Génie Logiciel
Introduction à la méthode B

Christine Paulin

Maîtrise 2001-2002

Table des matières

1 Introduction

1.1 Objectifs du cours

Les méthodes de modélisation (Fusion, UML) permettent d'analyser un *cahier des charges* pour identifier et concevoir les différents *composants* du logiciel à développer et préciser les *comportements attendus*.
Des notations (essentiellement des schémas) sont introduits pour faciliter une description non ambiguë du problème. Cependant ces méthodes restent *informelles*.
– le sens des schémas dans des cas complexes n'est pas toujours évident,
– le passage de la description au code est fait manuellement,
– les propriétés attendues des opérations sont généralement décrites en langue naturelle et ne sont pas vérifiées. Le langage OCL permet une description plus formelle, mais la sémantique n'est pas toujours claire et la vérification est laissée à l'utilisateur.
Dans cette partie du cours, nous allons présenter une méthode rigoureuse de développement logiciel. Il s'agit d'introduire dans le procédé de modélisation:
– la notion de spécification formelle
– la notion de preuve
Le cours permettra également une initiation à la méthode B qui est utilisée dans certaines entreprises (Alsthom, Matra, Gemplus ...) pour la modélisation de logiciels critiques.

1.2 Contexte et motivations

1.2.1 Modélisation

Pour construire un logiciel, on démarre de spécifications générales informelles, nécessitant une modélisation de l'environnement dans lequel le logiciel sera utilisé. Cette modélisation est forcément approximative. Petit-à-petit le modèle est raffiné aussi bien dans la représentation des données que dans celle des opérations. On passe à un modèle *mathématique* qui permet le traitement rigoureux du problème.

1.2.2 Spécification formelle

C'est l'écriture de ce qui est attendu d'un programme dans un langage *formel* (analogue aux mathématiques) non ambiguë. Les spécifications formelles permettent de raisonner rigoureusement sur les propriétés attendues du programme (établir des invariants, montrer la complétude des spécifications ...)

1.2.3 Modèle d'un programme

Pour prouver qu'un programme réalise bien ce que l'on attend de lui, il faut avoir un *modèle* de l'exécution du programme.
Un modèle usuel d'un programme, est de le voir comme une fonction qui transforme la mémoire de la machine. Nous reviendrons plus en détail sur ce point.
Lorsque le programme et sa spécification sont exprimés dans un langage mathématique, on peut envisager de *prouver* qu'un programme exécute bien les tâches demandées.

1.2.4 Raffinement et réutilisation

Un programme exécute en général des tâches très complexes, le modèle est donc très gros, raisonner dessus est alors impraticable.
Mais il est également illusoire de penser écrire 100000 lignes d'assembleur sans erreur. Les programmes sont construits suivant deux principes : le raffinement et la réutilisation.

Définitions Le *raffinement* est la décomposition d'un problème en sous-problèmes plus simples.
La *réutilisation* est la résolution d'un problème en utilisant des bibliothèques génériques dont les propriétés sont bien établies (efficacité, correction).
La preuve peut suivre le même cheminement, le raffinement correspond à une phase d'*analyse*, la réutilisation à une phase de *synthèse* où on cherche à appliquer des résultats connus (dont on n'est pas forcé de maîtriser la démonstration).

1.3 Applications

Le coût de développement d'une spécification formelle et de la preuve de correction d'un programme est très important. Ceci limite l'application de ces méthodes à des domaines particuliers où les besoins du *zéro-défaut* sont particulièrement importants.
Ce peut être pour diverses raisons :
- mise en danger de vies humaines (transport)
- coût du développement
  - matériel (processeurs)
  - Ariane
- attaques malveillantes (commerce, santé)

La présence de composants logiciels dans des fonctions de plus en plus critiques accroît de manière considérable le besoin de méthodes sûres et efficaces. D'autre part, le développement des logiciels est souvent confié à des organismes de sous-traitance, il faut être capable de vérifier leur qualité, de les intégrer, de les adapter.

1.3.1 Comment s'assurer qu'un programme critique est correct

Certification Les programmes *critiques* ne peuvent être mis en service qu'après avoir fait l'objet d'une *certification* par des instances compétentes. Pour l'instant les critères de certification reposent essentiellement sur le test, et les labels *qualité du procédé de développement* des entreprises. La certification se fait souvent sur des critères d'appréciation plus subjectifs que scientifiques. Cependant, depuis quelques années, des normes sont apparues dans le domaine de la sécurité des informations (critères communs) qui exigent des modélisations formelles et des preuves de correction, pour les normes les plus exigeantes.

Tests Le test est un élément important de la *validation* d'un programme. On peut espérer que des programmes très largement utilisés contiendront moins de défaut que des programmes peu utilisés. Cependant ce n'est vrai que pour des programmes qui évoluent peu : on peut comparer par exemple un traitement de texte commercial dans lequel de nouvelles fonctionnalités (et de nouveaux bugs) sont constamment introduits et un compilateur d'un langage stable. Il n'est pas toujours pertinent de déployer un programme avant d'être sûr de son fonctionnement (penser à Ariane, ou aux cartes bancaires). Tester de manière efficace (en temps, en couverture) un programme est très difficile. Lorsqu'une modification est apportée à un programme, celle-ci peut avoir des répercussions dans d'autres parties du programme, en conséquence les tests doivent souvent être repris à zéro.
Les tests sont indispensables, mais ils ne garantissent pas en général une correction totale et ont également un coût très élevé.

Ce qu'il faut retenir
- La résolution d'un problème informatique nécessite une phase de *modélisation* d'un problème réel à résoudre résultant en une *description des données et des opérations* que le programme devra réaliser.
- La place grandissante du logiciel dans des fonctions *critiques* rend particulièrement importante la *qualité* du logiciel.

– Les *tests* sont indispensables à la *validation* mais ils n'offrent qu'une garantie partielle et sont coûteux à mettre en œuvre.

1.3.2 Langage de spécification

Lorsqu'on modélise un problème, il faut choisir le langage dans lequel seront écrites les spécifications. En Angleterre, s'est développée une méthode appelée Z qui repose sur la théorie des ensembles [6]. L'idée est que les modélisations de systèmes d'information nécessitent de décrire des ensembles, des fonctions et des relations entre ces ensembles. Cette méthode reste très abstraite et n'est pas directement supportée par des outils de programmation ou de preuve. Cependant elle a beaucoup influencé le développement de la méthode B.

1.3.3 Qu'est-ce que B ?

La méthode B a été inventée par Jean-Raymond Abrial au milieu des années 80. Jean-Raymond Abrial était alors consultant auprès de la RATP. On lui a demandé d'évaluer la sécurité des programmes de freinage pour le RER A (la distance entre les rames devait être réduite ce qui rendait critique cette partie du code). Il a rendu un avis négatif sur le code ce qui a entraîné un retard important dans la mise en service.

Son idée a donc été de proposer une méthode à la fois rigoureuse et effectivement utilisable en milieu industriel. La méthode développée se devait de rester proche des habitudes des programmeurs, et être intégrée à un environnement de preuve.

La RATP a beaucoup soutenu le développement des méthodes formelles (en imposant leur utilisation à ses sous-traitants) et en particulier de la méthode B. La partie critique du métro Meteor en particulier a été développée ainsi.

Les grands principes de la méthode B sont :

– Elle intervient tôt dans le développement du logiciel.
– Les spécifications et programmes sont représentés en utilisant une notation de *machine abstraite*.
– Les propriétés sont décrites dans une théorie des ensembles générale.
– Un programme est développé par raffinements successifs de machines abstraites représentant la spécification du programme puis par implantation en utilisant les opérations d'autres machines.
– Chaque étape de raffinement ou d'implantation ne peut être réalisée que si certaines propriétés sont satisfaites. Ces propriétés peuvent être calculées (mais en général pas résolues) automatiquement.

Une machine abstraite se présente comme un objet avec des variables représentant l'état de la machine et des opérations décrites par leur propriétés. Par ces aspects, la méthode B se rapproche des méthodes de modélisation objet. C'est cette notion de machine abstraite et les procédés de raffinement et d'implantation qui lui sont associés qui constitue la partie la plus intéressante de la méthode B.

La phase d'implantation permet d'établir une chaîne sûre entre des spécifications de haut niveau et un code exécutable qui peut être engendré automatiquement.

1.3.4 Outils

Un grand intérêt de la méthode B est d'être supportée par des outils. Il existe essentiellement deux systèmes développés par des sociétés commerciales sous l'impulsion (passée ou présente) de J.-R. Abrial. L'un est `B-tool` développé en Angleterre par la société B-core [3]. L'autre est l'atelier B [2] développé en France par la société ClearSy (anciennement Steria) .

