Spécification en B

Support de cours

Ecole des Jeunes Chercheurs en Programmation

EJCP 2007

3-15 juin 2007
Dinard - Rennes

Didier Bert, Marie-Laure Potet
LSR-IMAG, Grenoble

Table des matières

Table des figures

Chapitre 1

Introduction aux notations logiques, ensemblistes et aux types de données prédéfinis

Résumé

Ce chapitre rappelle les notations logiques et ensemblistes et donne leur signification. Il donne la liste des types de données prédéfinis avec leurs opérations. Ces notations sont utilisées dans la méthode B pour spécifier les états, les invariants, les conditions, donner les propriétés des constantes etc.

1.1 Notations logiques

Les expressions logiques dénotent des prédicats qui ont une interprétation dans le domaine des valeurs de vérité : btrue et bfalse. Notons que ces constantes n'existent pas en tant que telles dans le langage. Les opérateurs qui permettent de combiner des prédicats sont les connecteurs usuels de la logique :

Symbole	Signification	Définition
$\wedge$	et logique	
$\neg$	négation	
$\vee$	ou logique	$a \vee b \stackrel{def}{=\!=} \neg(\neg a \wedge \neg b)$
$\Rightarrow$	implication	$a \Rightarrow b \stackrel{def}{=\!=} \neg a \vee b$
$\Leftrightarrow$	équivalence	$a \Leftrightarrow b \stackrel{def}{=\!=} (a \Rightarrow b) \wedge (b \Rightarrow a)$

On peut quantifier les prédicats par les quantificateurs universel et existentiel :

Symbole	Signification	Syntaxe	Définition
$\forall$	pour tout	$\forall\, Id\_liste \cdot (Prédicat)$	
$\exists$	il existe	$\exists\, Id\_liste \cdot (Prédicat)$	$\exists\, x \cdot (P) \stackrel{def}{=\!=} \neg \forall\, x \cdot (\neg\, P)$

Le non-terminal Id_liste représente une liste d'identificateurs séparés par des virgules. Le non-terminal $Prédicat$ désigne un prédicat quelconque. Dans le cas du "$\forall$" le prédicat quantifié est toujours de la forme : $P \Rightarrow Q$, où P définit le typage ensembliste des variables liées par le quantificateur. Dans le deuxième cas, le prédicat est de la forme $P \ \wedge \ Q$.

Exemple : $\forall xx \cdot (xx \in PERSONNE \Rightarrow age(pere(xx)) > age(xx))$.

Occurrences libres, occurrences liées.

Les prédicats, comme ensuite les expressions, contiennent des occurrences de variables. Ces occurrences sont *liées* si elles apparaissent sous la portée d'un quantificateur, sinon elles sont *libres*. Dans ce document, on utilise la notation :

$$x \backslash P \qquad\qquad x \text{ n'est pas libre dans } P$$

Notation de la "substitution".

En B, la notion de substitution est fondamentale[1]. Elle sera la base de la spécification des opérations (cf. chapitre 2). Une substitution est notée :

$$[x := E]\, P$$

Elle indique le remplacement uniforme des occurrences libres de x par E dans P. On peut aussi avoir une substitution multiple avec une signification évidente :

$$[x_1, x_2, \ldots := E_1, E_2, \ldots]\, P$$

Exemple : $[xx, yy := yy + 1, xx]\, (xx * yy) \ = \ (yy + 1) * xx$.

Remarques :

1. Les identificateurs des spécifications B doivent avoir au minimum deux lettres (ou une lettre et un chiffre).
2. Les sous-expressions d'une expression logique peuvent être parenthésées.
3. On ne redonne pas ici les axiomes qui s'appliquent aux connecteurs logiques (associativité, commutativité, distributivité, etc.).

Prédicat d'égalité.

Toutes les valeurs (expressions) d'ensemble ou les valeurs (expressions) de types prédéfinis possèdent un prédicat égalité.

[1] C'est évidemment le cas de toutes les théories logiques et du lambda-calcul.

Symbole	Signification	Définition
$=$	est égal à	
$\neq$	n'est pas égal à	$x \neq y \stackrel{def}{=\!=} \neg(x = y)$

1.2 Notations de la théorie des ensembles

1.2.1 Construction d'ensembles

Les ensembles peuvent être des ensembles d'éléments sans structure, ou bien des produits cartésiens d'ensembles, ou encore des parties d'un ensemble. Les notations de construction d'ensembles sont :

Symbole	Signification	Syntaxe
$\emptyset$	ensemble vide	
$\times$	produit cartésien	$Ensemble \times Ensemble$
$\mathbb{P}$	ensemble des sous-ensembles	$\mathbb{P}(Ensemble)$
$\mathbb{P}_1$	ensemble des sous-ensembles non vides	$\mathbb{P}_1(Ensemble)$
$\mathbb{F}$	ensemble des sous-ensembles finis	$\mathbb{F}(Ensemble)$
$\mathbb{F}_1$	ensemble des sous-ensembles finis non vides	$\mathbb{F}_1(Ensemble)$
$\{\ ,\ , \ldots\}$	ensembles définis en extension	$\{Expression\_liste\}$
$\{\mid\}$	ensembles définis en compréhension	$\{Id\_liste \mid Prédicat\}$

Le non-terminal *Ensemble* désigne une expression construisant un ensemble. Le non-terminal *Expression_liste* représente une liste d'expressions quelconques séparées par des virgules. Les éléments d'un produit cartésien sont des paires d'éléments.

1.2.2 Prédicats sur les ensembles

On dispose des prédicats suivants relatifs aux ensembles :

Symbole	Signification	Définition
$\in$	appartient à	
$\notin$	n'appartient pas à	$x \notin s \stackrel{def}{=\!=} \neg\,(x \in s)$
$\subseteq$	est inclus dans	$s \subseteq t \stackrel{def}{=\!=} s \in \mathbb{P}(t)$
$\not\subseteq$	n'est pas inclus dans	$s \not\subseteq t \stackrel{def}{=\!=} \neg\,(s \subseteq t)$
$\subset$	est strictement inclus dans	$s \subset t \stackrel{def}{=\!=} (s \subseteq t \ \wedge\ s \neq t)$
$\not\subset$	n'est pas strictement inclus dans	$s \not\subset t \stackrel{def}{=\!=} \neg\,(s \subset t)$

Le prédicat d'appartenance permet de "donner un sens" (observationnel) aux constructions d'ensembles. C'est ainsi qu'on a les axiomes :

$$x, y \in E_1 \times E_2 \ \Leftrightarrow\ (x \in E_1 \ \wedge\ y \in E_2)$$

Une paire peut aussi être notée : $(x \mapsto y)$. Le produit cartésien est associatif à gauche :

$$E_1 \times E_2 \times E_3 \ =\ (E_1 \times E_2) \times E_3$$

Les éléments d'un ensemble "parties d'un ensemble" sont eux-mêmes des ensembles. L'appartenance est définie par :

$$s \in \mathbb{P}(t) \ \Leftrightarrow\ \forall x \cdot (x \in s \Rightarrow x \in t)$$

Les éléments d'un ensemble de la forme : $\{x \mid x \in s \ \wedge\ P\}$ sont les éléments de s qui satisfont P, autrement dit :

$$y \in \{x \mid x \in s \ \wedge\ P\} \ \Leftrightarrow\ y \in s \ \wedge\ [x := y]\,P$$

L'égalité des ensembles est une notion dérivée des prédicats précédents, puisqu'on a :

$$\forall\, x \cdot (x \in s \Leftrightarrow x \in t) \ \Leftrightarrow\ s = t$$

L'inclusion jouit des propriétés habituelles (que l'on peut démontrer à partir des axiomes) :

$s \subseteq s$	(réflexivité)
$s \subseteq t \ \wedge\ t \subseteq u \Rightarrow s \subseteq u$	(transitivité)
$s \subseteq t \ \wedge\ t \subseteq s \Rightarrow s = t$	(anti-symétrie)

1.2.3 Expressions d'ensembles

Le premier jeu d'opérateurs concerne les opérations simples entre ensembles. On suppose, comme hypothèses $s_1 \subseteq t$ et $s_2 \subseteq t$:

	Signification	Définition
$\cup$	union	$s_1 \cup s_2 \stackrel{def}{=\!=} \{x \mid x \in t \ \wedge \ (x \in s_1 \ \vee \ x \in s_2)\}$
$\cap$	intersection	$s_1 \cap s_2 \stackrel{def}{=\!=} \{x \mid x \in t \ \wedge \ (x \in s_1 \ \wedge \ x \in s_2)\}$
$-$	différence d'ensembles	$s_1 - s_2 \stackrel{def}{=\!=} \{x \mid x \in t \ \wedge \ (x \in s_1 \ \wedge \ x \notin s_2)\}$

Le deuxième jeu d'opérateurs traite d'union et d'intersection de familles d'ensembles. Ce sont :

Symbole	Signification	Syntaxe
union	union généralisée	union($Ensemble$)
inter	intersection généralisée	inter($Ensemble$)
$\bigcup$	union quantifiée	$\bigcup Id\_liste \cdot (Prédicat \mid Ensemble)$
$\bigcap$	intersection quantifiée	$\bigcap Id\_liste \cdot (Prédicat \mid Ensemble)$

Dans les deux premières opérations, *Ensemble* est un ensemble d'ensembles et l'union (resp. l'intersection) réalise l'union (resp. l'intersection) de cette famille. Pour l'intersection, la famille doit être non vide.

Exemple : union$(\{\{1,2\},\{1,2,3\},\{2,4,5\}\}) = \{1,2,3,4,5\}$
inter$(\{\{1,2\},\{1,2,3\},\{2,4,5\}\}) = \{2\}$

Pour l'union ou l'intersection quantifiées, la partie *Ensemble* est une expression d'ensembles qui dépend des variables Id_liste. Le $Prédicat$ caractérise le domaine de variation de ces variables, sur lequel se réalise l'union ou l'intersection.

Exemple : si $E = \{2,4\}$ alors :

$$\bigcup x \cdot (x \in E \mid \{y \mid y \in \mathbb{N} \ \wedge \ x-1 \leq y \leq x+1\}) \ = \ \{1,2,3\} \cup \{3,4,5\} \ = \ \{1,2,3,4,5\}$$

Les définitions mathématiques sont :

Condition	Expression	Définition
$U \in \mathbb{P}(\mathbb{P}(t))$	$\mathsf{union}(U)$	$\{x \mid x \in t \ \wedge \ \exists y \cdot (y \in U \ \wedge \ x \in y)\}$
$U \in \mathbb{P}_1(\mathbb{P}(t))$	$\mathsf{inter}(U)$	$\{x \mid x \in t \ \wedge \ \forall y \cdot (y \in U \Rightarrow x \in y)\}$
$\forall x \cdot (P \Rightarrow E \subseteq t)$	$\bigcup x \cdot (P \mid E)$	$\{y \mid y \in t \ \wedge \ \exists x \cdot (P \ \wedge \ y \in E)\}$
$\forall x \cdot (P \Rightarrow E \subseteq t)$	$\bigcap x \cdot (P \mid E)$	$\{y \mid y \in t \ \wedge \ \forall x \cdot (P \ \Rightarrow \ y \in E)\}$

1.2.4 Relations

Les relations sont un cas particulier de construction d'ensembles. Elles sont très utilisées dans les spécifications (invariants, propriétés, etc.). Il s'agit simplement d'ensembles de couples d'éléments. La définition est :

Symbole	Signification	Définition
$\leftrightarrow$	relation entre deux ensembles	$E_1 \leftrightarrow E_2 \stackrel{def}{=\!=} \mathbb{P}(E_1 \times E_2)$

On définit le *domaine* d'une relation comme les éléments du premier ensemble E_1 qui sont effectivement en relation avec des éléments du second ensemble E_2. Le *codomaine*[2] est l'ensemble des points du second ensemble E_2 qui sont en relation avec des éléments du premier ensemble E_1. L'*image* d'un ensemble par une relation, noté $r[F]$ est l'ensemble des éléments de E_2 qui sont en relation avec les éléments de F par la relation r. Plus formellement :

Condition	Expression	Définition
$r \in E_1 \leftrightarrow E_2$	$\mathsf{dom}(r)$	$\{x \mid x \in E_1 \ \wedge \ \exists y \cdot (y \in E_2 \ \wedge \ (x \mapsto y) \in r)\}$
$r \in E_1 \leftrightarrow E_2$	$\mathsf{ran}(r)$	$\{y \mid y \in E_2 \ \wedge \ \exists x \cdot (x \in E_1 \ \wedge \ (x \mapsto y) \in r)\}$
$r \in E_1 \leftrightarrow E_2$ et $F \subseteq E_1$	$r[F]$	$\{y \mid y \in E_2 \ \wedge \ \exists x \cdot (x \in F \ \wedge \ (x \mapsto y) \in r)\}$

Opérations sur les relations

[2] Traduit en anglais par *range*.

Il existe de nombreuses opérations sur les relations. Les opérations sur les ensembles s'appliquent évidemment aux relations (qui sont effectivement des ensembles de couples). En particulier, une relation vide est la même chose qu'un ensemble vide. Néanmoins, il y a quelques opérations spécifiques, dont la plupart sont très usuelles. On a la relation *identité* sur un ensemble, qui à chaque élément associe le même élément. La relation *inverse* d'une relation donnée. On distingue ensuite trois formes de *composition* de relations. Enfin, on a les opérations de *projection* qui construisent les relations entre les ensembles paramètres et les éléments projetés droite ou gauche.

Symbole	Signification	Syntaxe
id	relation identité	$\mathsf{id}(Ensemble)$
$^{-1}$	inverse d'une relation	$Relation^{-1}$
;	composition séquentielle	$Relation$; $Relation$
$\otimes$	produit direct	$Relation \otimes Relation$
$\parallel$	produit parallèle	$Relation \parallel Relation$
prj_1	première projection	$\mathsf{prj}_1(Ensemble, Ensemble)$
prj_2	deuxième projection	$\mathsf{prj}_2(Ensemble, Ensemble)$

Dans cette syntaxe, la métanotion *Relation* est une expression d'ensemble de "type" relation, c'est-à-dire de la forme $\mathbb{P}(E_1 \times E_2)$. Comme exemple de calcul sur les relations, on a :

Exemples : Soient $R_1 = \{(0 \mapsto 2), (1 \mapsto 5), (2 \mapsto 5), (2 \mapsto 7)\}$,
$R_2 = \{(0 \mapsto 0), (2 \mapsto -1), (5 \mapsto 8), (6 \mapsto 9)\}$,
$R_3 = \{(4 \mapsto 5), (3 \mapsto 2)\}$,
$R_4 = \{(11 \mapsto 12), (21 \mapsto 22)\}$

alors : $\mathsf{id}(\{1,2,3\}) = \{1 \mapsto 1), (2 \mapsto 2), (3 \mapsto 3)\}$
$R_1^{-1} = \{(2 \mapsto 0), (5 \mapsto 1), (5 \mapsto 2), (7 \mapsto 2)\}$
$R_1 \ ; \ R_2 = \{(0 \mapsto -1), (1 \mapsto 8), (2 \mapsto 8)\}$
$R_1 \otimes R_2 = \{(0 \mapsto (2,0)), (2 \mapsto (5,-1)), (2 \mapsto (7,-1))\}$
$R_3 \parallel R_4 = \{((4 \mapsto 11) \mapsto (5 \mapsto 12)), ((4 \mapsto 21) \mapsto (5 \mapsto 22)),$
$((3 \mapsto 11) \mapsto (2 \mapsto 12)), ((3 \mapsto 21) \mapsto (2 \mapsto 22))\}$

Les projections construisent à la fois la relation entre les éléments des ensembles paramètres et les éléments des projections :

$\mathsf{prj}_1(\{1,2\},\{3,4\}) = \{((1 \mapsto 3) \mapsto 1), ((1 \mapsto 4) \mapsto 1), ((2 \mapsto 3) \mapsto 2), ((2 \mapsto 4) \mapsto 2)\}$
$\mathsf{prj}_2(\{1,2\},\{3,4\}) = \{((1 \mapsto 3) \mapsto 3), ((1 \mapsto 4) \mapsto 4), ((2 \mapsto 3) \mapsto 3), ((2 \mapsto 4) \mapsto 4)\}$

Les définitions précises des opérations sont :

Condition	Expression	Définition
	$\mathsf{id}(E)$	$\{x, y \mid (x \mapsto y) \in E \times E \ \wedge \ x = y\}$
$r \in s \leftrightarrow t$	r^{-1}	$\{y, x \mid (y \mapsto x) \in t \times s \ \wedge \ (x \mapsto y) \in r\}$
$r_1 \in t \leftrightarrow u$ et $r_2 \in u \leftrightarrow v$	$r_1 \ ; \ r_2$	$\{x, z \mid x, z \in t \times v \wedge \exists y \cdot (y \in u \ \wedge \ (x \mapsto y) \in r_1 \ \wedge \ (y \mapsto z) \in r_2)\}$
$r_1 \in t \leftrightarrow u$ et $r_2 \in t \leftrightarrow v$	$r_1 \otimes r_2$	$\{x, (y, z) \mid x, (y, z) \in t \times (u \times v) \wedge (x \mapsto y) \in r_1 \ \wedge \ (x \mapsto z) \in r_2\}$
$r_1 \in t \leftrightarrow u$ et $r_2 \in v \leftrightarrow w$	$r_1 \parallel r_2$	$\{(x, z), (y, a) \mid (x, z), (y, a) \in (t \times v) \times (u \times w) \wedge (x \mapsto y) \in r_1 \ \wedge \ (z \mapsto a) \in r_2\}$
	$\mathsf{prj}_1(E, F)$	$\{x, y, z \mid x, y, z \in E \times F \times E \ \wedge \ z = x\}$
	$\mathsf{prj}_2(E, F)$	$\{x, y, z \mid x, y, z \in E \times F \times F \ \wedge \ z = y\}$

Itérations.

Il existe des opérations pour *itérer* une relation, c'est-à-dire la composer séquentiellement avec elle-même un certain nombre de fois. Dans le tableau suivant, on a $r \in s \leftrightarrow s$. Les notations d'itération sont :

Notation	Signification	Définition
r^n	itération n fois de r	$r^n \stackrel{def}{=\!=} r \ ; \ r^{n-1}$ si $n > 0$ $r^0 \stackrel{def}{=\!=} \mathsf{id}(s)$
r^+	fermeture transitive	$r^+ \stackrel{def}{=\!=} \bigcup n \cdot (n \in \mathbb{N}_1 \mid r^n)$
r^*	fermeture réflexive transitive	$r^* \stackrel{def}{=\!=} \bigcup n \cdot (n \in \mathbb{N} \mid r^n)$

Exemple : Si $R = \{(0 \mapsto 2), (1 \mapsto 0), (2 \mapsto 1)\}$, alors :

$$\begin{array}{lcl} R^2 & = & \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 0)\} \\ R^3 & = & \{(0 \mapsto 0), (1 \mapsto 1), (2 \mapsto 2)\} \\ R^4 & = & R \\ R^* & = & \{(0 \mapsto 0), (0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 0), (1 \mapsto 1), (1 \mapsto 2), (2 \mapsto 0), (2 \mapsto 1), (2 \mapsto 2)\} \end{array}$$

Restrictions.

Enfin, les derniers opérateurs sur les relations consistent à restreindre ces relations sur des sous-ensembles du domaine ou du codomaine. On a également l'opération de modification d'une relation (on dit aussi surcharge d'une relation[3]). Dans le tableau des définitions, on a : $R \in s \leftrightarrow t$, $E \subseteq s$, $F \subseteq t$ et $Q \in s \leftrightarrow t$.

Signification	Expression	Définition
restriction sur le domaine	$E \lhd R$	$\{x, y \mid (x \mapsto y) \in R \ \wedge \ x \in E\}$
soustraction sur le domaine	$E \lhd\!\!\!- R$	$\{x, y \mid (x \mapsto y) \in R \ \wedge \ x \notin E\}$
restriction sur le codomaine	$R \rhd F$	$\{x, y \mid (x \mapsto y) \in R \ \wedge \ y \in F\}$
soustraction sur le codomaine	$R \rhd\!\!\!- F$	$\{x, y \mid (x \mapsto y) \in R \ \wedge \ y \notin F\}$
modification ou surcharge	$R \lhd\!\!\!+ Q$	$\{x, y \mid (x \mapsto y) \in s \times t \wedge$ $(((x \mapsto y) \in R \ \wedge \ x \notin \mathsf{dom}(Q))$ $\vee \ (x \mapsto y) \in Q)\}$

Exemples : Soient les relations $R_1 = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$ et $R_2 = \{(3 \mapsto 3), (5 \mapsto 4)\}$ et les ensembles $E = \{1, 2, 3\}$ et $F = \{5, 7, 9\}$, alors :

$$
\begin{array}{lcl}
E \lhd R_1 & = & \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9)\} \\
E \lhd\!\!\!- R_1 & = & \{(4 \mapsto 7), (4 \mapsto 9)\} \\
R_1 \rhd F & = & \{(3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\} \\
R_1 \rhd\!\!\!- F & = & \{(2 \mapsto 1), (2 \mapsto 8)\} \\
R_1 \lhd\!\!\!+ R_2 & = & \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 3), (4 \mapsto 7), (4 \mapsto 9), (5 \mapsto 4)\}
\end{array}
$$

Exercices : Il y a de nombreuses propriétés intéressantes que l'on peut démontrer sur les relations. Avec $r \in s \leftrightarrow t$, $p \subseteq s$, $q \subseteq s$, $m \subseteq t$, $v \in t \leftrightarrow u$, et $w \in s \leftrightarrow t$, on a :

$$
\begin{array}{lcl}
\mathsf{ran}(r) & = & r[s] \\
\mathsf{dom}(r) & = & r^{-1}[t] \\
r[p] & = & \mathsf{ran}(p \lhd r) \\
r[p \ \cup \ q] & = & r[p] \ \cup \ r[q] \\
r[p \ \cap \ q] & \subseteq & r[p] \ \cap \ r[q] \\
 & & \\
p \subseteq q & \Rightarrow & r[p] \subseteq r[q] \\
r \subseteq w & \Rightarrow & r[p] \subseteq w[p] \\
 & & \\
r \ ; (m \times u) & = & r^{-1}[m] \times u
\end{array}
\qquad
\begin{array}{lcl}
p \lhd (r \ ; v) & = & (p \lhd r) \ ; v \\
p \lhd (q \lhd r) & = & (p \ \cap \ q) \lhd r \\
(r \rhd m) \ ; v & = & r \ ; (m \lhd v) \\
 & & \\
\mathsf{dom}(p \lhd r) & = & p \ \cap \ \mathsf{dom}(r) \\
\mathsf{ran}(r \rhd m) & = & \mathsf{ran}(r) \ \cap \ m \\
 & & \\
(r \ ; v)^{-1} & = & v^{-1} \ ; r^{-1} \\
(r \ \cup \ w) \ ; v & = & (r \ ; v) \ \cup \ (w \ ; v) \\
(r \ \cap \ w) \ ; v & \subseteq & (r \ ; v) \ \cap \ (w \ ; v)
\end{array}
$$

[3]En anglais *overriding*.

1.2.5 Fonctions

Les fonctions sont des relations dont chaque élément du domaine n'est associé qu'à un seul élément du codomaine. Les fonctions les plus générales sont les fonctions partielles, ensuite, on peut définir les fonctions totales, injectives, surjectives, etc.

Signification	Expression	Définition
fonctions partielles	$s \pfun t$	$\{r \mid r \in s \rel t \land \forall x, y, z \cdot (x, y \in r \land x, z \in r \Rightarrow y = z)\}$
fonctions totales	$s \tfun t$	$\{f \mid f \in s \pfun t \land \mathsf{dom}(f) = s\}$
injectives partielles	$s \pinj t$	$\{f \mid f \in s \pfun t \land f^{-1} \in t \pfun s\}$
injectives totales	$s \tinj t$	$s \pinj t \cap s \tfun t$
surjectives partielles	$s \psur t$	$\{f \mid f \in s \pfun t \land \mathsf{ran}(f) = t\}$
surjectives totales	$s \tsur t$	$s \psur t \cap s \tfun t$
bijectives partielles	$s \pbij t$	$s \pinj t \cap s \psur t$
bijectives totales	$s \tbij t$	$s \tinj t \cap s \tsur t$

Il y a plusieurs manières de construire des fonctions autrement que par l'énumération des éléments. Ce sont la "lambda-notation", la notation de fonction constante et la transformée d'une relation en une fonction. À l'inverse, on peut construire une relation à partir d'une fonction. Enfin, on a l'évaluation d'une fonction en un point, comme d'habitude :

Signification	Syntaxe
lambda-expression	$\lambda\, Id\_liste \cdot (Prédicat \mid Expression)$
fonction constante	$Expression \times \{Expression\}$
transformée en fonction	$\mathsf{fnc}(Expression)$
transformée en relation	$\mathsf{rel}(Expression)$
évaluation de fonction	$Expression(Expression)$

Exemples : La lambda-expression : $\lambda\, x \cdot (x \in \mathbb{N} \mid x + 2)$ est la fonction sur les entiers qui ajoute deux à l'argument. Si cette fonction est appelée f, alors $f(3) = 5$ (notation de l'évaluation de f au point $x = 3$). Si on a : $R_1 = \{(0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 1), (1 \mapsto 7), (2 \mapsto 3), \}$ et $R2 = \{(0 \mapsto \{0, 2\}), (1 \mapsto \{4, 6, 8\})\}$, alors :

$$\begin{array}{lcl} \mathsf{fnc}(R_1) & = & \{(0 \mapsto \{1, 2\}), (1 \mapsto \{1, 7\}), (2 \mapsto \{3\})\} \\ \mathsf{rel}(R_2) & = & \{(0 \mapsto 0), (0 \mapsto 2), (1 \mapsto 4), (1 \mapsto 6), (1 \mapsto 8)\} \end{array}$$

Pour terminer, on donne quelques définitions ou propriétés des fonctions. On suppose l'axiome suivant de définition de l'évaluation d'une fonction :

$$f \in s \mathrel{+\!\!\!\rightarrow} t \ \wedge\ x \in \mathsf{dom}(f) \ \Rightarrow\ x, f(x) \in f$$

Alors on peut démontrer que l'évaluation d'une expression "lambda" est obtenue par substitution (note : on peut avoir un prédicat plus complexe que la simple appartenance $x \in s$, qui indique ici le domaine de la fonction) :

$$\begin{array}{l} \forall\, x \cdot (x \in s \Rightarrow E \in t) \\ z \in s \\ \Rightarrow \\ \lambda\, x \cdot (x \in s \mid E)(z) = [x := z]\, E \end{array}$$

On a également :

$$s \times \{v\} = \lambda\, x \cdot (x \in s \mid v)$$

Si on a une relation $r \in s \leftrightarrow t$, alors $\mathsf{fnc}(r) \in s \mathrel{+\!\!\!\rightarrow} \mathbb{P}(t)$. Le passage d'une relation à une fonction est défini par :

$$r \in s \leftrightarrow t \ \Rightarrow\ \mathsf{fnc}(r) = \lambda\, x \cdot (x \in \mathsf{dom}(r) \mid r[\{x\}])$$

Si une fonction f est de la forme : $f \in s \mathrel{+\!\!\!\rightarrow} \mathbb{P}(t)$, alors $\mathsf{rel}(f) \in s \leftrightarrow t$ avec :

$$f \in s \mathrel{+\!\!\!\rightarrow} \mathbb{P}(t) \ \Rightarrow\ \mathsf{rel}(f) = \{x, y \mid x, y \in s \times t \ \wedge\ x \in \mathsf{dom}(f) \ \wedge\ y \in f(x)\}$$

Pour n'importe quelle fonction $f \in s \mathrel{+\!\!\!\rightarrow} t$, la fonction $\mathsf{fnc}(f^{-1}) \in t \mathrel{+\!\!\!\rightarrow} \mathbb{P}(s)$ réalise une partition du domaine de f (exercice), c'est-à-dire :

$$\forall\, y, y' \cdot (y, y' \in \mathsf{ran}(f) \times \mathsf{ran}(f) \ \wedge\ y \neq y' \Rightarrow \mathsf{fnc}(f^{-1})(y) \cap \mathsf{fnc}(f^{-1})(y') = \emptyset$$

et

$$\bigcup y \cdot (y \in \mathsf{ran}(f) \mid \mathsf{fnc}(f^{-1})(y)) = \mathsf{dom}(f)$$

Exercice : Si $r \in s \leftrightarrow s$, $p \subseteq s$ et $q \subseteq s$, alors les prédicats ci-dessous sont tous équivalents (en notant $\overline{a}$ le complémentaire de $a \subseteq s$ par rapport à s) :

$$\begin{array}{lll} \forall\, x, y \cdot (x \in p \ \wedge\ x, y \in r \ \Rightarrow\ y \in q) & & \\ \Leftrightarrow\ r[p] & \subseteq & q \\ \Leftrightarrow\ p \lhd r & \subseteq & r \rhd q \\ \Leftrightarrow\ p & \subseteq & \overline{r^{-1}[\overline{q}]} \\ \Leftrightarrow\ r & \subseteq & \overline{p} \times s \ \cup\ s \times q \end{array}$$

1.3 Types de données prédéfinis

Les constructeurs d'ensembles présentés au paragraphe précédent (1.2) sont "génériques" en ce sens qu'on ne fixe pas le type des éléments qu'ils contiennent (de même pour les relations ou les fonctions). Il faut donc pouvoir définir et utiliser des éléments concrets (ou des valeurs de types concrets), avec lesquels on pourra modéliser une application. En B, on peut introduire des ensembles simplement par leur nom au moment de leur première déclaration. On ne précise pas leurs éléments, mais ils sont supposés non vides. Ils sont appelés des "ensembles différés" ou ensembles abstraits. Leur véritable contenu ne sera connu qu'à la phase d'implémentation. D'autres ensembles peuvent être définis avec leurs éléments sous forme d'énumération. Ils sont appelés "ensembles définis". D'autre part, on peut construire des expressions d'ensembles avec des types effectifs que l'on décrit dans le reste de ce paragraphe.

1.3.1 Les constantes symboliques et les booléens

On a vu que l'on pouvait définir un ensemble par énumération. Les valeurs des éléments sont des constantes symboliques (identificateurs), comme dans les types énumérés du langage Ada. Il n'y a pas d'opération prédéfinie sur ces valeurs, excepté le prédicat d'égalité (et de non égalité). Par exemple, on peut avoir les déclarations :

$$COULEUR = \{bleu, blanc, rouge, vert, jaune\}$$
$$CARTE = \{Trefle, Carreau, Coeur, Pique\}$$

Les booléens sont un cas particulier de type énuméré. Il est prédéfini par :

$$\mathsf{BOOL} = \{\mathsf{FALSE}, \mathsf{TRUE}\}$$

Il ne faut pas confondre les valeurs de cet ensemble des booléens qui font partie de la catégorie des *Expressions*, avec les *Prédicats*, qui constituent une autre catégorie syntaxique (cf. 1.1). Néanmoins, il est possible de convertir explicitement un prédicat en une valeur boolénne par l'opérateur prédéfini ci-dessous :

Symbole	Signification	Syntaxe
bool	conversion d'un prédicat en valeur booléenne	bool(*Prédicat*)

Exemple : L'expression :

$$\mathsf{bool}(\exists\, x \cdot (x \in \mathbb{N} \ \wedge \ x = x * 2))$$

est une expression bien formée qui a la valeur TRUE.

1.3.2 Les nombres entiers

Les nombres entiers sont les valeurs numériques habituelles avec laquelle on peut effectuer des calculs arithmétiques. Ils englobent les entiers relatifs (positifs et négatifs). Le type

le plus général est noté $\mathbb{Z}$ et il représente l'ensemble mathématique des entiers relatifs. On peut dénoter les valeurs d'entiers avec la syntaxe habituelle : $0, 1, \ldots, 342, \ldots, -4, \ldots$ etc. Deux sous-ensembles usuels sont prédéfinis : $\mathbb{N}$ les entiers positifs, et $\mathbb{N}_1$ les entiers strictement positifs. Il y a deux constantes qui jouent un rôle très important, ce sont maxint et minint. Elles représentent respectivement le plus grand et le plus petit entier représentable dans une machine donnée. À l'aide de ces constantes, on définit le sous-ensemble des entiers représentables, celui des positifs et des strictement positifs représentables. On a aussi la notation d'intervalle d'entiers, qui constitue une construction particulière d'ensemble d'entiers.

Il faut savoir que dans les machines B, on peut utiliser des objets infinis au moment des spécifications de haut niveau, mais qu'une "implémentation" ne peut mettre en œuvre que des objets finis et représentables en machine. Les contraintes de limites ou de bornes sont toujours décrites explicitement au plus tard au moment des implémentations, pour pouvoir vérifier que le programme implémenté final sera correct par rapport à ces contraintes (preuves de non débordement). Comme l'objectif est de développer des logiciels sécuritaires, on évite ainsi les erreurs de débordement à l'exécution[4].

D'une façon plus générale, l'implémentabilité des machines B est une preuve de l'existence d'un modèle satisfaisant la spécification, autrement dit une preuve de la *faisabilité* de la spécification abstraite[5].

