cela revient à remplacer dans la liste des obligations de preuve ci-dessus, l'obligation de preuve I (invariant vrai avant la boucle) par $[T]\, I$.

La définition suivante décrit l'application des substitutions simples dans les formules logiques.

Definition 4 *Substitution dans les formules*

Substitution	*Définition*	*Condition*
$[x := e]\ (P \wedge Q)$	$[x := e]\ P \wedge [x := e]\ Q$	
$[x := e]\ (P \vee Q)$	$[x := e]\ P \vee [x := e]\ Q$	
$[x := e]\ (P \Rightarrow Q)$	$[x := e]\ P \Rightarrow [x := e]\ Q$	
$[x := e]\ \neg P$	$\neg [x := e]\ P$	
$[x := e]\ \forall x\ P$	$\forall x\ P$	
$[x := e]\ \forall y\ P$	$\forall y\ [x := e] P$	$y \setminus x, e$
$[x := e]\ \exists x\ P$	$\exists x\ P$	
$[x := e]\ \exists y\ P$	$\exists y\ [x := e] P$	$y \setminus x, e$

Le calcul d'une formule $[S]R$ s'effectue de la manière suivante :

1. réduction de la substitution S à sa forme mathématique ;
2. application des axiomes (définition 2) ;
3. application des règles de substitution dans les formules (définition 4) ;
4. substitution dans les termes [3].

2.4 Preuve d'invariants

Soit M une machine de la forme [4] :

```
MACHINE M(p)
CONSTRAINTS C
VARIABLES x
INVARIANT I
INITIALISATION U
OPERATIONS
    u ⟵ nom_op(w) ≙ PRE P THEN S END ;
    ...
END
```

3. Ces règles ne sont pas décrites dans ce document (voir le B-Book).
4. Certaines clauses ont été volontairement omises, pour alléger les obligations de preuve.

La substitution U de l'initialisation ne peut être une substitution préconditionnée.

Les obligations de preuve ci-dessous permettent de vérifier que l'initialisation établit l'invariant et que tout appel d'opération, dans sa précondition, préserve l'invariant (s'il est vrai avant appel, il est vrai après appel). Ceci permet d'établir des invariants inductifs.

$C \;\Rightarrow\; [U]I$	initialisation
$C \;\wedge\; I \;\wedge\; P \;\Rightarrow\; [S]I$	opérations

Ces formules sont implicitement quantifiées universellement sur les variables x de la machine. Remarquons qu'ici la précondition de l'opération est traitée de manière particulière. En effet lors de la *définition* de l'opération la précondition est mise en hypothèse. L'axiome $[P|S]R \equiv P \wedge [S]R$ s'applique à *l'appel* d'opération.

2.5 Preuve de l'exemple

L'invariant de la machine M est, rappelons le :

$$occupes \subseteq SIEGES \tag{4}$$
$$\wedge\; nb\_libre \in 0..nb\_max \tag{5}$$
$$\wedge\; nb\_libre = nb\_max - card(occupes) \tag{6}$$

2.5.1 Preuves pour l'initialisation

La variable nb_libre est initialisée à nb_max et la variable $occupes$ à $\emptyset$. Les obligations de preuve sont donc :

$$\emptyset \subseteq SIEGES \tag{7}$$
$$nb\_max \in 0..nb\_max \tag{8}$$
$$nb\_max = nb\_max - card(\emptyset) \tag{9}$$

La seconde obligation de preuve nécessite de montrer que l'ensemble $0..nb\_max$ est non vide. Ceci se prouve à partir de l'hypothèse $nb\_max \in 1..MAXINT$.

2.5.2 Preuves pour l'opération `place_libre`

L'opération `place_libre` ne modifiant aucune variable de la machine, elle préserve trivialement l'invariant.

2.5.3 Preuves pour l'opération `réserver`

Rappelons que l'opération `réserver` a pour effet de choisir une place non occupée et de décrémenter le nombre de places libres. Les obligations de preuve sont :

$$
\begin{aligned}
occupes \cup \{place\} &\subseteq SIEGES & (10)\\
nb\_libre - 1 &\in 0..nb\_max & (11)\\
nb\_libre - 1 &= nb\_max - card(occupes \cup \{place\}) & (12)
\end{aligned}
$$

sachant :

$$
\begin{aligned}
occupes &\subseteq SIEGES & (a)\\
nb\_libre &\in 0..nb\_max & (b)\\
nb\_libre &= nb\_max - card(occupes) & (c)\\
card(occupes) &\neq nb\_max & (d)\\
place &\in SIEGES - occupes & (e)
\end{aligned}
$$

Les hypothèses a à c sont relatives au fait que l'invariant est vrai avant appel. L'hypothèse d décrit la précondition de l'opération `réserver`. L'hypothèse e décrit les conditions sur la variable locale $place$ choisie.