Ce qu'il faut retenir

– Dans le cas de fonctions simples où le comportement du programme et sa spécification sont bien compris, le développeur doit pouvoir apporter la preuve de la correction de son programme.

– Les phases de construction d'un programme (raffinement, réutilisation) correspondent à des étapes de raisonnement logique.
– Le développement de la spécification et de la preuve ne peut se faire qu'à l'aide d'un *langage formel* bien défini et d'*outils de développement* adaptés.

1.4 Plan du cours

1. Bases du langage
2. Machines abstraites
3. Raffinement
4. Implémentation
5. Application

2 Les bases du langage de la méthode B

La méthode B permet de formaliser des spécifications et des programmes. Pour cela, elle utilise son propre langage formel. Au niveau des spécifications, il s'agit d'un langage logique reposant sur une version simplifiée ad-hoc de la théorie des ensembles. Ce choix est un peu arbitraire mais peut se justifier par les arguments suivants :
- il est important de pouvoir décrire des ensembles, des relations et des fonctions (ce sont les concepts centraux dans les modélisations objet),
- la théorie des ensembles est bien établie dans la culture mathématique,
- la théorie des ensembles permet de construire les objets mathématiques évolués (entiers, arbres, ...) à partir d'un ensemble réduit de composants de base,
- la théorie des ensembles permet une écriture concise de notions évoluées.

Nous n'allons pas décrire ici en détail tout le langage de la méthode B, nous nous contenterons de détailler un petit nombre de concepts qui nous seront utiles par la suite. Une description plus détaillée est donnée dans l'annexe A de ce document. Nous donnons pour chaque notion à la fois la notation mathématique et une notation ASCII utilisée dans les outils.

2.1 Les expressions

2.1.1 Expressions de base

Parmi les expressions, nous trouvons les constantes entières ainsi que les variables. Nous trouvons également les expressions formées par application des opérateurs arithmétiques usuels ($+, -, \times, /,$ `mod`$, \ldots$).

Exemple $3{\times}(n{+}3)/k$

Remarque Dans les outils, une variable est formée d'au moins deux caractères, l'expression précédente s'écrira en fait : `3 * (nn + 3) / kk`

2.1.2 Combiner deux expressions

Il est souvent utile de composer deux éléments ensemble pour former une paire ordonnée. La paire formée des expressions e_1 et e_2 est notée e_1, e_2 ou bien $e_1 \mapsto e_2$. Cette seconde notation sera utilisée le plus souvent lorsque la paire représente un élément d'une fonction.

Exemple $x \mapsto x{+}2$ représente la paire $(x, x{+}2)$ et l'ensemble des paires $x \mapsto x{+}2$ pour tous les x entiers représente la fonction qui à x associe $x{+}2$.

2.1.3 Les ensembles

Le langage de B permet également de manipuler des ensembles.

Ensembles de base

$\emptyset$	`{}`	L'ensemble vide
$\mathbb{N}$	`NAT`	L'ensemble des entiers naturels
$\mathbb{Z}$	`INTEGER`	L'ensemble des entiers relatifs
`INT`	`INT`	L'ensemble des entiers relatifs compris entre `MININT` et `MAXINT`
$\{e\}$	$\{e\}$	L'ensemble réduit à un élément e
$\{e_1, \ldots, e_n\}$	$\{e_1, \ldots, e_n\}$	L'ensemble composé des éléments $e_1, \ldots, e_n$

Exemple $\{1, 2, 3\}$ est un ensemble. $\{\emptyset\}$ est un ensemble composé d'un élément.

Compréhension Soit P une formule, on peut définir un ensemble par *compréhension* comme l'ensemble de tous les objets qui vérifient la formule P. On note cet ensemble $\{z \mid P\}$.

2.1.4 Autres constructions

Quelques autres constructions primitives nous seront utiles.

$n_1..n_2$	$(n_1\,.\,.\,n_2)$	L'ensemble des entiers compris entre n_1 et n_2 (inclus) avec n_1 et n_2 des expressions représentants des entiers positifs.
$\mathrm{card}(E)$	`card`(E)	Le cardinal (nombre d'éléments) d'un ensemble E fini.

2.2 Les formules

2.2.1 Formules de base

On peut utiliser dans B tous les connecteurs logiques de base à savoir ceux du calcul des prédicats du premier ordre avec égalité. Le tableau suivant introduit les formules atomiques : e_1 et e_2 représentent des expressions quelconques (entiers, ensembles, paires ...), n_1 et n_2 sont des expressions qui représentent des entiers.

math&ASCII
$e_1 = e_2$
$n_1 > n_2$
$n_1 < n_2$

math	ASCII
$e_1 \neq e_2$	e_1`/=`e_2
$n_1 \geq n_2$	n_1`>=`n_2
$n_1 \leq n_2$	n_1`=<`n_2

2.2.2 Formules ensemblistes

Une formule atomique particulière importante est la formule qui dit qu'une expression appartient à un ensemble. Nous avons également des notations pour indiquer les relations d'inclusion ou de non inclusion. Dans le tableau suivant, e représente une expression tandis que E, E_1 et E_2 représentent des ensembles.

math	ASCII	
$e \in E$	e `:` E	L'expression e est un objet de l'ensemble E
$e \notin E$	e`/:`E	L'expression e n'est pas un objet de l'ensemble E.
$E_1 \subseteq E_2$	E_1`<:`E_2	E_1 est un sous-ensemble de E_2.
$E_1 \not\subseteq E_2$	E_1`/<:`E_2	E_1 n'est pas un sous-ensemble de E_2.

2.2.3 Formules composées

Le tableau suivant introduit les formules composées : F, F_1 et F_2 représentent des formules.

math	ASCII	
$\neg F$	`not`(F)	négation
$F_1 \wedge F_2$	F_1 `&` F_2	conjonction
$F_1 \vee F_2$	F_1 `or` F_2	disjonction
$F_1 \Rightarrow F_2$	F_1 `=>` F_2	implication
$F_1 \Leftrightarrow F_2$	F_1 `<=>` F_2	équivalence
$\exists$ var . F	`#` var . F	quantification existentielle
$\forall$ var . F	`!` var . F	quantification universelle

Quantification multiple Une particularité de B est que la quantification peut avoir lieu simultanément sur plusieurs variables, elles sont alors entourées de parenthèses et séparées par des virgules. Ainsi, on peut écrire la formule suivante :

$$\forall(x, y).(x = y \Rightarrow y = x)$$

qui est équivalente à $\forall x.\forall y.(x = y \Rightarrow y = x)$

2.2.4 Les formules bien typées

Dans le langage B, les ensembles servent à contraindre les données; en ce sens ils jouent le rôle des sortes ou des types d'autres langages. Cependant, il est en général indécidable de savoir si un objet e appartient à un ensemble E. En effet vérifier que $e \in \{z|P(z)\}$ est équivalent à vérifier que $P(e)$ est vrai, or la formule P est arbitraire. Cependant, le langage B propose un système rudimentaire de typage qui permet de distinguer les entiers, les paires, les ensembles ...
$\mathbb{N}$ est un type, les machines pourront également être paramétrés par des variables représentant des ensembles arbitraires. Ces variables sont également des types. Si T_1 et T_2 sont des types, il en est de même du produit cartésien de T_1 et de T_2, noté $T_1 \times T_2$ qui est le type des paires (t_1, t_2) d'éléments t_1 de type T_1 et t_2 de type T_2. Si T est un type, il en est de même du type des sous-ensembles de T noté $\mathbb{P}(T)$. Ainsi si $e_i \in T$ pour $i = 1..n$, alors $\{e_1, \ldots, e_n\} \in \mathbb{P}(T)$.
Le langage de B impose que les formules écrites soient bien "typées". Cela permet d'empêcher a priori de construire des expressions qui ne représentent effectivement pas des ensembles comme $\{x|x \notin x\}$. Pour cela on impose que chaque variable introduite soit contrainte à vivre dans un certain ensemble et que ces ensembles s'organisent de manière cohérente.
Les quantifications universelles, existentielles ainsi que les constructions d'ensemble par compréhension auront donc toujours la forme suivante :

$$\forall \text{ var } . \ P \Rightarrow Q \qquad \exists \text{ var } . \ P \wedge Q \qquad \{\text{var} \mid P \wedge Q\}$$

avec P une formule qui contraint chacune des variables z de var. Par exemple $z = e$, $z \in E$, $z \subseteq E$ ou $z \subset E$ avec z qui n'apparaît ni dans e ni dans E.
Il faut vérifier que les opérations ensemblistes apparaissant dans la formule sont compatibles avec la hiérarchie. Par exemple si on a $x = y$ alors les types de x et de y doivent être égaux. Dans la relation $x \in X$ on doit vérifier que X est de type $\mathbb{P}(U)$ pour un certain type U et que x est de type U. Si on a $X \subseteq Y$, on doit vérifier que X et Y sont de type $\mathbb{P}(U)$ pour un certain type U.