En résumé, les ensembles et constantes prédéfinis pour les valeurs numériques sont :

Symbole	Signification	Définition
$\mathbb{Z}$	ensemble des entiers relatifs	
$\mathbb{N}$	ensemble des entiers positifs	$\{x \mid x \in \mathbb{Z} \ \wedge \ x \geq 0\}$
$\mathbb{N}_1$	entiers strictement positifs	$\{x \mid x \in \mathbb{Z} \ \wedge \ x > 0\}$
$n..m$	ensemble d'entiers entre n et m	$\{x \mid x \in \mathbb{Z} \ \wedge \ n \leq x \ \wedge \ x \leq m\}$
maxint	entier maximum représentable	maxint $\in \mathbb{N}_1$
minint	entier minimum représentable	minint $\in \mathbb{Z}$
INT	entiers relatifs représentables	minint..maxint
NAT	entiers positifs représentables	0..maxint
NAT_1	entiers strict. pos. représentables	1..maxint

[4]On sait que l'echec de la fusée Ariane 5 est dû à un débordement dans une conversion arithmétique dont l'exception n'était pas traitée.

[5]Cela est à comparer avec VDM, par exemple, où les preuves existentielles font partie des obligations de preuve. En B, de telles preuves ne sont pas faites initialement, mais sont résolues constructivement par la possibilité de trouver une implémentation.

Prédicats de comparaison.

Sur toutes les valeurs entières on a le prédicat d'égalité (déjà cité) et les prédicats de comparaison usuels. Ce sont :

Symbole	Signification
$\leq$	inférieur ou égal
$<$	strictement inférieur
$\geq$	supérieur ou égal
$>$	strictement supérieur

Les propriétés d'ordre total (réflexivité, anti-symétrie, transitivité) sont vraies pour sur $\leq$ (ordre croissant) et $\geq$ (ordre décroissant).

Opérations sur les entiers.

On a les opérations arithmétiques usuelles sur les entiers $\mathbb{Z}$ ou des sous-ensembles dans le cas de certains opérateurs :

Symbole	Typage	Signification
succ	$\mathbb{Z} \to \mathbb{Z}$	fonction successeur
pred	$\mathbb{Z} \to \mathbb{Z}$	fonction prédécesseur
$-$	$\mathbb{Z} \to \mathbb{Z}$	moins unaire
$+$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	addition
$-$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	soustraction ou différence
$*$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	multiplication ou produit
$/$	$\mathbb{Z} \times (\mathbb{Z} - \{0\}) \to \mathbb{Z}$	quotient de la division entière
mod	$\mathbb{Z} \times \mathbb{N}_1 \to \mathbb{Z}$	reste de la division entière
x^y	$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$	opération puissance entière

Il y a des opérateurs de somme et produit généralisés :

Symbole	Signification	Syntaxe
Σ	somme d'expressions quantifiées	$\Sigma\, Id\_liste \cdot (Prédicat \mid Expression)$
Π	produit d'expressions quantifiées	$\Pi\, Id\_liste \cdot (Prédicat \mid Expression)$

La partie *Expression* doit être de type entier lorsque les variables liées satisfont le *Prédicat*.

Exemples : Avec $E_1 = \{1, 2, 3, 4\}$, on a :

$$\Sigma\, x \cdot (x \in E_1 \mid x * 2) = (1 * 2) + (2 * 2) + (3 * 2) + (4 * 2) = 20$$
$$\Pi\, x \cdot (x \in E_1 \mid x) = 1 * 2 * 3 * 4 = 24$$

Il y a enfin des opérateurs à résultat entier sur des ensembles :

Symbole	Signification	Syntaxe
card	nombre d'éléments d'un ensemble quelconque	card($Expression$)
min	minimum d'un ensemble fini non vide d'entiers	min($Expression$)
max	maximum d'un ensemble fini non vide d'entiers	max($Expression$)

Exemples : avec $E_2 = \{-1, 2, 9, -4\}$, on a :

$$\mathsf{max}(E_2) = 9, \quad \mathsf{min}(E_2) = -4, \quad \mathsf{card}(E_2) = 4$$

1.3.3 Les chaînes de caractères

Il existe deux ensembles prédéfinis :

CHAR
STRING

qui dénotent respectivement l'ensemble des caractères et celui des chaînes de caractères. Les valeurs de ces ensembles sont utilisables surtout dans les entrées/sorties sur fichier et écran pour la mise en page des résultats. Les opérations sur ces ensembles sont définies dans les machines abstraites de base qui exportent ces déclarations (cf. chapitre 5). Les constantes STRING sont écrites entre guillemets : ". Les constantes CHAR sont un ensemble énuméré isomorphe à l'intervalle 0..255.

Exemple : "Voici une chaine &(1,2(* de caractères".

1.3.4 Les séquences

Les séquences (ou suites) sont un cas particulier de constructeurs de fonctions dont le domaine est un intervalle d'entiers et dont le codomaine est l'ensemble support des éléments des séquences considérées. De ce fait, les séquences héritent de toutes les opérations définies sur les relations et les fonctions. Le résultat de ces opérations est une séquence si les opérations préservent les conditions propres au domaine des séquences. Comme le domaine des séquences est fini, on caractérise uniquement les *séquences finies* d'éléments. Les ensembles prédéfinis de séquences sont :

Notation	Signification	Définition
$\mathsf{seq}(E)$	séquences finies d'éléments de E	$\bigcup n \cdot (n \in \mathbb{N} \mid 1..n \rightarrow E)$
$\mathsf{seq}_1(E)$	séquences non vides	$\mathsf{seq}(E) - \emptyset$
$\mathsf{iseq}(E)$	séquences injectives	$\mathsf{seq}(E) \ \cap \ (\mathbb{N}_1 \rightarrowtail E)$
$\mathsf{iseq}_1(E)$	séquences injectives non vides	$\mathsf{iseq}(E) - \emptyset$
$\mathsf{perm}(E)$	séquences bijectives ou permutations	$\mathsf{seq}(E) \ \cap \ (\mathbb{N}_1 \rightarrowtail\!\!\!\!\!\rightarrow E)$

Les séquences injectives sont aussi les séquences sans répétition d'éléments. La longueur d'une séquence bijective ($\mathsf{perm}(E)$) est égale au cardinal de l'ensemble E autrement dit, c'est la mise en séquence des éléments de l'ensemble E. On a la propriété bien connue[6] :

$$\mathsf{card}(\mathsf{perm}(E)) = \mathsf{card}(E)\,!$$

Les séquences ont une dénotation syntaxique prédéfinie :

$[\,]$	représente une séquence vide d'éléments de E,
$[3, 7, 5]$	est la séquence des trois entiers 3, 7 et 5.

D'après la définition de seq, on a :

$$\begin{array}{rcl} [\,] & \stackrel{def}{=\!=} & 1..0 \rightarrow E \qquad (= \emptyset) \\ [3, 7, 5] & \stackrel{def}{=\!=} & \{1 \mapsto 3, 2 \mapsto 7, 3 \mapsto 5\} \end{array}$$

Il y a un certain nombre d'opérateurs usuels sur les séquences. Ils sont définis récursivement à partir de la séquence vide $[\,]$ et de l'opérateur d'ajout d'un élément à gauche qui est noté : $a \mathrel{->} s$ avec $a \in E$ et $s \in \mathsf{seq}(E)$. L'opérateur $\mathrel{->}$ est défini de façon primitive par :

$$\begin{array}{rcl} & a \mathrel{->} [\,] & \stackrel{def}{=\!=} \quad \{1 \mapsto a\} \\ s \neq [\,] \ \Rightarrow & a \mathrel{->} s & \stackrel{def}{=\!=} \quad \{1 \mapsto a\} \ \cup \ (\mathsf{pred} \ ; \ s) \end{array}$$

[6] D'après la formation des séquences, $\mathsf{card}(E)$ est forcément fini.

Il est clair que toute séquence peut être construite uniquement à l'aide de $[\,]$ et $\mathrel{->}$:

$$[3,7,5] \;=\; (3 \mathrel{->} (7 \mathrel{->} (5 \mathrel{->} [\,])))$$

On donne ci-après les opérateurs sur les séquences définis récursivement par cas. L'appartenance des variables est : $a \in E$, $b \in E$, $s \in \mathsf{seq}(E)$, $t \in \mathsf{seq}(E)$ et $u \in \mathsf{seq}(\mathsf{seq}(E))$.

Notation	Signification	Définition
size	longueur de séquence	$\mathsf{size}([\,]) \;=\; 0$ $\mathsf{size}(a \mathrel{->} s) \;=\; \mathsf{size}(s) + 1$
$\frown$	concaténation	$[\,] \frown t \;=\; t$ $(a \mathrel{->} s) \frown t \;=\; a \mathrel{->} (s \frown t)$
$\mathrel{<-}$	ajout à droite	$[\,] \mathrel{<-} b \;=\; b \mathrel{->} [\,]$ $(a \mathrel{->} s) \mathrel{<-} b \;=\; a \mathrel{->} (s \mathrel{<-} b)$
rev	inversion de séquence	$\mathsf{rev}([\,]) \;=\; [\,]$ $\mathsf{rev}(a \mathrel{->} s) \;=\; \mathsf{rev}(s) \mathrel{<-} a$
conc	concaténation généralisée	$\mathsf{conc}([\,]) \;=\; [\,]$ $\mathsf{conc}(s \mathrel{->} u) \;=\; s \frown \mathsf{conc}(u)$

Exemples : $\mathsf{size}([3,7,5,6]) \;=\; 4$
$[3,7,5] \frown [4,8] \;=\; [3,7,5,4,8]$
$\mathsf{rev}([3,7,5,4]) \;=\; [4,5,7,3]$
$\mathsf{conc}([[2,3],[\,],[5,7,8],[0]]) \;=\; [2,3,5,7,8,0]$

D'autres opérateurs permettent de sélectionner des sous-parties d'une séquence donnée. Ce sont la restriction à partir du bas, notée $s \uparrow n$ qui signifie que l'on garde les n premiers éléments de s, et la restriction à partir du haut notée $s \downarrow n$, qui signifie que l'on supprime les n premiers éléments :

Condition	Expression	Définition
$n \in 0..\mathsf{size}(s)$	$s \uparrow n$	$(1..n) \lhd s$
$n \in 0..\mathsf{size}(s)$	$s \downarrow n$	$\lambda\, i \cdot (i \in 1..(\mathsf{size}(s) - n) \mid s(n+i))$

Enfin, lorsque la séquence s en paramètre est non vide ($s \neq [\,]$), on a les opérateurs suivants :

Signification	Expression	Définition
premier élément	$\mathsf{first}(s)$	$s(1)$
dernier élément	$\mathsf{last}(s)$	$s(\mathsf{size}(s))$
éléments sauf le premier	$\mathsf{tail}(s)$	$s \downarrow 1$
éléments sauf le dernier	$\mathsf{front}(s)$	$s \uparrow (\mathsf{size}(s) - 1)$

Exemples : $\mathsf{first}([4,5,7,3]) = 4$
$\mathsf{last}([4,5,7,3]) = 3$
$\mathsf{tail}([4,5,7,3]) = [5,7,3]$
$\mathsf{front}([4,5,7,3]) = [4,5,7]$

Exercices : Avec $s \in \mathsf{seq}(E)$, $t \in \mathsf{seq}(E)$, $u \in \mathsf{seq}(\mathsf{seq}(E))$, $v \in \mathsf{seq}(\mathsf{seq}(E))$ et en supposant que les opérations sont bien définies, démontrer les propriétés :

$$\begin{array}{lcl} \mathsf{size}(s) & = & \mathsf{card}(\mathsf{dom}(s)) \\ \mathsf{size}(s \frown t) & = & \mathsf{size}(s) + \mathsf{size}(t) \\ \mathsf{rev}(s \frown t) & = & \mathsf{rev}(t) \frown \mathsf{rev}(s) \\ \mathsf{conc}(u \frown v) & = & \mathsf{conc}(u) \frown \mathsf{conc}(v) \end{array}$$

$$\begin{array}{lcl} (s \uparrow n) \frown (s \downarrow n) & = & s \\ \mathsf{first}(s) \rightarrow \mathsf{tail}(s) & = & s \\ \mathsf{front}(s) \leftarrow \mathsf{last}(s) & = & s \end{array}$$

Chapitre 2

Substitutions généralisées

Résumé

Ce chapitre décrit ce que sont les substitutions généralisées dans la méthode B. Il donne les bases mathématiques de ces notions. Il définit des prédicats utiles dans l'analyse des spécifications et donne une interprétation ensembliste des substitutions logiques. Ce chapitre permet de relier la notation B à des formalismes de spécifications ensemblistes comme Z.

2.1 De l'axiomatisation à la construction des programmes

Le but de la programmation est de transformer les idées intuitives que l'on a sur les fonctionnalités d'un logiciel en une suite d'instructions pour l'ordinateur, autrement dit à transformer une *connaissance* en un *programme*, ou de l'abstrait en du concret[1]. À l'inverse, on a longtemps vu la preuve de programmes comme le fait de donner une signification à une suite d'instructions[2]. Néanmoins, les deux points de vue (construction et preuve) sont intimement reliés. Dans une assertion de la logique de C. A. R. Hoare[3] (on dit aussi, logique des programmes), on trouve des "formules" de la forme :

$$P\{S\}R$$

qui signifient que si l'assertion (le prédicat) P est vraie, et si l'instruction S se termine, alors l'assertion R est vraie après l'exécution de S. Les assertions portent sur les variables du programme. Dans ces formules, P est appelée la *précondition* et R la *postcondition* de l'instruction S. De telles règles ne sont pas constructives du point de vue de la programmation, car on a l'inférence :

$$\frac{P' \Rightarrow P, \quad P\{S\}R, \quad R \Rightarrow R'}{P'\{S\}R'}$$

ce qui veut dire que l'on peut toujours renforcer une précondition et affaiblir une postcondition, sans changer la validité d'une formule de logique des programmes[4]. Un autre

[1] J.-R. Abrial, *Spécifier ou comment matérialiser l'abstrait*, TSI, 3(3), pp. 201-219. 1984.

[2] R. W. Floyd, *Assigning meaning to programs,* In Mathematical Aspects of Computer Science, J. T. Schwartz (ed.), AMS, pp. 19–32, 1967.

[3] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming.* In Comm. of the ACM, Vol 12, pp. 576–583, 1969.

[4] L'implication induit une relation d'ordre dans les prédicats, dans le sens où, si $P \Rightarrow Q$, alors, on peut dire que P est "plus fort" que Q.

point important est que les formules de Hoare ne traitent que de la correction partielle, car elles ne disent rien sur la terminaison des programmes. Or, prouver la terminaison, au moins pour les programmes de calcul, est essentiel. De plus, ce qui est important (et ce que l'on connaît généralement), c'est la postcondition, puisque c'est ce que le programme doit finalement satisfaire. La postcondition peut aussi être ce que le programme doit préserver, si l'on s'intéresse à des propriétés invariantes de l'état du programme. D'où l'idée, due à E. W. Dijkstra[5], de déterminer, pour une postcondition R et une instruction donnée S, quelle est la *plus faible précondition* P qui assure que S termine et que R est vraie après S. Il note cela : $wp_S(R)$ (pour *weakest precondition*) ou encore $wp(S, R)$. Dans le cas où on a pu prouver que S termine, une formule de Hoare de la forme $P\{S\}R$ est équivalente à l'assertion $P \Rightarrow wp_S(R)$.

Pour chaque instruction (ou suite d'instructions) particulière S, l'opérateur wp_S est un *transformateur de prédicat* puisque, appliqué à un prédicat, il retourne un autre prédicat. Il est ainsi possible de définir la sémantique des constructions de programme par des compositions d'opérateurs wp et des expressions de prédicats. Par exemple, pour le séquencement d'instruction, on a :

$$wp_{(S_1 \ ; \ S_2)} \stackrel{def}{=\!=} wp_{S_1} \circ wp_{S_2}$$

qui signifie que la plus faible précondition de S_1 ; S_2 revient à calculer la plus faible précondition de S_2, puis à calculer celle de S_1 à partir du résultat. Rappelons qu'une plus faible précondition part d'une postcondition et construit une précondition, en "remontant" dans le programme.

Dans cette démarche, la spécification de l'instruction $x := E$ des langages de programmation est donc le transformateur de prédicat noté : $wp_{x:=E}$. Il a été montré (c'est d'ailleurs l'axiome d'affectation de la logique de Hoare) que ce transformateur de prédicat est exactement la substitution de x par E dans le prédicat postcondition :

$$wp_{x:=E}(R) \ \Leftrightarrow \ [x := E]\, R$$

en prenant les notations du paragraphe 1.1 pour les substitutions. Considérant cette équivalence, la méthode B a choisi d'utiliser un langage proche des notations informatiques (par exemple "$x := E$"), dont le sens est donné par la "substitution" de cette notation dans un prédicat. C'est donc un langage pour la spécification fondé sur la logique. Les autres constructions de ce langage sont aussi analogues à des substitutions : c'est ce que l'on appelle le langage des *substitutions généralisées*. Dans la suite de ce chapitre, on décrit ce langage, avec sa forme mathématique et sa forme textuelle, ainsi que les définitions et les calculs liés à la théorie des substitutions généralisées.

2.2 Langage des substitutions généralisées

Les substitutions primitives sont des substitutions généralisées à partir desquelles on peut construire toutes les autres substitutions. Elles ont une notation concise qui facilite les formules et les calculs. Dans les sections suivantes, x, y, z sont des variables, E, F, V sont des expressions, S, T, U, W sont des substitutions généralisées et I, J, P, Q, R sont des prédicats.

[5] E. W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, In Comm. of the ACM, Vol 18, pp. 453–457, 1975.

2.2.1 Liste des substitutions primitives

$x := E$	substitution simple
$x, y := E, F$	substitution multiple simple
skip	substitution sans effet
$P \mid S$	substitution préconditionnée
$P \Longrightarrow S$	substitution gardée
$S \parallel T$	substitution de choix borné
$@z \cdot S$	substitution de choix non borné
$S \,;\, T$	séquencement de substitutions
$\mathcal{W}(P, S, J, V)$	substitution d'itération

2.2.2 Plus faible précondition des substitutions primitives

Pour chaque substitution, on calcule la plus faible précondition qui assure que la substitution termine et que l'état final satisfait la postcondition R (paragraphe 2.1). Le cas de base pour la substitution simple $[x := E]\, R$ est la substitution uniforme de x par E dans R. Pour une substitution multiple simple, il s'agit de substitutions uniformes "simultanées" des variables par les expressions associées. (paragraphe 1.1). La plus faible précondition des autres substitutions est calculée par induction sur la structure de la substitution. La substitution d'itération $\mathcal{W}$ est traitée au paragraphe 2.2.3.

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	
$[P \mid S]\, R$	$P \ \wedge \ [S]\, R$	
$[P \Longrightarrow S]\, R$	$P \Rightarrow [S]\, R$	
$[S \parallel T]\, R$	$[S]R \ \wedge \ [T]\, R$	
$[@z \cdot S]\, R$	$\forall z \cdot [S]\, R$	$z \backslash R$
$[S \,;\, T]\, R$	$[S]\,([T]\, R)$	

Exemples :

$$
\begin{array}{lll}
[x := x-1 \parallel x := x+1]\,(1 \le x \le 10) & \Leftrightarrow & 2 \le x \le 11 \ \wedge \ 0 \le x \le 9 \\
& \Leftrightarrow & 2 \le x \le 9 \\
\\
[x > 0 \mid x := x-1]\,(-1 \le x \le 1) & \Leftrightarrow & x > 0 \ \wedge \ 0 \le x \le 2 \\
& \Leftrightarrow & 0 < x \le 2 \\
\\
[@z \cdot (z > 0 \Longrightarrow x := x+z)]\,(x \ge 3) & \Leftrightarrow & \forall z \cdot (z > 0 \Rightarrow [x := x+z]\,(x \ge 3)) \\
& \Leftrightarrow & \forall z \cdot (z > 0 \Rightarrow x + z \ge 3)
\end{array}
$$

2.2.3 Plus faible précondition de la substitution d'itération

La substitution d'itération $\mathcal{W}(P, S, J, V)$ signifie "tant que le prédicat P est vrai, faire la substitution S". Le prédicat J est appelé l'*invariant de la boucle*. Il doit être vrai à l'entrée de la boucle et se maintenir vrai tant que la boucle "s'exécute". La boucle termine quand P devient faux. L'expression V est le *variant de la boucle*. Il s'agit d'une expression entière positive qui décroît strictement à chaque exécution de la boucle. Le variant assure donc que la boucle termine.

La plus faible précondition qui assure que $\mathcal{W}(P, S, J, V)$ termine dans un état qui satisfait la postcondition R contient toutes ces informations sur l'invariance de J et la décroissance de V. Elle assure également que, à la sortie de boucle, R est vrai. D'où la formule, où les variables x sont les variables utilisées dans la boucle et les divers prédicats ou expression :

$$
\begin{array}{lll}
[\,\mathcal{W}(P,S,J,V)\,]\,R & & \text{Commentaires :} \\
\Leftrightarrow & & \\
 & J \ \wedge & \text{L'invariant est une précondition ;} \\
\forall x \cdot ((J \ \wedge \ P) & \Rightarrow \ [S]\,J) \ \wedge & \text{il est conservé dans la boucle.} \\
\forall x \cdot (J & \Rightarrow \ V \in \mathbb{N}) \ \wedge & \text{Le variant est un entier naturel} \\
\forall x \cdot ((J \ \wedge \ P) & \Rightarrow \ [n := V][S](V < n)) \ \wedge & \text{qui décroît à chaque pas.} \\
\forall x \cdot ((J \ \wedge \ \neg P) & \Rightarrow \ R) & \text{La sortie de boucle implique } R.
\end{array}
$$

Pratiquement, il y a une initialisation de variables avant de rentrer dans une boucle. La boucle est donc précédée d'une autre substitution qui produit un état qui satisfait l'invariant de la boucle. La forme générale de "boucle avec initialisation" et obligations de preuves est :

$$
\begin{array}{ll}
[T]\,J \ \wedge & \text{l'initialisation établit } J \\
\forall x \cdot (J \ \Rightarrow \ [\,\mathcal{W}(P,S,J,V)\,]\,R) & J \text{ assure la terminaison de la boucle dans } R \\
\Rightarrow & \\
[\,T \ ; \mathcal{W}(P,S,J,V)\,]\,R &
\end{array}
$$

Cela revient à remplacer l'obligation de preuve J par $[T]\,J$ dans la liste des obligations de preuve de la substitution $\mathcal{W}$.

2.2.4 Règles d'équivalence du séquencement

Le séquencement de substitutions peut être éliminé d'une substitution généralisée complexe qui le contient. Dans le cas où il n'y a pas de substitution d'itération, les règles qui permettent d'éliminer le " ; " sont données dans le tableau suivant. On remarque que ces règles ne sont pas symétriques puisque le séquencement induit un ordre. On verra au paragraphe 2.3.4 que, même en présence de substitution d'itération, le séquencement peut être éliminé d'une substitution donnée. L'égalité de deux substitutions $S = T$ peut être prouvée, au second ordre, en montrant que les deux substitutions ont le même effet sur tout prédicat, c'est-à-dire $[S]\,R \Leftrightarrow [T]\,R$, pour tout R (voir encore ce même paragraphe 2.3.4).

Equivalence			Condition
$\mathsf{skip}\ ;\ S$	$=$	S	
$(P \mid S)\ ;\ T$	$=$	$P \mid (S\ ;\ T)$	
$(P \Longrightarrow S)\ ;\ T$	$=$	$P \Longrightarrow (S\ ;\ T)$	
$(S \parallel T)\ ;\ U$	$=$	$(S\ ;\ U) \parallel (T\ ;\ U)$	
$(@z \cdot S)\ ;\ T$	$=$	$@z \cdot (S\ ;\ T)$	$z \backslash T$
$(S\ ;\ T)\ ;\ U$	$=$	$S\ ;\ (T\ ;\ U)$	
$S\ ;\ \mathsf{skip}$	$=$	S	
$S\ ;\ (P \mid T)$	$=$	$[S]\,P \mid (S\ ;\ T)$	
$(x := E)\ ;\ (P \Longrightarrow T)$	$=$	$[x := E]\,P \Longrightarrow (x := E\ ;\ T)$	
$S\ ;\ (T \parallel U)$	$=$	$(S\ ;\ T) \parallel (S\ ;\ U)$	
$S\ ;\ @z \cdot T$	$=$	$@z \cdot (S\ ;\ T)$	$z \backslash S$

2.2.5 Substitution multiple généralisée

Il existe une notation pour "mettre ensemble" plusieurs substitutions. C'est la *substitution multiple généralisée*, notée "$S \mid\mid T$". Elle signifie "faire simultanément S et T". Elle n'est définie que si les variables modifiées par S et celles modifiées par T sont disjointes. Moyennant cette restriction, une substitution construite avec $\mid\mid$ peut se ramener à une substitution généralisée primitive par les règles ci-après. Le prédicat $\mathsf{trm}(S)$, qui signifie "terminaison de S", est expliqué au paragraphe 2.3.1.

Equivalence			Condition
$x := E \mid\mid y := F$	$=$	$x, y := E, F$	
$\mathsf{skip} \mid\mid S$	$=$	S	
$(P \mid T) \mid\mid S$	$=$	$P \mid (T \mid\mid S)$	
$(P \Longrightarrow T) \mid\mid S$	$=$	$\mathsf{trm}(S) \mid P \Longrightarrow (T \mid\mid S)$	
$(T \,[\!]\, U) \mid\mid S$	$=$	$(T \mid\mid S) \,[\!]\, (U \mid\mid S)$	
$(@z \cdot T) \mid\mid S$	$=$	$@z \cdot (T \mid\mid S)$	$z \backslash S$
$T \mid\mid S$	$=$	$S \mid\mid T$	

2.2.6 Définition des substitutions verbeuses

Les notations mathématiques ne peuvent pas être utilisées dans les textes des programmes B. A leur place, on utilise des notations verbeuses qui facilitent la lecture et la mise en page. L'équivalent verbeux des substitutions primitives est (celles qui ne sont pas mentionnées s'utilisent telles quelles) :

Substitution	Notation
$P \mid S$	PRE P THEN S END
$P \Longrightarrow S$	SELECT P THEN S END
$S \,[\!]\, T$	CHOICE S OR T END
$@z \cdot S$	VAR z IN S END
$\mathcal{W}(P, S, J, V)$	WHILE P DO S INVARIANT J VARIANT V END

De plus, de nombreuses substitutions dérivées sont des moyens d'écrire facilement une composition complexe de substitutions primitives (par exemple, la substitution IF). La substitution ASSERT assure que le prédicat P est vrai avant de faire S. Il devient alors une hypothèse utilisable dans S. La substitution ANY est très utile car elle permet d'introduire de nouvelles variables z qui doivent satisfaire le prédicat P pour faire S. Plusieurs jeux de valeurs de variables peuvent convenir, il s'agit donc d'une substitution non déterministe. On trouve aussi des abréviations bien pratiques : "$x :\in E$" signifie x prend une valeur quelconque dans l'ensemble E et "$x : P$" (substitution "devient tel que") signifie x prend une valeur qui satisfait le prédicat P.

Notation	Définition
BEGIN S END	S
ASSERT P THEN S END	$P \mid P \Longrightarrow S$
IF P THEN S ELSE T END	$P \Longrightarrow S \parallel \neg\, P \Longrightarrow T$
CHOICE S OR $T \ldots$ OR U END	$S \parallel T \parallel \cdots \parallel U$
ANY z WHERE P THEN S END	$@z \cdot (P \Longrightarrow S)$
LET $x, \ldots, y$ BE $x = E \ \wedge \ldots \wedge y = F$ IN S END	ANY $x, \ldots, y$ WHERE $x = E \ \wedge \ldots \wedge y = F$ THEN S END
$x :\in E$	ANY x' WHERE $x' \in E$ THEN $x := x'$ END
$x : P$	ANY x' WHERE $[x\$0, x := x, x']P$ THEN $x := x'$ END
$x := \mathsf{bool}(P)$	IF P THEN $x :=$ TRUE ELSE $x :=$ FALSE END
$f(x) := E$	$f := f \Leftarrow \{x \mapsto E\}$

Dans la cas de la substitution "devient tel que" $(x : P)$, si l'on veut caractériser la nouvelle valeur x par rapport à sa valeur avant la substitution, on utilise la notation $x\$0$. Par exemple, la substitution :

$$x \ : \ x > x\$0$$

signifie que la nouvelle valeur de x doit être strictement supérieure à son ancienne valeur. Sous forme de choix non borné, l'équivalent de cet exemple de substitution est :

$$\text{ANY } x' \text{ WHERE } x' > x \text{ THEN } x := x' \text{ END}$$

La notation $x\$0$ peut aussi être utilisée dans un invariant ou un variant de boucle pour parler de la valeur de la variable x avant l'entrée dans la boucle (cf. paragraphe 5.2.3, figure 5.4).

2.2.7 Variations sur la substitution de choix borné

Dans ce paragraphe, on donne les autres équivalences de notation définies dans le langage B.

Notation	Définition

IF P THEN
S
END
	IF P THEN
	S
	ELSE
	skip
	END

IF P THEN
S
ELSIF Q THEN
T
ELSE
U
END
	IF P THEN
	S
	ELSE
	IF Q THEN
	T
	ELSE
	U
	END
	END

IF P THEN
S
ELSIF Q THEN
T
END
	IF P THEN
	S
	ELSIF Q THEN
	T
	ELSE
	skip
	END

SELECT P THEN S WHEN Q THEN T ... WHEN R THEN U END	CHOICE $P \Longrightarrow S$ OR $Q \Longrightarrow T$ ... OR $R \Longrightarrow U$ END

Notation (suite)	Définition
SELECT P THEN S WHEN Q THEN T ... WHEN R THEN U ELSE W END	SELECT P THEN S WHEN Q THEN T ... WHEN R THEN U WHEN $\neg\ (P\ \vee\ Q \ldots R)$ THEN W END
CASE E OF EITHER l THEN S OR p THEN T ... OR q THEN U ELSE W END END	SELECT $E \in \{l\}$ THEN S WHEN $E \in \{p\}$ THEN T ... WHEN $E \in \{q\}$ THEN U ELSE W END
CASE E OF EITHER l THEN S OR p THEN T ... OR q THEN U END END	SELECT $E \in \{l\}$ THEN S WHEN $E \in \{p\}$ THEN T ... WHEN $E \in \{q\}$ THEN U END

2.3 Calcul dans les substitutions généralisées

2.3.1 Terminaison

On caractérise le fait qu'une substitution S "termine" par un prédicat noté $\mathsf{trm}(S)$. On donne ci-après la définition théorique du prédicat trm. Le prédicat btrue est la constante de prédicat "vrai", alors que bfalse est la constante prédicat "faux".

Notation	Définition
$\mathsf{trm}(S)$	$[S]\,\mathsf{btrue}$

On calcule facilement les résultats de terminaison des substitutions primitives et de quelques formes usuelles. Le prédicat prd_x est défini au paragraphe 2.3.3.

$$
\begin{array}{lcl}
\mathsf{trm}(x := E) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{trm}(\mathsf{skip}) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{trm}(P \mid S) & \Leftrightarrow & P \ \wedge\ \mathsf{trm}(S) \\
\mathsf{trm}(P \Longrightarrow S) & \Leftrightarrow & P \Rightarrow \mathsf{trm}(S) \\
\mathsf{trm}(S \parallel T) & \Leftrightarrow & \mathsf{trm}(S) \ \wedge\ \mathsf{trm}(T) \\
\mathsf{trm}(@z \cdot S) & \Leftrightarrow & \forall z \cdot \mathsf{trm}(S) \\
\mathsf{trm}(x :\in E) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{trm}(x : P) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{trm}(S \ ; T) & \Leftrightarrow & (\mathsf{trm}(S) \ \wedge\ \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']\,\mathsf{trm}(T))) \\
\mathsf{trm}(\mathcal{W}(P,S,J,V)) & \Leftrightarrow & J \ \wedge\ \forall x \cdot ((J \ \wedge\ P) \ \Rightarrow\ [S]\,J) \ \wedge \\
 & & \forall x \cdot (J \ \Rightarrow\ V \in \mathbb{N}) \ \wedge \\
 & & \forall x \cdot ((J \ \wedge\ P) \ \Rightarrow\ [n := V][S](V < n))
\end{array}
$$

2.3.2 Faisabilité

Certaines substitutions généralisées peuvent ne pas avoir de sens. Cela arrive en particulier lorsqu'on a une substitution gardée, dont la garde est toujours fausse : $\mathsf{bfalse} \Longrightarrow S$. En effet, dans ce cas quelle que soit la postcondition R, cette substitution établit R, puisque $[\mathsf{bfalse} \Longrightarrow S]R \ \Leftrightarrow\ \mathsf{bfalse} \ \Rightarrow\ [S]R$ qui est effectivement vrai. C'est théoriquement

une substitution "miraculeuse" puisqu'elle permet de spécifier n'importe quel programme. Malheureusement (ou plutôt heureusement) c'est une substitution irréalisable ou non "faisable". Le prédicat $\mathsf{fis}(S)$ caractérise les valeurs pour lesquelles S est faisable.