La première obligation de preuve découle des deux hypothèses a et e. La seconde obligation de preuve se déduit de a et de $nb\_libre \neq 0$. Ceci est vrai puisque le cardinal de l'ensemble $occupes$ est strictement inférieur à nb_max (hypothèses a et d). L'obligation de preuve 3 se déduit de c et du fait que $place$ n'est pas déjà un élément de l'ensemble $occupes$ (hypothèse e).

2.5.4 Preuves pour l'opération `libérer`

L'opération `libérer(place)` rend libre la place et incrémente le nombre de places libres. Ceci est correct car il est supposé, en précondition, que la place est effectivement occupée. Les obligations de preuve sont :

$$
\begin{aligned}
occupes - \{place\} &\subseteq SIEGES & (13)\\
nb\_libre + 1 &\in 0..nb\_max & (14)\\
nb\_libre + 1 &= nb\_max - card(occupes - \{place\}) & (15)
\end{aligned}
$$

sachant :

$$
\begin{aligned}
occupes &\subseteq SIEGES & (a)\\
nb\_libre &\in 0..nb\_max & (b)\\
nb\_libre &= nb\_max - card(occupes) & (c)\\
place &\in occupes & (d)
\end{aligned}
$$

Les hypothèses a à c sont relatives au fait que l'invariant est vrai avant appel. L'hypothèse d décrit la précondition de l'opération `libérer`. La première obligation de preuve se déduit directement de l'hypothèse a. Pour la seconde obligation de preuve il faut déduire que $nb\_libre \neq nb\_max$. Ceci découle des hypothèses a, c et d (il y a au moins une place occupée). La troisième obligation de preuve se prouve à partir des hypothèses c et d.

3 Raffinement

Le raffinement permet d'ajouter des précisions à une spécification, par exemple pour se rapprocher d'un cahier des charges (en phase de spécification) ou pour aller vers du code (en phase de conception). Dans ce dernier cas on s'intéressera :

- à réduire le non-déterminisme
- à représenter des données abstraites

En pratique un raffinement se présente sous la forme :

- d'une spécification (un état interne, une initialisation, des opérations, des propriétés invariantes) ;
- d'une relation entre **variables abstraites** et **variables concrètes** [5]. Cette relation fait partie de l'invariant du raffinement. Lorsque les variables abstraites sont conservées (même nom) la relation de raffinement est l'identité.

3.1 Exemple

Nous donnons ici un raffinement du service de réservation. Dans ce raffinement l'ensemble abstrait est représenté de manière concrète et l'opération `réserver` est rendue déterministe : on choisit systématiquement la place libre ayant le plus petit numéro.

L'ensemble $occupes$ est représenté par une fonction totale qui, à chaque siège, associe le booléen TRUE si le siège est occupé, le booléen FALSE si il est libre (notation $etat \in SIEGES \rightarrow BOOL$). Cette représentation est plus concrète dans le sens où une telle fonction correspond à la notion de tableau des langages de programmation. La relation de raffinement se décrit par :

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

5. Les termes abstraits et concrets ne correspondent pas ici à la notion de variables et constantes abstraites ou concrètes des composants B (voir section 4).

La notation $r[e]$ désigne l'ensemble $\{y \mid \exists x(x \in e \ \wedge (x, y) \in r)\}$. Les opérations de ce raffinement sont redéfinies pour cette nouvelle représentation.

Le raffinement hérite de certaines informations de la machine `RESERVATION`, en particulier des contraintes sur les paramètres, de l'invariant sur les variables abstraites et des préconditions abstraites.

REFINEMENT
  RESERVATION1 (*nb_max*)

REFINES
  RESERVATION

DEFINITIONS
  SIEGES == (1 .. *nb_max*)

VARIABLES
  etat, *nb_libre*

INVARIANT
  $etat \in SIEGES \rightarrow BOOL \wedge occupes{=}etat^{-1}[\{\mathsf{TRUE}\}]$

INITIALISATION
  $nb\_libre := nb\_max \mid\mid etat := SIEGES \times \{\mathsf{FALSE}\}$

OPERATIONS

  $nb \leftarrow place\_libre =$
    BEGIN
      $nb := nb\_libre$
    END;

  $pp \leftarrow reserver =$
    BEGIN
      LET *place* BE *place*=min($etat^{-1}[\{\mathsf{FALSE}\}]$)
      IN *pp*:=*place* || *nb_libre*:=*nb_libre*-1 || etat(*place*):=TRUE
      END
    END;

  liberer (*place*) =
    BEGIN
      etat(*place*) := FALSE || *nb_libre* := *nb_libre*+1
    END

END

La notation $f(x) := e$, où f est une fonction, est une abréviation pour $f := f \lhd\!\!\!- (x, e)$, où $\lhd\!\!\!-$ est l'opérateur de réécriture d'une fonction.