Exemple Nous illustrons la procédure de vérification sur un exemple. On suppose qu'un ensemble E arbitraire est donné. Soit la formule :

$$\forall(a, b).(a, b) \in \mathbb{P}(E) \times \mathbb{P}(E) \Rightarrow \{x|x \in E \wedge x \in a \wedge x \in b\} \subseteq E$$

Comme E est un ensemble paramétrique, les types sont de la forme $\mathbb{P}(E), E \times E, E \times \mathbb{P}(E), \ldots$
On a a de type $\mathbb{P}(E)$, b de type $\mathbb{P}(E)$, x de type E on vérifie que $x \in a$ et $x \in b$ sont des formules bien typées et $\{x|x \in E \wedge x \in a \wedge x \in b\}$ est de type $\mathbb{P}(E)$, de même E est de type $\mathbb{P}(E)$, donc l'inclusion est bien formée.

2.3 Substitution élémentaire

Une notion importante du langage de B est l'opération d'une substitution S sur une formule P qui est notée $[S]P$.
Une substitution élémentaire est de la forme var $:= E$. Dans ce cas $[\text{var} := E]P$ représente juste le résultat de la substitution des occurrences libres de var dans P par E.

Exemple $[n := 3]n > m$ est égal à $3 > m$.

2.3.1 Variables liées et substitution

Les quantificateurs ont la propriété de lier les variables qu'ils introduisent, c'est-à-dire que le nom de la variable n'intervient plus dans le sens de la formule, seuls les emplacements (on parle d'occurrences) où la variable apparaît dans la formule sont importants.
Ainsi $n = 4$ et $m = 4$ sont des formules différentes mais $\exists n.n = 4$ et $\exists m.m = 4$ représentent la même formule.
La variable z est *liée* dans les formules $\forall z.Q$ et $\exists z.Q$ ainsi que dans l'expression $\{z|Q\}$.
Un même nom de variable peut avoir simultanément des occurrences libres et des occurrences liées dans une formule :

$$n > 0 \Rightarrow \forall n.n \geq 0$$

Lors de la substitution $[\mathrm{var} := E]P$, il faut être attentif à ne remplacer que les occurrences libres de var dans P et de plus, il ne faut pas que les variables libres de E se retrouvent liées par des quantificateurs de P

Exemple Soit la formule P définie comme étant $n > 0 \Rightarrow \forall n.\exists m.m \geq n$ On souhaite calculer $[n := m{+}1]P$. Une opération naïve de remplacement donnerait le résultat **erroné** :

$$m{+}1 > 0 \Rightarrow \forall n.\exists m.m \geq m{+}1$$

En renommant les variables liées de P on obtient une formulation équivalente de P:

$$n > 0 \Rightarrow \forall n_0.\exists m_0.m_0 \geq n_0$$

Il est alors licite de calculer la substitution comme un simple remplacement et on obtient le résultat **correct** suivant :

$$m{+}1 > 0 \Rightarrow \forall n_0.\exists m_0.m_0 \geq n_0$$

Principe de la substitution Pour calculer le résultat d'une substitution $[\mathrm{var} := E]P$ on renomme les variables liées de P de manière à ce qu'elles soient différentes des variables libres de P et de E puis on procède à un remplacement de var par E dans la formule ainsi obtenue.

2.3.2 Substitutions multiples

Il est possible de substituer de manière simultanée plusieurs variables dans une formule. On notera cette opération

$$[x_1, \ldots, x_n := e_1, \ldots, e_n]P$$

Toutes les variables x_i doivent être distinctes. Une substitution multiple peut alternativement être notée :

$$x_1 := e_1 \| \ldots \| x_n := e_n$$

Remarque La substitution est simultanée, en particulier $[x, y := a, b]P$ n'est pas équivalent à $[x := a][y := b]P$. L'exemple suivant illustre ce phénomène :

$$[x, y := y, x]x = y \equiv y = x \qquad [x := y][y := x]x = y \equiv [x := y]x = x \equiv y = y$$

2.3.3 Substitutions généralisées

Le langage B permet de décrire des substitutions généralisées qui serviront à représenter des programmes. Celles-ci sont décrites dans la section 3.

2.4 Les relations et fonctions

En théorie des ensembles, les relations sont un cas particulier d'ensemble (des ensembles de paires) et les fonctions sont un cas particulier de relations. Cependant ces concepts essentiels bénéficient de notations spécifiques que nous décrivons ici.

2.4.1 Relations

Dans le tableau suivant, E, E_1 et E_2 représentent des ensembles tandis que R, R_1 et R_2 représentent des relations.

$E_1 \leftrightarrow E_2$	E_1`<->`E_2	L'ensemble des relations entre éléments de E_1 et de E_2
		$\mathbb{P}(E_1 \times E_2)$
$R_1; R_2$	R_1`;`R_2	Composition des relations R_1 et R_2
		$(x, y) \in (R_1; R_2) \Leftrightarrow \exists z.((x, z) \in R_1 \wedge (z, y) \in R_2)$
$\mathrm{id}(E)$	`id`(E)	la relation identité sur l'ensemble E
R^{-1}	R`~`	relation inverse de R
		$(x, y) \in R^{-1} \Leftrightarrow (y, x) \in R$
$E \lhd R$	E`<|`R	la restriction de la relation R au domaine E
		$(x, y) \in (E \lhd R) \Leftrightarrow (x \in E \wedge (x, y) \in R)$
$R \rhd E$	R`|>`E	la restriction de la relation R à l'image E
		$(x, y) \in (R \rhd E) \Leftrightarrow ((x, y) \in R \wedge y \in E)$

Ensembles définis à partir de relations (A.4.1)

$\mathrm{dom}(R)$	`dom`(R)	le domaine de la relation R
		$x \in \mathrm{dom}(R) \Leftrightarrow (\exists y.(x, y) \in R)$
$\mathrm{ran}(R)$	`ran`(R)	l'image de la relation R
		$y \in \mathrm{ran}(R) \Leftrightarrow (\exists x.(x, y) \in R)$

2.4.2 Fonctions

Les fonctions du système B sont, comme en mathématique, des cas particuliers de relations. Une fonction de E_1 dans E_2 est une relation binaire sur $E_1 \times E_2$, c'est-à-dire un ensemble de couples (x, y) aussi notés $x \mapsto y$. Une relation R est dite *fonctionnelle* si lorsque $x \mapsto y_1 \in R$ et $x \mapsto y_2 \in R$ alors $y_1 = y_2$. Une fonction f de E_1 dans E_2 est une relation fonctionnelle. L'ensemble des relations fonctionnelles de E_1 dans E_2 est noté $E_1 \nrightarrow E_2$. Le *domaine* d'une fonction f est l'ensemble dom(f) des x pour lesquels il existe y tel que $x \mapsto y \in f$. Si x est dans le domaine de f alors on note $f(x)$ l'unique y tel que $x \mapsto y \in f$.
Une fonction est totale si son domaine est égal à l'ensemble E_1 tout entier.
D'autres notions sur les fonctions sont définies en A.4.2.

2.4.3 Séquences

Les séquences sont des suites ordonnées d'objets d'un ensemble E. Une manière de les modéliser mathématiquement est de les voir comme des fonctions de $\mathbb{N}$ dans E dont le domaine est soit l'ensemble vide (dans le cas de la séquence vide) soit l'ensemble $1..n$ pour n un entier strictement positif. L'ensemble des séquences sur E est noté $\mathrm{seq}(E)$, les principales constructions sur les séquences sont rappelées en annexe (A.4.3).
Finalement, nous pouvons introduire des constantes correspondant à des chaînes de caractères écrites entre guillemets comme `"Hello"`.