Notation	Définition
$\mathsf{fis}(S)$	$\neg\ [S]\,\mathsf{bfalse}$

On calcule facilement les résultats de faisabilité des substitutions primitives et de quelques formes usuelles. Ce calcul est celui de la "garde" d'une substitution, aussi dans certains articles, on trouve la notation "$\mathsf{grd}(S)$". Dans la substitution $x : P$, la variable $x\$0$, que l'on peut trouver dans P, représente la valeur de x avant la substitution. Dans $\mathcal{W}$, la variable $x\$0$, que l'on peut trouver dans J ou V, représente la valeur de x avant l'entrée dans la boucle.

$$
\begin{array}{lcl}
\mathsf{fis}(x := E) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{fis}(\mathsf{skip}) & \Leftrightarrow & \mathsf{btrue} \\
\mathsf{fis}(P \mid S) & \Leftrightarrow & P \Rightarrow \mathsf{fis}(S) \\
\mathsf{fis}(P \Longrightarrow S) & \Leftrightarrow & P \ \wedge\ \mathsf{fis}(S) \\
\mathsf{fis}(S \parallel T) & \Leftrightarrow & \mathsf{fis}(S)\ \vee\ \mathsf{fis}(T) \\
\mathsf{fis}(@z \cdot S) & \Leftrightarrow & \exists z \cdot \mathsf{fis}(S) \\
\mathsf{fis}(x :\in E) & \Leftrightarrow & E \neq \emptyset \\
\mathsf{fis}(x : P) & \Leftrightarrow & \exists x' \cdot [x\$0, x := x, x']P \\
\mathsf{fis}(S\ ;\ T) & \Leftrightarrow & (\mathsf{trm}(S) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(S)\ \wedge\ [x := x']\,\mathsf{fis}(T))) \\
\mathsf{fis}(\mathcal{W}(P,S,J,V)) & \Leftrightarrow & \mathsf{trm}(\mathcal{W}(P,S,J,V))\ \Rightarrow\ \exists x' \cdot ([x\$0, x := x, x']\,(J\ \wedge\ \neg P))
\end{array}
$$

2.3.3 Prédicat avant-après

Le prédicat avant-après permet de déterminer la relation entre les valeurs d'une variable x avant et après une substitution S. D'une manière classique, la valeur avant est notée x et la valeur après est notée x' (il s'agit toujours d'un *nouvel* identificateur par rapport aux variables de la machine courante). On voit donc que la ou les variables dont on veut parler sont un paramètre supplémentaire du prédicat. On note $\mathsf{prd}_x(S)$ la relation avant-après

des variables x pour la substitution S. La double négation est nécessaire à cause du non-déterminisme des substitutions. La formule signifie que x' est une des valeurs possibles de x après l'opération.

Notation	Définition
$\mathsf{prd}_x(S)$	$\neg\,[S](x' \neq x)$

Le calcul donne en particulier les résultats suivants (pour la notation $x\$0$, voir les commentaires au paragraphe précédent) :

$$
\begin{array}{lcll}
\mathsf{prd}_x(x := E) & \Leftrightarrow & x' = E & \\
\mathsf{prd}_{x,y}(x := E) & \Leftrightarrow & x', y' = E, y & \\
\mathsf{prd}_x(\mathsf{skip}) & \Leftrightarrow & x' = x & \\
\mathsf{prd}_x(P \mid S) & \Leftrightarrow & P \Rightarrow \mathsf{prd}_x(S) & \\
\mathsf{prd}_x(P \Longrightarrow S) & \Leftrightarrow & P \ \wedge\ \mathsf{prd}_x(S) & \\
\mathsf{prd}_x(S \parallel T) & \Leftrightarrow & \mathsf{prd}_x(S) \ \vee\ \mathsf{prd}_x(T) & \\
\mathsf{prd}_x(@z \cdot S) & \Leftrightarrow & \exists z \cdot \mathsf{prd}_x(S) & \text{si } z \backslash x' \quad (*) \\
\mathsf{prd}_x(@y \cdot T) & \Leftrightarrow & \exists (y, y') \cdot \mathsf{prd}_{x,y}(T) & \text{si } y \backslash x' \quad (**) \\
\mathsf{prd}_x(x :\in E) & \Leftrightarrow & x' \in E & \\
\mathsf{prd}_x(x : P) & \Leftrightarrow & [x\$0, x := x, x']P & \\
\mathsf{prd}_x(S \ ; T) & \Leftrightarrow & (\mathsf{trm}(S) \Rightarrow \exists x'' \cdot ([x' := x'']\mathsf{prd}_x(S) \ \wedge\ [x := x'']\,\mathsf{prd}_x(T))) & \\
\mathsf{prd}_x(\mathcal{W}(P,S,J,V)) & \Leftrightarrow & \mathsf{trm}(\mathcal{W}(P,S,J,V)) \ \Rightarrow\ [x\$0, x := x, x']\,(J \ \wedge\ \neg P) &
\end{array}
$$

Le prédicat avant-après d'une substitution de choix non borné "$@z \cdot S$" est divisé en deux : dans le cas marqué (*), la variable z n'est pas modifiée par S. Dans le cas marqué (**), la variable y est modifiée et le développement du prédicat sur T est donc quantifié par y et y' qui vont intervenir dans la relation $\mathsf{prd}_{x,y}(T)$. Une propriété intéressante est de relier la faisabilité et l'existence d'une valeur "après" du prédicat avant-après. On peut en effet démontrer :

$$\mathsf{fis}(S) \quad \Leftrightarrow \quad \exists x' \cdot \mathsf{prd}_x(S)$$

2.3.4 Forme normalisée

Toute substitution généralisée peut se mettre sous la forme :

$$S = P \mid @x' \cdot (Q \Longrightarrow x := x') \qquad \text{si } x' \backslash P$$

Dans le B-Book [Abr96], la preuve de cette forme normalisée est donnée en la calculant pour chacune des substitutions primitives et en appliquant des transformations syntaxiques. En fait, on peut montrer que toute substitution généralisée est équivalente à la forme suivante, construite avec trm et de prd_x :

$$S \;=\; \mathsf{trm}(S) \mid @x' \cdot (\mathsf{prd}_x(S) \Longrightarrow x := x')$$

La plus faible précondition d'une forme normalisée est :

$$[\,P \mid @x' \cdot (Q \Longrightarrow x := x')]\,R \;\Leftrightarrow\; P \;\wedge\; \forall x' \cdot (Q \Rightarrow [x := x']\,R)$$

On a les équivalences :

$$\begin{array}{lcl} \mathsf{trm}(P \mid @x' \cdot (Q \Longrightarrow x := x') & \Leftrightarrow & P \\ \mathsf{fis}(P \mid @x' \cdot (Q \Longrightarrow x := x') & \Leftrightarrow & P \Rightarrow \exists x' \cdot Q \\ \mathsf{prd}_x(P \mid @x' \cdot (Q \Longrightarrow x := x') & \Leftrightarrow & P \Rightarrow Q \end{array}$$

On peut définir l'égalité de deux substitutions par :

$S = T$	$[S]\,R \Leftrightarrow [T]\,R$ pour tout prédicat R

En se basant sur la forme normalisée, ce résultat conduit à :

$S = T$	$(\mathsf{trm}(S) \Leftrightarrow \mathsf{trm}(T)) \;\wedge\; ((\mathsf{trm}(S) \Rightarrow \mathsf{prd}_x(S)) \Leftrightarrow (\mathsf{trm}(T) \Rightarrow \mathsf{prd}_x(T)))$

Dans le tableau qui suit, R et Q sont des prédicats et S une substitution généralisée quelconque. Les propriétés suivantes font partie de la théorie des substitutions généralisées et plus généralement des transformateurs de prédicats positivement conjonctifs :

$[S]\,(R \;\wedge\; Q) \;\Leftrightarrow\; [S]\,R \;\wedge\; [S]\,Q$	Distributivité
$\forall x \cdot (R \Rightarrow Q) \;\Rightarrow\; ([S]\,R \Rightarrow [S]Q)$	Monotonie

Les substitutions généralisées satisfont les propriétés :

Totalité	$\mathsf{prd}_x(S) \ \vee \ \mathsf{trm}(S)$
Terminaison	$[S]\,R \ \Rightarrow \ \mathsf{trm}(S)$

Preuve :

On a : $x = x' \ \vee \ \mathsf{btrue}$, c'est-à-dire $x \neq x' \Rightarrow \mathsf{btrue}$. Et, par la propriété de monotonie :

$$\begin{array}{l} [S]\,(x \neq x') \ \Rightarrow \ [S]\,\mathsf{btrue} \\ \Leftrightarrow \ \neg\,[S]\,(x \neq x') \ \vee \ [S]\,\mathsf{btrue} \\ \Leftrightarrow \ \mathsf{prd}_x(S) \ \vee \ \mathsf{trm}(S) \end{array}$$

La seconde propriété se démontre de la même manière. □

2.4 Interprétation ensembliste

2.4.1 Modèle de base

Pour construire ce modèle, on suppose que les variables appartiennent à un ensemble de typage[6] et que cette appartenance est un invariant de la spécification. Plus formellement, cela donne l'hypothèse que toute variable x appartient à un certain ensemble s tel que :

$$x \in s \ \wedge \ \mathsf{trm}(S) \ \Rightarrow \ [S](x \in s)$$

Sous cette hypothèse, on s'intéresse aux notions ensemblistes suivantes : $\mathsf{pre}(S)$ est l'ensemble des éléments pour lesquels la terminaison de S est vraie ; $\mathsf{rel}(S)$ est la relation qui relie les valeurs avant et les valeurs après de la substitution S et $\mathsf{dom}(S)$ est le domaine de la relation $\mathsf{rel}(S)$. D'où les définitions :

Notation	Définition
$\mathsf{pre}(S)$	$\{x \mid x \in s \ \wedge \ \mathsf{trm}(S)\}$
$\mathsf{rel}(S)$	$\{x, x' \mid x, x' \in s \times s \ \wedge \ \mathsf{prd}_x(S)\}$
$\mathsf{dom}(S)$	$\{x \mid x \in s \ \wedge \ \mathsf{fis}(S)\}$

Quelques exemples avec $x \in \mathbb{Z}$ (l'ensemble $\mathbb{N}_1$ est l'ensemble des entiers strictement positifs) :

[6] Il s'agit ici des "super-ensembles" qui structurent l'univers de typage en B, par exemple les entiers $\mathbb{Z}$.

$S =$	PRE $x > 0$ THEN	$\mathsf{pre}(S) = \{x \mid x > 0\} = \mathbb{N}_1$
	$x := x - 1$	$\mathsf{rel}(S) = (\mathbb{Z} - \mathbb{N}_1) \times \mathbb{Z} \ \cup \ \{x, x' \mid x, x' \in \mathbb{N}_1 \times \mathbb{Z} \ \wedge \ x' = x - 1\}$
	END	$\mathsf{dom}(S) = \mathbb{Z}$
$T =$	SELECT $x > 0$ THEN	$\mathsf{pre}(T) = \mathbb{Z}$
	$x := x - 1$	$\mathsf{rel}(T) = \{x, x' \mid x, x' \in \mathbb{N}_1 \times \mathbb{Z} \ \wedge \ x' = x - 1\}$
	END	$\mathsf{dom}(T) = \mathbb{N}_1$
$U =$	IF $x > 0$ THEN	$\mathsf{pre}(U) = \mathbb{Z}$
	$x := x - 1$	$\mathsf{rel}(U) = \mathsf{id}(\mathbb{Z} - \mathbb{N}_1) \ \cup \ \{x, x' \mid x, x' \in \mathbb{N}_1 \times \mathbb{Z} \ \wedge \ x' = x - 1\}$
	END	$\mathsf{dom}(U) = \mathbb{Z}$

Propriété caractéristique

Si $\mathsf{pre}(S)$ n'est pas vide, $\mathsf{rel}(S)$ contient l'ensemble des éléments complémentaires de la précondition associés à tout le domaine des variables. Cela est visible sur le premier exemple ci-dessus. Il s'agit de la contrepartie ensembliste de la propriété $\mathsf{trm}(S) \vee \mathsf{prd}_x(S)$. Sachant que $p \subseteq s$, on note $\overline{p} = s - p$. On a :

$$\overline{\mathsf{pre}(S)} \times s \quad \subseteq \quad \mathsf{rel}(S)$$

Forme normalisée ensembliste

La forme normalisée de S peut s'écrire en fonction de $\mathsf{pre}(S)$ et $\mathsf{rel}(S)$:

$$S \ = \ x \in \mathsf{pre}(S) \mid @x' \cdot ((x, x') \in \mathsf{rel}(S) \implies x := x')$$

2.4.2 Interprétation ensembliste des substitutions primitives

Pour un ensemble de valeurs satisfaisant un prédicat P, on utilise la notation :

$$\mathsf{set}(P) \ = \ \{x \mid x \in s \quad \wedge \quad P\}$$

Pour chaque substitution primitive S, on donne les valeurs $\mathsf{pre}(S)$, $\mathsf{rel}(S)$ et $\mathsf{dom}(S)$ en fonction des interprétations ensemblistes des arguments.

	$\mathsf{pre}()$	$\mathsf{rel}()$	$\mathsf{dom}()$
$x := E$	s	$\lambda x \cdot (x \in s \mid E)$	s
skip	s	$\mathsf{id}(s)$	s
$P \mid S$	$\mathsf{set}(P) \ \cap \ \mathsf{pre}(S)$	$(\overline{\mathsf{set}(P)} \times s) \ \cup \ \mathsf{rel}(S)$	$\overline{\mathsf{set}(P)} \ \cup \ \mathsf{dom}(S)$
$S \parallel T$	$\mathsf{pre}(S) \ \cap \ \mathsf{pre}(T)$	$\mathsf{rel}(S) \ \cup \ \mathsf{rel}(T)$	$\mathsf{dom}(S) \ \cup \ \mathsf{dom}(T)$
$P \Longrightarrow S$	$\overline{\mathsf{set}(P)} \ \cup \ \mathsf{pre}(S)$	$(\mathsf{set}(P) \times s) \ \cap \ \mathsf{rel}(S)$	$\mathsf{set}(P) \ \cap \ \mathsf{dom}(S)$
$S \ ; \ T$	$\mathsf{pre}(S) \cap$ $\overline{\mathsf{rel}(S)^{-1}[\overline{\mathsf{pre}(T)}]}$	$\overline{\mathsf{pre}(S)} \times s \cup$ $\mathsf{rel}(S) \ ; \ \mathsf{rel}(T)$	$\overline{\mathsf{pre}(S)} \cup$ $\mathsf{rel}(S)^{-1}[\mathsf{dom}(T)]$

Dans le cas du choix non borné, on considère un premier cas, noté $@z \cdot (z \in u \Longrightarrow S)$ où z est une variable auxiliaire qui n'est pas modifiée par S (S ne modifie éventuellement que les variables x de la machine) et un deuxième cas noté $@y \cdot (y \in t \Longrightarrow T)$, où T peut modifier à la fois x et y. Les resultats sont :

	$\mathsf{pre}()$	$\mathsf{rel}()$
$@z \cdot (z \in u \Longrightarrow S)$	$\bigcap z \cdot (z \in u \mid \mathsf{pre}(S))$	$\bigcup z \cdot (z \in u \mid \mathsf{rel}(S))$
$@y \cdot (y \in t \Longrightarrow T)$	$\overline{\mathsf{prj}_1(s,t)[\overline{\mathsf{pre}(T)}]}$	$\mathsf{prj}_1(s,t)^{-1} \ ; \ \mathsf{rel}(T) \ ; \ \mathsf{prj}_1(s,t)$

Le premier cas ci-dessus est une généralisation évidente du choix borné. Le deuxième cas est défini comme la substitution généralisée T dont l'effet est "projeté" sur les composantes (les variables) x de la machine, par des opérations de projection de relation.

2.4.3 Transformateur d'ensembles

Sachant que $x \in s$, on s'intéresse à l'assertion $x \in q$ où $q \subseteq s$. Plus précisément, on cherche à savoir quelle est la plus faible précondition qui établit $x \in q$ après une substitution donnée S. On a :

$$\{x \mid x \in s \ \wedge \ [S](x \in q)\} \ = \ \mathsf{pre}(S) \cap \overline{\mathsf{rel}(S)^{-1}[\overline{q}]}$$

Si on considère q comme un paramètre, on peut définir $\mathsf{str}(S)$ comme un *transformateur d'ensembles* qui fournit l'ensemble auquel doit appartenir x en entrée pour que la variable x appartient à q après "exécution" du programme spécifié par la substitution S. D'où la définition :

Notation	Définition
$\mathsf{str}(S)$	$\lambda p \cdot (p \in \mathbb{P}(s) \mid \{x \mid x \in s \ \wedge \ [S](x \in p)\})$

Il existe des relations entre les différents constructeurs de l'interprétation ensembliste (avec les notations habituelles) :

$$\begin{array}{lcll}
\mathsf{pre}(S) & = & \mathsf{str}(S)(s) & \\
\mathsf{rel}(S) & = & \{x, x' \mid (x, x') \in s \times s \ \wedge \ x \in \overline{\mathsf{str}(S)(\overline{\{x'\}})}\} & \\
\mathsf{dom}(S) & = & \overline{\mathsf{str}(S)(\emptyset)} & \\
\mathsf{str}(S)(p) & = & \mathsf{pre}(S) \cap \overline{\mathsf{rel}(S)^{-1}[\overline{p}]} & \\
\mathsf{rel}(S)^{-1}[p] & = & \overline{\mathsf{str}(S)(\overline{p})} & \text{if } p \neq \emptyset
\end{array}$$

Pour chaque constructeur de substitution généralisée S, on peut calculer la valeur de $\mathsf{str}(S)(p)$ en fonction des composants de S (exercice), ce qui donne le tableau suivant :

S	$\mathsf{str}(S)(p)$
$x := E$	$\{x \mid x \in s \ \wedge \ E \in p\}$
skip	p
$P \mid S$	$\mathsf{set}(P) \ \cap \ \mathsf{str}(S)(p)$
$S \parallel T$	$\mathsf{str}(S)(p) \ \cap \ \mathsf{str}(T)(p)$
$P \Longrightarrow S$	$\overline{\mathsf{set}(P)} \ \cup \ \mathsf{str}(S)(p)$
$@z \cdot (z \in u \Longrightarrow S)$	$\bigcap z \cdot (z \in u \mid \mathsf{str}(S)(p))$
$@y \cdot (y \in t \Longrightarrow T)$	$\overline{\mathsf{prj}_1(s,t)[\overline{\mathsf{str}(T)(p \times t)}]}$
$S \ ; \ T$	$\mathsf{str}(S)(\mathsf{str}(T)(p))$

Exemple 1 : Soit la substitution $S = (x := x + 1 \parallel x := x - 1)$ et l'ensemble $p = \mathsf{set}(1 \leq x \leq 5)$, c'est-à-dire $\{x \mid x \in 1..5\}$ où $x \in \mathsf{NAT}$. On a :

$$\begin{array}{rcl} \mathsf{str}(S)(p) & = & \mathsf{str}(x := x + 1 \parallel x := x - 1)(p) \\ & = & \mathsf{str}(x := x + 1)(p) \ \cap \ \mathsf{str}(x := x - 1)(p) \\ & = & \mathsf{set}(0 \leq x \leq 4) \ \cap \ \mathsf{set}(2 \leq x \leq 6) \\ & = & \{x \mid x \in 2..4\} \end{array}$$

Exemple 2 : Soit la substitution $T = (x > 0 \Longrightarrow x := x - 1)$ et l'ensemble $q = \mathsf{set}(x > 5)$, où $x \in \mathsf{INT}$. On a :

$$\begin{array}{rcl} \mathsf{str}(T)(q) & = & \mathsf{str}(x > 0 \Longrightarrow x := x - 1)(q) \\ & = & \overline{\mathsf{set}(x > 0)} \ \cup \ \mathsf{str}(x := x - 1)(q) \\ & = & \overline{\mathsf{set}(x > 0)} \ \cup \ \mathsf{set}(x > 6) \\ & = & \{x \mid \mathsf{minint} \leq x \leq 0 \ \vee \ 6 < x \leq \mathsf{maxint}\} \end{array}$$

Exemple 3 : Soit la substitution $U = (x > 0 \mid x := x - 1)$ et l'ensemble $q = \mathsf{set}(x > 5)$, où $x \in \mathsf{INT}$. On a :

$$
\begin{array}{lcl}
\mathsf{str}(U)(q) & = & \mathsf{str}(x > 0 \mid x := x - 1)(q) \\
& = & \mathsf{set}(x > 0) \ \cap \ \mathsf{str}(x := x - 1)(q) \\
& = & \mathsf{set}(x > 0) \ \cap \ \mathsf{set}(x > 6) \\
& = & \{x \mid 6 < x \leq \mathsf{maxint}\}
\end{array}
$$

L'égalité des substitutions généralisée peut être définie à partir de str, simplement par :

$S = T$	$\mathsf{str}(S) = \mathsf{str}(T)$

En déduire l'égalité à partir de pre et rel.

Chapitre 3

Machines abstraites

Résumé

Ce chapitre décrit les *machines abstraites*. C'est un composant B qui décrit le premier niveau d'un développement. On donne les clauses qui constituent une machine. On définit les obligations de preuve qui assurent que les opérations préservent l'invariant. Enfin, on termine par des exemples simples de machines.

3.1 Description des clauses d'une machine

3.1.1 Généralités

Les machines abstraites sont constituées d'un entête, d'un ensemble de données (la partie statique) et d'un ensemble d'opérations (la partie dynamique). L'entête contient le nom de la machine, suivi éventuellement de paramètres qui peuvent être des ensembles et/ou des valeurs scalaires. Par convention, les identificateurs des ensembles doivent commencer par une majuscule. Les ensembles sont supposés finis et non vides (la vérification formelle est faite à l'instanciation de la machine). Les valeurs scalaires sont spécifiées par un prédicat dans la rubrique CONSTRAINTS.

MACHINE
- *Partie entête :*
  - *nom de la machine*
  - *paramètres de la machine*
  - *contraintes sur les paramètres*
- *Partie statique :*
  - *déclaration d'ensembles*
  - *déclaration de constantes*
  - *propriétés des constantes*
  - *variables (état)*
  - *invariant (caractérisation de l'état)*
- *Partie dynamique :*
  - *initialisation de l'état*
  - *opérations*

END

3.1.2 Définition de données

Les données sont constituées (éventuellement) de :

1. Définition d'ensembles (clause SETS). Ces ensembles sont soit uniquement caractérisés par leur nom, soit définis par leur nom et les éléments qui le composent (ensemble énuméré). Les contraintes associées à ces déclarations sont : les ensembles sont finis et non vides ; les éléments énumérés d'un ensemble sont distincts.
2. Des constantes. Il s'agit simplement d'une liste de noms. Les constantes peuvent être *abstraites* (clause ABSTRACT_CONSTANTS) ou *concrètes* (clause CONCRETE_CONSTANTS). Par défaut (clause CONSTANTS), il s'agit de constantes concrètes.
3. Les propriétés des constantes (rubrique PROPERTIES). Cette rubrique contient un prédicat qui spécifie les constantes par des axiomes. Ces axiomes ne peuvent pas dépendre des paramètres de la machine. La seule contrainte (qui sera vérifiée) est qu'il existe bien une valeur qui satisfait l'axiomatisation.
4. Des variables. Comme pour les constantes, il s'agit simplement d'une liste de noms. Elles peuvent être abstraites (clause ABSTRACT_VARIABLES) ou concrètes (clause CONCRETE_VARIABLES). Par défaut (clause VARIABLES), ce sont des variables abstraites. Nous reviendrons sur l'usage des informations concrètes au paragraphe 5.1.
5. Un invariant (clause INVARIANT). C'est un prédicat qui donne le typage des variables et les contraintes qu'elles doivent satisfaire. Ces contraintes peuvent être du sous-typage par prédicat ou des relations entre variables ou entre variables et constantes.
6. Des assertions (clause ASSERTIONS). C'est une liste de prédicats qui sont des conséquences logiques des autres axiomes (et de l'invariant) déclarés dans la machine. On peut les voir comme des lemmes utiles pour les preuves ou pour la compréhension (elles devront être vérifiées de toute façon).

3.1.3 Définition des opérations

Dans la partie dynamique, on trouve la clause d'initialisation des variables (clause INITIALISATION) qui est une substitution généralisée et une liste d'opérations (clause OPERATIONS). Les opérations ont la forme générale :

$$r \longleftarrow nom\_op(p) = \text{PRE } P \text{ THEN } K \text{ END}$$

où r (optionnel) est une variable (ou une liste de variables) des paramètres de sortie, nom_op est l'identificateur de l'opération, p (optionnel) est la liste des paramètres d'entrée, P est un prédicat précondition de l'opération (qui doit donner le typage des paramètres d'entrée) et K est une substitution généralisée qui spécifie l'effet de l'opération. Le type du résultat r est déduit de la valeur qu'il reçoit dans K. Les restrictions sont les suivantes :

1. les noms des paramètres d'entrée et de sortie sont distincts et différents de ceux des variables ;
2. les paramètres d'entrée (passés par valeur) ne peuvent pas être affectés dans le corps de l'opération ;
3. les paramètres de sortie sont affectés dans le corps de l'opération et leur valeur n'est pas lue avant cette affectation.

Ces restrictions sont celles rencontrées dans les langages de programmation, si ce n'est qu'il n'y a pas de règle de masquage de noms pour les paramètres. Les paramètres d'entrée correspondent aux paramètres `in` et les paramètres de sortie aux paramètres `out` du langage Ada. Les deux premières contraintes sont vérifiées par la méthode B. La dernière n'est pas directement vérifiée mais donnera lieu à des obligations de preuves non prouvables si elle n'est pas respectée. La condition revient à vérifier que le prédicat $\mathsf{prd}_{x,r}(K)$ ne contient pas d'occurrence libre de r, les variables x désignant ici les variables de la machine dans laquelle l'opération est définie.

3.1.4 Restrictions sur les substitutions

Les substitutions généralisées séquencement et itération ne sont pas autorisées dans les composants MACHINE. Ces restrictions sont principalement d'ordre méthodologique. Dans la pratique ces substitutions ne s'avèrent pas nécessaires au niveau des spécifications.

3.2 Validation d'une machine simple

Une machine abstraite introduit une notion de composant prouvé : l'initialisation établit l'invariant et chaque définition d'opération préserve l'invariant, sous l'hypothèse de la précondition. Le principe sous-jacent à un composant machine abstraite est l'encapsulation : les variables de la machine ne peuvent être modifiées que par l'intermédiaire des opérations.

3.2.1 Forme générale d'une machine

La forme générale d'une machine est donnée à la figure 3.1.

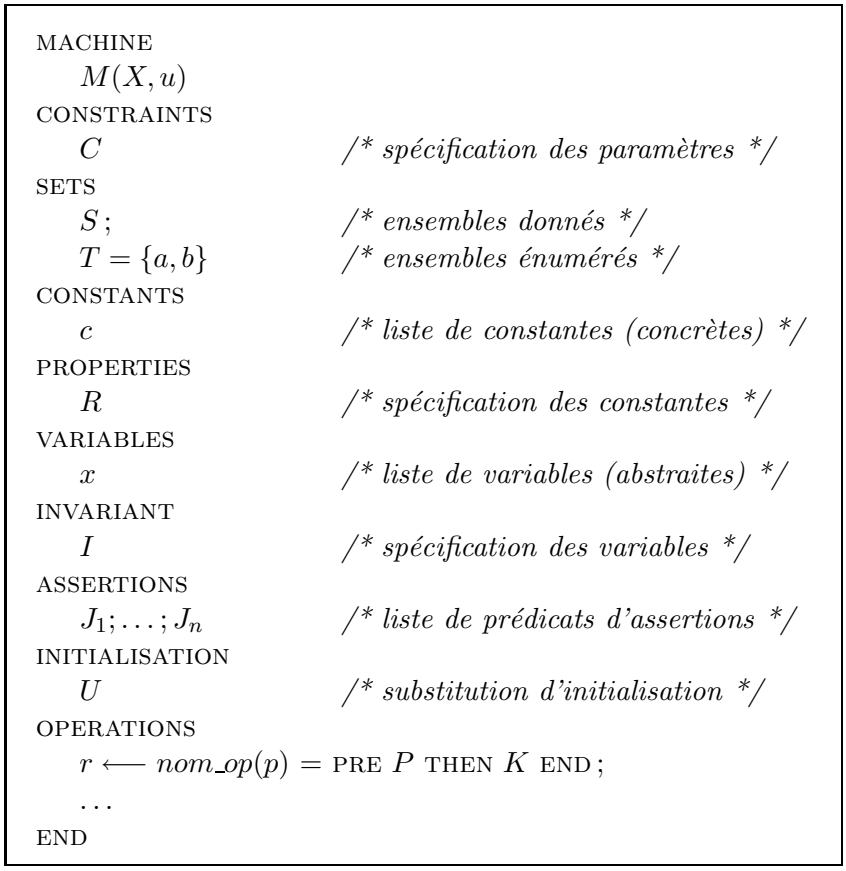


FIG. 3.1 – Forme générale d'une machine

3.2.2 Obligations de preuves

On utilise des abréviations A et B pour décrire les conditions sur les paramètres (contraintes sur les scalaires et ensembles finis et non vides) et sur les propriétés des constantes et des ensembles déclarés. Les méta-variables sont celles de la figure 3.1.

$$A \quad \Leftrightarrow \quad C \ \wedge \ X \in \mathbb{P}_1(\mathsf{INT})$$

$$B \quad \Leftrightarrow \quad R \ \wedge \ S \in \mathbb{P}_1(\mathsf{INT}) \quad \wedge \quad T \in \mathbb{P}_1(\mathsf{INT}) \quad \wedge \quad T = \{a,b\} \quad \wedge \quad a \neq b$$

Les assertions sont mises en séquence, ce qui signifie que l'assertion J_k peut être démontrée en utilisant les assertions $J_1, \ldots, J_{k-1}$. Dans la suite, on utilise l'abbréviation :

$$J \ \Leftrightarrow \ J_1 \ \wedge \ \ldots \ \wedge \ J_n$$

Les formules de preuve des assertions sont obtenues par

$A \ \wedge \ B \ \wedge \ I \ \Rightarrow \ J_1$	première assertion
$A \ \wedge \ B \ \wedge \ I \ \wedge \ J_1 \ \wedge \ \ldots \ \wedge \ J_{k-1} \ \Rightarrow \ J_k$	assertion k

Les conditions de validité ou "obligations de preuves" d'une machine indiquent que l'initialisation établit l'invariant I et que les opérations préservent l'invariant si elles sont appelées sous leur précondition P :

$A \ \wedge \ B \ \Rightarrow \ [U]\, I$	initialisation
$A \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P \ \Rightarrow \ [K]\, I$	opérations

3.3 Exemples de machines

3.3.1 Réservation des places

Cet exemple est une machine *RESERVATION* qui gère la réservation de places. Le nombre de places (nb_max) est une constante de la machine ainsi que l'ensemble des places, noté *SIEGES* tel que $SIEGES = 1..nb\_max$. L'état est constitué d'une variable *occupes* qui est l'ensemble des places occupées et d'une variable nb_libre qui indique le nombre de places libres. L'invariant indique le typage des variables et rend explicite la relation qui les lie, c'est-à-dire $nb\_libre = nb\_max - \mathsf{card}(occupes)$.

Les services de la machine sont les différentes opérations. L'opération *place_libre* donne le nombre de places libres. L'opération *reserver* réserve une place parmi les places libres. Ce choix non déterministe est réalisé par la substitution "ANY". La stratégie d'allocation des places n'est pas spécifiée dans cette machine, car on a une vision "haut-niveau" de l'opération de réservation. Un avantage important est que cette machine peut être réutilisée pour n'importe quel type de réservation. L'opération *liberer* libère la place donnée en paramètre si elle est réservée, sinon elle ne fait rien. Le résultat de cette opération est un booléen qui indique l'alternative réalisée. On remarque qu'il n'est pas possible dans cette spécification de savoir de l'extérieur si une place donnée est réservée ou non.

Notons que si les appels aux services de *RESERVATION* sont faits dans le cadre de la méthode B, alors les vérifications des obligations de preuve imposent que les préconditions d'appel sont satisfaites (voir chapitre 6, paragraphe 6.2.3). La spécification complète est donnée avec commentaires à la figure 3.2.