3.2 Preuves de raffinement

La notion de raffinement consiste à pouvoir substituer une spécification (la machine abstraite) par un comportement plus précis (le raffinement) et que celui-ci ne délivre que des résultats observables sur la spécification. Dans le cas de la méthode B le raffinement est observationnel : les opérations qui renvoient des résultats doivent donner des résultats compatibles avec leur définition abstraite. En particulier, contrairement à VDM, il n'est pas demandé de préserver tous les états abstraits, si ceux-ci ne sont pas observables par une opération.

Soit M une machine et soit N un raffinement de cette machine.

MACHINE M VARIABLES y INVARIANT I INITIALISATION B r1 $\leftarrow$ OPERATIONS op = PRE P THEN K END END	REFINEMENT N REFINES M VARIABLES z INVARIANT J INITIALISATION C r2 $\leftarrow$ OPERATIONS op = PRE Q THEN L END END

Nous supposons ici que les variables de M et N sont distinctes. Lorsque ce n'est pas le cas il suffit de renommer les variables dans le raffinement est d'introduire une égalité dans l'invariant de liaison. De la même manière nous supposons que les paramètres résultats des opérations ont des noms différents (ce qui n'est syntaxiquement pas admis dans l'atelier B, l'outil associé à la méthode B).

Les obligations de preuve du raffinement consiste à montrer que l'initialisation concrète est un raffinement de l'initialisation abstraite et que, pour chaque opération, la définition concrète est un raffinement de la définition abstraite.

3.2.1 Obligations de preuve pour l'initialisation

$[C] \neg[B] \neg J$	initialisation

3.2.2 Obligations de preuve pour les opérations

Les obligations de preuve du raffinement d'opération consistent à montrer que :

- l'opération concrète est au moins définie pour toutes les données vérifiant la précondition de l'opération abstraite ;
- l'opération concrète modifie les variables concrètes de manière compatible avec les modifications des variables abstraites de l'opération abstraite ;
- si l'opération retourne des résultats alors les résultats obtenus par l'opération concrète sont des résultats admis par l'opération abstraite.

Ceci donne :

$I \wedge J \wedge P \Rightarrow Q$	précondition
$I \wedge J \wedge P \Rightarrow [L] \neg[K] \neg J$	opération sans résultat
$I \wedge J \wedge P \Rightarrow [L] \neg[K] \neg(J \wedge (r1 = r2))$	opération avec résultat

Les obligations de preuve sont implicitement quantifiées universellement sur les variables y et z.

La négation a pour effet de " choisir " des valeurs de l'opération abstraite. Par exemple la substitution $x := 1 \; [] \; x := 2$ se raffine en $y := 1$ pour l'invariant $x = y$. En effet il faut montrer :

$$
\begin{array}{l}
[y := 1]\neg[x := 1 \; [] \; x := 2]\neg(x = y) \\
[y := 1]\neg\,([x := 1]\neg\,(x = y) \wedge [x := 2]\neg\,(x = y)) \\
([y := 1](\neg\,[x := 1]\neg\,(x = y)) \vee (\neg\,[x := 2]\neg\,(x = y))) \\
[y := 1](1 = y \vee 2 = y) \\
(1 = 1 \vee 2 = 1)
\end{array}
$$

3.2.3 Invariant local

L'invariant du raffinement peut contenir à la fois la relation de raffinement et des propriétés sur les variables concrètes. Si l'invariant J est de la forme $J_1(z) \wedge J_2(y, z)$ alors l'obligation de preuve relative à J est :

$$I \wedge J \wedge P \Rightarrow [L] \neg[K] \neg(J_1(z) \wedge J_2(y, z))$$

Cette obligation de preuve peut se décomposer en deux :

$$I \wedge J \wedge P \Rightarrow [L]J_1(z)$$
$$I \wedge J \wedge P \Rightarrow [L]\, \neg[K]\, \neg J_2(y,z)$$

La première obligation de preuve correspond à la preuve de l'invariant local J_1, la seconde à la preuve de la relation de raffinement J_2. Cette décomposition n'est correcte que parce que l'invariant J_1 ne porte pas sur la variable abstraite y.

3.3 Preuve de l'exemple

Nous montrons la preuve de raffinement pour l'exemple. Rappelons que l'invariant du raffinement `RESERVATION1` est :

$$etat \in SIEGES \to BOOL \wedge occupes = etat^{-1}[\{\mathsf{TRUE}\}]$$

Les obligations de preuve consistent donc à établir :

1. l'invariant liant les variables $occupes$ et $etat$:
$$occupes = etat^{-1}[\{\mathsf{TRUE}\}]$$
2. l'invariant implicite sur la variable nb_libre. Après renommage de nb_libre en nb_libre2 dans le raffinement il faudra montrer :
$$nb\_libre = nb\_libre2$$
3. l'invariant relatif à la variable $etat$:
$$etat \in SIEGES \to BOOL$$

Nous ne présentons que les obligations de preuve pour 1. Dans le cas 2 les preuves sont triviales puisque la variable nb_libre est modifiée de la même manière dans le raffinement et la machine. Pour le cas 3 les obligations de preuve correspondent à la préservation de l'invariant portant sur les variables du raffinement, comme montré précédemment.