2.4.4 Exemple

Les notions que nous avons introduites sont suffisantes pour modéliser un petit problème.
On s'intéresse à un ensemble de personnes et leurs liens de parenté et de mariage. On suppose donné un ensemble `personnes` représentant les personnes. On cherche à modéliser de manière ensembliste les notions de `femmes`, `hommes`, les relations de `a_pour_epoux`, `a_pour_épouse`, `a_pour_mère`, `a_pour_père`, `a_pour_frère`, `a_pour_enfant`, `est_marié_à` sachant que les principes suivants doivent être respectés :

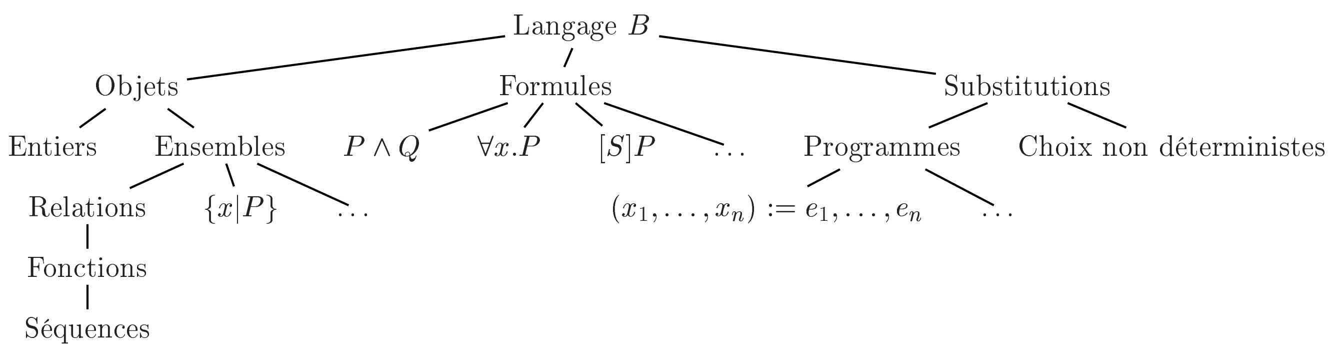
1. chaque personne est soit un homme soit une femme,
2. nulle personne n'est à la fois homme et femme,
3. seules les femmes ont des époux qui sont des hommes,
4. chaque femme a au plus un mari qui n'est le mari d'aucune autre femme,
5. une mère est une femme mariée,
6. le père est l'époux de la mère.

On identifie les données minimales qui doivent être primitives (par exemple l'ensemble `hommes`, les relations `a_pour_epoux` et `a_pour_mère`). Les contraintes imposent des restrictions sur la nature de ces objets. On peut ensuite définir les autres notions en fonction des notions primitives.

- $\texttt{hommes} \subseteq \texttt{personnes}$
- $\texttt{femmes} = \{p | p \in \texttt{personnes} \wedge p \notin \texttt{hommes}\}$
- $\texttt{a\_pour\_epoux} \in \texttt{femmes} \times \texttt{hommes}$
- `a_pour_epoux` est une fonction (partielle) de `femmes` dans `hommes` :
  $\texttt{a\_pour\_epoux} \in \texttt{femmes} \mathrel{+\!\!\!\rightarrow} \texttt{hommes}$ ce qui signifie :
  $(f \mapsto h_1 \in \texttt{a\_pour\_epoux} \wedge f \mapsto h_2 \in \texttt{a\_pour\_epoux}) \Rightarrow h_1 = h_2$
- `a_pour_epoux` est une fonction injective de `femmes` dans `hommes` :
  $\texttt{a\_pour\_epoux} \in \texttt{femmes} \mathrel{\rightarrowtail\!\!\!\!\!+} \texttt{hommes}$ ce qui signifie :
  $(f_1 \mapsto h \in \texttt{a\_pour\_epoux} \wedge f_2 \mapsto h \in \texttt{a\_pour\_epoux}) \Rightarrow f_1 = f_2$
- `a_pour_epouse` est la fonction inverse de `a_pour_epoux` :
  $\texttt{a\_pour\_epouse} \in \texttt{hommes} \mathrel{\rightarrowtail\!\!\!\!\!+} \texttt{femmes}$ et $\texttt{a\_pour\_epouse} = \texttt{a\_pour\_epoux}^{-1}$ ce qui signifie :
  $m \mapsto f \in \texttt{a\_pour\_epouse} \Leftrightarrow f \mapsto m \in \texttt{a\_pour\_epoux}$
- `a_pour_mère` est une fonction (partielle) de `personnes` dans `femmes` :
  $\texttt{a\_pour\_mère} \in \texttt{personnes} \mathrel{+\!\!\!\rightarrow} \texttt{femmes}$
- `a_pour_père` est une fonction (partielle) de `personnes` dans `hommes` qui est la composition de la relation `a_pour_mère` et de la relation : `a_pour_epoux`
  $\texttt{a\_pour\_père} = \texttt{a\_pour\_mère} ; \texttt{a\_pour\_epoux}$ ce qui signifie :
  $x \mapsto p \in \texttt{a\_pour\_père} \Leftrightarrow (\exists m . x \mapsto m \in \texttt{a\_pour\_mère} \wedge m \mapsto p \in \texttt{a\_pour\_epoux})$
- `a_pour_enfant` est une relation de $\texttt{personnes} \leftrightarrow \texttt{personnes}$, c'est la réunion des relations inverses de `a_pour_mère` et `a_pour_père` :
  $\texttt{a\_pour\_enfant} = \texttt{a\_pour\_mère}^{-1} \cup \texttt{a\_pour\_père}^{-1}$ ce qui signifie :
  $x \mapsto y \in \texttt{a\_pour\_enfant} \Leftrightarrow (y \mapsto x \in \texttt{a\_pour\_mere} \vee y \mapsto x \in \texttt{a\_pour\_pere})$
- `a_pour_frère` est une relation de $\texttt{personnes} \leftrightarrow \texttt{hommes}$. Pour trouver les frères, il faut chercher la mère puis composer avec les enfants de cette mère qui sont des hommes et qui ne sont pas la personne initiale. Cela donne :
  $\texttt{a\_pour\_frère} = ((\texttt{a\_pour\_mère} ; \texttt{a\_pour\_mère}^{-1}) \rhd \texttt{hommes}) - \mathrm{id}(\texttt{hommes})$

Le système B privilégie l'utilisation de symboles algébriques pour représenter les contraintes sur les données sans introduire de quantificateur. Cela permet d'automatiser plus simplement les preuves.

Ce qu'il faut retenir Le langage associé à la méthode B permet de manipuler différentes classes d'objet :



2.5 Méthodes de preuve

2.5.1 Raisonner sur les objets

Nous n'avons donné aucun système d'axiomes pour raisonner sur les objets introduits. Un tel système peut se trouver dans le livre "The B-Book" de J.-R. Abrial. Cependant une présentation effectuée de manière rigoureuse est très fastidieuse. Pour ce cours, il est suffisant de raisonner de manière semi-formelle, comme on peut le faire en mathématiques.

2.5.2 Raisonner sur les programmes

Dès lors qu'on connaît ce qui est attendu d'un programme (*sa spécification*) et qu'on connaît la manière dont le programme va s'exécuter (*sa sémantique*), il est possible de faire une *preuve de correction* du programme.

2.5.3 Logique de Hoare

Hoare dès les années 70, puis Floyd et Dijkstra ont développé l'idée qu'il fallait prouver les programmes. Pour pouvoir raisonner sur un programme, il faut savoir les représenter et se donner des règles et méthodes de preuve sur ces objets.

Modélisation du programme L'exécution d'un programme transforme l'état de la mémoire de la machine. La partie qui nous intéresse correspond aux valeurs associées aux variables du programme. On représente souvent l'environnement d'exécution du programme comme une fonction qui à chaque *variable* du programme associe une *valeur* (entier, chaîne de caractères, ensemble fini,...)
Il n'est pas raisonnable de raisonner globalement sur le programme comme une transformation de l'état de la mémoire. Hoare a donc proposé ce qui est connu comme la *sémantique axiomatique* ou la *logique de Hoare*. L'idée principale de la logique de Hoare, est de ne pas raisonner sur la mémoire globalement mais seulement sur les valeurs des variables apparaissant dans le programme.
Si en un point du programme, je démontre qu'une propriété $P(x)$ est vérifiée, cela signifie que la propriété $P(v)$ est vraie, avec v la valeur associée à la variable x dans la mémoire à ce point d'exécution du programme.

Exemple Que fait le programme suivant ? Comment le démontrer ?

```
n:=5;
i:=n; res:=1;
while i > 0 do res:=i*res; i:=i-1 end
print(res)
```

La logique de Hoare établit un système de déduction qui permet de démontrer des *triplets de Hoare*

$$\{\phi\}P\{\psi\}$$

dans lesquels P est un programme et ϕ et ψ sont des formules portant sur les variables du programme. Lorsqu'un triplet $\{\phi\}P\{\psi\}$ est vrai cela signifie que si les valeurs des variables de P satisfont la formule $\{\phi\}$ alors après l'exécution de P, les nouvelles valeurs de ces variables satisferont la propriété ψ.