En guise d'exercice, on peut calculer l'obligation de preuve de l'initialisation, en notant l'invariant par I :

Initialisation : $nb\_max \in 1..\mathsf{maxint} \ \wedge \ SIEGES = 1..nb\_max$
$\Rightarrow$
$[occupes, nb\_libre := \emptyset, nb\_max] \ I$

Cette obligation de preuve se réécrit en :

$nb\_max \in 1..\mathsf{maxint} \ \wedge \ SIEGES = 1..nb\_max$
$\Rightarrow$
$\emptyset \subseteq SIEGES$
$\wedge \ nb\_max \in 0..nb\_max$
$\wedge \ nb\_max = nb\_max - \mathsf{card}(\emptyset)$

```
MACHINE
    RESERVATION
CONSTANTS
    nb_max, SIEGES
PROPERTIES
    nb_max ∈ 1..maxint ∧          /* Nombre maximum de sièges */
    SIEGES = 1..nb_max            /* Ensemble des numéros de sièges */
VARIABLES
    occupes, nb_libre
INVARIANT
    occupes ⊆ SIEGES              /* Ensemble des sièges occupés */
     ∧ nb_libre ∈ 0..nb_max       /* Nombre de sièges libres */
     ∧ nb_libre = nb_max − card(occupes)
INITIALISATION
    occupes, nb_libre := ∅, nb_max
OPERATIONS
    nb ⟵ place_libre =            /* Nombre de places libres */
        BEGIN
            nb := nb_libre
        END ;
    place ⟵ reserver =            /* Le résultat est la place réservée */
        PRE
            nb_libre ≠ 0
        THEN                      /* On prend une certaine valeur pp dans */
            ANY pp WHERE          /* les numéros de sièges libres */
                pp ∈ SIEGES − occupes
            THEN
                place, occupes, nb_libre := pp, occupes ∪ {pp}, nb_libre − 1
            END
        END ;
    rr ⟵ liberer(place) =         /* Opération de libération */
        PRE
            place ∈ SIEGES
        THEN                      /* On libère la place indiquée en paramètre */
            IF place ∈ occupes THEN
                rr, occupes, nb_libre := TRUE, occupes − {place}, nb_libre + 1
            ELSE
                rr := FALSE
            END
        END
END
```

FIG. 3.2 – Machine *RESERVATION*

L'obligation de preuve de l'opération *reserver* est :

Réserver : $nb\_max \in 1..\mathsf{maxint} \ \wedge \ SIEGES = 1..nb\_max$
$\wedge \ I$
$\wedge \ nb\_libre \neq 0$
$\Rightarrow$
[ANY pp WHERE $pp \in SIEGES - occupes$
THEN $place, occupes, nb\_libre := pp, occupes \ \cup \{pp\}, nb\_libre - 1$
END] I

D'où la formule à démontrer, dans laquelle toutes les variables libres des hypothèses sont quantifiées universellement :

$nb\_max \in 1..\mathsf{maxint} \ \wedge \ SIEGES = 1..nb\_max$
$\wedge \ occupes \subseteq SIEGES$
$\wedge \ nb\_libre \in 0..nb\_max$
$\wedge \ nb\_libre = nb\_max - \mathsf{card}(occupes)$
$\wedge \ nb\_libre \neq 0$
$\wedge \ pp \in SIEGES - occupes$
$\Rightarrow$
$(occupes \ \cup \{pp\}) \subseteq SIEGES$
$\wedge \ nb\_libre - 1 \in 0..nb\_max$
$\wedge \ nb\_libre - 1 = nb\_max - \mathsf{card}(occupes \ \cup \{pp\})$

3.3.2 Petit problème de division entière

C'est l'exemple de base de la preuve de programmes en logique de Hoare[1]. On sait qu'il s'agit de trouver le quotient entier q et le reste r de la division de a par b, sachant que $a \geq 0$ et $b > 0$. Les relations mathématiques qui lient ces quatre valeurs sont : $a = b \times q + r \ \wedge \ r < b$.

En B, on ne peut pas décrire un algorithme seul. Il faut l'intégrer à une machine qui propose le "service" division entière. Cette machine n'a pas d'état (ni invariant, ni initialisation). L'opération va délivrer deux résultats. On a une spécification d'opération qui s'écrit à peu près textuellement comme la spécification, sachant que l'on caractérise la relation entre les entrées et les sorties à l'intérieur de la partie WHERE de la substitution qui définit la division. La machine de la division entière est donnée à la figure 3.3.

On reviendra sur l'implémentation de cette machine et de cette opération au chapitre 5.2.1, en montrant comment les obligations de preuve de l'implémentation sont équivalentes à la preuve de programme au sens habituel du terme.

3.3.3 Tri d'un tableau

Le tri de tableau est le problème algorithmique par excellence. L'état de la machine est constitué par le tableau à trier (fonction totale). L'intervalle des indices de tableau est $1..nn$ où nn est une constante concrète qui devra être initialisée à l'implémentation du tableau. En B, on ne peut décrire que des variables dont la taille est connue statiquement (contrainte nécessaire dans les logiciels critiques, en vue d'éviter les erreurs à l'exécution).

L'opération "*trier*" est définie (figure 3.4) en indiquant quelles sont les propriétés du tableau résultat noté "*tt*". Cette spécification est fournie par la partie ANY-WHERE de la

[1] C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*, Déjà cité.

```
MACHINE
    DIVISION
OPERATIONS                          /* qq et rr sont le quotient et le reste de la */
    qq, rr ⟵ divi(aa, bb) =         /* division entière de aa par bb */
        PRE
            aa ∈ NAT  ∧  bb ∈ NAT1
        THEN
            ANY ss, tt WHERE
                ss ∈ NAT  ∧  tt ∈ NAT ∧
                aa = bb ∗ ss + tt  ∧  tt < bb
            THEN
                qq, rr := ss, tt
            END
        END
END
```

FIG. 3.3 – Machine de la division entière

substitution. L'effet est simplement que l'état devient la valeur du tableau trié. On peut noter que cette spécification est valable quelle que soit l'implémentation choisie (tri par insertion, tri rapide, etc.).

Le point intéressant est la définition que le tableau final est une "permutation" du tableau initial. On traduit cela par le fait qu'il existe une bijection "hh" sur les indices telle que $(hh \ ; \ tt) = TT$. On remarque que, bien que le tableau trié soit finalement unique, il peut exister plusieurs fonctions différentes de permutation des indices pour les valeurs égales du tableau. Si tous les éléments du tableau sont différents (le tableau est une injection : $TT \in 1..nn \rightarrowtail \mathsf{NAT}$), alors on peut simplement spécifier la permutation par : $\mathsf{ran}(TT) = \mathsf{ran}(tt)$.

La machine exporte aussi les opérations $vv \longleftarrow val\_TT(ii)$ qui fournit la valeur du tableau pour l'indice ii et $str\_TT(ii, vv)$ qui range la valeur de vv à l'indice ii.

Le tableau est spécifié comme une fonction totale. Cet invariant est vérifié, car l'initialisation indique que le tableau prend pour valeur une valeur quelconque du type spécifié (initialisation non déterministe). Cette "initialisation" correspond à l'allocation d'un tableau en mémoire, dans la mesure où l'emplacement des éléments de tableau représente exactement des valeurs du type NAT.

MACHINE
$TRI1$
CONCRETE_CONSTANTS
nn
PROPERTIES
$nn \in \mathsf{NAT}_1$
VARIABLES
TT
INVARIANT
$TT \in 1..nn \rightarrow \mathsf{NAT}$
INITIALISATION
$TT :\in 1..nn \rightarrow \mathsf{NAT}$
OPERATIONS
$vv \longleftarrow val\_TT(ii) =$
PRE $ii \in 1..nn$ THEN $vv := TT(ii)$ END
;
$str\_TT(ii, vv) =$
PRE $ii \in 1..nn \wedge vv \in \mathsf{NAT}$ THEN $TT(ii) := vv$ END
;
$trier =$
ANY tt, hh WHERE
$tt \in 1..nn \rightarrow \mathsf{NAT} \wedge$
$(\forall(i_1, i_2) \cdot (i_1 \in 1..nn \ \wedge \ i_2 \in 1..nn \ \wedge \ i_1 < i_2 \Rightarrow tt(i_1) \leq tt(i_2))) \wedge$
$hh \in 1..nn \rightarrowtail\!\!\!\!\!\rightarrow 1..nn \wedge$
$(hh \ ; \ tt) = TT$
THEN
$TT := tt$
END
END

FIG. 3.4 – Machine de tri

Chapitre 4

Raffinement

Résumé

Les raffinements constituent les étapes de développement d'une machine. Ils consistent à détailler et à préciser les structures de données et les opérations de la machine initiale. Des "obligations de preuve" assurent que les raffinements sont cohérents avec la sémantique spécifiée au plus haut niveau d'abstraction.

4.1 Raffinement des substitutions et des composants

4.1.1 Raffinement simple

Raffiner une substitution S signifie trouver une autre substitution T dont l'effet sur les variables x de l'état est un des effets possibles de la substitution S. De ce fait, utiliser T à la place de S n'est pas perceptible par un observateur extérieur. Si S et T sont des substitutions généralisées, on note $S \sqsubseteq T$ la relation "S est raffiné par T". Une première définition du raffinement des substitutions est :

$S \sqsubseteq T$	$\forall R \cdot ([S]\, R \;\Rightarrow\; [T]\, R)$

(1)

Des exemples de telles substitutions sont :

$$S \stackrel{def}{=\!=} (x := 1 \parallel x := 2)$$
$$T \stackrel{def}{=\!=} x := 1$$

ou encore :

$$S \stackrel{def}{=\!=} (x > 4 \mid x := x + 1)$$
$$T \stackrel{def}{=\!=} (x \geq 0 \mid x := x + 1)$$

D'un point de vue ensembliste, le raffinement peut s'exprimer sur les ensembles pre et la relation rel :

$S \sqsubseteq T$	$\mathsf{pre}(S) \subseteq \mathsf{pre}(T)$ $\mathsf{rel}(T) \subseteq \mathsf{rel}(S)$

(2)

D'après les définitions du paragraphe 2.4.3, si l'on suppose que les variables de la machine sont x avec l'invariant $x \in s$, alors le raffinement peut se définir par :

$S \sqsubseteq T$	$\forall a \cdot (a \subseteq s \;\Rightarrow\; \mathsf{str}(S)(a) \subseteq \mathsf{str}(T)(a))$

(3)

Des définitions et des exemples donnés, on voit que le raffinement d'une substitution a principalement deux effets :

- diminution du non déterminisme
- affaiblissement des préconditions

4.1.2 Raffinement avec changement de représentation

Le raffinement en B permet également le *changement de l'espace* des variables. Ce point est essentiel car en général, on part d'une spécification riche en variables d'états "mathématiques" (ensembles, relations, etc.) et l'on aboutit, par raffinements successifs, à une "implémentation" où il y a peut-être davantage de variables, mais celles-ci sont simplifiées et sont proches des structures informatiques usuelles (tableaux, scalaires, etc.). De plus, l'état des machines contient des informations utiles pour le spécifieur et explicite le "quoi" alors que l'état des implémentations ne contient que l'information indispensable au calcul, qui est caractéristique du "comment".

Le raffinement simple de substitution se place dans un cas où S et T portent sur le même espace de variables. On suppose maintenant que le raffinement fait passer d'un espace de variables x, défini par l'ensemble de valeurs b ($x \in b$), à un ensemble de variables y, défini sur c ($y \in c$). La relation entre les deux espaces doit permettre de faire correspondre chaque valeur concrète avec au moins une valeur abstraite :

$v \;\in\; c \leftrightarrow b \;\wedge\; \mathsf{dom}(v) = c$

(C-Raffinement)

D'une manière ensembliste, le raffinement avec changement de variables caractérisé par la relation v, notée $\sqsubseteq_v$, où v satisfait la condition *C-Raffinement*, est défini par :

$S \sqsubseteq_v T$	$v^{-1}\,[\mathsf{pre}(S)] \subseteq \mathsf{pre}(T)$ $v^{-1} \;;\; \mathsf{rel}(T) \subseteq \mathsf{rel}(S) \;;\; v^{-1}$

(4)

On peut se ramener à des prédicats en posant $v = \{\ (y, x) \mid L\ \}$. La relation de raffinement qui dépend d'un prédicat L est notée $\sqsubseteq_L$. La définition (4) est alors équivalente à :

$S \sqsubseteq_L T$	$L \;\wedge\; \mathsf{trm}(S) \;\Rightarrow\; \mathsf{trm}(T)$ $L \;\wedge\; \mathsf{prd}_y(T) \;\Rightarrow\; \exists x' \cdot (\mathsf{prd}_x(S) \;\wedge\; [x, y := x', y']\, L)$

(5)

Finalement, le raffinement de substitutions avec changement de variables peut s'exprimer directement en terme des substitutions généralisées, par la définition suivante :

$S \sqsubseteq_L T$	$L \ \wedge \ \mathsf{trm}(S) \ \Rightarrow \ [T] \neg [S] \neg L$

(6)

L'équivalence entre les définitions (5) et (6) peut être établie à partir des formes normalisées des substitutions S et T. En utilisant la forme normalisée de T la formule (6) peut se réécrire en :

$$\begin{array}{ll} (a) & L \ \wedge \ \mathsf{trm}(S) \ \Rightarrow \ \mathsf{trm}(T) \\ (b) & L \ \wedge \ \mathsf{trm}(S) \ \wedge \ \mathsf{prd}_y(T) \ \Rightarrow \ [y := y'] \, (\neg [S] \neg L) \end{array}$$

La formule (a) correspond à la première condition de la définition (5). Pour (b), la forme normalisée de la substitution S donne les réécritures suivantes :

$$\begin{array}{lcl} [S] \neg L & \Leftrightarrow & (\mathsf{trm}(S) \ \wedge \ \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x'] \neg L)) \\ \neg [S] \neg L & \Leftrightarrow & (\mathsf{trm}(S) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(S) \wedge [x := x'] \, L)) \end{array}$$

D'où (b) devient :

$$L \ \wedge \ \mathsf{trm}(S) \ \wedge \ \mathsf{prd}_y(T) \ \Rightarrow \ \exists x' \cdot (\mathsf{prd}_x(S) \wedge [x, y := x', y'] \, L)$$

Pour montrer l'équivalence avec la seconde condition de la définition (5), nous allons établir que l'implication $L \ \wedge \ \neg \, \mathsf{trm}(S) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(S) \wedge [x, y := x', y'] \, L)$ est toujours vraie. En effet :

$\forall x' \cdot (\mathsf{trm}(S) \ \vee \ \mathsf{prd}_x(S))$	propriété des substitutions gén.
$\neg \, \mathsf{trm}(S) \Rightarrow \forall x' \cdot \mathsf{prd}_x(S)$	conséquence logique
$L \Rightarrow \forall y' \cdot \exists x' \cdot [x, y := x', y'] \, L$	relation L totale
$L \ \wedge \ \neg \, \mathsf{trm}(S) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(S) \wedge [x, y := x', y'] \, L)$	par composition

Notons enfin que la relation de raffinement simple $\sqsubseteq$ est le cas particulier de la relation $\sqsubseteq_v$ où v est l'identité.

4.1.3 Raffinement pas à pas

Une propriété intéressante du raffinement est la transitivité, qui permet de construire un raffinement par étape et, par là même, de couper la complexité du développement et de la preuve. Cette propriété s'énonce par :

$$S \sqsubseteq_{v_1} T \ \wedge \ T \sqsubseteq_{v_2} U \ \Rightarrow \ S \sqsubseteq_{v_2 ; v_1} U$$

Si la spécification R_1 est raffinée en R_2 pour la relation $\{(x_2, x_1) \mid J_1\}$, et si cette spécification est elle-même raffinée en R_3 pour la relation $\{(x_3, x_2) \mid J_2\}$ alors on peut déduire que la spécification R_1 est raffinée par R_3 pour la relation $\{(x_3, x_1) \mid \exists x_2 \cdot (J_1 \wedge J_2)\}$.

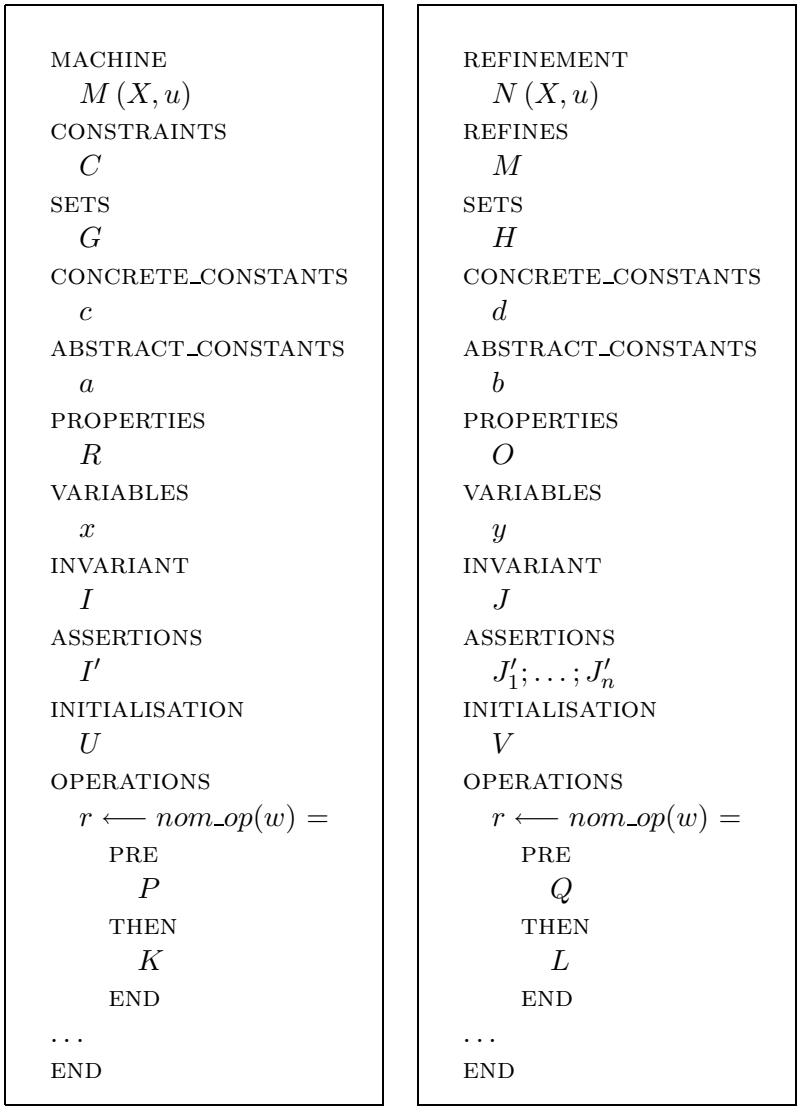


FIG. 4.1 – Machine et raffinement

4.1.4 Raffinement des machines

Un raffinement se présente sous la forme d'un incrément (un *delta*) par rapport à une spécification. Il ne correspond donc pas à un composant indépendant. Il permet le raffinement des données et des traitements. Nous donnons dans la figure 4.1 la forme d'un raffinement N par rapport à une machine M.

La clauses REFINES établit le lien entre le raffinement et sa spécification abstraite. A part cette clause, les composants REFINEMENT contiennent les mêmes rubriques que les machines.

Les paramètres doivent être répétés dans l'entête. Ils gardent la même spécification que celle donnée dans la machine initiale et donc, la clause CONSTRAINTS ne doit pas être répétée.

Les ensembles G sont hérités dans le raffinement. Les ensembles H du raffinement

sont de "nouveaux" ensembles qui s'ajoutent à ceux hérités de la machine. Les constantes concrètes c sont aussi héritées avec leurs propriétés R. Les constantes abstraites de la machine M, en revanche, ne sont pas héritées. Elles peuvent être raffinées par des constantes de N (concrètes d ou abstraites b) et la relation de liaison se fait dans le prédicat de propriété O. Les constantes concrètes d du raffinement sont de "nouvelles" constantes, qui s'ajoutent à celles héritées de la machine.

Les variables sont des variables abstraites ou concrètes. Les variables concrètes sont héritées comme les constantes concrètes. Les variables abstraites sont raffinées par des variables concrètes ou abstraites du raffinement. Des variables abstraites peuvent être communes à la spécification et au raffinement (les variables concrètes, qui ne doivent pas être répétées dans le raffinement, sont considérées comme faisant partie de ce cas). Cette possibilité est une facilité syntaxique qui permet d'alléger notablement l'écriture d'un raffinement.

L'invariant J spécifie les variables y et définit *la liaison* entre les variables x de la machine M et les variables y du raffinement. La relation de raffinement v entre les états concrets et abstraits est la suivante :

$$v \ = \ \{(y, x) \mid I \wedge J\}$$

Si les listes de variables x et y contiennent des noms identiques, disons z, alors dans les obligations de preuves, ces variables sont renommées dans N en $z\$1$ et l'invariant J est complété par l'égalité $z = z\$1$[1].

Les assertions de la machine I' sont une liste d'assertions qui sont vues dans les obligations de preuves comme une conjonction de prédicats. Les assertions du raffinement J' sont aussi une liste d'assertions qu'on a explicité dans la figure 4.1.

La liste des opérations de la machine M doit être la même que la liste des opérations du raffinement N, avec les mêmes entêtes (liste de résultats, nom de l'opération et liste de paramètres). Si une opération garde la même définition, il n'est pas nécessaire de la redéfinir.

Les noms des constantes et des variables abstraites de la machine M ne sont visibles dans le raffinement N que dans les clauses INVARIANT et ASSERTIONS et dans le prédicat de la substitution ASSERT. Si le raffinement est une implémentation (chapitre 5) alors ces constantes et variables sont aussi visibles dans les variants et invariants de boucle.

Un raffinement peut être raffiné à son tour pour construire une chaîne de raffinements.

4.2 Validation des raffinements

4.2.1 Forme des obligations de preuve

Les preuves de raffinement d'un composant suivent le même principe que les preuves d'une machine (paragraphe 3.2.2). Les obligations de preuve supposent que les listes de variables soient disjointes. Si tel n'est pas le cas, il y a renommage comme indiqué précédemment.

Il doit être établi que les assertions sont des conséquences des déclarations et de l'invariant, que l'initialisation concrète est un raffinement de l'initialisation abstraite et que

[1] C'est grâce à cela que l'on peut identifier les variables abstraites et concrètes au niveau de la description des obligations de preuve et de la machine équivalente à un raffinement.

chaque raffinement d'opération est correct. Les obligations de preuves du composant N sont données ci-après, en reprenant les notations de la figure 4.1. A représente les propriétés sur les paramètres, B les propriétés sur les ensembles et les constantes de la machine et B_r les propriétés sur les constantes et les ensembles du raffinement.

Obligations de preuve relative aux assertions :

$$A \wedge B \wedge B_r \wedge I \wedge I' \wedge J \Rightarrow J'_1$$
$$A \wedge B \wedge B_r \wedge I \wedge I' \wedge J \wedge J'_1 \ldots \wedge J'_{k-1} \Rightarrow J'_k$$

Obligation de preuve de l'initialisation :

$$A \wedge B \wedge B_r \Rightarrow [V] \neg[U] \neg J$$

Obligation de preuve pour chaque opération :

$A \wedge B \wedge B_r \wedge I \wedge I' \wedge J \wedge J' \wedge P \Rightarrow Q$	précondition
$A \wedge B \wedge B_r \wedge$ $I \wedge I' \wedge J \wedge J' \wedge P \Rightarrow [L] \neg[K] \neg J$	opération sans résultat
$A \wedge B \wedge B_r \wedge$ $I \wedge I' \wedge J \wedge J' \wedge P \Rightarrow [[r := r']L] \neg[K] \neg(J \wedge r = r')$	opération avec résultat

La notation $[r := r']\,L$ signifie le renommage de r par r' dans la subsitution L. L'application des substitutions aux substitutions est détaillé au paragraphe 6.2.2.

Dans les obligations de preuve des opérations, on voit que la précondition des opérations de la machine est mise en hypothèse des preuves. Une démarche classique est donc de ne pas mettre de précondition dans les opérations du raffinement (c'est-à-dire une précondition implicite à btrue), ce qui a pour effet de supprimer l'obligation de preuve sur les préconditions.

Dans le cas d'une chaîne de raffinements les hypothèses contiennent tous les invariants et assertions introduits dans les niveaux supérieurs (sauf pour l'initialisation).

4.2.2 Machine abstraite associée à un raffinement

Les raffinements s'écrivent comme des incréments par rapport aux spécifications de plus haut niveau. Comme nous l'avons dit, ceci évite de répéter certaines informations. Il est néanmoins possible de construire un composant complet qui correspond à un raffinement. Dans la méthode B, cette possibilité n'est que théorique. Nous donnons dans la figure 4.2 la machine correspondant au raffinement de la figure 4.1. Nous supposons que pour les noms initialement communs aux variables abstraites de la machine et du raffinement et pour les variables concrètes, le mécanisme de renommage indiqué au paragraphe 4.1.4 a été appliqué. Donc les listes de variables x et y considérées ici sont disjointes.

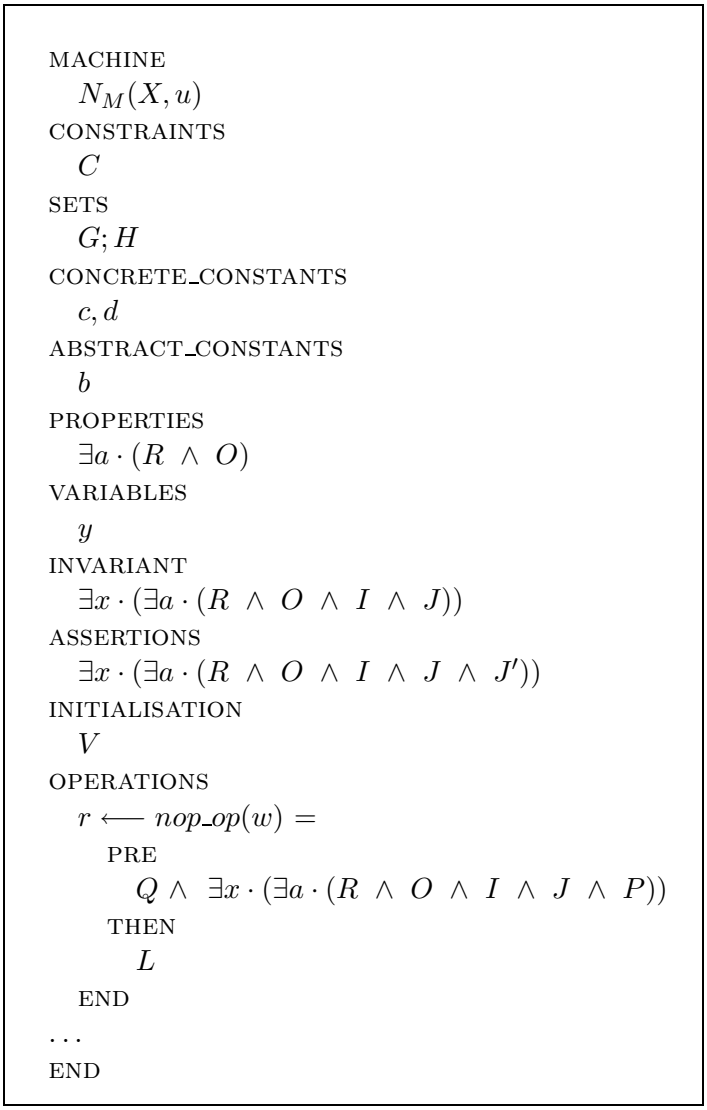


FIG. 4.2 – Machine construite à partir d'un raffinement

On peut déduire des obligations de preuve de la machine M et du raffinement N que le composant N_M est valide, c'est-à-dire que la formule $\exists x \cdot (\exists a \cdot (R \wedge O \wedge I \wedge J))$ est bien un invariant de cette machine.

4.3 Application à l'exemple de la réservation

On reprend l'exemple de réservation (paragraphe 3.3.1). Le raffinement est donné à la figure 4.3. L'état est constitué d'une fonction totale qui, à chaque numéro de siège associe une valeur booléenne. L'ensemble des sièges *occupes* (variable de la machine à raffiner) est exactement l'ensemble des places ayant la valeur TRUE par la fonction *etat*. Cette relation constitue l'invariant de liaison entre la machine *RESERVATION* et son raffinement. Il s'agit d'un raffinement de données. A l'initialisation, tous les sièges sont libres (requis

par l'initialisation de la machine abstraite). L'opération *liberer* met systématiquement la valeur de l'état du siège à FALSE.

```
REFINEMENT
    RESERVATION1
REFINES
    RESERVATION
VARIABLES
    etat, nb_libre
INVARIANT
    etat ∈ SIEGES → BOOL          /* chaque siège a une valeur booléenne */
     ∧ occupes = etat⁻¹[{TRUE}]   /* invariant de liaison */
INITIALISATION
    nb_libre := nb_max ||
    etat := SIEGES × {FALSE}
OPERATIONS
    /* l'opération nb_libre est heritée sans changement */

    place ⟵ reserver =            /* réserver une place */
        ANY pp1 WHERE
            pp1 ∈ etat⁻¹[{FALSE}]
        THEN
            place, etat(pp1), nb_libre := pp1, TRUE, nb_libre − 1
        END ;
    rr ⟵ liberer(place) =         /* libérer place */
        BEGIN
            IF etat(place) = TRUE THEN
                rr, nb_libre := TRUE, nb_libre + 1
            ELSE
                rr := FALSE
            END
            || etat(place) := FALSE
        END
END
```

FIG. 4.3 – Raffinement de *RESERVATION*

L'invariant du raffinement est celui qui est écrit dans le texte, auquel le système ajoute l'égalité entre la variable *nb_libre* de la machine et *nb_libre*\$1 qui est celle, renommée, du raffinement. De plus, la déclaration d'une fonction totale est transformée en une fonction partielle dont le domaine est complet :

$$J \;\Leftrightarrow\; etat \in SIEGES \nrightarrow BOOL \;\wedge\; \mathsf{dom}(etat) = SIEGES \\ \wedge\; occupes = etat^{-1}[\{\mathsf{TRUE}\}] \;\wedge\; nb\_libre = nb\_libre\$1$$

Les hypothèses des constantes sont héritées :

$$H \;\Leftrightarrow\; nb\_max \in 1..\mathsf{maxint} \;\wedge\; SIEGES = 1..nb\_max$$

Initialisation

L'obligation de preuve de l'initialisation est :

$$H \;\Rightarrow\; [nb\_libre := nb\_max \parallel etat := SIEGES \times \{\mathsf{FALSE}\}]\; \neg[occupes, nb\_libre := \emptyset, nb\_max]\,\neg J$$

qui produit la formule à prouver :

$$\begin{array}{rl} H \;\Rightarrow\; & SIEGES \times \{\mathsf{FALSE}\} \in SIEGES \mathrel{+\!\!\!\rightarrow} BOOL \\ & \wedge\; \mathsf{dom}(SIEGES \times \{\mathsf{FALSE}\}) = SIEGES \\ & \wedge\; \emptyset = (SIEGES \times \{\mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}] \\ & \wedge\; nb\_max = nb\_max \end{array}$$

Opération "reserver"

L'opération raffinée n'a pas de précondition, donc la première obligation de preuve est trivialement vérifiée. L'obligation de preuve pour le corps de *reserver* est :

$$H \;\wedge\; J \;\wedge\; nb\_libre \neq 0 \;\Rightarrow\; [reserver_C]\,\neg[reserver_A]\,\neg(J \;\wedge\; place = place')$$

Les corps des opérations sous forme de substitutions primitives mathématiques sont :

$$\begin{array}{rcl} reserver_A & = & @pp \cdot pp \in SIEGES - occupes \Longrightarrow \\ & & \quad place, occupes, nb\_libre := pp, occupes \;\cup \{pp\}, nb\_libre - 1 \\ reserver_C & = & @pp1 \cdot pp1 \in etat^{-1}[\{\mathsf{FALSE}\}] \Longrightarrow \\ & & \quad place', etat(pp1), nb\_libre\$1 := pp1, \mathsf{TRUE}, nb\_libre\$1 - 1 \end{array}$$

On note que $etat(pp1) := \mathsf{TRUE}$ est équivalent à $etat := etat \mathrel{<\!\!+} \{pp1 \mapsto \mathsf{TRUE}\}$. On utilise la règle de transformation : $\exists\, x \cdot (Q \;\wedge\; x = e) \;\Leftrightarrow\; [x := e]\, Q$. En appliquant les règles du calcul des substitutions et les règles des opérateurs logiques, on obtient l'obligation de preuve :

$$\begin{array}{l} nb\_max \in 1..\mathsf{maxint} \;\wedge\; SIEGES = 1..nb\_max \\ \;\wedge\; etat \in SIEGES \mathrel{+\!\!\!\rightarrow} BOOL \;\wedge\; \mathsf{dom}(etat) = SIEGES \\ \;\wedge\; occupes = etat^{-1}[\{\mathsf{TRUE}\}] \;\wedge\; nb\_libre = nb\_libre\$1 \\ \;\wedge\; nb\_libre \neq 0 \\ \Rightarrow \\ \forall\, pp1 \cdot (pp1 \in etat^{-1}[\{\mathsf{FALSE}\}] \Rightarrow \\ \quad pp1 \in SIEGES - occupes \\ \quad \wedge\; (etat \mathrel{<\!\!+} \{pp1 \mapsto \mathsf{TRUE}\}) \in SIEGES \mathrel{+\!\!\!\rightarrow} BOOL \\ \quad \wedge\; \mathsf{dom}(etat \mathrel{<\!\!+} \{pp1 \mapsto \mathsf{TRUE}\}) = SIEGES \\ \quad \wedge\; occupes \;\cup \{pp1\} = (etat \mathrel{<\!\!+} \{pp1 \mapsto \mathsf{TRUE}\})^{-1}[\{\mathsf{TRUE}\}] \\ \quad \wedge\; nb\_libre - 1 = nb\_libre\$1 - 1) \end{array}$$