3.3.1 Preuves pour l'initialisation

Dans le cas de l'initialisation l'obligation de preuve revient à montrer qu'initialiser la fonction à faux partout revient à initialiser l'ensemble abstrait à l'ensemble vide. C'est à dire :

$$\emptyset = (SIEGES \times \{\mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$$

ce qui est trivialement vrai.

3.3.2 Preuves pour l'opération `libérer`

L'opération abstraite enlève la place à l'ensemble $occupes$ et l'opération concrète affecte à cette place le booléen FALSE. L'obligation de preuve est donc :

$$occupes - \{place\} = (etat \lhd\!\!\!+ \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$$

sachant que l'invariant est vrai avant appel, c'est à dire $occupes = etat^{-1}[\{\mathsf{TRUE}\}]$. Ceci se prouve facilement.

3.3.3 Preuves pour l'opération `réserver`

L'opération abstraite choisit une place libre et l'ajoute à l'ensemble $occupes$. L'opération concrète affecte la place libre ayant le plus petit numéro au booléen TRUE. Ici les obligations de preuve consistent à montrer que :

- l'invariant de liaison $occupes = etat^{-1}[\{\mathsf{TRUE}\}]$ est vérifié par les affectations abstraites et concrètes ;
- la valeur résultat au niveau concret, $place = min(etat^{-1}[\{\mathsf{FALSE}\}])$ est bien une valeur qui vérifie la propriété abstraite attendue, $place \in SIEGES - occupes$.

Les obligations de preuve de l'opération `réserver` sont donc :

$$occupes \cup \{place\} = (etat \lhd\!\!\!+ \{place \mapsto \mathsf{TRUE}\})^{-1}[\{\mathsf{TRUE}\}] \tag{16}$$

$$place = min(etat^{-1}[\{\mathsf{FALSE}\}]) \Rightarrow place \in SIEGES - occupes \tag{17}$$

La première obligation de preuve de déduit du fait que l'invariant est vrai avant appel, c'est à dire $occupes = etat^{-1}[\{\mathsf{TRUE}\}]$. La seconde est aussi vraie puisque $SIEGES - occupes = etat^{-1}[\{\mathsf{FALSE}\}]$.

4 Implémentation

Un programme est un raffinement particulier qui vérifie certaines contraintes correspondant à celles des langages de programmation classiques. Il se présente avec le mot réservé IMPLEMENTATION.

Les variables doivent correspondre à des structures de données du langage de programmation (entiers bornés, booléens, tableaux ...). Les variables admises dans les implémentations en B sont qualifiées de concrètes (clause CONCRETE_VARIABLES). Dans sa version actuelle (3.5) l'atelier B admet les types de base, des formes restreintes de tableaux (les tableaux de tableaux ne sont pas admis) et les records.

Les substitutions généralisées autorisées doivent correspondre à des instructions classiques des langages de programmation. Les instructions B admises sont l'affectation, différentes formes de conditionnelles, le séquencement, l'itération et la déclaration de variables locales. Ce langage est appelé B0.

D'autre part le langage B0 offre un ensemble d'opérateurs prédéfinis sur les types permis (opérateurs arithmétiques, accès aux éléments d'un tableau ...). Nous ne les détaillons pas ici.

4.1 Exemple

Dans l'exemple les variables choisies, $etat$ et nb_libre, sont des variables concrètes. Elles sont conservées au niveau de l'implémentation. De la même manière l'opération `place_libre` est déjà exprimée en terme d'instructions. Elle n'est pas modifiée. Dans l'opération `libérer` l'affectation simultanée est remplacée par un séquencement.

Dans l'opération `réserver`, la recherche de la place libre ayant le plus petit numéro, est implantée par l'itération suivante :

```
VAR ind , val IN
   ind := 1;
   val := etat(ind) ;
   WHILE  val=TRUE  DO
     ind := ind + 1 ;
     val := etat(ind)
   END;
```

La forme des expressions est limitée dans le langage B0, ici nous ne pouvons pas écrire directement la condition $etat(ind) = TRUE$, c'est pour cela que nous avons introduit la variable locale val. Cette restriction est liée aux obligations de preuve produites pour les implémentations [6] (section 4.2).