Exemple-suite On peut annoter le programme précédent par des assertions

n:=5;
$\{n \geq 0\}$ i:=n; res:=1;
$\{res{\cdot}i! = n!\}$
while i $>$ 0 do res:=i*res; i:=i-1 end
$\{i = 0 \wedge res = n!{\cdot}i!\}$
print(res) $\{res = n!\}$

Un programme vu comme un transformateur de prédicat La logique de Hoare permet de voir les programmes non plus comme des fonctions qui transforment les états de la mémoire mais comme des *transformateurs de prédicats*. Si on se donne un programme P et une propriété ψ portant sur les variables de P alors on peut définir en fonction de P *la plus faible condition* ϕ pour que $\{\phi\}P\{\psi\}$ soit vérifié. On note $\mathsf{wp}(P, \psi)$ la propriété ϕ ainsi définie.
Elle doit vérifier les deux conditions suivantes :

$$\{\mathsf{wp}(P, \psi)\}P\{\psi\} \qquad \forall\phi.\{\phi\}P\{\psi\} \Rightarrow (\phi \Rightarrow \mathsf{wp}(P, \psi))$$

Les règles de la logique de Hoare permettent de calculer effectivement $\mathsf{wp}(P, \psi)$ pour un ψ quelconque en fonction de P, lorsque P ne contient pas de boucle. Par exemple :

$$\begin{array}{l}
\mathsf{wp}(x := e, \psi) = \psi[x \leftarrow e] \\
\mathsf{wp}(\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{end}, \psi) = (C \Rightarrow \mathsf{wp}(P, \psi)) \wedge (\neg C \Rightarrow \mathsf{wp}(Q, \psi)) \\
\mathsf{wp}(P_1; P_2, \psi) = \mathsf{wp}(P_1, \mathsf{wp}(P_2, \psi))
\end{array}$$

Le cas des boucles L'introduction de boucles dans les programmes complique le raisonnement. Il est nécessaire de produire un *invariant* qui est une propriété qui sera vérifiée à l'entrée dans la boucle et préservée par le corps de la boucle. D'autre part, une boucle peut ne pas terminer. On distingue alors deux variantes de la logique de Hoare que l'on qualifie de *partielle* et *totale*.

- La logique de Hoare *partielle* ne garantit pas la terminaison; le triplet $\{\phi\}P\{\psi\}$ s'interprète comme : si ϕ est vérifié dans l'état initial et si P termine alors ψ est vérifié dans l'état final. En particulier la formule $\{\phi\}P\{\mathtt{false}\}$ est équivalente à P ne termine jamais à partir d'un état satisfaisant ϕ.
- La logique de Hoare *totale* garantit la terminaison; le triplet $\{\phi\}P\{\psi\}$ s'interprète comme : si ϕ est vérifié dans l'état initial alors P termine et ψ est vérifié dans l'état final. En particulier la formule $\{\phi\}P\{\mathtt{true}\}$ est équivalente à P termine toujours à partir d'un état satisfaisant ϕ..

Des exemples de logique de Hoare seront développés en exercice (A).

3 Machines abstraites

3.1 Introduction

La notion centrale de la méthode B est la notion de machine abstraite. Afin de modulariser les développements, les programmes sont décomposés en unités qui ont un rôle analogue à celui des objets. Chaque machine déclare ses propres variables et opérations. Les variables ne peuvent être modifiées que par les opérations de la machine.

Dans la méthode B, les machines comportent également des assertions logiques qui permettent de préciser des propriétés qui seront satisfaites par les variables et les opérations. Dans un premier temps les opérations sont spécifiées par leurs propriétés et non par la manière dont elles seront implémentées.

Les machines abstraites sont décrites à partir de substitutions que nous décrivons dans le paragraphe 3.2. Nous décrirons ensuite la structure d'une machine abstraite et les propriétés qui doivent être vérifiées pour assurer que la machine abstraite est bien construite. Finalement, nous indiquons les principales opérations de composition des machines abstraites. Dans les sections suivantes, nous décrirons les opérations de raffinement et d'implantation.

3.2 Les substitutions

Le langage de la méthode B permet de manipuler des programmes, c'est-à-dire des transformations (éventuellement non-déterministes) de la mémoire.

3.2.1 Substitutions de base

Les programmes sont mathématiquement représentés par des *substitutions généralisées* qui sont définies par leur action sur les prédicats. Une substitution généralisée S est complètement déterminée par la définition de la formule $[S]P$ pour une formule P arbitraire.

Si on interprète S comme un programme alors $[S]P$ représente la plus faible précondition pour que après n'importe quelle exécution de S, la propriété P soit vérifiée.

Les substitutions généralisées sont formées à partir des substitutions élémentaires que nous avons introduites dans le paragraphe 2.3. Les substitutions élémentaires représentent l'affectation dans les programmes.

Dans le tableau suivant qui définit des substitutions généralisées de base, P et Q représentent des formules et S, S_1 et S_2 des substitutions généralisées.

math	ASCII	$[S]P$	
$S_1;S_2$	`S1;S2`	$[S_1;S_2]P \Leftrightarrow [S_1][S_2]P$	Séquencement
skip	`skip`	$[\text{skip}]P \Leftrightarrow P$	instruction nulle
$Q \Longrightarrow S$	`Q==>S`	$[Q \Longrightarrow S]P \Leftrightarrow (Q \Rightarrow [S]P)$	Substitution gardée par Q
$Q\|S$	`Q|S`	$[Q\|S]P \Leftrightarrow Q \wedge [S]P$	Substitution préconditionnée, assertion de Q
$S_1[]S_2$	`S1[]S2`	$[S_1[]S_2]P \Leftrightarrow [S_1]P \wedge [S_2]P$	Choix non-déterministe
$@z.S$	`@z.S`	$[@z.S]P \Leftrightarrow \forall\ z.[S]P$	Choix non borné

3.2.2 $Q|S$ versus $Q \Longrightarrow S$

Il est parfois délicat de comprendre la différence entre les substitutions $Q|S$ et $Q \Longrightarrow S$.

La substitution $Q|S$ correspond à affirmer que Q est vrai avant l'exécution de la substitution S. D'un point de vue programme, la propriété Q n'est pas retestée, on fait confiance au programmeur pour n'exécuter la substitution S que lorsque l'état vérifie Q.

La substitution $Q \Longrightarrow S$ par contre peut être utilisée dans n'importe quel état. D'un point de vue programmation, la propriété Q est d'abord vérifiée avant d'exécuter S.

Supposons que Q soit faux alors on ne peut rien dire sur l'état de la mémoire après l'exécution de $Q|S$ en effet il ne sera jamais possible de valider la plus faible précondition.
Dans le cas de $Q \Longrightarrow S$ si Q est faux alors on peut montrer (par l'absurde) que $[Q \Longrightarrow S]P$ est vrai et cela pour tout P donc on peut établir que toute propriété P est vérifiée après l'exécution de ce programme, en particulier la propriété absurde. Cela signifie que le programme correspondant à S n'est jamais exécuté.

3.2.3 Construction de programmes

Des notations sont introduites pour représenter des substitutions correspondant à des programmes usuels.

PRE Q THEN S END	$Q\|S$
SELECT Q THEN S END	$Q \Longrightarrow S$
IF Q THEN S_1 ELSE S_2 END	$(Q \Longrightarrow S_1)[](\neg Q \Longrightarrow S_2)$
IF Q THEN S END	$(Q \Longrightarrow S)[](\neg Q \Longrightarrow \text{skip})$
CHOICE S_1 OR $S_2 \ldots$ OR S_n END	$S_1[]S_2[]\ldots[]S_n$

Exercice Calculer [IF Q THEN S_1 ELSE S_2 END]P et [IF Q THEN S END]P.

Choix non borné Une opération importante sur les substitutions est le choix non borné. Il permet de choisir arbitrairement un objet et sera utile pour spécifier des opérations de création d'objet ou d'initialisation.

VAR x IN S END	$@x.S$
ANY x WHERE Q THEN S END	$@x.(Q \Longrightarrow S)$
$x :\in E$	$@y.(y \in E \Longrightarrow x := y)$
	n'importe quel objet dans l'ensemble E
$x : P$	$@y.([x := y]P \Longrightarrow x := y)$
	n'importe quel objet qui vérifie P

Restrictions La substitution VAR ne peut être utilisée dans une machine abstraite de spécification. La variable introduite par ANY ne peut apparaître en partie gauche d'une substitution.

Mise à jour de fonctions Une fonction est un ensemble de couples de la forme $x \mapsto y$, où on a au plus une valeur associée à chaque entrée x. Une opération utile est celle de mise à jour. Soit f une variable représentant une fonction, on écrira

$$f(e_1) := e_2$$

pour représenter la substitution $f := \{x \mapsto y | (x \neq e_1 \wedge x \mapsto y \in f) \vee (x = e_1 \wedge y = e_2)\}$

Choix booléen BOOL est un ensemble prédéfini qui comprend les deux valeurs true et false.
Si P est une formule, la substitution

$$x := \text{bool}(P)$$

représente la substitution

IF P THEN $x :=$ true ELSE $x :=$ false END

Cette substitution ne correspond pas forcément à un programme exécutable car rien n'assure que la condition P puisse être calculée par un ordinateur, elle peut être de la forme $\forall n.f(n) = 0$ avec f une fonction quelconque.