Opération "liberer"

L'opération *liberer* se traite de la même façon. Le paramètre *place* apparaît comme une nouvelle variable libre. L'obligation de preuve est :

$$H \;\wedge\; J \;\wedge\; place \in SIEGES \;\Rightarrow\; [liberer_C]\,\neg[liberer_A]\,\neg(J \;\wedge\; rr = rr')$$

Les corps des opérations sous forme de substitutions primitives mathématiques sont :

$$
\begin{array}{lcl}
liberer_A & = & (place \in occupes \implies \\
 & & \quad rr, occupes, nb\_libre := \mathsf{TRUE}, occupes - \{place\}, nb\_libre + 1) \\
 & & [\!] \ (place \notin occupes \implies rr := \mathsf{FALSE}) \\
liberer_C & = & (etat(place) = \mathsf{TRUE} \implies \\
 & & \quad rr', nb\_libre, etat(place) := \mathsf{TRUE}, nb\_libre + 1, \mathsf{FALSE}) \\
 & & [\!] \ (etat(place) \neq \mathsf{TRUE} \implies rr', etat(place) := \mathsf{FALSE}, \mathsf{FALSE})
\end{array}
$$

Les choix bornés dans l'opération et l'opération raffinée produisent un calcul combinatoire des obligations de preuve qui peut produire une explosion lorsqu'il y a trop de cas à traiter. Ici, on obtient les cas suivants sous les hypothèses $H \ \wedge \ J \ \wedge \ place \in SIEGES$:

$$
\begin{array}{l}
etat(place) = \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow \\
\quad [rr', nb\_libre\$1, etat(place) := \mathsf{TRUE}, nb\_libre\$1 + 1, \mathsf{FALSE}] \\
\quad\quad ([rr, occupes, nb\_libre := \mathsf{TRUE}, occupes - \{place\}, nb\_libre + 1] \, (J \wedge rr = rr'))
\end{array}
$$

$$
\begin{array}{l}
etat(place) = \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow \\
\quad [rr', nb\_libre\$1, etat(place) := \mathsf{TRUE}, nb\_libre\$1 + 1, \mathsf{FALSE}] \\
\quad\quad ([rr := \mathsf{FALSE}] \, (J \wedge rr = rr'))
\end{array}
$$

$$
\begin{array}{l}
etat(place) \neq \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow \\
\quad [rr', etat(place) := \mathsf{FALSE}, \mathsf{FALSE}] \\
\quad\quad ([rr, occupes, nb\_libre := \mathsf{TRUE}, occupes - \{place\}, nb\_libre + 1] \, (J \wedge rr = rr'))
\end{array}
$$

$$
\begin{array}{l}
etat(place) \neq \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow \\
\quad [rr', etat(place) := \mathsf{FALSE}, \mathsf{FALSE}]([rr := \mathsf{FALSE}] \, (J \wedge rr = rr'))
\end{array}
$$

D'où les obligations de preuves (sans simplification) :

$$
\begin{array}{l}
etat(place) = \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow \\
\quad\quad etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL \\
\quad \wedge \ \mathsf{dom}(etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\}) = SIEGES \\
\quad \wedge \ occupes - \{place\} = (etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}] \\
\quad \wedge \ nb\_libre + 1 = nb\_libre\$1 + 1 \\
\quad \wedge \ \mathsf{TRUE} = \mathsf{TRUE}
\end{array}
$$

$$
\begin{array}{l}
etat(place) = \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow \\
\quad\quad etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL \\
\quad \wedge \ \mathsf{dom}(etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\}) = SIEGES \\
\quad \wedge \ occupes = (etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}] \\
\quad \wedge \ nb\_libre = nb\_libre\$1 + 1 \\
\quad \wedge \ \mathsf{FALSE} = \mathsf{TRUE}
\end{array}
$$

$$
\begin{array}{l}
etat(place) \neq \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow \\
\quad etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL \\
\quad \wedge \ \mathsf{dom}(etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\}) = SIEGES \\
\quad \wedge \ occupes - \{place\} = (etat \lhd\!\!+ \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}] \\
\quad \wedge \ nb\_libre + 1 = nb\_libre\$1 \\
\quad \wedge \ \mathsf{TRUE} = \mathsf{FALSE}
\end{array}
$$

$$
\begin{array}{l}
etat(place) \neq \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow \\
\quad etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL \\
\quad \wedge \ \mathsf{dom}(etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\}) = SIEGES \\
\quad \wedge \ occupes = (etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}] \\
\quad \wedge \ nb\_libre = nb\_libre\$1 \\
\quad \wedge \ \mathsf{FALSE} = \mathsf{FALSE}
\end{array}
$$

Après simplification, les obligations de preuves non évidentes à prouver sont les suivantes dans l'atelier B. Les OP de 6 à 11 ont des hypothèses contradictoires, à cause de l'invariant : $occupes = etat^{-1}[\{\mathsf{TRUE}\}]$. Les OP avec hypothèses contradictoires sont vérifiées à cause de l'équivalence $(\mathsf{bfalse} \Rightarrow R) \ \Leftrightarrow \ \mathsf{btrue}$.

1− $etat(place) = \mathsf{TRUE} \Rightarrow$
$etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL$

2− $etat(place) = \mathsf{TRUE} \Rightarrow$
$\mathsf{dom}(etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\}) = SIEGES$

3− $etat(place) \neq \mathsf{TRUE} \Rightarrow$
$etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\} \in SIEGES \leftrightarrow BOOL$

4− $etat(place) \neq \mathsf{TRUE} \Rightarrow$
$\mathsf{dom}(etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\}) = SIEGES$

5− $etat(place) = \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow$
$occupes - \{place\} = (etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$

6− $etat(place) = \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow$
$occupes = (etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$

7− $etat(place) = \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow$
$nb\_libre = nb\_libre\$1 + 1$

8− $etat(place) = \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow$
$\mathsf{FALSE} = \mathsf{TRUE}$

9− $etat(place) \neq \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow$
$occupes - \{place\} = (etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$

10− $etat(place) \neq \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow$
$nb\_libre + 1 = nb\_libre\$1$

11− $etat(place) \neq \mathsf{TRUE} \ \wedge \ place \in occupes \Rightarrow$
$\mathsf{TRUE} = \mathsf{FALSE}$

12− $etat(place) \neq \mathsf{TRUE} \ \wedge \ place \notin occupes \Rightarrow$
$occupes = (etat \lhd\!\!\!- \{place \mapsto \mathsf{FALSE}\})^{-1}[\{\mathsf{TRUE}\}]$

Chapitre 5

Implémentation

Résumé

Les machines d'implémentation sont le dernier maillon d'une chaîne de raffinements. On donne les conditions syntaxiques que doivent vérifier les modules d'implémentation. On termine en donnant les implémentations des exemples utilisés au long des chapitres précédents.

5.1 Conditions syntaxiques d'une implémentation

5.1.1 Variables concrètes

Les variables concrètes peuvent apparaître à n'importe quel stade de développement. La différence avec les variables abstraites est qu'elles ne peuvent pas être raffinées (elles doivent traverser les niveaux de raffinement sans modification) et qu'elles peuvent être implémentées telles quelles. Pour qu'une telle réalisation soit possible, les variables concrètes doivent satisfaire des restrictions fortes de typage. Les variables concrètes, de même que les constantes concrètes, doivent avoir un typage de "type concret" qui peut être :

- un entier représentable (entier concret),
- une valeur booléenne,
- une valeur d'un ensemble abstrait (différé) ou énuméré,
- un "tableau".

Les trois premiers items constituent ce que l'on appelle des ensembles "simple concrets" ou "scalaires". Un ensemble différé est finalement instancié avec un ensemble d'entiers concrets (voir paragraphe 5.1.2). Par "tableau", on entend une fonction totale dont le domaine est un ensemble simple concret ou un produit d'ensembles simples concrets, et dont le codomaine est un ensemble simple concret. Cette fonction totale peut être de plus surjective, injective ou bijective. Le domaine est appelé le domaine des *indices* du tableau. Exemple de déclaration de variables concrètes :

CONCRETE_VARIABLES
nb, ind, TT
INVARIANT
$nb \in \mathsf{NAT} \ \wedge \ ind \in 0..10 \ \wedge$
$TT \in 1..10 \rightarrow 1..3$

Un autre exemple est fourni avec l'implémentation de la réservation (paragraphe 5.2.2).

5.1.2 Valuation des ensembles et des constantes concrètes

Les constantes concrètes et les ensembles différés sont hérités depuis la machine initiale dans la chaîne de raffinements. Un exemple de déclaration d'ensemble et de constantes est :

SETS
$PERSONNES$
CONCRETE_CONSTANTS
$pp, dernier, ff$
PROPERTIES
$pp \in PERSONNES \ \wedge \ dernier \in \mathsf{NAT}_1 \ \wedge$
$ff \in 0..3 \rightarrow \mathsf{NAT}_1$

Ces entités concrètes doivent recevoir une valeur au moment de l'implémentation. Cela se fait par une clause spéciale, appelée "VALUES". On ne rentre pas dans les détails des possibilités d'évaluation des ensembles et des constantes (voir le manuel de référence). Les valeurs acceptables pour instancier des ensembles différés sont des intervalles d'entiers. Les valeurs des constantes doivent être des valeurs du type concret déclaré pour ces constantes. Il existe en B des notations de fonctions constantes (voir le paragraphe 1.2.5).

Exemple : VALUES
$PERSONNES = 1..20\,000$;
$pp = 1$;
$dernier = 20\,000$;
$ff = \{0 \mapsto 0, 1 \mapsto 10, 2 \mapsto 21, 3 \mapsto 32\}$

La valuation des constantes génère une obligation de preuve pour s'assurer que la valuation est correcte par rapport à la spécification. Cette obligation de preuve assure également qu'il existe des valeurs possibles pour les constantes abstraites des niveaux précédents. On reprend les notations des machines et raffinements de la figure 4.1. Pour la machine et chaque niveau de raffinement, on note D_i le prédicat qui rassemble les propriétés implicites et explicites des ensembles et des constantes déclarées. Pour la machine M, cela donne $D_1 \ \Leftrightarrow \ A \wedge B \wedge R$, et ainsi de suite dans les raffinements successifs juqu'à l'implémentation où la spécification des ensembles et des constantes concrètes est notée D_n. Rappelons que les ensembles et les constantes ne peuvent pas dépendre des paramètres de la machine. Si l'on note W la substitution correspondant à la clause VALUES (il suffit de remplacer "=" par " :="), cette clause fait la substitution des ensembles différés et de toutes les constantes concrètes de la chaîne de raffinement par leur valeur. Les constantes abstraites rencontrées au cours du développement sont notées $a, b, \ldots$. L'obligation de preuve générée dans le composant d'implémentation est alors :

$$\exists(a, b, \ldots) \cdot [W] \, (D_1 \ \wedge \ D_2 \ \wedge \ \ldots \ \wedge \ D_n)$$

5.1.3 Substitutions et prédicats valides

Au niveau d'une implémentation, on ne peut utiliser que des substitutions déterministes, que l'on appelle des "instructions". L'équivalence avec un programme habituel est quasi immédiate. Les expressions E sont des expressions arithmétiques, des constantes d'énumération

ou des variables. Dans l'affectation $f(E_1) := E_2$, f est l'identificateur d'un tableau. La liste des substitutions (c'est-à-dire instructions) admises est :

substitution simple ou affectation	$x := E$ $f(E_1) := E_2$ $b := \mathsf{bool}(C)$
appel d'opération	$x_1, \ldots \longleftarrow nom\_op(E, \ldots)$
instruction vide	skip
instruction fermée	BEGIN I END
instruction bloc	VAR $y_1, \ldots, y_k$ IN I END
séquence	I_1 ; I_2
instruction "si" (cf. paragraphe 2.2.7)	IF C THEN I_1 ELSE I_2 END IF C THEN I_1 END IF C THEN I_1 ELSIF ...END
instruction "cas" (cf. paragraphe 2.2.7)	CASE E OF EITHER E_1 THEN I_1 ... etc.
itération	WHILE C DO I INVARIANT J VARIANT V END

Les prédicats sont réduits aux "conditions" C qui sont des expressions évaluables opérationnellement. Ce sont des conditions simples de la forme :

$C_1 \wedge C_2$, $C_1 \vee C_2$, $\neg C$
E_1 *oprel* E_2 avec *oprel* de la forme $\geq, \leq, <, >, =, \neq$ et E_1 et E_2 des identificateurs ou des constantes.

Le type des entiers représentable est INT. Les opérations arithmétiques $+, -, *, /$ sont des opérations *sans débordement* dans ce type. Les opérations de l'implémentation génèrent des obligations de preuve qui assurent que le résultat des opérations arithmétiques ne déborde pas de l'intervalle minint..maxint.

5.1.4 Opérations locales

Dans une implémentation, il est possible de définir des *opérations locales* auxiliaires. Ces opérations ne sont pas visibles de l'extérieur. Elles ne peuvent être appelées que dans le corps des opérations de l'implémentation (évidemment sans récursivité).

Les opérations locales sont spécifiées dans une rubrique appelée LOCAL_OPERATIONS. La spécification doit se conformer à l'écriture des opérations dans une machine (paragraphes 3.1.3, 3.1.4). Pour ces opérations locales, on doit associer immédiatement une implémentation dans la partie OPERATIONS qui doit être un raffinement de la spécification au sens habituel (obligations de preuves).

L'invariant de l'implémentation n'est pas supposé vérifié au moment de l'appel des opérations locales. Seul le "typage" au sens B des variables de l'état est mis en hypothèse des obligations de preuve du raffinement. L'utilisateur peut indiquer les propriétés qu'il sait être vraies à l'appel des opérations locales dans leur précondition.

On donne un exemple d'opération locale dans le paragraphe 5.2.3 d'implémentation du tri du tableau.

5.2 Exemples d'implémentation de machines

5.2.1 Implémentation et preuve de la division entière

On donne maintenant l'algorithme de la division entière sous forme d'un programme avec boucle. Cet algorithme a été spécifié au paragraphe 3.3.2. L'idée de l'implémentation vient de la relation invariante :

$$bb * ss + tt \;=\; bb * (ss + 1) + (tt - bb)$$

lorsque $tt \geq bb$. Si l'on commence avec $ss = 0$ et $tt = aa$, on a l'égalité : $aa \;=\; bb * 0 + aa$ qui est conservée par la modification simultanée de ss et tt, grâce à la première identité, jusquà ce que $bb > tt$. C'est la condition d'arrêt qui donne le résultat cherché dans ss et tt. L'algorithme est décrit à la figure 5.1.

Cette implémentation engendre des obligations de preuves. Ce sont celles décrites par le raffinement (paragraphe 4.2). On indice par 0 les abbréviations d'éléments de la machine et par 1 ceux de l'implémentation. De même, ss_0 et tt_0 sont les variables ss et tt de la machine et ss_1 et tt_1 celles de l'implémentation. En notant :

$$\begin{array}{lcl}
P_0 & \Leftrightarrow & aa \in \mathsf{NAT} \;\wedge\; bb \in \mathsf{NAT}_1 \\
Q_0 & \Leftrightarrow & ss_0 \in \mathsf{NAT} \;\wedge\; tt_0 \in \mathsf{NAT} \;\wedge\; aa = bb * ss_0 + tt_0 \;\wedge\; tt_0 < bb \\
T_1 & = & ss_1 := 0 \;;\; tt_1 := aa \\
B_1 & = & ss_1 := ss_1 + 1 \;;\; tt_1 := tt_1 - bb \\
J_1 & \Leftrightarrow & ss_1 \in \mathsf{NAT} \;\wedge\; tt_1 \in \mathsf{NAT} \;\wedge\; aa = bb * ss_1 + tt_1 \;\wedge\; tt_1 \geq 0 \\
V_1 & = & tt_1 \\
F_1 & = & qq := ss_1 \;;\; rr := tt_1
\end{array}$$

alors, la substitution de l'opération *divi* de la machine abstraite *DIVISION* (paragraphe 3.3.2) est :

$$(P_0 \mid @ss_0, tt_0 \cdot (Q_0 \implies qq, rr := ss_0, tt_0))$$

La substitution de l'opération du raffinement est :

```
IMPLEMENTATION
    DIVISION_I
REFINES
    DIVISION
OPERATIONS
    qq, rr ⟵ divi(aa, bb) =
        VAR ss, tt IN              /* variables locales auxiliaires : */
            ss := 0 ;              /* initialisations */
            tt := aa ;
            WHILE tt ≥ bb DO
                ss := ss + 1 ;     /* corps de la boucle */
                tt := tt − bb
            INVARIANT              /* conditions invariantes */
                ss ∈ NAT ∧ tt ∈ NAT ∧
                aa = bb ∗ ss + tt ∧ tt ≥ 0
            VARIANT                /* valeur entière qui décroît */
                tt
            END ;
            qq := ss ;             /* retour du résultat */
            rr := tt
        END
END
```

FIG. 5.1 – Implémentation de la division entière

$$@ss_1, tt_1 \cdot (T_1 \ ; \mathcal{W}(tt_1 \geq bb, B_1, J_1, V_1) \ ; F_1)$$

L'obligation de preuve du raffinement est :

$$\begin{array}{rcl} P_0 & \Rightarrow & [@ss_1, tt_1 \cdot (T_1 \ ; \mathcal{W}(tt_1 \geq bb, B_1, J_1, V_1) \ ; [qq, rr := qq', rr']\, F_1)] \\ & & \quad \neg\, [@ss_0, tt_0 \cdot (Q_0 \implies qq, rr := ss_0, tt_0)]\, \neg\, (qq = qq' \ \wedge \ rr = rr') \end{array}$$

On détaille le calcul en utilisant les règles données pour les substitutions appliquées à des prédicats (paragraphe 2.2.2). La réduction de la deuxième substitution donne :

$$\begin{array}{rcl} P_0 & \Rightarrow & \forall ss_1, tt_1 \cdot ([T_1 \ ; \mathcal{W}(tt_1 \geq bb, B_1, J_1, V_1) \ ; [qq, rr := qq', rr']\, F_1] \\ & & \qquad (\exists ss_0, tt_0 \cdot (Q_0 \ \wedge \ ss_0 = qq' \ \wedge \ tt_0 = rr'))) \end{array}$$

qui se simplifie en :

$$P_0 \ \Rightarrow \ \forall ss_1, tt_1 \cdot ([T_1 \ ; \mathcal{W}(tt_1 \geq bb, B_1, J_1, V_1) \ ; [qq, rr := qq', rr']\, F_1]\, Q_1)$$

avec $\begin{array}{rcl} Q_1 & \Leftrightarrow & [ss_0, tt_0 := qq', rr']\, Q_0 \\ & \Leftrightarrow & (qq' \in \mathsf{NAT} \ \wedge \ rr' \in \mathsf{NAT} \ \wedge \ aa = bb * qq' + rr' \ \wedge \ rr' < bb) \end{array}$

Et en substituant $[qq, rr := qq', rr']\, F_1$:

$$P_0 \quad \Rightarrow \quad \forall ss_1, tt_1 \cdot ([T_1 \ ; \mathcal{W}(tt_1 \geq bb, B_1, J_1, V_1)]\, Q_2)$$

avec $Q_2 \quad \Leftrightarrow \ (ss_1 \in \mathsf{NAT} \ \wedge \ tt_1 \in \mathsf{NAT} \ \wedge \ aa = bb * ss_1 + tt_1 \ \wedge \ tt_1 < bb)$

En appliquant la règle de la boucle avec initialisation (paragraphe 2.2.3) et en enlevant le quantificateur universel qui est appliqué par défaut à toute nouvelle variable dans une preuve, on obtient les obligations de preuve :

$$\begin{array}{rcl} P_0 & \Rightarrow & [T_1]\, J_1 \ \wedge \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & [B_1]\, J_1 \ \wedge \\ P_0 \ \wedge \ J_1 & \Rightarrow & V_1 \in \mathbb{N} \ \wedge \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & [n := V_1][B_1](V_1 < n) \ \wedge \\ P_0 \ \wedge \ J_1 \ \wedge \ \neg\, (tt_1 \geq bb) & \Rightarrow & Q_2 \end{array}$$

qui s'écrivent, d'une manière détaillée, en séparant les conjonctions de prédicats dans les conséquents :

Initialisation :

$$\begin{array}{rcl} P_0 & \Rightarrow & 0 \in \mathsf{NAT} \\ P_0 & \Rightarrow & aa \in \mathsf{NAT} \\ P_0 & \Rightarrow & aa = bb * 0 + aa \\ P_0 & \Rightarrow & aa \geq 0 \end{array}$$

Corps de boucle :

$$\begin{array}{rcl} P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & ss_1 + 1 \in \mathsf{NAT} \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & tt_1 - bb \in \mathsf{NAT} \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & aa = bb * (ss_1 + 1) + (tt_1 - bb) \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & tt_1 - bb \geq 0 \end{array}$$

Variant :

$$\begin{array}{rcl} P_0 \ \wedge \ J_1 & \Rightarrow & tt_1 \in \mathbb{N} \\ P_0 \ \wedge \ J_1 \ \wedge \ tt_1 \geq bb & \Rightarrow & tt_1 - bb_1 < tt_1 \end{array}$$

Sortie de boucle :

$$\begin{array}{rcl} P_0 \ \wedge \ J_1 \ \wedge \ \neg\, (tt_1 \geq bb) & \Rightarrow & ss_1 \in \mathsf{NAT} \\ P_0 \ \wedge \ J_1 \ \wedge \ \neg\, (tt_1 \geq bb) & \Rightarrow & tt_1 \in \mathsf{NAT} \\ P_0 \ \wedge \ J_1 \ \wedge \ \neg\, (tt_1 \geq bb) & \Rightarrow & aa = bb * ss_1 + tt_1 \\ P_0 \ \wedge \ J_1 \ \wedge \ \neg\, (tt_1 \geq bb) & \Rightarrow & tt_1 < bb \end{array}$$

Ces obligations de preuve sont en général faciles à prouver. Il est intéressant de remarquer comment les informations sont transmises entre la machine abstraite et l'implémentation, de telle sorte que l'on retrouve la preuve traditionnelle des programmes algorithmiques.

5.2.2 Implémentation de la réservation

On termine les raffinements de la machine *RESERVATION* (paragraphes 3.3.1, 4.3) par une implémentation. Toutes les variables deviennent des variables concrètes de même nom qui sont donc égales aux variables du dernier raffinement. L'invariant est inutile. On remarque l'utilisation de la substitution :

$$\text{VAR } y_1, \ldots, y_k \text{ IN } S \text{ END}$$

qui joue le rôle d'un bloc d'instructions avec déclarations locales. Les variables sont typées au moment de leur utilisation en partie gauche d'affectation. Dans les opérations, on ne donne que *reserver*. Les autres opérations sont quasiment identiques à celles du raffinement : les substitutions multiples deviennent une séquence de substitutions simples (en respectant l'ordre des modifications).

Dans cette opération, on choisit de rechercher la première place libre. La variable *ind* parcourt les numéros de places à partir de 1 et s'arrête dès que la place en question est libre :

$etat(ind) = \mathsf{FALSE}$. Le variant de la boucle est simplement le nombre de sièges qui n'ont pas encore été testés. La deuxième ligne de l'invariant : $etat[1..ind-1] \subseteq \{\mathsf{TRUE}\}$ indique qu'on n'a pas encore trouvé de place libre. Donc, grâce à la précondition de l'opération : $nb\_libre \neq 0$, on sait qu'il en existe toujours bien au moins une et que le test de sortie deviendra finalement vrai.

Dans l'opération *reserver*, la variable *val* est nécessaire car les arguments d'un opérateur de comparaison (dans le test d'entrée dans la boucle) ne peuvent pas être des expressions complexes (paragraphe 5.1.3) comme l'accès à un élément de tableau.

```
IMPLEMENTATION
    CODE_RESERVATION
REFINES
    RESERVATION1
VALUES
    nb_max = 4 ;                    /* valuation des constantes */
    SIEGES = 1..4
CONCRETE_VARIABLES
    etat, nb_libre
INITIALISATION
    nb_libre := nb_max ;
    etat := SIEGES × {FALSE}
OPERATIONS
    place ⟵ reserver =
        VAR
            ind, val                /* variables auxiliaires locales */
        IN
            ind := 1 ;              /* initialisation des variables */
            val := etat(ind) ;
            WHILE val = TRUE DO
                ind := ind + 1 ;    /* recherche d'une place libre */
                val := etat(ind)
            INVARIANT
                ind ∈ 1..nb_max  ∧  val = etat(ind)
                 ∧ etat[1..ind − 1] ⊆ {TRUE}
            VARIANT
                nb_max − ind
            END ;
            etat(ind) := TRUE ;     /* mises à jour de l'état */
            nb_libre := nb_libre − 1 ;
            place := ind            /* valeur résultat */
        END ;
        ...
END
```

FIG. 5.2 – Implémentation de la machine de réservation

5.2.3 Implémentation de l'algorithme de tri

On a vu au paragraphe 3.3.3 la spécification d'un tri de tableau. On va réaliser une implémentation à l'aide de l'algorithme de recherche du minimum. L'idée est de découper le tableau à trier TT en deux parties : entre les indices 1 et $jj-1$ ($1 \leq jj$), on a un sous-tableau trié. La formulation mathématique est :

$$\forall (i_1, i_2) \cdot (i_1 \in 1..jj-1 \wedge i_2 \in 1..jj-1 \wedge i_1 < i_2 \Rightarrow TT(i_1) \leq TT(i_2))$$

Les valeurs du tableau pour les indices entre 1 et $jj-1$ sont toutes inférieures (ou égales) à celles des indices compris entre jj et nn (nn est l'indice maximum), c'est-à-dire :

$$\forall (i_3, i_4) \cdot (i_3 \in 1..jj-1 \wedge i_4 \in jj..nn \Rightarrow TT(i_3) \leq TT(i_4))$$

```
trier =
    VAR jj, ll IN
        jj := 1;
        ll := nn − 1 ;
        WHILE jj ≤ ll DO            /* on ne trie pas le dernier élément */
            VAR kk, tmj IN          /* on pourrait tester si kk ≠ jj */
                kk ⟵ trouver_min(jj);
                tmj := TT(jj);      /* élément à déplacer */
                TT(jj) := TT(kk);   /* élément minimum */
                TT(kk) := tmj       /* échange des valeurs du tableau */
            END ;
            jj := jj + 1
        INVARIANT                   /* TT est la valeur du tableau dans la boucle */
            jj ∈ 1..nn ∧            /* TT$0 est la valeur du tableau d'entrée */
            TT ∈ 1..nn → NAT ∧
            (∀(i1, i2) · (i1 ∈ 1..jj − 1 ∧ i2 ∈ 1..jj − 1 ∧ i1 < i2 ⇒ TT(i1) ≤ TT(i2))) ∧
            (∀(i3, i4) · (i3 ∈ 1..jj − 1 ∧ i4 ∈ jj..nn ⇒ TT(i3) ≤ TT(i4))) ∧
            (∃hh · (hh ∈ 1..nn ⤖ 1..nn ∧ (hh ; TT) = TT$0))
        VARIANT
            nn − jj
        END
    END
```

FIG. 5.3 – Algorithme du tri d'un tableau

Parmi les variantes de spécification, on pourrait dire aussi que les éléments du sous-tableau $jj..nn$ sont plus grands que le dernier élément trié $TT(jj-1)$ avec $jj > 1$. Cela conduirait à des preuves différentes. La deuxième assertion serait alors :

$$\forall\, i_5 \cdot (i_5 \in jj..nn \Rightarrow TT(jj-1) \leq TT(i_5))$$

On pourrait aussi dire, en une seule assertion que les éléments entre 1 et $jj-1$ sont plus petits que les éléments plus grands, ce qui conduit à exprimer la spécification par :

$$\forall (i_1, i_2) \cdot (i_1 \in 1..jj-1 \wedge i_2 \in 1..nn \wedge i_1 < i_2 \Rightarrow TT(i_1) \leq TT(i_2))$$

Le calcul qui permet d'avancer d'un indice dans la boucle principale est simplement le fait de rechercher l'élément minimum du sous-tableau $TT[jj..nn]$ qui se trouve à l'indice kk (opération *trouver_min*(jj)) et d'échanger les deux valeurs de tableau $TT(jj)$ et $TT(kk)$. Il est facile de voir que lorsqu'on arrive à l'indice $nn - 1$, le tableau entier est trié. La description de l'algorithme de tri est donné à la figure 5.3. Les assertions de l'invariant sont exactement les conditions formelles qui définissent un état intermédiaire de l'algorithme. De plus, on a toujours une permutation des valeurs (fonction hh) avec le tableau initial avant l'entrée de la boucle, noté $TT\$0$.

Pour compléter l'algorithme, il faut définir l'opération *trouver_min* qui est la recherche de l'indice du minimum de la tranche de tableau $TT[jj..nn]$. Cette opération est définie comme une opération locale. L'entête du module d'implémentation qui donne la spécification de l'opération de recherche du minimum est donné à la figure 5.4.

```
IMPLEMENTATION
    TRI1R
REFINES TRI1
VALUES                          /* valuation de la constante concrète */
    nn = 100
CONCRETE_VARIABLES              /* déclaration du tableau concret */
    TT
LOCAL_OPERATIONS                /* spécification des opérations locales */
    kk ⟵ trouver_min(jj) =
        PRE                     /* la précondition assure l'existence d'un */
            jj : 1..nn − 1 ∧    /* élément dans la tranche de tableau */
            TT ∈ 1..nn → NAT    /* à l'appel, TT est toujours un tableau */
        THEN
            ANY pp WHERE        /* spécification du minimum */
                pp ∈ jj..nn  ∧  TT(pp) = min(TT[jj..nn])
            THEN
                kk := pp
            END
        END
...
END
```

FIG. 5.4 – Entête du composant d'implémentation du tri d'un tableau

Comme on l'a indiqué au paragraphe 5.1.4, l'invariant n'est pas forcément vérifié au moment de l'appel de *trouver_min*. La variable TT est déclarée comme $TT \in 1..nn \rightarrow \mathsf{NAT}$, mais elle a comme "type B" un ensemble de couples, c'est-à-dire $TT \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$. Pour faire la preuve de l'algorithme de l'opération *trouver_min*, on ajoute les propriétés de TT en précondition (elles feront partie des obligations de preuve de l'appel).

L'algorithme de l'opération de recherche du minimum utilise une boucle avec initialisation sur la première valeur du sous-tableau à parcourir. Le minimum du sous-tableau $jj..pp - 1$ qui se trouve à l'indice ll est gardé dans la variable aa (voir l'invariant de la boucle). Pour chaque élément du sous-tableau d'indice pp, il suffit de comparer la valeur

$TT(pp)$ avec aa et si cette valeur est inférieure, il faut faire les mises à jour nécessaires. Cet algorithme est donné à la figure 5.5. Il s'agit d'un raffinement de la spécification donnée dans la partie des opérations locales. Il doit se trouver dans la rubrique des opérations. C'est ainsi que se termine l'implémentation de la machine de tri.

```
kk ⟵ trouver_min(jj) =
    VAR pp, ll, aa IN              /* précondition : jj ∈ 1..nn − 1 */
        pp := jj + 1;              /* ici pp est au plus nn */
        ll := jj;
        aa := TT(ll);
        WHILE pp < nn DO           /* on compare jusqu'à l'avant-dernier élément : */
            VAR tpp IN             /* c'est nécessaire pour éviter */
                tpp := TT(pp);     /* le débordement de pp */
                IF tpp < aa THEN
                    aa := tpp;
                    ll := pp
                END
            END;
            pp := pp + 1
        INVARIANT
            pp ∈ jj + 1..nn  ∧  ll ∈ jj..nn − 1 ∧
            aa ∈ NAT  ∧  aa = min(TT$0[jj..pp − 1])  ∧  aa = TT$0(ll)
        VARIANT
            nn − pp + 1
        END;
        VAR tnn IN
            tnn := TT(nn);
            IF tnn < aa THEN       /* on compare avec le dernier élément */
                kk := nn
            ELSE
                kk := ll
            END
        END
    END
```

FIG. 5.5 – Algorithme de recherche du minimum

Chapitre 6

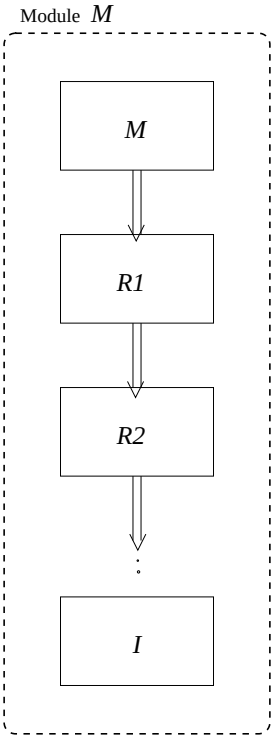
Modularité

Résumé

Un projet B est le plus souvent constitué d'un ensembles de machines qui sont raffinées jusqu'à des implémentations. Les chaînes de développement, appelées "modules" sont reliées par les relations de structuration des projets B. Les modules encapsulent des données et des états. Ils communiquent essentiellement par appel d'opérations. Ce chapitre présente les relations de structuration des modules en B et les résultats qui justifient la validité de ces primitives. Le point important de la structuration modulaire est que la validation locale des composants B n'est pas remise en cause lors de l'assemblage de modules.