La notation :

$$\boxed{\text{VAR } x_1, ..., x_n \text{ IN } S \text{ END}}$$

est définie par :

$$\boxed{@\, x_1, \ldots x_n \cdot S}$$

L'invariant de la boucle décrit le fait que l'indice qui parcourt le tableau, la variable locale ind, appartient à l'intervalle $1\ ..\ nb\_max$, que la variable val contient la valeur

6. Bien que cette restriction pourrait être assouplie.

courante de $etat(ind)$ et qu'aucune place libre n'a déjà été rencontrée. Cet invariant est :

$$ind \in 1 \,..\, nb\_max \wedge val = etat(ind) \wedge etat[1 \,..\, ind - 1] \subseteq \{\mathsf{TRUE}\}$$

On peut remarquer que cette boucle s'arrête puisqu'il y a nécessairement une place telle que $etat(place) = \mathsf{FALSE}$. Ceci est garanti par la précondition initiale de l'opération `réserver` ($card(occupes) \neq nb\_max$) qui garantit $etat^{-1}[\{\mathsf{FALSE}\}] \neq \emptyset$.

L'initialisation n'est pas faite par une boucle qui affecte la valeur FALSE à chaque élément du tableau, mais par une affectation en bloc. Ceci est dû à une limitation actuelle de l'atelier B qui impose, dans sa version actuelle,[7] une affectation en bloc des tableaux.

```
IMPLEMENTATION
   CODE (nb_max)

REFINES
   RESERVATION1

CONCRETE_VARIABLES
  etat, nb_libre

INITIALISATION
  etat := (1 .. nb_max) × {FALSE} ; nb_libre := nb_max

OPERATIONS
  nb ← place_libre =
     BEGIN
        nb := nb_libre
     END;

  pp ← reserver  =
     BEGIN
        VAR ind , val IN
           ind := 1;
           val := etat(ind) ;
           WHILE  val=TRUE  DO
             ind := ind + 1 ;
             val := etat(ind)
           INVARIANT  ind ∈ 1 .. nb_max ∧ val=etat(ind) ∧ etat[1..ind-1] ⊆ {TRUE}
           VARIANT  nb_max - ind
           END;
            etat(ind):=TRUE;
           nb_libre := nb_libre - 1;
           pp := ind
        END
```

7. Version 3.5. Ceci permet de garantir que toutes les variables ont reçu une valeur initiale.

END;

liberer (*place*) =
BEGIN
etat(*place*):= FALSE; *nb_libre* := *nb_libre* + 1
END

END

4.2 Obligations de preuve

Les obligations de preuve sur une implémentation comportent les obligations de preuve classiques d'un raffinement ainsi que des obligations de preuve relatives aux opérateurs prédéfinis du langage B0.

En effet les opérateurs prédéfinis ne sont pas nécessairement les mêmes que dans le modèle mathématique. Ces opérateurs sont généralement partiels : ils peuvent imposer des restrictions sur les données et, en particulier, ils se prémunissent contre les débordements arithmétiques. Ces opérateurs sont implicitement préconditionnés et sont alors traités comme les opérations définies par le spécifieur.

Par exemple l'opérateur $+$ du langage B0 est défini par :

$$r \leftarrow x + y = \text{PRE } x \in \mathsf{INT} \wedge y \in \mathsf{INT} \wedge x{+}y \in \mathsf{INT} \text{ THEN } r{:=}x{+}y \text{ END}$$

où INT désigne les entiers bornés ($\mathsf{INT} = MININT..MAXINT$). Dans $x + y$ l'occurrence de l'opérateur $+$ désigne l'opérateur mathématique d'addition.

Pour les opérateurs arithmétiques les préconditions sont relatives aux débordements arithmétiques et, dans le cas de la division et du modulo au fait que le second argument est non nul. Pour l'opérateur d'indexation de tableau $t(e)$ la précondition implicite est $e \in dom(t)$. Ceci correspond à la vérification du non débordement d'intervalle.

4.2.1 Erreurs à l'exécution

La méthode B se prémunit, a priori, contre les erreurs à l'exécution, ceci dépendant bien sur des contraintes du langage de programmation. En particulier le langage B0 n'offre pas de constructions complexes introduisant des erreurs potentielles à l'exécution (pas d'allocation dynamique, pas d'enregistrement à champ variant ...) qui rendraient les preuves plus complexes.

D'autre part B garantit que toutes les variables sont initialisées, par construction[8]. Remarquons que dans un langage de programmation, l'accès à une variable non ini-

8. comme ceci est fait dans le langage Java.

tialisée n'est pas en soit une erreur à l'exécution [9], mais peut potentiellement en introduire.