3.2.4 Opérations

Dans la construction des machines abstraites qui forment le cœur du langage B, on sera amené à définir de nouvelles substitutions généralisées et à leur donner des noms, ce sont les *opérations*. Ces opérations peuvent être paramétriques c'est-à-dire dépendre de variables. En général on distingue dans un programme les variables correspondant aux entrées que l'on instancie au moment de l'appel de celles correspondant aux sorties dans lesquelles on récupère les résultats. Les opérations correspondent aux procédures et fonctions des langages de programmation.
Si S est une substitution généralisée et var, var_1 et var_2 des listes de variables alors une opération opn du langage B est introduite par une des déclarations suivantes :

1. routines : $\mathrm{opn} \hat{=} S$
2. procédures : $\mathrm{opn}(\mathrm{var}) \hat{=} S$
3. fonction sans paramètre : $\mathrm{var} \longleftarrow \mathrm{opn} \hat{=} S$
4. fonction avec paramètres : $\mathrm{var}_1 \longleftarrow \mathrm{opn}(\mathrm{var}_2) \hat{=} S$

On pourra ensuite utiliser opn à la place de S dans de nouvelles substitutions. L'appel se fera de manière correspondante à la définition :

1. opn est équivalent à S
2. $\mathrm{opn}(e_1, \ldots, e_n)$ est équivalent à S dans laquelle les variables de var ont été remplacées par $(e_1, \ldots, e_n)$
3. $\mathrm{var}' \longleftarrow \mathrm{opn}$ est équivalent à S dans laquelle les variables de var ont été remplacées par les variables de var'.
4. $\mathrm{var} \longleftarrow \mathrm{opn}(e_1, \ldots, e_n)$ est équivalent à S dans laquelle les variables de var_1 ont été remplacées par les variables de var et les variables var_2 ont été remplacées par $(e_1, \ldots, e_n)$

Exemple On peut par exemple définir une opération `moyenne` qui calcule dans la variable m la moyenne des variables a et b. Cette opération correspond à la substitution simple :

$$m := (a+b)/2$$

Il est commode de la voir comme une opération prenant comme entrée a et b et ayant comme sortie m. On écrira donc :

$$m \longleftarrow \texttt{moyenne}(a, b) = m := (a+b)/2$$

On pourra ensuite définir la moyenne de 4 nombres en utilisant cette opération :

$$\begin{array}{rl} m \longleftarrow \texttt{moyenne4}(a, b, c, d) \hat{=} & \mathsf{VAR}\ \mathrm{tmp}\ \mathsf{IN} \\ & \mathrm{tmp} \longleftarrow \texttt{moyenne}(a, b); \\ & m \longleftarrow \texttt{moyenne}(c, d); \\ & m \longleftarrow \texttt{moyenne}(\mathrm{tmp}, m); \\ & \mathsf{END} \end{array}$$

3.3 Structure d'une machine abstraite

Une machine abstraite est un programme composé d'une suite de déclarations de différentes natures (constantes, variables, opérations ...).

3.3.1 Machine de base

La machine de base est formée d'un nom, un ensemble de variables, un invariant portant sur les variables de la machine, une initialisation de ces variables ainsi que des opérations.
Une machine *élémentaire* est déclarée de la manière suivante :

MACHINE *nom*	nom de la machine
VARIABLES $x_1, \ldots, x_n$	noms des variables
INVARIANT I	propriété
INITIALISATION S_0	*substitution*
OPERATIONS	*substitutions*
$\mathrm{var}_1 \longleftarrow$ *nom_ de_ l'opération*$(\mathrm{var}_2) \;\hat{=}\; S;$	
$\vdots$	
END	

Exemple : notion de point

```
MACHINE point
VARIABLES x, y
INVARIANT x ∈ 0..1024 ∧ y ∈ 0..768
INITIALISATION x, y := 0, 0
OPERATIONS
v ⟵ abs ≙ v := x;
v ⟵ ord ≙ v := y;
move(dx, dy)≙
    PRE x+dx ≤ 1024 ∧ y+dy ≤ 768
    THEN (x, y) := (x+dx, y+dy)
    END
END
```

Conditions Une machine abstraite doit suivre certaines contraintes.
- Les formules et substitutions apparaissant dans la machine doivent être correctement construites. En particulier chaque variable introduite doit être contrainte de manière à assurer son appartenance à un ensemble bien construit.
- Les variables apparaissant dans les paramètres d'entrée ou de sortie d'une opération doivent être toutes distinctes et distinctes des variables de la machine. Elles ne sont visibles que dans la substitution S qui définit l'opération.
- L'invariant I de la machine doit être satisfait dans l'état initial de la machine décrit par la substitution d'initialisation S_0. Ceci signifie qu'il faut montrer

$$[S_0]I$$

L'invariant doit être préservé par chaque opération. Si S est la substitution associée à l'opération alors il faut montrer une propriété de la forme $I \Rightarrow [S]I$ cependant la substitution est souvent de la forme $P|S$ avec P une précondition, cette précondition indique sous quelles hypothèses (portant souvent sur les paramètres) l'opération peut être exécutée. On demande donc simplement que l'invariant soit préservé dans le cas où la précondition est vérifiée. L'opération ne décrit pas juste une substitution mais une classe de substitutions suivant l'instanciation des paramètres. On doit vérifier la préservation de l'invariant pour toutes les valeurs possibles des paramètres. Finalement, pour une opération de la forme :

$$\mathrm{var}_1 \longleftarrow \textit{nom\_ de\_ l'opération}(\mathrm{var}_2) \hat{=} P|S$$

. on obtient l'obligation de preuve :

$$I \Rightarrow \forall(\mathrm{var}_1, \mathrm{var}_2)P \Rightarrow [S]I$$

Dans le cas de notre exemple il faudrait vérifier :
- le cas initial : $0 \in 0..1024 \land 0 \in 0..768$
- opération `abs` : $x \in 0..1024 \land y \in 0..768 \Rightarrow \forall vv.[vv := x](x \in 0..1024 \land y \in 0..768)$
  qui est équivalent à la propriété triviale : $x \in 0..1024 \land y \in 0..768 \Rightarrow (x \in 0..1024 \land y \in 0..768)$
  en effet cette opération ne modifie pas l'état de la machine.
- opération `ord` analogue à `abs`
- opération `move` :

$$\begin{aligned} x \in 0..1024 \land y \in 0..768 \Rightarrow \ & \forall(dx, dy).x{+}dx \leq 1024 \land y{+}dy \leq 768 \\ & \Rightarrow [(x, y) := (x{+}dx, y{+}dy)](x \in 0..1024 \land y \in 0..768) \end{aligned}$$

La précondition sur la substitution est essentielle pour assurer que l'invariant est préservé.

Opérations spécifiées Il n'est pas nécessaire d'implanter complètement une opération, les machines servent alors à décrire des spécifications, c'est-à-dire ce que fait l'opération en évitant de décrire comment l'opération le fait. On peut par exemple vouloir implanter une opération qui étant donné un ensemble fini de points nous renvoie l'abscisse la plus petite.

$px \longleftarrow$ `pp_abs`$(E) \widehat{=}$
PRE $E \in \mathbb{P}(0..1024 \times 0..768) \land E \neq \emptyset$
THEN ANY x WHERE $\exists y.(x, y) \in E \land \forall(x', y').(x', y') \in E \Rightarrow x \leq x'$
    THEN $px := x$ END
END

3.3.2 Ensembles et constantes

Ensembles Les modélisations manipulent souvent des ensembles qui restent non spécifiés jusqu'à la phase d'implantation. Par exemple pour modéliser une base de données, on introduira l'univers global des objets dans la base qui est un ensemble fini, on pourra piocher dans cet ensemble pour ajouter de nouveaux objets. Une variable de la machine abstraite décrit le sous-ensemble de cet univers qui correspond à l'état courant de la base.
Il est également utile de pouvoir introduire de nouveaux ensemble finis par extension en énumérant les objets de cet ensemble.
Les ensembles ainsi introduits sont toujours finis et non vide et on convient de les identifier par des noms en majuscule. Les objets introduits dans l'énumération d'un ensemble sont tous distincts.
Cette déclaration se fait de la manière suivante :

SETS
ENSEMBLE_ABSTRAIT;
ENSEMBLE_EN_EXTENSION $= \{a_1, \ldots, a_n\}$

Constantes Il est utile également d'introduire des constantes correspondant à des valeurs particulières qui restent "symboliques" jusqu'à l'implantation. C'est le cas par exemple de la taille des ensembles introduits. La déclaration de constantes se fait de la manière suivante :

CONSTANTS
$c_1, \ldots, c_n$

Propriétés Si les valeurs des constantes et ensembles ainsi introduits ne seront précisées qu'au moment de l'implantation (ils correspondront à des constantes du programme à exécuter), il peut être utile d'exprimer des contraintes sur ces objets. C'est le rôle de la déclaration **PROPERTIES** des machines abstraites. Cette déclaration introduit une formule portant sur les constantes et ensembles déclarés qui est supposée être vérifiée.