6.1 Notion de module

On appelle *composant* en B une unité de "compilation" qui est soit une machine, soit un raffinement, soit une implémentation. Un *module* est un développement constitué d'une machine M, éventuellement d'un certain nombre de raffinements de cette machine $R1$, $R2$,...Rn et d'une implémentation I lorsque le développement est complet. Un module est appelé du même nom que la machine racine du développement. On appelle *interface* d'un module M la liste de ses opérations avec leur profil. Tous les composants d'un module ont la même interface. Les obligations de preuve assurent la validité des composants (l'initialisation établit l'invariant et celui-ci est préservé par les opérations si elles sont appelées sous leur précondition (paragraphe 3.2.2)) et elles assurent également le raffinement (les comportements de chaque opération d'un raffinement sont inclus dans les comportements possibles de l'abstraction (paragraphe 4.2)). Grâce aux obligations de preuve et à la transitivité des raffinements, on peut affirmer qu'une implémentation I est bien un raffinement de la machine M racine du développement et que les invariants sont préservés.



Dans un module, la machine racine est la spécification et l'implémentation représente le code exécutable. Le texte de l'implémentation est celui qui est traduit par le traducteur de l'atelier B dans un langage comme C ou Ada.

Dans un projet, il peut aussi y avoir des *modules abstraits* qui sont des modules ne contenant qu'une machine sans raffinements ni implémentation. On y reviendra en parlant de la primitive d'inclusion.

Un projet est *modulaire* en ce sens qu'il peut comporter plusieurs modules reliés par des *primitives de structuration*. Ces primitives[1] sont explicitées dans le paragraphe 6.3.

6.1.1 Déclaration de modules et instances de modules

En B, on écrit des *déclarations* de machines, raffinements, implémentations et donc de modules. Ces modules peuvent avoir des paramètres (paragraphe 3.1). Les paramètres ne peuvent pas être utilisés dans les parties statiques du module (ensembles et constantes) qui sont non paramétrables.

Les modules qui sont utilisés effectivement dans un projet sont des *instances* de modules déclarés. Une instance de module M est créée lorsqu'elle apparaît dans une primitive de structuration "instanciante". Les primitives instanciantes sont IMPORTS (paragraphe 6.3.2) et INCLUDES (paragraphe 6.3.5). Lors de l'instanciation, les modules paramétrés doivent avoir des paramètres effectifs qui correspondent à leur nature (ensembles ou scalaires) et qui satisfont les contraintes exprimées dans la clause CONSTRAINTS (obligations de preuve). Une instance de module peut aussi être *renommée* dans une primitive de structuration instanciante. Si un module s'appelle M, alors les renommages $nn.M$ et $pp.M$ sont des instances distinctes de M. Une instance de module peut être vue comme une "allocation" d'un exemplaire de la déclaration dans lequel les paramètres formels sont remplacés par les paramètres effectifs. Les variables et les opérations déclarées dans une machine renommée sont également renommées de la même façon. Par contre, les entités statiques (ensembles et constantes) ne sont pas renommées et sont communes à toutes les instances du même module, qu'il soit paramétré ou non paramétré.

Dans un projet B, lorsqu'un module n'est pas utilisé à travers une primitive de structuration (il s'agit alors du module principal du projet non paramétré), on identifie son instance à sa déclaration.

Dans la suite du chapitre, sauf indication explicite, les termes composants, machines, modules, etc. doivent être compris comme instances de composants, instances de machines, instances de modules, etc.

6.1.2 Spécification des composants et des modules d'un projet B

Les composants sont identifiés par des *NOMS*. Les sous-ensembles disjoints des noms de composants sont les machines (*mach*), les raffinements (*raff*) et les implémentations (*impl*). On a l'axiomatisation suivante dans les composants :

$composants = mach \cup raff \cup impl$	ensemble des composants d'un projet
$c\_raffinants = raff \cup impl$	ensemble des composants raffinants
$c\_raffinables = mach \cup raff$	ensemble des composants raffinables
$mach \cap c\_raffinants = \emptyset$	
$raff \cap impl = \emptyset$	

[1] Au moins les plus importantes. On ne détaille pas ici la primitive USES qui est peu utilisée.

Par construction, tous les composants raffinants raffinent un seul autre composant (clause REFINES) et un composant ne peut pas être raffiné plusieurs fois. Donc, la relation *raffine_directement* est injective totale sur *c_raffinants*. La relation inverse du raffinement (*est_raffine_par*) est aussi une fonction injective. Un composant ne peut pas (par construction) être un raffinement de lui-même (directement ou indirectement). Les relations modélisant le raffinement[2] entre composants peuvent donc s'exprimer par :

$raffine\_directement \in c\_raffinants \rightarrowtail c\_raffinables$	
$raffine = raffine\_directement^+$	fermeture transitive
$raffine \cap \mathsf{id}(NOMS) = \emptyset$	il n'y a pas de boucle
$est\_raffine\_par = raffine^{-1}$	relation inverse
$est\_raffine\_par \in c\_raffinables \rightarrowtail c\_raffinants$	

Il est utile de privilégier la relation d'implémentation entre une machine et son implémentation :

$implemente \in impl \rightarrowtail mach$
$implemente = impl \lhd raffine \rhd mach$
$est\_impl\_par = implemente^{-1}$

6.2 Les appels d'opérations

6.2.1 Forme d'un appel d'opération

Un appel d'opération a exactement la forme de l'entête de l'opération, c'est-à-dire il a une des formes suivantes :

op	Opération sans paramètre ni résultat
$op(E, F, \ldots)$	Opération avec paramètres et sans résultat
$z, t, \ldots \longleftarrow op$	Opération sans paramètre et avec résultats
$z, t, \ldots \longleftarrow op(E, F, \ldots)$	Opération avec paramètres et avec résultats

Si l'on prend le cas le plus général, la déclaration d'une opération est de la forme (r et p pouvant être des listes de variables) :

$$r \longleftarrow op(p) \;=\; S$$

Un appel de cette même opération est de la forme (avec le même nombre d'arguments et de résultats que dans la définition) :

$$v \longleftarrow op(e)$$

où v est une liste de variables et e une liste d'expressions compatibles avec les paramètres formels. Les restrictions sont les suivantes :

1. v est une liste de noms de variables ;
2. v ne contient pas de doublon ;
3. les variables de la liste v sont disjointes des variables de la liste x, où x désigne les variables de la machine dans laquelle l'opération est définie.

[2] Il ne s'agit pas ici du raffinement noté "$\sqsubseteq$", mais de la relation induite par la clause REFINES.

La restriction 1 interdit des appels de la forme $t(i) \leftarrow op$, qui peuvent introduire des ambiguïtés liées à l'ordre d'évaluation. La restriction 2 élimine des appels de la forme $a, a \leftarrow op$, qui peuvent aussi être difficiles à interpréter (quelle est la valeur finale de a ?). La dernière restriction impose que les variables sur lesquelles est définie l'opération soient disjointes des paramètres effectifs résultats. Cette restriction découle du principe d'encapsulation : les variables x ne peuvent être modifiées à l'extérieur d'une machine que par l'intermédiaire d'un appel d'opération.

6.2.2 Sémantique par substitution

La définition ci-dessous est celle donnée dans le B-Book [Abr96]. Elle ne s'applique que si S est définie en terme de substitutions autorisées dans les machines. Soit $r \leftarrow op(p) = S$ la définition d'une opération. Le résultat de l'appel $v \leftarrow op(e)$ est défini par :

$$[p, r := e, v]\ S$$

L'application d'une substitution sur une substitution est définie dans le tableau ci-après. Il n'existe pas de règle générale définissant l'application d'une substitution sur les substitutions séquencement et itération. La définition ci-dessus est applicable à toute substitution, en prenant la forme normalisée qui ne contient que des substitutions mathématiques. Une substitution appliquée à une expression est, de la même manière, définie inductivement sur la structure de l'expression.

Substitution	Définition	Condition
$[x := E](y := F)$	$[x := E]y := [x := E]F$	
$[x := E]\mathsf{skip}$	skip	
$[x := E](P \mid S)$	$[x := E]P \mid [x := E]S$	
$[x := E](S \parallel T)$	$[x := E]S \parallel [x := E]T$	
$[x := E](P \Longrightarrow S)$	$[x := E]P \Longrightarrow [x := E]S$	
$[x := E](@y \cdot S)$	$@y \cdot [x := E]S$	$y \backslash x, y \backslash E$
$[x := E]y$	y	$y \neq x$
$[x := E]y$	E	$y = x$

Exemple : Soit op l'opération définie par :

$$r \leftarrow op(p) = \ \text{PRE}\ \ p \geq 0\ \text{THEN}\ \ r := p + 1\ \ \text{END}$$

L'appel $v \leftarrow op(3)$ se réduit à $3 \geq 0 \mid v := 3 + 1$, c'est-à-dire à $v := 4$. L'appel $a \leftarrow op(a)$, quant à lui, se réduit à $a \geq 0 \mid a := a + 1$. □

6.2.3 Préservation de l'invariant d'une machine appelée

Du point de vue formel, la sémantique de l'appel d'opération doit permettre de réutiliser les preuves faites sur la définition des opérations. On peut garantir que, si la définition d'une opération préserve l'invariant I, alors il en est de même des appels de cette opération. La propriété est la suivante :

Propriété 1 :
Soit $r \leftarrow op(p) = P \mid S$ la définition d'une opération et soit $v \leftarrow op(e)$ un appel vérifiant les restrictions sur les appels, alors :

$$\begin{array}{c} \forall r, p \cdot ((I \wedge P \Rightarrow [S]\, I) \\ \Rightarrow \\ (I \wedge [p := e]\, P \Rightarrow [[p, r := e, v]\, S]\, I) \end{array}$$

Preuve :
En appliquant la propriété de monotonie sur l'antécédent (page 30, chapitre 2), pour la substitution $p, r := e, v$, on obtient :

$$I \wedge [p := e]P \Rightarrow [p, r := e, v][S]\, I$$

puisque r est non libre dans I et P et p est non libre dans I. Or la formule $[p, r := e, v][S]\, I$ est équivalente à la formule $[[p, r := e, v]S]\, I$, puisque les variables p et r ne sont pas libres dans I. Ceci permet de déduire l'obligation de preuve relative à la préservation de l'invariant pour l'appel $v \leftarrow op(e)$. □

Nous nous intéressons uniquement aux appels d'opérations qui terminent. En fait, la preuve de terminaison de l'appel se réduit à établir la formule $[p := e]P$. En effet la terminaison de l'appel est définie par la formule $[p := e]P \wedge \mathsf{trm}([p, r := e, v]S)$. Or, de la preuve de préservation de l'invariant, on peut déduire $I \wedge [p := e]P \Rightarrow \mathsf{trm}([p, r := e, v]S)$ (propriété de terminaison, page 31, chapitre 2). Il suffit donc d'établir la formule $[p := e]P$ (obligation de preuve générée par l'atelier B) pour assurer la terminaison de l'appel d'opération.

La préservation des invariants découle du fait que ces formules ne font pas intervenir les paramètres des opérations. Comme contrexemple, on peut montrer que l'appel d'opération ne préserve pas les postconditions. Par exemple, l'opération *op* définie par :

$$r \leftarrow op(p) = \ \text{PRE}\ \ p \geq 0\ \text{THEN}\ \ r := p + 1\ \ \text{END}$$

vérifie la postcondition $r = p + 1$. En effet, on a bien $p \geq 0 \Rightarrow [r := p + 1](r = p + 1)$. Par contre, l'appel $v \leftarrow op(v)$ ne vérifie pas la postcondition $[p, r := v, v](r = p + 1)$, qui se réduit à $v = v + 1$, c'est-à-dire à faux.

6.2.4 Préservation des raffinements par l'appel d'opération

L'appel d'opération a la sémantique définie par la substitution des paramètres de l'appel dans le corps de l'opération donné dans la machine où cette opération est déclarée (paragraphe 6.2.2) Mais un appel d'opération d'une machine se traduit finalement par l'appel de l'implémentation de cette opération. Peut-on alors assurer que l'appel de l'implémentation de l'opération est bien un raffinement de l'appel de la spécification de cette opération ?

Pour prouver cela, nous allons nous appuyer sur la propriété de la monotonie du raffinement par rapport à l'appel d'opération.

Propriété 2 :
Soit $r \longleftarrow op(p) = P_1 \mid S_1$ la définition abstraite d'une opération op et soit $r \longleftarrow op(p) = P_2 \mid S_2$ la définition concrète de cette même opération. Soit $v \longleftarrow op(e)$ un appel de cette opération qui vérifie les restrictions sur les appels et tel que l'expression e ne contient aucune occurrence des variables de la machine dans laquelle est définie l'opération[3] (ni de son raffinement). Alors on a :

$$\begin{array}{c} P_1 \mid [r := r_1]S_1 \sqsubseteq_{L \wedge r_1 = r_2} P_2 \mid [r := r_2]S_2 \\ \Rightarrow \\ [p := e]P_1 \mid [p, r := e, v_1]S_1 \sqsubseteq_{L \wedge v_1 = v_2} [p := e]P_2 \mid [p, r := e, v_2]S_2 \end{array}$$

avec L l'invariant de raffinement.

Preuve :
Les obligations de preuve relatives au raffinement de l'opération sont :

$$\begin{array}{ll} (a) & \forall p \,.\, (L \wedge P_1 \Rightarrow P_2) \\ (b) & \forall p, r_1, r_2 \,.\, (L \wedge P_1 \Rightarrow [[r := r_2]S_2]\, \neg[[r := r_1]S_1]\, \neg(L \wedge r_1 = r_2)) \end{array}$$

et nous voulons en déduire :

$$\begin{array}{ll} (c) & L \wedge [p := e]P_1 \Rightarrow [p := e]P_2 \\ (d) & L \wedge [p := e]P_1 \Rightarrow [[p, r := e, v_2]S_2]\, \neg[[p, r := e, v_1]S_1]\, \neg(L \wedge v_1 = v_2) \end{array}$$

La formule (c) se déduit directement de la formule (a) en appliquant la propriété de monotonie (page 30, chapitre 2), pour la substitution $p := e$. Nous utilisons cette même propriété, à partir de (b), pour la substitution $p, r_1, r_2 := e, v_1, v_2$. Nous obtenons :

$$L \wedge [p := e]P_1 \Rightarrow [p, r_1, r_2 := e, v_1, v_2]([[r := r_2]S_2]\, \neg[[r := r_1]S_1]\, \neg(L \wedge r_1 = r_2))$$

puisque p, r_1, r_2 sont non libres dans L et r_1, r_2 sont non libres dans P_1. La partie droite peut être réécrite en utilisant la propriété suivante :

$$[x := f][S]R \Leftrightarrow [[x := f]S][x := f]R$$

si f ne contient aucune occurrence des variables de S.

La condition est vérifiée par la substitution $p, r_1, r_2 := e, v_1, v_2$ et les substitutions $[r := r_2]S_2$ et $[r := r_2]S_2$, puisque v_1 et v_2 sont des variables externes à la machine dans laquelle l'opération est définie et puisque nous avons cette hypothèse sur l'expression e. La partie droite se réécrit donc en :

$$\begin{array}{l} [[p, r_1, r_2 := e, v_1, v_2][r := r_2]S_2] \\ \quad \neg[[p, r_1, r_2 := e, v_1, v_2][r := r_1]S_1] \\ \quad\quad \neg[p, r_1, r_2 := e, v_1, v_2](L \wedge r_1 = r_2) \end{array}$$

[3]Ceci correspond à la vue encapsulée qui permet de substituer une définition abstraite par son raffinement.

Puisque r_1 est non libre dans $[r := r_2]S_2$, r_2 est non libre dans $[r := r_1]S_1$ et p non libre dans $L \wedge r_1 = r_2$, on simplifie en :

$$\begin{array}{l}[[p, r_2 := e, v_2][r := r_2]S_2] \\ \quad \neg[[p, r_1 := e, v_1][r := r_1]S_1] \\ \quad\quad \neg[r_1, r_2 := v_1, v_2](L \wedge r_1 = r_2)\end{array}$$

qui se réécrit en la partie droite de la formule (d). □

6.3 Primitives de structuration de modules

6.3.1 Combinaison de machines

On appelle *combinaison* de machines, le fait de considérer une liste (ou un ensemble) d'instances de machines comme une seule machine. Il s'agit simplement d'une définition théorique, car il est clair que les machines restent physiquement des entités séparées. La combinaison intervient dans les primitives de structuration qui portent toujours sur une *liste* de noms de machines.

Une machine fournit des "services" qui sont de diverse nature : les déclarations statiques, c'est-à-dire les ensembles G, les constantes concrètes c et abstraites a, spécifiées par des propriétés R, les variables concrètes v et les variables abstraites x de l'état, spécifiées par un invariant I, l'initialisation U et les opérations visibles O de la forme $r \longleftarrow op(p) = (P \mid S)$.

On note $Liste_{i=1}^{k} A_i$ la liste des éléments A_i dans l'ordre des indices. Si l'on combine k machines $M_1, M_2, \ldots, M_k$ on obtient l'équivalent d'une machine ayant les services suivants :

Entités	Valeur combinée
Ensembles	$Liste_{i=1}^{k} G_i$
Constantes concrètes	$Liste_{i=1}^{k} c_i$
Constantes abstraites	$Liste_{i=1}^{k} a_i$
Propriétés	$\bigwedge_{i=1}^{k} R_i$
Variables concrètes	$Liste_{i=1}^{k} v_i$
Variables abstraites	$Liste_{i=1}^{k} x_i$
Invariant	$\bigwedge_{i=1}^{k} I_i$
Initialisation	$\|\|_{i=1}^{k} U_i$
Opérations	$Liste_{i=1}^{k} O_i$

Une combinaison de machines doit être valide, c'est-à-dire que son invariant doit être établi par l'initialisation et préservé par les opérations. Il y a deux manières (non exclusives) d'obtenir cette validité : elle peut être obtenue par construction, par exemple si les espaces de variables des machines sont disjoints deux à deux, ou elle peut être obtenue par vérification d'obligations de preuves spécifiques. En B, il y a l'un et/ou l'autre cas suivant les primitives de structuration.

D'un point de vue théorique, chaque machine de la combinaison est plongée dans un espace de noms plus grand, qui est l'espace des noms de la combinaison. Lorsqu'on calcule, par exemple, le prédicat avant-après d'une opération de M_i, on n'obtient pas le même résultat, suivant qu'on le calcule dans M_i ou dans la combinaison $M_1, M_2, \ldots, M_k$.

6.3.2 Relation IMPORTS

Un développement de modules en B peut se faire *par couches*. Cela signifie que l'implémentation d'un module de couche n "importe" des instances de machines (donc de modules) de la couche $n-1$. Un tel développement peut se poursuivre jusqu'à la couche des *machines de base* qui sont des machines pour lesquelles l'implémentation n'est pas donnée en B, mais directement dans le langage de programmation cible (par exemple en C). C'est par le moyen des machines de base que l'on peut faire communiquer des développements B avec des composants systèmes ou des programmes développés de manière conventionnelle. Evidemment, l'implémentation des machines de base doit être validée par des techniques ad hoc.

La clause de l'importation se présente de la manière suivante :

IMPLEMENTATION *IMPL*
REFINES *REFI*
IMPORTS *M_IMPORTEES*
...
END

La clause d'importation peut mentionner plusieurs instances de machines qui sont alors combinées pour former la machine *M_IMPORTEES*. L'invariant de *IMPL* doit indiquer la liaison entre les variables de l'état hérité du raffinement *REFI* et les variables de l'état de la machine importée. Les variables concrètes héritées peuvent ne pas faire partie de l'invariant de liaison, dans la mesure où elles sont implantables directement (paragraphe 5.1.1).

L'effet de l'importation est de permettre à l'implémentation *IMPL* d'accéder aux services de la machine importée. Les entités statiques G, c, a, peuvent être utilisées dans l'implémentation sans restriction. Par exemple, elles peuvent servir à valuer les ensembles et les constantes concrètes héritées dans l'implémentation, si les types sont compatibles. Les variables concrètes v de l'implémentation sont visibles en lecture uniquement, dans les instructions.

Pour ce qui concerne les variables abstraites de l'état de la machine importée x et, plus généralement, pour les modifications de cet état, le principe d'encapsulation s'applique : dans les instructions des opérations de l'implémentation, on ne peut modifier l'état de la machine importée qu'à travers des appels aux opérations O de cette machine (préservant ainsi son invariant, comme indiqué au paragraphe 6.2.3).

Cependant, on peut noter que dans les prédicats du composant implémentation (invariant, invariants de boucles, assertions) ainsi que dans les variants des boucles, on peut utiliser directement les variables concrètes et abstraites de la machine importée, comme les autres entités statiques. Le principe d'encapsulation ne joue pas dans les parties "logiques" du composant.

La modélisation de la relation d'importation est la suivante :

$importe\_directement \in impl \leftrightarrow mach$	relation de la clause IMPORTS
$importe \in mach \leftrightarrow mach$	importation entre modules
$importe = (est\_impl\_par \ ; \ importe\_directement)$	définition de *importe*
$importe^{-1} \in mach \nrightarrow mach$	propriété de l'inverse

La dernière propriété, qui est vérifiée dans l'atelier B, indique qu'une instance de machine (module) ne peut être importée qu'une seule fois dans un projet donné. Cette restriction est nécessaire pour éviter que plusieurs modules accèdent en écriture à un même module. Cela invaliderait la relation de raffinement théorique construite globalement sur les modules reliés par des relations d'importation.

6.3.3 Exemple de développement en couche

On complète l'exemple de la réservation (paragraphes 3.3.1, 4.3 et 5.2.2) par un module principal pour appeler les services du module *RESERVATION*. Par convention, un module principal contient une opération sans paramètres, appelée *main*. On donne la machine de ce module principal à la figure 6.1.

```
MACHINE
    USER
OPERATIONS
    main = skip                    /* Programme principal */
END
```

FIG. 6.1 – Machine principale d'appel de *RESERVATION*

Le raffinement de cette machine principale est une implémentation. Elle doit importer la machine *RESERVATION* ainsi qu'une machine de base fournie avec la bibliothèque prédéfinie de l'atelier B pour réaliser des entrées-sorties standards clavier-écran. Cette machine s'appelle *BASIC_IO* et fournit en particulier les opérations suivantes :

$STRING\_WRITE(p)$	affiche à l'écran la chaîne de caractère p
$INT\_WRITE(i)$	affiche à l'écran la valeur de l'entier i
$r \longleftarrow INTERVAL\_READ(a, b)$	lit un entier compris entre a et b et donne sa valeur dans r

Les déclarations de l'implémentation sont décrites à la figure 6.2.

```
IMPLEMENTATION
    USER_i
REFINES
    USER
IMPORTS                            /* machines importées */
    RESERVATION, BASIC_IO
```

FIG. 6.2 – Entête de l'implémentation du module principal

Le raffinement de *main* est un programme qui attend des requêtes de l'utilisateur et fournit les réponses données par les opérations du module *RESERVATION*. Le nombre de pas de la boucle du programme principal est borné par la valeur de l'entier maximum représentable (rappelons que la méthode B impose de prouver que les boucles terminent). Une partie de cette opération est donnée à la figure 6.3.

Après traitement des différents composants par l'atelier B (type checking, génération des OPs, preuve, vérification du niveau B0 (conditions spécifiques pour pouvoir générer du code)), il est possible de passer à la génération de code. On a choisi la génération en C, ce qui produit un répertoire contenant les modules C générés. Il suffit de faire "`make`" pour obtenir un programme exécutable. Après ces étapes, l'état du projet est le suivant :

```
Project status
+--------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| COMPONENT    | TC | POG | Obv | nPO | nUn | %Pr | BOC |  C  | Ada | C++ | HIA |
+--------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| CODE         | OK | OK  |  33 |  40 |   0 | 100 | OK  | OK  |  -  |  -  |  -  |
| RESERVATION  | OK | OK  |  11 |   8 |   0 | 100 | OK  |     |     |     |     |
| RESERVATION1 | OK | OK  |  18 |  20 |   0 | 100 | OK  |     |     |     |     |
| USER1        | OK | OK  |   3 |   0 |   0 | 100 | OK  |     |     |     |     |
| USER1_i      | OK | OK  |  76 |  95 |   0 | 100 | OK  | OK  |  -  |  -  |  -  |
+--------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| TOTAL        | OK | OK  | 141 | 163 |   0 | 100 | OK  | OK  |  -  |  -  |  -  |
+--------------+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

La troisième colonne contient le nombre d'obligation de preuves évidentes qui ont été générées par l'Atelier B, alors que la quatrième contient le nombre d'OP qui nécessitent une preuve. La cinquième colonne indique le nombre d'OP non prouvées ; et la sixième fournit le pourcentage de prouvées par rapport à celles à prouver. Dans la colonne C, on voit les composants qui ont été traduits en C : il s'agit des composants d'implémentation. Une session d'exécution du programme `RESERVATION` est retranscrite ci-après :

```
horus(50) ./RESERVATION
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
1
Place_libre : 4
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
2
Reservation : Reservation de la place : 1
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
2
Reservation : Reservation de la place : 2
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
3
Liberation : Tapez le numero de place
4
La place etait libre
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
3
Liberation : Tapez le numero de place
1
Liberation effectuee
```

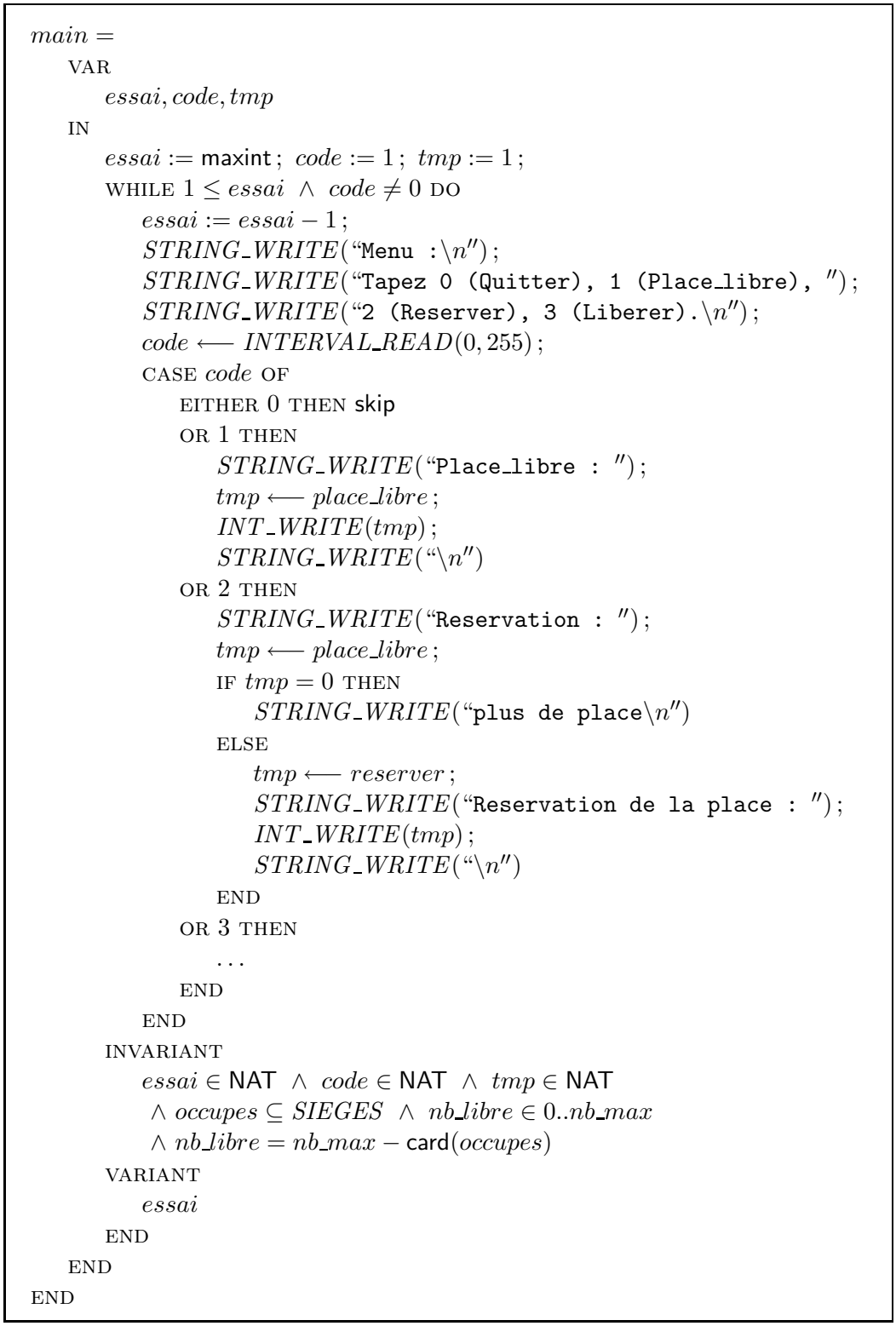


FIG. 6.3 – Raffinement de l'opération *main*