4.2.2 Vérification des préconditions

Nous n'avons pas encore vu la vérification des préconditions à l'appel. Les obligations de preuve sont :

$\mathsf{trm}(U)$	pour une initialisation U
$I \wedge P \Rightarrow \mathtt{trm}(S)$	pour une opération de la forme $P\|S$ et un invariant I

avec :

Definition 5 *Terminaison*

Notation	*Définition*
$\mathsf{trm}(S)$	$[S]true$

Le prédicat trm définit une notion de terminaison des substitutions. Ce prédicat peut être vu comme " un ramasseur " de préconditions. Il contient aussi la preuve d'arrêt des itérations. Ceci correspond à la notion classique de terminaison d'un programme. Ces obligations de preuve s'appliquent aussi bien entendu à l'appel d'opérations définies par le spécifieur.

Exemple :

$$\mathsf{trm}(x := 2 \,;\ \text{PRE } x \neq 0 \text{ THEN } y := y \ mod\ x \text{ END})$$
$$[x := 2]((x \neq 0) \wedge [y := y \ mod\ x]true)$$
$$2 \neq 0 \wedge true$$
$$true$$

Ces obligations sont en général redondantes avec les obligations de preuve de préservation d'invariant et de raffinement. Elles ne sont donc pas produites, sauf si la

9. ceci correspond, par exemple, à la notion d'exécution erronée du langage Ada.

préservation d'invariant ou la preuve de raffinement ne produisent pas d'obligation de preuve. En effet pour toute formule R on a :

$R \Rightarrow true$	règle logique
$[S]R \Rightarrow [S]true$	monotonie de la substitution

et donc, dès qu'une subtitution S établit une formule quelconque R, la terminaison de S est établie.

4.3 Preuve de l'exemple

4.3.1 Initialisation

Pour l'initialisation, le raffinement se prouve trivialement puisque l'initialisation concrète est presque identique à l'initialisation abstraite, si ce n'est qu'il y a introduction d'un séquencement à la place de l'affectation simultanée. L'affectation $nb\_libre := nb\_max$ produit une obligation de preuve supplémentaire qui vérifie que l'expression nb_max est un entier borné ($nb\_max \in \mathsf{INT}$).

4.3.2 Opération `place_libre`

Le code concret et le code abstrait sont les mêmes, la preuve de raffinement est triviale. Les obligations de preuve supplémentaires vérifient que l'affectation $nb := nb\_libre$ est valide sur les entiers bornés. Or $nb\_libre \in 0..nb\_max$ et $nb\_max \in 1..MAXINT$ d'où $nb\_libre \in \mathsf{INT}$.

4.3.3 Opération `libérer`

La preuve de raffinement est triviale puisque la seule modification est l'introduction du séquencement à la place de l'affectation simultanée. Les obligations de preuve liées aux préconditions des opérateurs prédéfinis sont les suivantes :

$$place \in dom(etat) \tag{18}$$

$$nb\_libre + 1 \in \mathsf{INT} \tag{19}$$

La première obligation de preuve se prouve facilement à partir de la précondition initiale de l'opération `libérer` ($place \in occupes$) et des hypothèses $occupes \subseteq SIEGES$ (invariant de la machine `RESERVATION`) et $etat \in SIEGES \rightarrow BOOL$ (invariant du raffinement `RESERVATION1`).

La seconde obligation de preuve nécessite de montrer que $nb\_libre < nb\_max$. Cette preuve à déjà été faite lors de la preuve de préservation de l'invariant ($nb\_libre \in 0..nb\_max$) de la machine `RESERVATION` par l'opération `libérer`. Nous ne la redétaillons pas.

4.3.4 Opération `réserver`

Pour l'opération `réserver` il faut prouver l'itération (invariant et variant), faire la preuve de raffinement et vérifier les préconditions implicites des opérateurs prédéfinis.

Preuve de l'invariant de boucle. Rappelons que l'invariant est :

$$ind \in 1..nb\_max \wedge val = etat(ind) \wedge etat[1..ind-1] \subseteq \{\mathsf{TRUE}\}$$

L'initialisation de la boucle est $ind := 1 \ ; \ val := etat(ind)$. Les obligations de preuve sont donc :

$$1 \in 1 \,..\, nb\_max \tag{20}$$
$$etat(1) = etat(1) \tag{21}$$
$$etat[1 \,..\, 0] \subseteq \{\mathsf{TRUE}\} \tag{22}$$

Pour la première obligation de preuve il faut montrer que l'ensemble $1 \,..\, nb\_max$ est non vide, ce qui est vrai puisque $nb\_max \in 1..MAXINT$. La seconde obligation de preuve est trivialement vraie et la troisième se réduit à $\emptyset \subseteq \{TRUE\}$.