Exemple

```
MACHINE Mariages
SETS
  PERSONNES;
  STATUT = {marié,célibataire};
  SEXE = {homme,femme}
CONSTANTS max_pers
PROPERTIES
  max_pers ∈ ℕ−{0} ∧ card(PERSONNES) = max_pers
VARIABLES
  personnes, sexe, statut
INVARIANT
  personnes ⊆ PERSONNES
  ∧ sexe ∈ personnes ⟶ SEXE
  ∧ statut ∈ personnes ⟶ STATUT
```

La machine sera complétée par une substitution d'initialisation. On peut choisir d'initialiser les ensembles en prenant l'ensemble vide pour les personnes et les fonctions.

```
INITIALISATION
  personnes, sexe, statut := ∅, ∅, ∅
```

Si on voulait spécifier qu'initialement `personnes` est un ensemble quelconque de femmes célibataires, on écrirait :

```
INITIALISATION
  personnes :∈ ℙ(PERSONNES)
  ||sexe := personnes×{femme}
  ||statut := personnes×{célibataire}
```

On peut également ajouter une opération de création :

```
OPERATIONS
  ajout(s, m) ≙
    PRE s ∈ SEXE ∧ m ∈ STATUT ∧ PERSONNES−personnes ≠ ∅
    THEN
      ANY x WHERE x ∈ PERSONNES−personnes
      THEN personnes := personnes∪{x}||sexe(x) := s||statut(x) := m
      END
    END
```

Exercice Spécifier une opération de mariage.

3.3.3 Machine paramétrée

Il est possible de paramétrer une machine abstraite. Les paramètres jouent un rôle analogue à celui des constantes et ensembles. Ces paramètres représentent d'ailleurs des ensembles finis non vides de valeurs numériques ou des valeurs numériques.

La machine paramétrée pourra être utilisée avec différentes instances pour les paramètres.
Les conditions sur les paramètres sont exprimées dans un champ appelé **CONSTRAINTS** par une formule ne portant que sur les paramètres de la machine.

Exemple Nous décrivons une machine abstraite pour spécifier des tableaux.

MACHINE `Array(INDEX,VAL)`
VARIABLES $table$
INVARIANT $table \in \mathtt{INDEX} \longrightarrow \mathtt{VAL}$
INITIALISATION $table :\in \mathtt{INDEX} \longrightarrow \mathtt{VAL}$
OPERATIONS
$\mathtt{enter}(i, v) \hat{=}$
  PRE $i \in \mathtt{INDEX} \wedge v \in \mathtt{VAL}$
  THEN $table(i) := v$
  END;
$v \longleftarrow \mathtt{access}(i) \hat{=}$
  PRE $i \in \mathtt{INDEX}$
  THEN $v := table(i)$
  END;
$i \longleftarrow \mathtt{search}(v) \hat{=}$
  PRE $v \in \mathtt{ran}(table)$
  THEN $i :\in table^{-1}[\{v\}]$
  END;
$b \longleftarrow \mathtt{test}(v) \hat{=}$
  PRE $v \in \mathtt{VAL}$
  THEN $b := \mathtt{bool}(v \in \mathtt{ran}(table))$
  END;

3.3.4 Autres champs

Les machines abstraites peuvent contenir d'autres champs tels que **DEFINITIONS** qui permet d'introduire des abréviations améliorant la lisibilité ou **ASSERTIONS** qui permet d'énoncer des propriétés qui sont toujours vérifiées par la machine car conséquences de l'invariant.

3.4 Combiner des machines abstraites

Plusieurs opérations permettent de combiner des machines entre elles. La déclaration d'une machine M peut contenir une ou plusieurs des déclarations d'inclusion suivante **INCLUDES**, **IMPORTS**, **USES**, **SEES**, **PROMOTES**, **EXTENDS**.

3.4.1 Substitution multiple généralisée

On a déjà utilisé la notation $x_1 := e_1 \| x_2 := e_2$ comme abbréviation pour la substitution multiple $x_1, x_2 := e_1, e_2$. Cette notation peut s'étendre pour combiner deux substitutions généralisées $S_1 \| S_2$ portant sur des variables distinctes.

3.4.2 INCLUDES

INCLUDES prend en argument une liste de machines $M_1, \ldots, M_k$. Si la machine M inclut une machine $M'(X, x)$ paramétrée par un ensemble X et un scalaire x, alors les paramètres seront instanciés dans la clause d'inclusion, on écrira donc **INCLUDES** $M_i(P, p)$ où P et p sont respectivement un ensemble et un scalaire de la machine M.
Les ensembles, constantes et variables incluses deviennent des ensembles, constantes et variables de la machine M. Les variables de la machine incluse M_i ne peuvent être modifiées que par les opérations de la machine M_i.
Si certaines opérations $op_1, \ldots, op_n$ de la machine incluse doivent devenir des opérations de M alors on utilisera la déclaration **PROMOTES** $op_1, \ldots, op_n$. On peut également utiliser **EXTENDS** au lieu de **INCLUDES**, dans ce cas toutes les opérations des machines incluses deviennent des opérations de la machine M. On implante ainsi l'analogue de l'héritage entre des classes.
Le fait d'utiliser une clause **INCLUDES** demande de vérifier des obligations de preuve. Si la machine M inclut une machine $M'(P, p)$, la condition principale à vérifier est que l'instanciation a de la machine incluse satisfait bien les contraintes indiquées dans M'. Cette instanciation peut se faire à l'aide des paramètres, constantes et ensembles de M.
L'opération d'initialisation de la machine M est composée de l'initialisation de M' exécutée conjointement à la déclaration d'initialisation indiquée dans M. Pour montrer que l'initialisation et les opérations de M préservent l'invariant de M, on peut utiliser le fait que l'initialisation et les opérations de M' préservent l'invariant de M'.
Lorsque deux machines M_1 et M_2 sont incluses dans une même machine M alors les variables de M_1 et M_2 doivent être différentes (de manière à éviter de casser les invariants de chaque machine). Au besoin, on utilisera une opération de renommage sur les variables et opérations des machines incluses.
La commande d'inclusion est une facilité d'écriture, il serait possible d'expanser les déclarations des machines incluses dans la machine principale.

Exemple On aurait pu découper notre machine `Mariages` en deux en introduisant tout d'abord une machine `Personnes` ne contenant que les personnes et leur sexe :

```
MACHINE Personnes
SETS
  PERSONNES;
  SEXE = {homme,femme}
CONSTANTS max_pers
PROPERTIES
  max_pers ∈ ℕ−{0} ∧ card(PERSONNES) = max_pers
VARIABLES personnes, sexe
INVARIANT
  personnes ⊆ PERSONNES  ∧  sexe ∈ personnes ⟶ SEXE
INITIALISATION  personnes, sexe := ∅, ∅
```

L'opération de création d'une personne prend en argument la personne créée :

```
OPERATIONS
  ajout_personne(x, s) ≙
     PRE s ∈ SEXE ∧ x ∈ PERSONNES−personnes
     THEN personnes := personnes∪{x}||sexe(x) := s
     END
```

On peut alors construire la machine des mariages au-dessus de la machine des personnes :

```
MACHINE Mariages
SETS  STATUT = {marié,célibataire};
INCLUDES Personnes
VARIABLES  statut
INVARIANT  statut ∈ personnes ⟶ STATUT
INITIALISATION  statut := ∅
```

Pour implanter l'opération de création, on va utiliser l'opération `ajout_personne` de la machine `Personnes`.

```
OPERATIONS
ajout(s, m) ≙
     PRE s ∈ SEXE ∧ m ∈ STATUT ∧ PERSONNES−personnes ≠ ∅
     THEN
       ANY x WHERE x ∈ PERSONNES−personnes
       THEN ajout_personne(x, s)||statut(x) := m END
     END
```

3.4.3 USES

USES prend en argument une liste de noms de machines $M_1, \ldots, M_k$. Les variables, paramètres, constantes et ensembles des machines utilisées sont visibles dans la machine M, peuvent être utilisés dans les invariants, substitutions de M. Cependant la machine M ne peut voir les opérations des machines de la clause `USES` et ne peut en modifier les variables. Les éventuels paramètres des machines M_i ne sont pas instanciés dans M et doivent être distincts de ceux de M.
L'idée est que dans le futur, une machine N sera construite qui incluera la machine M et toutes les machines des clauses `USES` de M. La vérification de la cohérence globale du système sera faite dans la machine N.
La déclaration **USES** sert à partager des informations statiques entre deux machines incluses dans une troisième.