```
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
1
Place_libre : 3
Menu : Tapez 0 (Quitter), 1 (Place_libre), 2 (Reserver), 3 (Liberer).
0
horus(51)
```

6.3.4 Relation SEES

La clause SEES a été introduite pour permettre l'accès en lecture des informations d'une machine. Cette clause peut apparaître dans n'importe quel composant. En fait, la visibilité porte sur les déclarations statiques, les variables et les opérations. On peut donc accéder aux valeurs des variables vues, mais sans pouvoir les modifier. On peut aussi appeler une opération d'une machine vue, à condition que cette opération ne modifie pas l'état.

Cette clause a la forme syntaxique :

SEES *M_VUES*

où *M_VUES* représente une liste de noms de machines, éventuellement renommées, qui sont combinées, comme indiqué au paragraphe 6.3.1. La clause SEES n'est pas "instanciante". Les noms des machines vues permettent à l'analyseur de déterminer quelle est l'instance effectivement vue. Cette instance fait partie des instances importées à un autre niveau dans l'arbre d'importation. Le schéma 6.4 indique comment se place la relation SEES dans un arbre d'importation. On remarque sur ce schéma que la relation d'importation de $M2$ dans I produit une instance qui est celle repérée par la relation *voit_directement* de C.

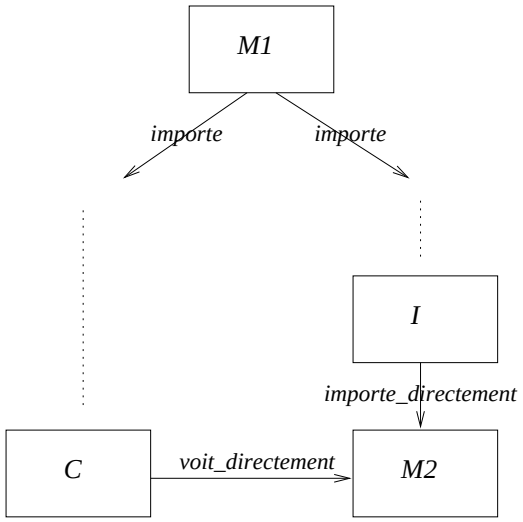


FIG. 6.4 – Relation SEES et instanciation

La relation SEES est modélisée (sur les composants instanciés) par :

$voit\_directement \in composants \leftrightarrow mach$	relation de la clause SEES
$voit\_directement \subseteq (est\_raffine\_par \ ; voit\_directement)$	(*)
$voit \in mach \leftrightarrow mach$	visibilité entre modules
$voit = (est\_impl\_par \ ; voit\_directement)$	définition de *voit*

La contrainte (*) indique que si un composant C voit directement une machine, alors le raffinement de C doit voir aussi cette machine. Cette condition fait qu'il n'y a pas de machine vue directement dans une chaîne de raffinements qui ne soit pas vue par l'implémentation. Cette remarque justifie la définition de la relation *voit* (pour un module complet).

Les relations IMPORTS et SEES induisent des dépendances entre les modules. La modélisation de ces dépendances est la suivante :

$utilise = voit \cup importe$	relation *voit* ou *importe*
$depend\_de = utilise^+$	fermeture de la relation *utilise*
$peut\_consulter = (utilise^* \ ; voit)$	accès transitif en lecture
$peut\_modifier = (utilise^* \ ; importe)$	accès transitif en lecture-écriture

Ces relations doivent satisfaire des contraintes qui sont vérifiées par l'atelier B. La première contrainte est que les relations n'induisent pas des dépendances circulaires. La deuxième contrainte indique qu'il ne doit pas être possible à un composant C de voir une machine à laquelle il peut accéder par ailleurs en lecture-écriture. Il s'agit d'une condition suffisante de validité d'une architecture de projet. Si cette condition n'est pas respectée, le raffinement n'est plus garanti. En particulier, une implémentation ne peut pas voir et importer la même instance de machine. Ces contraintes s'expriment par :

$depend\_de \ \cap \ \mathsf{id}(NOMS) \ = \ \emptyset$	non circularité
$peut\_modifier \ \cap \ voit \ = \ \emptyset$	condition d'accès

6.3.5 Relation INCLUDES

Alors que les relation IMPORTS et SEES sont des relations d'architecture de modules, c'est-à-dire que les programmes générés respectent cette modularité, la relation INCLUDES est une relation entre les spécifications. Elle permet de décrire des machines et des raffinements par aggégation de composants plus simples. Cette spécification "par morceaux" peut être ou ne pas être reflétée dans l'implémentation, comme on le verra au paragraphe 6.3.6.

La clause d'inclusion dans une machine ou un raffinement a la forme suivante :

MACHINE (ou REFINEMENT)
  M
INCLUDES
  M_INCL

Comme précédemment, M_INCL est une liste d'instances de machines qui sont combinées. Les noms des constantes, variables et opérations des machines incluses doivent être disjoints de ceux de la machine incluante.

La sémantique de l'inclusion est donnée par le calcul de la machine équivalente sans inclusion explicite. Les services exportés de la machine résultant de l'inclusion de M_INCL dans M sont donnés dans le tableau ci-après. Pour l'initialisation et les opérations, les expressions $[O_i := S_i]\,U$ et $[O_i := S_i]\,O$ signifient que les appels des opérations des machines incluses sont remplacées par leur corps après substitution des paramètres effectifs,

comme indiqué dans la sémantique de l'appel au paragraphe 6.2.2. La liste des opérations exportées est la liste des opérations de la machine incluante.

Entités	M	M_INCL	Résultat
Ensembles	G	G_i	$G_i \ ; \ G$
Constantes concrètes	c	c_i	c_i, c
Constantes abstraites	a	a_i	a_i, a
Propriétés	R	R_i	$R_i \ \wedge \ R$
Variables concrètes	v	v_i	v_i, v
Variables abstraites	x	x_i	x_i, x
Invariant	I	I_i	$I_i \ \wedge \ I$
Initialisation	U	U_i	$U_i \ ; \ [O_i := S_i] \, U$
Opérations	O	$O_i = S_i$	$[O_i := S_i] \, O$

L'utilisation des variables et des opérations des machines incluses dans l'initialisation et les opérations de la machine incluante est restreinte pour que les invariants des machines incluses soient préservés. Ces restrictions sont :

- les variables des machines incluses v_i et x_i ne peuvent être utilisées qu'en lecture dans l'initialisation de la machine incluante U et dans le corps des opérations O,
- les opérations des machines incluses peuvent être appelées dans l'initialisation et dans les opérations de la machine incluante, à condition qu'il n'y ait pas des appels simultanés d'opérations modifiant l'état d'une même machine incluse.

La première restriction a déjà été rencontrée dans l'importation, pour les variables concrètes. La deuxième restriction s'explique par le fait que, dans les machines et les raffinements, on peut écrire des substitutions "simultanées" par le constructeur || (paragraphe 2.2.5). Une condition suffisante pour que les invariants des machines incluses soient préservés est qu'il n'y ait pas appel en parallèle de deux opérations qui modifient l'état d'une même machine. En revanche, l'atelier B autorise l'appel en parallèle d'opérations qui ne modifient pas l'état ou avec une seule opération qui modifie l'état.

Enfin, comme pour les importations, une instance de machine ne peut être incluse qu'une seule fois dans un projet.

6.3.6 Architecture de modules avec inclusion

Les machines incluses sont "sémantiquement" intégrées dans les machines incluantes. La première façon de représenter les modules avec inclusion de machines et donc de considérer qu'il n'y a qu'une seule chaîne de développement composée de composants incluants. D'autre part, l'inclusion est transitive, en ce sens que, si une machine $M3$ est incluse dans une machine $M2$, alors le fait d'inclure $M2$ dans $M1$ transporte aussi les services de $M3$ dans $M1$, via l'inclusion dans $M2$. Le schéma de ce type de module avec inclusion est donné à la figure 6.5.

Les machines sans développement propre, comme $M1$, $M2$ et $M3$ sur le schéma, sont appelées *modules abstraits* dans la terminologie de l'Atelier B.

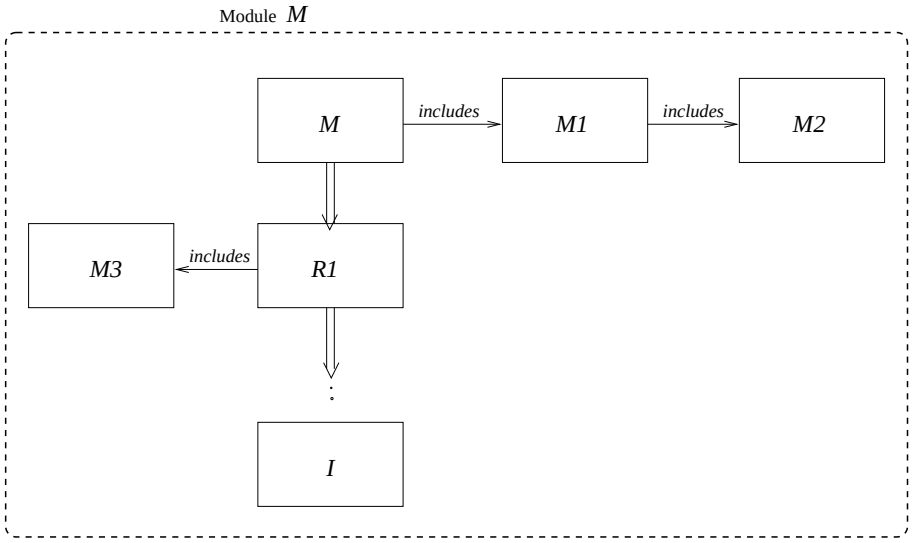


FIG. 6.5 – Module avec machines incluses intégrées

Il existe une autre façon d'utiliser l'inclusion, tout en ayant la même sémantique. Cela consiste à adjoindre systématiquement la *même machine* le long d'une chaîne de développement jusqu'à l'implémentation. A ce niveau, la machine en question ne peut plus être incluse, mais elle devient une machine importée de l'implémentation. Dans le niveau inférieur, cette machine peut alors être à son tour raffinée indépendamment en un nouveau module. Ce schéma est représenté à la figure 6.6.

6.3.7 Clauses PROMOTES et EXTENDS

Promotion des opérations

Les opérations exportées par une machine (ou un module) sont celles qui sont déclarées dans la liste des opérations de cette machine. Lorsqu'un composant C inclut ou importe une machine M, il est possible de *promouvoir* des opérations de M dans l'interface de C. Ces opérations promues font alors partie de l'interface visible de C. Des obligations de preuve sont générées pour assurer que les opérations promues préservent l'invariant de la machine, comme pour les autres opérations. La forme syntaxique de la promotion est :

	IMPLEMENTATION
	IMPL
MACHINE (ou REFINEMENT)	REFINES
M	*REFI*
INCLUDES	IMPORTS
M_INCL	*M_IMPORTEES*
PROMOTES	PROMOTES
$op_1, op_2, \ldots op_m$	$op_1, op_2, \ldots op_m$

Les noms des opérations $op_1, op_2, \ldots op_m$ sont des noms d'opérations visibles dans la combinaison des machines incluses ou importées.

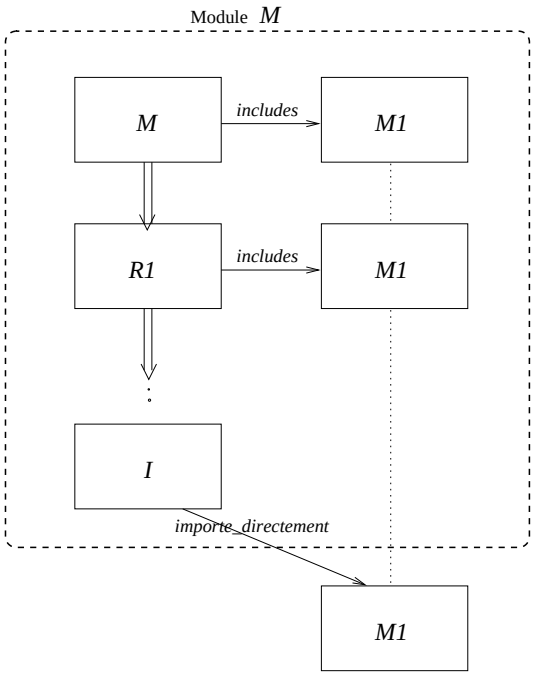


FIG. 6.6 – Module avec machines incluses puis importées

Extension d'interface

Lorsque toutes les opérations d'une machine incluse ou d'une machine importée doivent être promues, on peut utiliser la clause EXTENDS suivie d'une instance de machine ou d'une liste d'instances de machines.

MACHINE ou REFINEMENT ou IMPLEMENTATION
C
...
EXTENDS M_j

Si cette clause se trouve dans une machine ou un raffinement, elle est équivalente à la clause INCLUDES sur la même liste de machines suivie de la clause PROMOTES de toutes les opérations de ces machines. Si cette clause se trouve dans une implémentation, elle est équivalente à la clause IMPORTS suivie de la clause PROMOTES de toutes les opérations des machines importées.

Chapitre 7

B événementiel

Résumé

Ce chapitre présente l'extension de B connue sous le nom de B événementiel ou B système. Dans cette extension, un modèle représente un système dynamique pris dans son ensemble, et décrit une simulation d'une réalité observable. Ce chapitre donne la définition d'un composant système et de ses raffinements. Il donne les obligations de preuve au sens de la cohérence interne (propriétés de sûreté). Il définit aussi des modalités pour caractériser la vivacité et les propriétés dynamiques.

7.1 Composant "système"

7.1.1 Principes généraux

Dans le B classique, la notion de base est la *machine*. Une machine définit un composant qui propose des "services". Les composants extérieurs (ou les utilisateurs) peuvent utiliser les services fournis en faisant des appels d'opérations. Les opérations définissent les conditions d'appel dans une *précondition*. On a une structure classique de type client-serveur. Cette façon de faire correspond bien à la modélisation d'applications informatiques.

Cependant, on peut vouloir modéliser des *systèmes* d'une manière globale, comme par exemple un système réactif, où des "événements" se produisent dans certaines conditions. Les événements modifient l'état du système. Prenons le cas d'un système "ascenseur". Les événements qui peuvent arriver sont :

- des utilisateurs appuient sur les boutons extérieurs ou intérieurs
- les portes s'ouvrent ou se ferment
- l'ascenseur démarre, s'arrête
- l'ascenseur change de direction.

Comme pour une machine B, un système B possède un *état* qui enregistre l'information pertinente (les boutons où l'on a appuyé et qui n'ont pas été servis, par exemple). Les événements ne peuvent s'exécuter que si certaines conditions sont remplies. Toujours sur l'exemple, il est clair que l'événement "ouverture des portes" ne doit pas se produire lorsque l'ascenseur est en mouvement. On dit que les événements sont *gardés* par des conditions.

Le modèle d'exécution d'un système B est le suivant : un événement s'exécute si sa garde est vraie dans l'état courant (on dit que l'événement est *habilité*). Si plusieurs événements sont habilités, alors un de ces événements, choisi aléatoirement, s'exécute. Les

événements ne s'exécutent pas en parallèle : à un instant donné, il n'y a, au plus, qu'un seul événement qui s'exécute. Enfin, un événement n'est pas interruptible. On considère, dans l'abstraction, qu'il s'exécute dans une durée de temps nulle. Un système B n'est pas bloqué tant qu'il existe au moins une garde vraie.

7.1.2 Syntaxe d'un composant système

La forme générale d'un modèle B événementiel est très proche de celle d'une machine. Les clauses de même nom (cf. paragraphe 3.1) sont les mêmes clauses que celles des machines. Le schéma d'un modèle B événementiel est donné à la figure 7.1.

SYSTEM
    nom du modèle
SETS
    déclaration d'ensembles
CONSTANTS
    déclaration de constantes
PROPERTIES
    propriétés des constantes
VARIABLES
    variables (état)
INVARIANT
    invariant (caractérisation de l'état)
ASSERTIONS
    assertions supplémentaires
INITIALISATION
    substitution d'initialisation
EVENTS
    nom_ev = substitution d'événement
    ...
END

FIG. 7.1 – Forme générale d'un système d'événements

Les événements ont un nom, mais pas de paramètres, ni de résultat. La substitution d'événement est une substitution généralisée qui prend seulement une des formes suivantes :

nom_ev =
    BEGIN
        corps d'événement
    END

nom_ev =
    SELECT
        condition
    THEN
        corps d'événement
    END

nom_ev =
    ANY *variables* WHERE
        condition
    THEN
        corps d'événement
    END

Une substitution de *corps d'événement* est une substitution de machine abstraite (paragraphe 3.1.4)[1]. Les notions de terminaison, faisabilité et de prédicat avant-après s'appliquent aux substitutions du B événementiel, de la même manière qu'à celles du B classique.

On utilise la notion syntaxique de *garde* des événements qui est définie d'après la forme des événements par le tableau ci-après :

Evénement E	Garde $\mathsf{grd}(E)$
BEGIN S END	btrue
SELECT $G(x)$ THEN S END	$G(x)$
ANY t WHERE $G(t,x)$ THEN S END	$\exists t \cdot G(t,x)$

En fait, le premier cas d'événement BEGIN peut être vu comme un événement SELECT dans lequel la condition est le prédicat btrue. Par la suite, pour un événement E, on parlera de sa garde $\mathsf{grd}(E)$ qui est un prédicat, et de son corps : $\mathsf{bod}(E)$ qui est une substitution généralisée.

7.1.3 Obligations de preuve d'un système

On se place dans un système avec déclarations D, variables x et invariant $I(x)$. Une première règle est que la garde de chaque événement doit être suffisamment forte pour assurer la faisabilité de cet événement. Cela signifie que, pour chaque événement E, on doit avoir la condition :

$$D \ \wedge \ I(x) \ \wedge \ \mathsf{grd}(E) \ \Rightarrow \ \exists x' \cdot \mathsf{prd}_x(\mathsf{bod}(E)) \qquad (\mathsf{FIS})$$

De même, la terminaison doit être assurée :

$$D \ \wedge \ I(x) \ \wedge \ \mathsf{grd}(E) \ \Rightarrow \ \mathsf{trm}(\mathsf{bod}(E)) \qquad (\mathsf{TRM})$$

De manière plus classique, comme pour les machines, l'initialisation U doit établir l'invariant, sous l'hypothèse des déclarations de propriétés :

$$D \ \Rightarrow \ [U]\, I(x) \qquad (\mathsf{INIT})$$

Chaque événement E doit préserver l'invariant :

$$D \ \wedge \ I(x) \ \Rightarrow \ [E]\, I(x) \qquad (\mathsf{INV})$$

[1] Certains documents sur le B événementiel donnent des conditions restrictives sur les substitutions de corps d'événements, comme par exemple, le projet RODIN [MAV05].

On peut détailler la formule de préservation de l'invariant sur chaque cas de substitution d'événement :

Evénement E	Préservation de l'invariant
BEGIN S END	$D \ \wedge \ I(x) \ \Rightarrow \ [S]\, I(x)$
SELECT $G(x)$ THEN S END	$D \ \wedge \ I(x) \ \wedge \ G(x) \ \Rightarrow \ [S]\, I(x)$
ANY t WHERE $G(t,x)$ THEN S END	$D \ \wedge \ I(x) \ \wedge \ G(t,x) \ \Rightarrow \ [S]\, I(x)$

7.1.4 Exemple du parking

On veut modéliser le contrôle de l'entrée des voitures dans un parking. Dans l'abstraction, on donne les conditions évidentes qui font qu'une voiture peut entrer dans un parking et en sortir. Le modèle sera raffiné pour dériver le système de contrôle (informatique) pour gérer les feux d'entrée dans le parking. On considère donc un parking qui contient *nbPlaces* et deux événements : *entrer* et *sortir*.

```
SYSTEM
    Parking
CONSTANTS
    nbPlaces
PROPERTIES
    nbPlaces ∈ NAT1              /* Nombre maximum de places */
VARIABLES
    vehi
INVARIANT
    vehi ∈ 0..nbPlaces           /* Nombre de véhicules dans le parking */
INITIALISATION
    vehi := 0
EVENTS
    entrer =                     /* Un véhicule entre à condition*/
        SELECT                   /* qu'il y ait de la place */
            vehi < nbPlaces
        THEN
            vehi := vehi + 1
        END ;
    sortir =                     /* Un véhicule sort : il y a au moins */
        SELECT                   /* un véhicule dans le parking */
            vehi > 0
        THEN
            vehi := vehi − 1
        END
END
```

FIG. 7.2 – Modèle du parking

7.1.5 Vues comportementales des systèmes d'événements

Un système fermé ne produit pas de résultats. Il ne modélise pas des fonctions où l'on peut définir des sorties en fonction des entrées et de l'état. Pour l'étudier, il y a deux manières classiques d'observer le *comportement* d'un système. La vue "state-based" qui examine les traces de l'état, et la vue "event-based" qui s'intéresse aux suites d'événements.

Sans entrer dans les détails, dans la vue basée sur les états, on distingue souvent des variables *observables*, qui donnent les valeurs visibles de l'état, et les variables *cachées*. De même, dans la vue basée sur les événements, on peut distinguer des événements visibles et des événements cachés. La sémantique complète du B événementiel est basée sur les traces d'états avec variables observables. Elle ne sera pas abordée dans ce document.

Il est souvent commode de représenter un système B (fini ou infini) par une approximation sous forme de diagramme fini d'états-transitions. Les états sont décrits par des prédicats, et les transitions sont les événements qui peuvent se déclencher entre ces états. Par exemple, pour le parking, une approximation est le diagramme décrit à la figure 7.3.

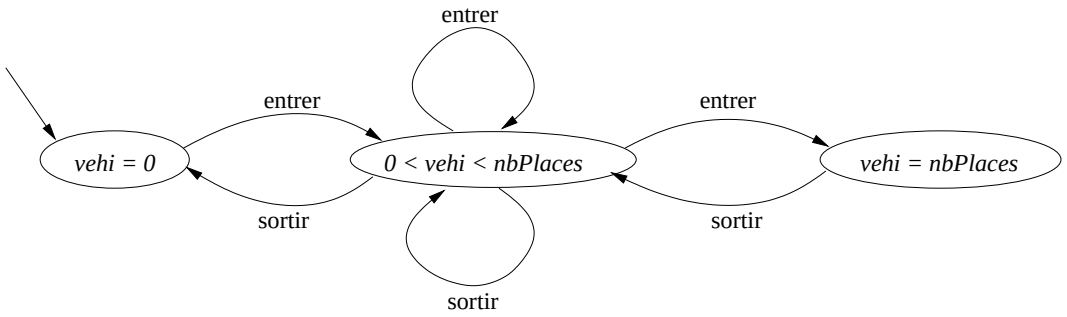


FIG. 7.3 – Diagramme états-transitions du parking

7.2 Raffinement de système

7.2.1 Composant de raffinement

Les systèmes B peuvent être raffinés. Le raffinement peut porter sur les données et sur les traitements (événements), comme dans le cas classique (opérations). On peut introduire la séquentialité du calcul par la substitution " ;". La nouveauté du B événementiel est que l'on peut *raffiner le temps*. Cela signifie que l'on peut observer les événements avec une granularité plus fine, et donc, introduire des détails de comportement, c'est-à-dire de nouveaux événements, qui n'étaient pas "visibles" au niveau plus abstrait.

Par exemple, dans le parking, immédiatement après chaque entrée (événement *entrer*), le raffinement va introduire un événement *controler_entree*. Cet événement consistera à activer le contrôleur, dès qu'un événement "une voiture entre" sera détecté. De même, après chaque sortie, il va falloir traiter l'événement pour, éventuellement, mettre à jour les feux. Ces nouveaux événements n'étaient pas visibles dans l'abstraction.

Pour continuer l'exemple, le premier pas de raffinement du parking consiste à construire l'automate qui gère la séquentialité des événements (contrôle du comportement des événements). Rappelons que l'on ne peut contraindre l'exécution des événements que par des variables

de l'état. Cet automate est modélisé par une variable entière cc qui prend trois valeurs dans l'intervalle $-1..1$:

- $cc = 0$: le système est en attente d'une entrée ou d'une sortie de voiture
- $cc = 1$: une voiture vient de rentrer, il faut activer le contrôleur d'entrée
- $cc = -1$: une voiture vient de sortir, il faut activer le contrôleur de sortie.

Ce premier raffinement est donné à la figure 7.4. Le traitement du contrôleur n'est pas encore précisé. Il sera déterminé dans les raffinements ultérieurs. On sait simplement que le système revient en attente d'entrée ou de sortie de voiture ($cc = 0$) après chaque phase de contrôle.

```
REFINEMENT
    ParkingR1
REFINES
    Parking
VARIABLES
    vehi, cc
INVARIANT
    cc ∈ −1..1 ∧
    (cc = −1 ⇒ vehi < nbPlaces) ∧   /* Conditions liées à cc */
    (cc = 1 ⇒ vehi > 0)
INITIALISATION
    vehi, cc := 0, 0
EVENTS
    entrer =                        /* Un véhicule entre */
        SELECT
            vehi < nbPlaces ∧ cc = 0
        THEN
            vehi := vehi + 1 || cc := 1
        END ;
    controler_entree =              /* Nouvel événement */
        SELECT cc = 1 THEN cc := 0 END ;
    sortir =                        /* Un véhicule sort */
        SELECT
            vehi > 0 ∧ cc = 0
        THEN
            vehi := vehi − 1 || cc := −1
        END ;
    controler_sortie =              /* Nouvel événement */
        SELECT cc = −1 THEN cc := 0 END
END
```

FIG. 7.4 – Premier raffinement du parking

7.2.2 Obligations de preuve d'un raffinement

Les obligations de preuve des raffinements d'un système événementiel doivent permettre de prouver les propriétés suivantes :

- les événements raffinés modifient l'état de manière compatible avec l'abstraction
- les nouveaux événement raffinent skip (pas de modification de l'état abstrait par les nouveaux événements)
- les nouveaux événements ne peuvent pas prendre le contrôle indéfiniment ("livelock" interdit)
- la vivacité de l'abstraction est préservée.

Ces principes se traduisent par les obligations de preuve ci-après. Soient D les déclarations constantes des deux niveaux, $I(x)$ l'invariant de l'abstraction et $J(y)$ l'invariant du raffinement (on suppose que les variables x et y sont distinctes). On note E^a un événement de l'abstraction et E^c le même événement raffiné (concret). La cohérence du raffinement entre les deux niveaux est donnée par :

$$D \ \wedge \ I(x) \ \wedge \ J(y) \ \Rightarrow \ [E^c] \neg [E^a] \neg J(y) \qquad \text{(REF1)}$$

Les nouveaux événements N^c raffinent skip. Si l'on remplace E^a par skip dans la formule ci-dessus, on obtient après simplification la simple préservation de l'invariant du raffinement :

$$D \ \wedge \ I(x) \ \wedge \ J(y) \ \Rightarrow \ [N^c] J(y) \qquad \text{(REF2)}$$

Les nouveaux événements ne doivent pas prendre le contrôle indéfiniment. On peut vérifier cette propriété en donnant une expression V (un variant) sur un ordre bien fondé, qui décroît strictement à chaque exécution d'un nouvel événement. On en déduit qu'il n'y a pas de boucle infinie d'exécution avec uniquement des nouveaux événements. Cette obligation de preuve s'écrit, avec v, une variable "fraîche" :

$$D \ \wedge \ I(x) \ \wedge \ J(y) \ \wedge \ v = V \ \Rightarrow \ [N^c] (V < v) \qquad \text{(REF3)}$$

La conservation de la vivacité de l'abstraction signifie que, pour toute trace d'événements abstraits, $E_1^a \longrightarrow E_2^a \ldots \longrightarrow E_k^a \ldots$, il existe une trace d'événements concrets qui passe par les mêmes événements raffinés, entrecoupés éventuellement de nouveaux événements : $N_1^c \longrightarrow E_1^c \longrightarrow N_2^c \longrightarrow N_3^c \longrightarrow E_2^c \ldots \longrightarrow E_k^c \ldots$. Cette condition s'écrit :

$$D \ \wedge \ I(x) \ \wedge \ J(y) \ \wedge \ \mathsf{grd}(E^a) \ \Rightarrow \ \mathsf{grd}(E^c) \ \vee \ \bigvee_j \mathsf{grd}(N_j^c) \qquad \text{(REF4)}$$

pour j parcourant l'ensemble des nouveaux événements.

Il est intéressant de développer la règle du raffinement des événements (REF1) qui est la même que celle du raffinement des opérations (paragraphe 4.1.2, définition (6)), sachant que la terminaison de l'événement abstrait est vraie pour les systèmes. Pour simplifier, prenons l'événement E^a = SELECT P THEN S END qui est raffiné par E^c = SELECT Q THEN T END. La formule donne (en omettant les variables libres des prédicats) :

$$\begin{array}{l} D \ \wedge \ I \ \wedge \ J \ \Rightarrow \ [Q \Longrightarrow T] \neg [P \Longrightarrow S] \neg J \\ D \ \wedge \ I \ \wedge \ J \ \Rightarrow \ [Q \Longrightarrow T] \neg (P \Rightarrow [S] \neg J) \\ D \ \wedge \ I \ \wedge \ J \ \Rightarrow \ (Q \Rightarrow [T](P \ \wedge \ \neg [S] \neg J)) \\ D \ \wedge \ I \ \wedge \ J \ \Rightarrow \ (Q \Rightarrow (P \ \wedge \ [T] \neg [S] \neg J)) \end{array}$$

La dernière transformation est correcte car les variables de P (abstraction) sont distinctes de celles de T (raffinement). D'après les propriétés des substitutions, cette obligation de preuve se décompose en deux cas :

$$\begin{array}{l} D \ \wedge \ I(x) \ \wedge \ J(y) \ \wedge \ Q \ \Rightarrow \ P \\ D \ \wedge \ I(x) \ \wedge \ J(y) \ \wedge \ Q \ \Rightarrow \ [T] \neg [S] \neg J(y) \end{array}$$

Le premier cas indique que la garde du raffinement Q doit impliquer la garde de l'abstraction P. Donc, dans le raffinement des événements, il y a un *renforcement des gardes*. Ceci est à comparer au raffinement des opérations, où il y a un *affaiblissement* des préconditions. C'est pour cela que la condition REF4 est importante, car sinon, les systèmes pourraient restreindre les comportements au fur et à mesure des raffinements, à cause de ce renforcement des gardes.

Exercice 1 : Calculer les obligations de preuve de raffinement dans les autres cas syntaxiques d'événements. Vérifier que l'on a bien toujours un renforcement de la garde.

Exercice 2 : Calculer les obligations de preuve du raffinement de la figure 7.4.

7.3 Propriétés dynamiques

7.3.1 Vivacité

Les propriétés de sûreté définissent les conditions dans lesquelles le système doit fonctionner. Il donne donc les conditions néfastes à éviter. Les propriétés de vivacité expriment au contraire ce qui doit arriver quand le système fonctionne. Ce sont, en quelque sorte, les bonnes conditions à remplir. Une première série de propriétés concerne la vivacité et l'absence de blocage ("deadlock freeness"). Pour un système, la condition de non-blocage est donnée, dans les notations du paragraphe 7.1, par :

$$\boxed{D \ \wedge \ I(x) \ \Rightarrow \ \bigvee_i \mathsf{grd}(E_i)} \qquad (\mathsf{LIV1})$$

avec i parcourant l'ensemble des événements.

La préservation de cette vivacité par un raffinement est donnée, avec les notations du paragraphe 7.2, par :

$$D \wedge I(x) \wedge J(y) \wedge \bigvee_i \mathsf{grd}(E_i^a) \Rightarrow \bigvee_i \mathsf{grd}(E_i^c) \vee \bigvee_j \mathsf{grd}(N_j^c) \qquad \text{(LIV2)}$$

7.3.2 Modalités

Les modalités sont des propriétés sur les traces d'états. Un des formalismes les plus connus est la logique linéaire temporelle (LTL). Les opérateurs usuels sont $\Box P$ qui signifie que sur chaque trace, tous les états satisfont le prédicat P, et $\Diamond P$ qui signifie que sur chaque trace, il existe des états qui satisfont P. On a aussi la notation $\bigcirc P$ pour un état, qui signifie que P est vrai dans l'état suivant.

En B événementiel, l'article [AM98] donne les façons de spécifier quelques modalités et la façon de les prouver.

Futur immédiat

Il y a deux manières d'exprimer qu'une propriété P est vraie au pas suivant d'exécution, en tenant éventuellement compte d'une condition avant. Ces formes de modalités sont :

BEGIN
E_1 OR ...OR E_n
ESTABLISH
$P(x_0, x)$
END

SELECT
$C(x)$
THEN
E_1 OR ...OR E_n
ESTABLISH
$P(x_0, x)$
END

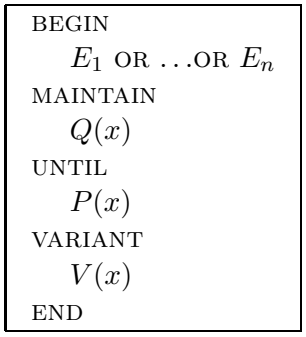
Elles signifient : par un pas d'exécution d'un des événements E_1,..., E_n, le système doit se trouver dans un état qui satisfait $P(x_0, x)$, sans condition initiale (cas BEGIN) ou avec condition initiale $C(x)$ (cas SELECT). Les obligations de preuve générées par ces modalités sont, pour tous les E_i dans la liste (cas SELECT) :

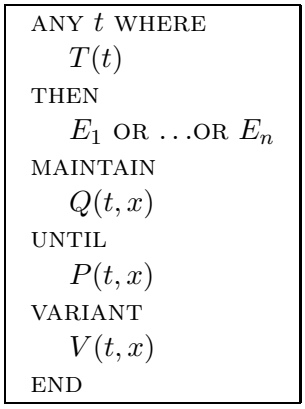
$$D \wedge I \wedge C(x) \wedge x = x_0 \Rightarrow [E_i]\, P(x_0, x) \qquad \text{(MOD1)}$$

Futur imprécis

Cette modalité est de la forme $\Box(R \Rightarrow \Diamond P)$, qui signifie que, sur chaque trace, si R est vrai pour un certain état, alors fatalement, P sera vrai dans un état futur de la trace. Cette propriété est aussi notée dans d'autres formalismes $R \leadsto P$, c'est-à-dire R conduit à P. En B, il faut s'assurer que les propriétés soient démontrables par un processus de déduction. Le fait d'atteindre un but s'apparente à la terminaison d'une boucle. Il faut donc, comme dans une boucle en B, disposer *syntaxiquement* d'un variant et, le plus souvent d'un invariant local. Donc, plutôt que d'avoir une propriété "R conduit à P", on s'intéresse à des modalités du type "Q est vrai jusqu'à ce que P soit vrai", Q jouant le rôle d'un invariant. Comme pour la modalité de futur immédiat, on peut donner une liste d'événements qui doivent être pris en compte dans la progression (cette liste pouvant être

tous les événements, mot-clé ALL). Il est aussi possible d'utiliser des variables auxiliaires t qui doivent être typées par un prédicat $T(t)$. Dans les formules, $V(t,x)$ est une expression sur un ordre bien fondé. Cela donne :

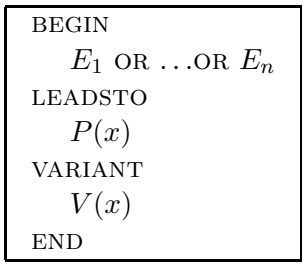


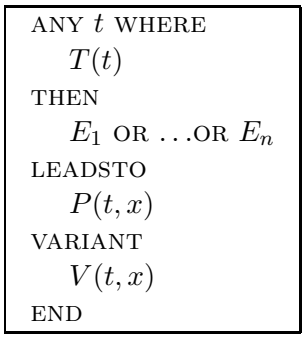


Les obligations de preuve engendrées pour cette modalité sont (cas ANY), avec K mis pour $D \wedge I(x) \wedge T(t) \wedge Q(t,x) \wedge \neg P(t,x)$:

$$
\begin{array}{l}
K \Rightarrow \mathsf{grd}(E_1) \vee \ldots \vee \mathsf{grd}(E_n) \\
K \Rightarrow [E_i]\,(P(t,x) \vee Q(t,x)) \\
K \wedge v = V(t,x) \Rightarrow [E_i]\,(\neg P(t,x) \Rightarrow V(t,x) < v)
\end{array}
\qquad \text{(MOD2)}
$$

Le prédicat K est vrai, tant qu'on progresse vers P sans l'avoir atteint. La première obligation de preuve indique que le système ne se bloque pas dans la progression. La deuxième dit que chaque événement atteint P ou maintient Q. La troisième dit que chaque événement atteint P ou sinon fait décroître le variant V. Comme V repose sur un ordre bien fondé, cette décroissance est finie, et donc P est fatalement atteint. Pour terminer, notons qu'il existe une simplification des modalités lorsque Q est inutile (toujours vrai) :





Dans le **raffinement**, il faut s'assurer que les modalités sont préservées. Une modalité de futur immédiat est validée par une obligation de preuve de plus faible précondition qui est préservée par raffinement. Mais la propriété devient une propriété de futur imprécis, à cause de l'introduction des nouveaux événements. Cependant, par les obligations de preuve REF3 et REF4, ces nouveaux événements ne peuvent pas prendre le contrôle indéfiniment

et il ne peut pas y avoir de nouveaux blocages, donc les modalités de futur immédiat sont préservées.

Le raisonnement est sensiblement le même pour la préservation des modalités de futur imprécis. Le seul point à vérifier est que le raffinement des événements cités dans la modalités permette la progression tant que l'état qui satisfait le prédicat P n'est pas atteint. Cela conduit à l'obligation de preuve (avec les mêmes notations que ci-dessus) :

$$K \ \wedge \ J(x,y) \ \wedge \ \mathsf{grd}(E_1^a) \ \vee \ \ldots \ \vee \ \mathsf{grd}(E_n^a) \\ \Rightarrow \ \mathsf{grd}(E_1^c) \ \vee \ \ldots \ \vee \ \mathsf{grd}(E_n^c) \ \vee \ \bigvee_j \mathsf{grd}(N_j^c) \qquad \text{(REF-MOD2)}$$

Si la modalité concerne tous les événements (ce qui est le cas le plus fréquent), alors cette obligation de preuve est la même que LIV2, c'est-à-dire la préservation du non-blocage du système.

7.4 Fin de l'exemple du parking

7.4.1 Deuxième raffinement

Dans ce raffinement, on introduit le feu d'entrée. Le feu peut être positionné à vert ou à rouge. On a donc un type énuméré qui représente l'état du feu. Dans l'invariant, on trouve le lien entre l'état du feu et le nombre de places dans le cas d'attente d'entrée et de sortie ($cc = 0$) ainsi que les conséquences dans les cas $cc = 1$ et $cc = -1$.

REFINEMENT *ParkingR2*
REFINES
ParkingR1
SETS
$COULEUR\_FEU = \{vert, rouge\}$
VARIABLES
$vehi, cc, feu$
INVARIANT
$feu \in COULEUR\_FEU \ \wedge$
$(cc = 0 \ \wedge \ feu = vert \Rightarrow vehi < nbPlaces) \ \wedge$
$(cc = 0 \ \wedge \ feu = rouge \Rightarrow vehi = nbPlaces) \ \wedge$

$(cc = 1 \Rightarrow feu = vert) \ \wedge$ /* une voiture rentre */
$(cc = -1 \ \wedge \ vehi = nbPlaces - 1 \Rightarrow feu = rouge) \ \wedge$
$(cc = -1 \ \wedge \ vehi < nbPlaces - 1 \Rightarrow feu = vert)$

FIG. 7.5 – Déclarations du deuxième raffinement

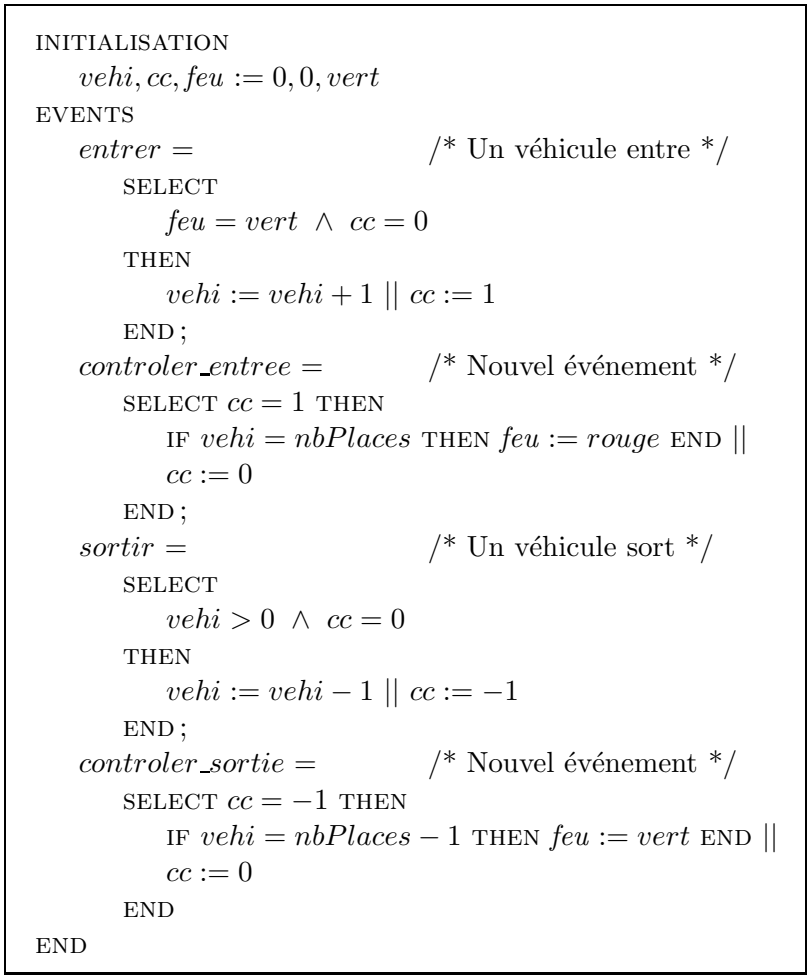


FIG. 7.6 – Evénements du deuxième raffinement

7.4.2 Troisième raffinement

Dans le dernier raffinement, tous les calculs sont reportés dans les événements de contrôle. Le contrôleur gère donc lui-même le nombre voitures par un compteur local nv, qui est l'information *qu'il connaît* sur le nombre de véhicules dans le parking. De même, on supprime le test sur la valeur des feux. Le contrôleur ne sait que positionner à *vert* ou à *rouge*. C'est un choix technique : envoi d'un signal à l'équipement. La variable *feu* est considérée, ici, comme un canal et l'affectation est l'envoi de l'information dans le canal. Le canal est relié à l'équipement qui est censé être fiable (hypothèse à introduire dans le cahier des charges). Si le canal ou l'équipement n'est pas fiable, il convient de raffiner encore, avec une méthode "d'acknowledgement" et de reprise, suivant la politique des protocoles. On touche du doigt le raffinement du temps, où une simple affectation à un certain niveau abstrait peut devenir une suite complexe de pas élémentaires. Le dernier raffinement du "Parking" se trouve à la figure 7.7.

Quelques remarques :
(1) Du point de vue du contrôleur, les gardes de *entrer* et *sortir* sont des conditions qui doivent être vraies lorsque le signal d'entrée ou de sortie d'une voiture lui parvient. Si ces