Il faut montrer que le corps de l'itération ($ind := ind + 1; val := etat(ind)$) préserve l'invariant, lorsque la condition de l'itération est vraie ($val = \mathsf{TRUE}$). Les obligations de preuve sont :

$$ind + 1 \in 1 \,..\, nb\_max \tag{23}$$
$$etat(ind + 1) = etat(ind + 1) \tag{24}$$
$$etat[1 \,..\, ind + 1 - 1] \subseteq \{\mathsf{TRUE}\} \tag{25}$$

sachant :

$$\begin{array}{ll}
ind \in 1 \,..\, nb_max & (a) \\
val = etat(ind) & (b) \\
etat[1 \,..\, ind-1] \subseteq \{\mathsf{TRUE}\} & (c) \\
val = \mathsf{TRUE} & (d) \\
card(occupes) \neq nb_max & (e) \\
occupes = etat^{-1}[\{\mathsf{TRUE}\}] & (f) \\
etat \in 1 \,..nb_max \rightarrow BOOL & (g)
\end{array}$$

Les hypothèses a à c correspondent à l'invariant de l'itération. L'hypothèse d est la condition de l'itération. L'hypothèse e est la précondition initiale de l'opération `réserver` (donnée dans la machine `RESERVATION`) et les hypothèses f et g correspondent à l'invariant de liaison entre la variable abstraite $occupes$ et la variable concrète $etat$ (invariant du raffinement `RESERVATION1`).

Pour la première obligation de preuve il faut montrer que l'ensemble $1 \,..\, nb\_max$ est non vide, comme pour l'initialisation. D'autre part il faut déduire que $ind < nb\_max$ sachant $ind \in 1..nb\_max$. Supposons $ind = nb\_max$. On a alors :

$$\begin{array}{ll} val = etat(nb\_max) & (b) \\ etat[1 .. nb\_max - 1] \subseteq \{\mathsf{TRUE}\} & (c) \\ val = \mathsf{TRUE} & (d) \end{array}$$

A partir de b, c, d et g on déduit $etat^{-1}[\{\mathsf{TRUE}\}] = 1..nb\_max$ et donc, par f, $card(occupes) = nb\_max$, ce qui est contradictoire avec l'hypothèse e.

La seconde obligation de preuve est trivialement vraie et la troisième se déduit facilement des hypothèses b, c et d.

Preuve du variant.

Les obligations de preuve sur le variant ($nb\_max - ind$) consiste à montrer que l'expression reste positive ou nulle et qu'elle décroit à chaque passage, c'est à dire :

$$0 \leq nb\_max - ind \tag{26}$$

$$nb\_max - (ind + 1) < nb\_max - ind \tag{27}$$

Puisque l'invariant de l'itération garantit $ind \in 1 .. nb\_max$, ces deux obligations de preuve sont trivialement vraies.

Preuve de raffinement.

L'invariant de liaison est :

$$etat = etat_2 \wedge nb\_libre = nb\_libre_2$$

où $etat_2$ et nb_libre_2 désignent les variables de l'implémentation. Il faut aussi montrer que la valeur résultat de la définition concrète de l'opération `réserver` raffine la valeur résultat abstraite.

Les obligations de preuve sont donc :

$$etat \Leftarrow\!\!+ (min(etat^{-1}[\{\mathsf{FALSE}\}]), \mathsf{TRUE}) = etat_2 \Leftarrow\!\!+ (ind, \mathsf{TRUE}) \tag{28}$$

$$nb\_libre_2 - 1 = nb\_libre - 1 \tag{29}$$

$$ind = min(etat^{-1}[\{\mathsf{FALSE}\}]) \tag{30}$$

sachant :

$$\begin{array}{ll} etat = etat_2 & (a) \\ nb\_libre = nb\_libre_2 & (b) \\ ind \in 1..nb\_max & (c) \\ val = etat(ind) & (d) \\ etat[1 .. ind - 1] \subseteq \{\mathsf{TRUE}\} & (e) \\ etat \in 1..nb\_max \rightarrow BOOL & (f) \\ val = \mathsf{FALSE} & (g) \end{array}$$

Les hypothèses a et b correspondent à l'invariant de raffinement. Les hypothèses c à e correspondent à l'invariant de l'itération. L'hypothèse f est l'invariant sur la variable état (donné dans le raffinement `RESERVATION1`) et l'hypothèse g est la condition de sortie de la boucle.

La première obligation de preuve se prouve à partir de la troisième et de l'hypothèse a. La seconde obligation de preuve se prouve directement à partir de b. Pour la troisième on peut montrer que $ind \in etat^{-1}[\{\mathsf{FALSE}\}]$ à partir de d et g. D'autre part, par les hypothèses e et f on déduit que ind est le minimum.

Preuve pour les opérateurs prédefinis. Nous donnons ci-dessous les obligations de preuves introduites par les opérateurs prédéfinis.

1. Affectation $val := etat(ind)$ avant l'itération :
$$1 \in dom(etat) \tag{31}$$
2. Affectations $ind := ind + 1$ et $val := etat(ind)$ dans l'itération :
$$ind + 1 \in \mathsf{INT} \tag{32}$$
$$ind + 1 \in dom(etat) \tag{33}$$
3. Affectations $etat(ind) := \mathsf{TRUE}$, $nb\_libre := nb\_libre - 1$ et $place := ind$ après l'itération :
$$ind \in dom(etat) \tag{34}$$
$$nb\_libre - 1 \in \mathsf{INT} \tag{35}$$
$$ind \in \mathsf{INT} \tag{36}$$

Les seules obligations de preuve qui posent quelques difficultés sont celles dans le corps de boucle (cas 2) car il faut prouver $ind < nb\_max$. Cette preuve a déjà été faite pour montrer l'invariant de boucle $ind \in 1..nb\_max$.

5 L'atelier B

L'atelier B, développé par Stéria-méditerranée maintenant ClearSy, outille la méthode B. Il offre un analyseur, un générateur d'obligations de preuve, un prouveur dédié (automatique et interactif) et un générateur de code (C et Ada). L'utilisation classique consiste, une fois les obligations de preuve produites, à lancer le démonstrateur automatique. Pour les obligations de preuve restantes le prouveur interactif permet de guider la preuve.

Les obligations de preuves sont classées en différentes catégories :

– `NbObv` le nombre d'obligations de preuves évidentes, c'est à dire pour lesquelles aucune activité de preuve n'est nécessaire ;

– `NbPO` les obligations de preuve non évidentes (pour lesquelles la preuve doit être faite). Ces preuves se découpent en deux catégories : `NbPRa` le nombre prouvé automatiquement et `NbPRi` le nombre prouvé interactivement.

– Le pourcentage final correspond à la couverture des preuves (automatiques ou interactives).

Nous présentons ci-dessous les résultats après utilisation du prouveur automatique.

5.1 Machine `RESERVATION`

Ceci correspond à la sous-section 2.5.

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	0	3	0	3	100
place_libre	4	0	0	0	100
reserver	3	2	0	1	50
liberer	1	3	0	2	66
RESERVATION	8	8	0	6	75

L'obligation de preuve non prouvée dans l'opération `réserver` est celle consistant à montrer que $nb\_libre - 1 \in 0..nb\_max$. Rappelons qu'elle nécessite de prouver que $nb\_libre > 0$, à partir de la précondition de l'opération `réserver` (sous-section 2.5.3).

Pour l'opération `libérer` c'est l'invariant $nb\_libre+1 = nb\_max-card(occupes-\{place\})$ qui n'est pas prouvé automatiquement.

5.2 Raffinement `RESERVATION1`

Ceci correspond à la sous-section 3.3.

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	2	3	0	3	100
place_libre	7	0	0	0	100
reserver	3	5	0	4	80
liberer	3	3	0	2	66

RESERVATION1	15	11	0	9	81

Les obligations de preuve qui restent sont relatives au raffinement de $occupes$ en $etat$.

Pour l'opération `réserver` (3.3.3) :

$$place = min(etat^{-1}[\{\mathsf{FALSE}\}]) \Rightarrow place \in SIEGES - occupes$$

Pour l'opération `libérer` (3.3.2) :

$$occupes - \{place\} = (etat \mathbin{<\!\!+} \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$$

5.3 Implémentation `CODE`

	NbObv	NbPO	NbPRi	NbPRa	%Pr
ValuesLemmas	1	0	0	0	100
Initialisation	3	3	0	3	100
place_libre	4	3	0	3	100
reserver	8	27	0	21	77
liberer	4	8	0	6	75
CODE	20	41	0	33	80

Pour l'opération `libérer` (sous-section 4.3.3) il reste à prouver $nb\_libre + 1 \in \mathsf{INT}$ qui nécessite de montrer que $nb\_libre < nb\_max$ et $MININT \leq nb\_libre + 1$.

Pour l'operation `réserver` (sous-section 4.3.4) il reste :

– deux preuves relatives au raffinement :

$$etat \mathbin{<\!\!+} (min(etat^{-1}[\{\mathsf{FALSE}\}]), \mathsf{TRUE}) = etat_2 \mathbin{<\!\!+} (ind, \mathsf{TRUE}) \quad (37)$$

$$ind = min(etat^{-1}[\{\mathsf{FALSE}\}]) \quad (38)$$

– deux preuves relatives à la préservation de l'invariant dans l'itération :

$$ind + 1 \in 1 \,..\, nb\_max \quad (39)$$

$$etat[1 \,..\, ind + 1 - 1] \subseteq \{\mathsf{TRUE}\} \quad (40)$$

– deux preuves relatives aux opérateurs prédéfinis dans l'itération, $ind + 1 \in \mathsf{INT}$ et $ind + 1 \in dom(etat)$, qui nécessitent de prouver $ind < nb\_max$.

Références

[1] J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.