```
REFINEMENT ParkingR3
REFINES
    ParkingR2
VARIABLES
    cc, feu, nv
INVARIANT
    nv ∈ NAT  ∧  nv ∈ 0..nbPlaces
    (cc = 0 ⇒ nv = vehi) ∧                /* invariant de liaison */
    (cc = 1 ⇒ nv = vehi − 1) ∧
    (cc = −1 ⇒ nv = vehi + 1) ∧
                                  /* retranscription des liens avec le feu */
    (cc = 0  ∧  nv = nbPlaces ⇒ feu = rouge) ∧
    (cc = 0  ∧  nv < nbPlaces ⇒ feu = vert)
INITIALISATION
    cc, feu, nv := 0, vert, 0
EVENTS
    entrer =                              /* Un véhicule entre */
        SELECT
            cc = 0  ∧  nv < nbPlaces
        THEN
            cc := 1
        END ;
    controler_entree =
        SELECT cc = 1 THEN
            nv := nv + 1  ;
            IF nv = nbPlaces THEN feu := rouge END  ;
            cc := 0
        END ;
    sortir =                              /* Un véhicule sort */
        SELECT
            cc = 0  ∧  nv > 0
        THEN
            cc := −1
        END ;
    controler_sortie =
        SELECT cc = −1 THEN
            IF nv = nbPlaces THEN feu := vert END  ;
            nv := nv − 1  ;
            cc := 0
        END
END
```

FIG. 7.7 – Troisième raffinement

conditions ne sont pas remplies, cela lui indique qu'il y a un disfonctionnement matériel

(une panne du feu, par exemple). Ce pourrait être le point de départ d'une vérification supplémentaire, ou d'émission d'un signal d'alarme.
(2) Dans cette version, le contrôleur n'a pas une image de l'état des feux. On aurait pu l'ajouter facilement. En règle générale, les contrôleurs gardent souvent une image de l'environnement contrôlé, justement pour détecter les pannes, s'il y a une divergence entre leur image et ce que leur renvoient les capteurs.
(3) Les substitutions des événements du contrôleur sont mises en séquence pour se rapprocher d'une implémentation.

Exercice 3 : Montrer que le système "Parking" ne se bloque pas.

Exercice 4 : Peut-on démontrer que toute voiture qui est entrée dans le parking finit par en sortir ? Pourquoi ? Quelle information faudrait-il introduire dans le système pour avoir une telle propriété ?

Le système Parking a été prouvé entièrement avec la *Balbulette*[2]. Le nombre des obligations de preuve et leur type de preuve (automatique ou interactif) est donné ci-dessous.

	Nbre d'OPs	automatiques	interactives	auto/total
Parking	3	3	0	100%
ParkingR1	11	11	0	100%
ParkingR2	19	15	4	79%
ParkingR3	43	36	7	84%

7.5 L'atelier B RODIN

7.5.1 Principes généraux

L'atelier B RODIN a été conçu et implémenté dans le cadre du projet européen RODIN : "Rigourous Open Development Environment for Complex Systems". Il s'est accompagné d'une mise à jour de la syntaxe du B sans modifier les notions mathématiques sous-jacentes. On peut dire que le nouveau langage "Event-B"[3] est un B événementiel primitif, donc plus simple, mais est aussi plus puissant que le B classique étendu pour "faire" des systèmes d'actions. Cependant, sa puissance ne peut s'exprimer que si de nombreux outils sont intégrés au noyau. C'est pour cela que la structure de l'atelier RODIN est celle d'un logiciel libre développé en java, qui propose une plateforme (dérivée de "Eclipse") sur laquelle on peut déposer des plugins, ce qui facilite l'extension de la plateforme. Au jour d'aujourd'hui, l'atelier RODIN est téléchargeable et utilisable[4], mais il est encore dans une phase expérimentale.

Les idées-force de RODIN sont :

– La construction d'un modèle se fait d'une manière incrémentale.

[2]URL : http ://www.b4free.com/.
[3]Pour la clarté, nous l'appelons ainsi, pour indiquer la différence avec le B événementiel décrit dans les paragraphes précédents.
[4]URL : http ://rodin-b-sharp.sourceforge.net/

– L'atelier prend les éléments d'un modèle "comme ils viennent" et l'utilisateur peut demander à tout moment de valider ou d'invalider ses fragments de modèles par la preuve. Donc, la preuve devient un outil d'aide à la modélisation.

– Tous les objets d'un modèle sont nommés. Cela était déjà le cas pour les variables, les constantes et les événements. Dans Event-B, on nomme aussi les prédicats (dans les invariants et les propriétés), les gardes, et les modifications élémentaires de l'état (les actions). L'objectif est de pouvoir tracer l'origine des obligations de preuve qui sont générées par l'atelier, justement pour pouvoir utiliser les résultats de la preuve dans la mise au point des modèles.

– Les événements sont les briques de base qui opèrent des changements d'états sous certaines conditions. L'agencement de ces événements primitifs permet de construire des logiciels séquentiels, distribués, de contrôle, réactifs, etc. mais cet agencement est fait dans d'autres étapes qui ne sont pas encore prises en compte dans l'atelier. On peut trouver des exemples de ces agencements dans des études de cas.

– De même que la notion d'événement est simplifiée, les théories mathématiques prédéfinies sont aussi réduites à un noyau qui comprend la logique du premier ordre avec égalité, les ensembles (comme définis dans le chapitre 1, à quelques détails près) et les entiers naturels. Ce noyau peut être enrichi par l'utilisateur en développant des bibliothèques spécialisées. On sait, par exemple, que les types inductifs sont définissables à l'aide d'axiomes du premier ordre quantifiés sur des sous-ensembles.

7.5.2 Les composants Event-B

Il y a deux sortes de composants en Event-B :

– les **modèles** (appelés aussi "machines" dans l'atelier, mais nous gardons le nom "modèles" pour éviter la confusion) : ce sont les composants qui décrivent l'état et les événements d'un système.

– les **contextes** : composants qui fournissent les déclarations statiques et les propriétés des structures de données du système à modéliser.

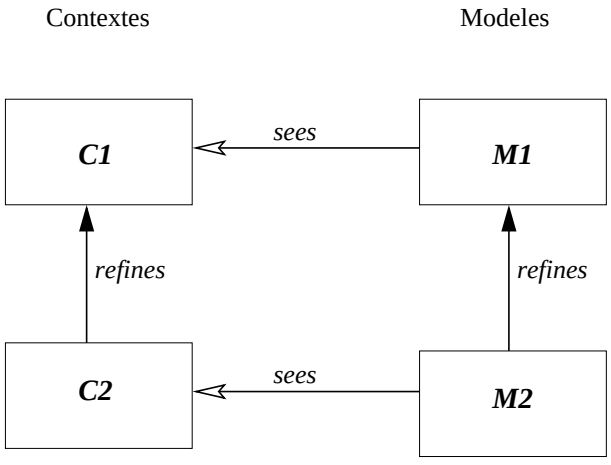


FIG. 7.8 – Relations entre composants en RODIN

Les modèles peuvent **voir** des contextes (clause SEES), pour accéder à la théorie sur laquelle un modèle est construit. Un modèle peut être **raffiné** par un autre modèle, de la même manière que le raffinement vu au paragraphe 7.2. Une différence importante vient du fait qu'un événement peut être raffiné par plusieurs événements (*découpage des événements*) et aussi, inversement, un événement peut en raffiner plusieurs (*fusion des événements).* Une fusion de deux événements E_1 et E_2 est équivalente au choix borné des deux événements $E_1 \parallel E_2$. Un contexte peut être **raffiné** par un autre contexte. Le raffinement de contexte est simplement une *extension* des déclarations du premier contexte. Un modèle raffiné M_2 peut voir le raffinement C_2 d'un contexte C_1 vu par l'abstraction M_1 du modèle raffiné, comme indiqué à la figure 7.8. Les relations de raffinement entre composants sont transitives (et réflexives). Les ensembles et les constantes déclarées dans un contexte sont, en quelque sorte, des **paramètres** du modèle. Il est prévu d'avoir une opération **d'instantiation** des modèles, c'est-à-dire remplacer un contexte abstrait par des ensembles et des constantes d'une théorie plus spécifique.

La fenêtre de l'éditeur de l'atelier Rodin est visualisée à la figure 7.9, avec l'exemple du "Parking". On remarque le nommage du prédicat de l'invariant, de la garde et de l'action de l'événement *entrer*. Ce modèle voit le contexte *Places* qui a été défini pour déclarer et axiomatiser la constante *nbPlaces*.

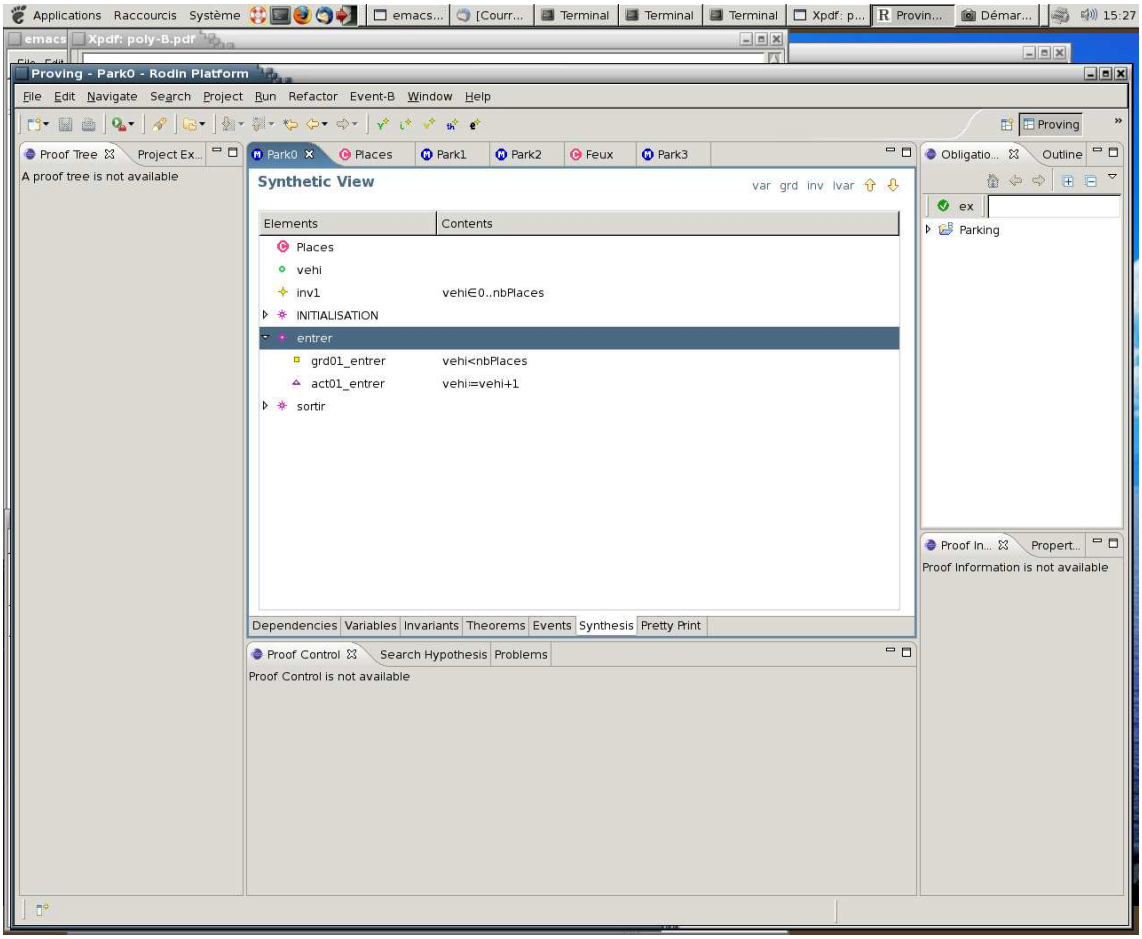


FIG. 7.9 – Editeur RODIN

7.5.3 Les preuves dans l'atelier RODIN

L'atelier RODIN offre deux "perspectives" (vues) : édition et preuve. Le prouveur reprend les principes de l'outil "Click'n Prove" de la Balbulette[5]. Les hypothèses d'une formule à prouver sont divisées en hypothèses *sélectionnées*, hypothèses dans *le cache* et autres hypothèses. Les procédures de preuve sont essentiellement le **prouveur de prédicat** et le **prouveur général**, avec plus ou moins de "force" dans les stratégies proposées. Si les procédures automatiques de décision échouent, il reste à faire une preuve "à la main" en appliquant des règles d'inférence. Ces règles sont accessibles par des boutons, et dépendent du but ou de la forme de certaines hypothèses (l'outil indique quelles sont les règles applicables). L'utilisateur peut toujours introduire des lemmes auxiliaires pour avancer dans la preuve. Pour plus de détails sur les principes du prouveur de prédicat, voir l'article [AC03].

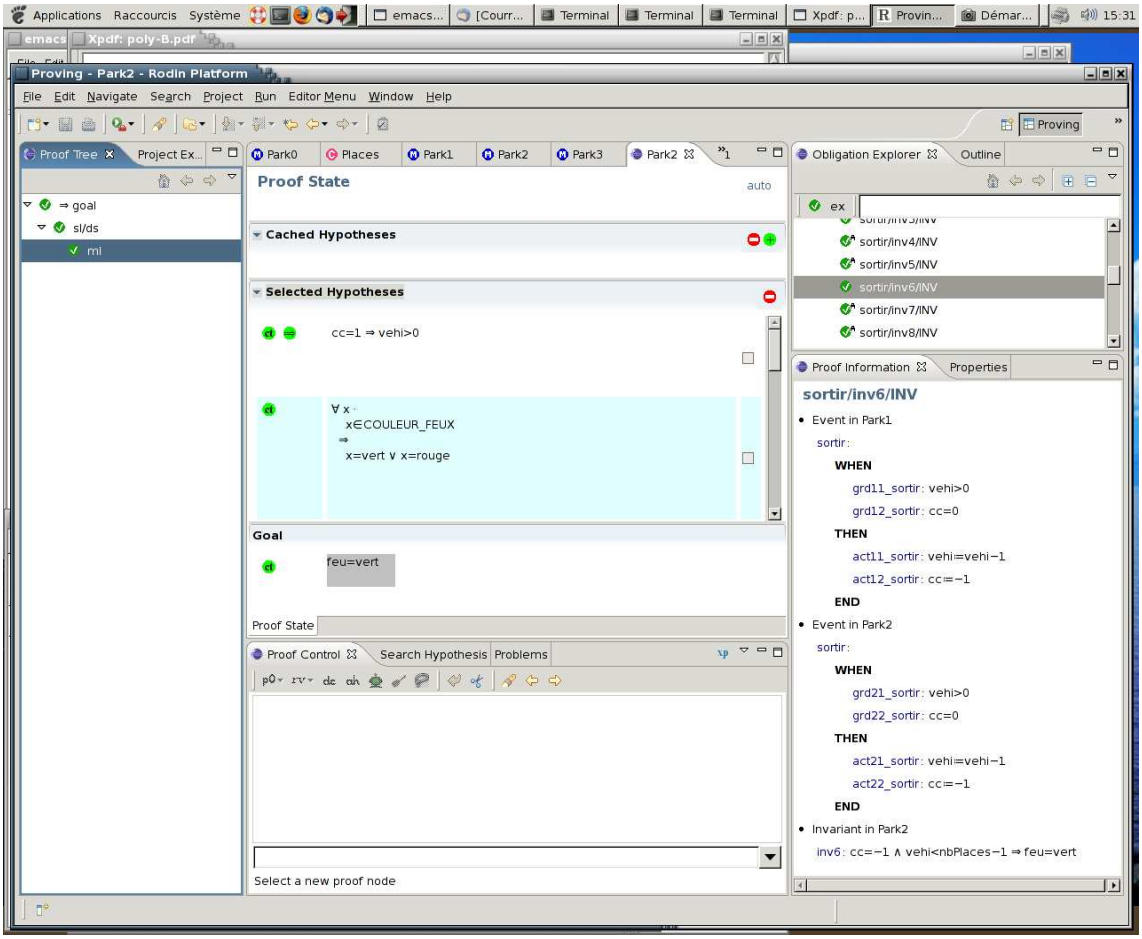


FIG. 7.10 – Prouveur RODIN

Les fenêtres de l'environnement de preuve sont visualisées à la figure 7.10. A droite, en haut, il y a le composant considéré et les obligations de preuves prouvées (rond vert) ou non prouvées. En dessous, on trouve les éléments permettant de connaître d'où vient

[5]URL : http ://www.b4free.com/

cette OP (ici, c'est le raffinement d'un événement et un invariant de collage). Au centre, on trouve le but courant et les hypothèses sélectionnées. On peut y activer les règles d'inférence et les procédures de décision. A gauche, il y a l'arbre de la preuve en cours.

Bibliographie

[Abr96] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[AC03] J.-R. Abrial and Dominique Cansell. Click'n Prove : Interactive Proofs Within Set Theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2003.

[AM98] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98 : Recent Advances in the Development and Use of the B Method, Proc. of the 2nd International B Conference, LNCS 1393*, pages 83–128. Springer-Verlag, 1998.

[Dun99] S. Dunne. The Safe Machine : A New Specification Construct for B. In *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, september 1999.

[Hab01] H. Habrias. *Spécification formelle avec B*. Hermès Science Publications, 2001.

[MAV05] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B Language. Technical report, Project IST-511599, RODIN, URL :http ://rodin.cs.ncl.ac.uk, 2005.

[Pot02] M-L. Potet. *Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B*. Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, 5 décembre 2002.

Notations du langage B classique

Composants

MACHINE	machine abstraite
REFINEMENT	raffinement
IMPLEMENTATION	implémentation
END	fin de composant, de substitution

Relations entre composants (non décrites dans le polycopié)

EXTENDS	extension d'une machine
IMPORTS	importation d'une machine
INCLUDES	inclusion de machine
PROMOTES	promotion des opérations
REFINES	raffinement
SEES	accès en lecture à une autre machine
USES	inclusion avec partage

Clauses des machines, raffinements ou implémentations

ABSTRACT_CONSTANTS	déclaration de constantes abstraites
ABSTRACT_VARIABLES	déclaration (explicite) de variables abstraites
ASSERTIONS	déclaration d'assertions
CONCRETE_CONSTANTS	déclaration (explicite) de constantes concrètes
CONCRETE_VARIABLES	déclaration de variables concrètes
CONSTANTS	déclaration de constantes concrètes
CONSTRAINTS	contraintes sur les paramètres de machines
DEFINITIONS	définitions (macros)
INITIALISATION	initialisation
INVARIANT	invariant (propriétés des variables)
OPERATIONS	opérations
PROPERTIES	propriété des constantes
SETS	déclaration d'ensembles
VALUES	valuation des constantes et des ensembles différés
VARIABLES	déclaration de variables abstraites

Mots-clés des substitutions généralisées

ANY	substitution gardée de choix non borné
ASSERT	substitution d'assertion
BEGIN	substitution simple ou séquentielle
CASE	substitution de choix par cas
CHOICE	substitution de choix borné
IF	substitution conditionnelle
LET	substitution de choix affecté
PRE	substitution préconditionnée
SELECT	substitution gardée
VAR	substitution de choix non borné (ou entrée de bloc)
WHILE	substitution de boucle
BE	séparateur dans LET
DO	séparateur dans WHILE
EITHER	séparateur dans CASE
ELSE	séparateur dans IF, SELECT, CASE
ELSIF	séparateur dans IF
IN	séparateur dans VAR, LET
INVARIANT	séparateur dans WHILE
OF	séparateur dans CASE
OR	séparateur dans CHOICE, CASE
THEN	séparateur dans PRE, IF, SELECT, CASE, ANY
VARIANT	séparateur dans WHILE
WHEN	séparateur dans SELECT
WHERE	séparateur dans ANY

Dans la suite, la deuxième colonne indique la notation ASCII des symboles (si le symbole peut être utilisé dans les programmes B).

Symboles dans les substitutions généralisées

$\mid$		symbole de précondition
$\Longrightarrow$		symbole de garde `==>`
$[\!]$		symbole de choix `[]`
@		symbole @
$\sqsubseteq$		symbole de raffinement
$:=$	`:=`	substitution simple
skip	`skip`	substitution sans effet
$:\in$	`::`	prend une valeur dans
:	`:`	devient tel que
$\mid\mid$	`||`	substitution multiple
;	`;`	composition des substitutions
$\longleftarrow$	`<--`	symbole "retour de fonction"

Opérations sur les substitutions généralisées

trm	terminaison
fis	faisabilité
prd	prédicat avant-après
pre	ensemble “pre” d’une substitution
rel	relation “rel” d’une substitution
str	transformateur d’ensemble
set	ensemble satisfaisant un prédicat
\	symbole “n’est pas libre dans”

Symboles logiques

$\forall$	`!`	quantificateur pour tout
$\exists$	`#`	quantificateur “il existe”
$\cdot$	`.`	point qui suit les quantificateurs
λ	`%`	définition de fonction
$\neg$	`not`	symbole négation logique
$\wedge$	`&`	symbole et logique
$\vee$	`or`	symbole ou logique
$\Rightarrow$	`=>`	symbole implique logique
$\Leftrightarrow$	`<=>`	symbole equivalent logique
$=$	`=`	égalité de deux valeurs
$\neq$	`/=`	non égalité de deux valeurs
btrue		prédicat constant “vrai”
bfalse		prédicat constant “faux”

Constructeurs d’ensembles

$\emptyset$	`{}`	ensemble vide
$\times$	`*`	produit cartésien
$\mapsto$	`|->`	maplet (doublet de valeurs)
$\mathbb{P}$	`POW`	parties d’un ensemble
$\mathbb{P}_1$	`POW1`	parties non vides d’un ensemble
$\mathbb{F}$	`FIN`	parties finies d’un ensemble
$\mathbb{F}_1$	`FIN1`	parties finies non vides
..	`..`	intervalle

Relations et fonctions

$\leftrightarrow$	`<->`	ensemble des relations
$\rightarrow$	`-->`	ensemble des fonctions totales
$\nrightarrow$	`+->`	... fonctions partielles
$\rightarrowtail$	`>->`	... fonctions injectives totales
$\nrightarrowtail$	`>+>`	... fonctions injectives partielles
$\twoheadrightarrow$	`-->>`	... fonctions surjectives totales
$\nrightarrow\!\!\!\!\twoheadrightarrow$	`+->>`	... fonctions surjectives partielles
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	`>->>`	... fonctions bijectives totales
$\nrightarrowtail\!\!\!\!\twoheadrightarrow$	`>+>>`	... fonctions bijectives partielles

Prédicats sur les ensembles

$\in$	`:`	appartient
$\notin$	`/:`	n'appartient pas
$\subseteq$	`<:`	est inclus dans
$\not\subseteq$	`/<:`	n'est pas inclus dans
$\subset$	`<<:`	est strictement inclus dans
$\not\subset$	`/<<:`	n'est pas strictement inclus dans

Opérations sur les ensembles

$\cup$	`\/`	union d'ensembles
$\cap$	`/\`	intersection d'ensembles
$-$	`-`	soustraction d'ensembles
union	`union`	union d'un ensemble d'ensembles
inter	`inter`	intersection d'un ensemble d'ensembles
$\bigcup$	`UNION`	union quantifiée
$\bigcap$	`INTER`	intersection quantifiée

Opérations sur les relations et les fonctions

dom	`dom`	ensemble domaine
ran	`ran`	ensemble rang ou codomaine
id	`id`	relation identité
prj_1	`prj1`	première projection d'une relation
prj_2	`prj2`	deuxième projection
rel	`rel`	transformée d'une fonction en relation
fnc	`fnc`	transformée d'une relation en fonction
;	`;`	composition séquentielle des relations
$\otimes$	`><`	produit direct de relations
$\|\|$	`||`	produit parallèle de relations

R^{-1}	`R~`	relation inverse
R^*	`closure(`R`)`	fermeture réflexive transitive
R^+	`closure1(`R`)`	fermeture transitive

$\lhd$	`<|`	restriction de domaine
$\not\lhd$	`<<|`	soustraction de domaine
$\rhd$	`|>`	restriction de codomaine
$\not\rhd$	`|>>`	soustraction de codomaine
$\lhd\!\!+$	`<+`	overriding ou surcharge
$r[s]$	`r[s]`	image de l'ensemble s par la relation r
$f(x)$	`f(x)`	valeur de la fonction f au point x

Ensembles et constantes arithmétiques prédéfinis

$\mathbb{N}$	`NATURAL`	entiers naturels
$\mathbb{N}^+$	`NATURAL1`	entiers positifs
$\mathbb{Z}$	`INTEGER`	entiers relatifs
NAT	`NAT`	entiers naturels représentables
NAT_1	`NAT1`	entiers positif représentables
INT	`INT`	entiers relatifs représentables
minint	`MININT`	entier minimum représentable
maxint	`MAXINT`	entier maximum représentable

Opérations arithmétiques

$+$	`+`	addition
$-$	`-`	soustraction et moins unaire
$\times$	`*`	multiplication
$/$	`/`	quotient de la division entière
mod	`mod`	reste de la division entière
x^y	`x**y`	puissance entière
succ	`succ`	fonction succ des entiers
pred	`pred`	fonction pred des entiers
Σ	`SIGMA`	somme quantifiée d'entiers
Π	`PI`	produit quantifié d'entiers
card	`card`	cardinal d'un ensemble
min	`min`	minimum d'un ensemble d'entiers
max	`max`	maximum d'un ensemble d'entiers

Prédicats arithmétiques

$\leq$	`<=`	plus petit ou égal
$<$	`<`	strictement plus petit
$\geq$	`>=`	plus grand ou égal
$>$	`>`	strictement plus grand

Chaînes de caractères et booléens

CHAR		ensemble des caractères
STRING	`STRING`	ensemble des chaînes de caractères
BOOL	`BOOL`	ensemble des booléens
bool	`bool`	bool (prédicat devient booléen)
TRUE	`TRUE`	constante booléenne true
FALSE	`FALSE`	constante booléenne false

Séquences prédéfinies

seq	`seq`	les séquences
seq_1	`seq1`	les séquences non vides
iseq	`iseq`	les séquences injectives
iseq_1	`iseq1`	les séquences injectives non vides
perm	`perm`	les permutations (bijectives)
$[\,]$	`[]`	la séquence vide
$\rightarrow$	`->`	insertion à gauche
$\leftarrow$	`<-`	insertion à droite
$\frown$	`^`	concaténation des séquences
conc	`conc`	concaténation généralisée
$\uparrow$	`/|\`	restriction de bas vers le haut
$\downarrow$	`|/`	restriction du haut vers le bas
size	`size`	longueur de séquence
first	`first`	premier élément
last	`last`	dernier élément
front	`front`	début de séquence (sauf le dernier)
tail	`tail`	fin de séquence (sauf le premier)
rev	`rev`	inversion de séquence