Exemple En reprenant notre exemple de mariage, une alternative serait de construire une machine `Statut` qui utilise la machine `Personnes` et implante le statut, une opération de mise-à-jour du statut est introduite. Il n'est pas possible dans la machine `Statut` de maintenir l'invariant que `statut` est une fonction totale sur `personnes`, en effet la mise à jour du statut se fait indépendemment des modifications apportées à la variable `personnes`. On se contente donc de déclarer `statut` comme une fonction partielle de l'ensemble `PERSONNES` dans `STATUT`.
La machine `Mariage` est alors construite dans un troisième temps par inclusion des machines `Personnes` et `Statut`, cette machine peut déclarer l'opération d'ajout et contraindre la fonction `statut` à être totale sur `personnes` :

```
MACHINE Statut
SETS
 STATUT = {marié,célibataire}
USES Personnes
VARIABLES statut
INVARIANT statut ∈ PERSONNES ⇸ STATUT
INITIALISATION statut := ∅
OPERATIONS
 set_statut(x,m) ≙
    PRE m ∈ STATUT ∧ x ∈ PERSONNES THEN statut(x) := m END
END
```

La machine `Mariages` est alors construite de la manière suivante :

```
MACHINE Mariages
INCLUDES
 Personnes, Statut
INVARIANT
 dom(statut) = personnes
OPERATIONS
ajout(s,m) ≙
    PRE s ∈ SEXE ∧ m ∈ STATUT ∧ PERSONNES−personnes ≠ ∅
    THEN ANY x WHERE x ∈ PERSONNES−personnes
          THEN ajout_personne(x,s)||set_statut(x,m) END
    END
END
```

3.4.4 SEES

SEES prend en argument une liste de noms de machines $M_1, \dots, M_k$. Les constantes, ensembles, variables, et les opérations ne modifiant pas les variables des machines apparaissant dans la clause `SEES` sont visibles dans la machine M. Cependant les variables des machines vues sont complètement cachées et ne peuvent être utilisée dans l'invariant de M, elles sont par contre accessibles en lecture dans les opérations de M. Lorsqu'une machine M a une clause **SEES** M', alors c'est que l'implantation de M comportera en particulier une instance de M'.

4 Raffinement

4.1 Introduction

Le raffinement est une technique utilisée au cours du processus de développement logiciel pour transformer un modèle abstrait d'un système logiciel (la spécification) en un modèle plus *concret*, c'est-à-dire qui contient plus de détails sur la spécification ou bien qui est plus près d'une implémentation.
Lors d'un raffinement, une machine M_1 est remplacée par une autre machine M_2 qui va fournir des opérations de même nom et de même *signature* mais qui seront implantées à l'aide de variables d'états différentes ou qui satisferont une spécification plus forte. On dira que deux opérations ont la même signature s'ils ont le même nombre de paramètres d'entrée et de sortie et que ces paramètres sont contraints à habiter les mêmes univers.
Si une opération op_2 est un raffinement d'une opération op_1 alors toute utilisation de op_1 doit pouvoir être remplacée par une utilisation de op_2, sans casser le fonctionnement du programme.

4.2 Substitution et programme

Nous avons dit que les substitutions représentaient des programmes. Cependant les substitutions sont définies par leur action sur des prédicats $[S]P$. Nous allons préciser le lien entre les substitutions et les relations de transformation sur la mémoire (représentée par les valeurs des variables manipulées dans la substitution).

4.2.1 Terminaison

L'interprétation de $[S]P$ est la plus faible condition telle que si elle est vérifiée alors toute exécution de S termine dans un état qui vérifie P.
Une substitution S correspond donc à un programme qui termine lorsque l'on peut démontrer $[S](x = x)$. On pourrait en fait remplacer la formule $x = x$ par n'importe qu'elle formule toujours vraie.

Exemples
- $P|S$ termine lorsque $[P|S](x = x)$ est vérifié, ie lorsque $P \wedge [S](x = x)$ c'est-à-dire lorsque P est vrai et que S termine.
- $P \Longrightarrow S$ termine lorsqu'on peut montrer que S termine sous la condition que P est vrai (donc si P est faux on peut montrer que $P \Longrightarrow S$ termine).
- $S_1[]S_2$ termine lorsque S_1 et S_2 terminent.

4.2.2 Substitutions comme des relations

Soit S une substitution portant sur une variable x, nous voulons définir la relation $\mathrm{rel}(S)(a, a')$ qui dit que la substitution S sur une entrée x où $x = a$ peut s'exécuter et terminera dans un état où $x = a'$.
Une première idée serait de prendre $x = a \Rightarrow [S](x = a')$.
Cependant, certaines des substitutions que nous définissons sont non déterministes ou correspondent à des programmes qui ne terminent pas. La formule $x = a \Rightarrow [S](x = a')$ signifie que pour toute exécution de S, cette exécution termine sur un état où $x = a'$. Nous sommes intéressés par une relation plus faible de correction partielle qui sera notée $\mathrm{rel}(S)(a, a')$ et qui signifie qu'il existe une exécution de S partant d'un état où $x = a$ qui soit ne termine pas soit termine dans un état où $x = a'$.
Pour montrer cela il suffit d'établir qu'aucune exécution de S ne termine à coup sûr dans un état où $x \neq a'$. La propriété $[S]x \neq a'$ représente la plus faible condition telle que si elle est vérifiée alors après toute exécution de S, on peut établir $x \neq a'$. Donc cela signifie qu'aucune exécution de S n'aboutit à $x = a'$. La condition qui nous intéresse qui dit qu'une exécution de S ne termine pas ou conduit à

$x = a'$ est donc exactement la négation de cette propriété. On définit donc :

$$\boxed{\mathrm{rel}(S)(a, a') \equiv x = a \Rightarrow \neg([S](x \neq a'))}$$

Lorsqu'on prend $a = x$, on obtient la formule équivalente $\neg([S](x \neq x'))$.

Exemples
- $\mathrm{rel}(P|S)(x, x')$ est équivalent à $P \Rightarrow \mathrm{rel}(S)(x, x')$
- $\mathrm{rel}(P \Longrightarrow S)(x, x')$ est équivalent à $P \wedge \mathrm{rel}(S)(x, x')$
- $\mathrm{rel}(S_1[]S_2)(x, x')$ est équivalent à $\mathrm{rel}(S_1)(x, x') \vee \mathrm{rel}(S_2)(x, x')$.

4.3 Raffiner une substitution

4.3.1 Définition du raffinement

On suppose que l'on se donne deux substitutions S_1 et S_2 qui transforment une variable x. Alors S_2 est un raffinement de S_1 si S_2 termine lorsque S_1 termine et si la relation entrées-sorties définie par S_2 est incluse dans la relation entrées-sorties de S_1, ce qui nous donne les deux conditions :

$$[S_1](x = x) \Rightarrow [S_2](x = x) \qquad \neg([S_2](x \neq x')) \Rightarrow \neg([S_1](x \neq x'))$$

Lorsque cette condition est vérifiée, on écrira $S_1 \sqsubseteq S_2$ et on dira que S_1 est raffiné par S_2.
On remarque que la seconde condition peut s'écrire de manière équivalente :

$$[S_1](x \neq x') \Rightarrow [S_2](x \neq x')$$

4.3.2 Propriétés du raffinement

La relation de raffinement satisfait des bonnes propriétés, elle est réflexive, transitive. Elle est monotone par rapport aux opérations de construction des substitutions.

Exercice Montrer les propriétés suivantes :
- $S \sqsubseteq S$
- si $S_1 \sqsubseteq S_2$ et $S_2 \sqsubseteq S_3$ alors $S_1 \sqsubseteq S_3$
- si $S_1 \sqsubseteq S_2$ alors
  - $P|S_1 \sqsubseteq P|S_2$
  - $P \Longrightarrow S_1 \sqsubseteq P \Longrightarrow S_2$
- $S_1 \sqsubseteq S_1[]S_2$

4.4 Raffiner une machine abstraite

Nous nous intéressons à la relation de raffinement entre deux machines M_1 et M_2. L'idée est que les deux machines implantent les mêmes opérations mais en utilisant des *variables d'états différentes*.

4.4.1 Exemple

On peut se donner une première machine `Ens1` qui a pour variable d'état, un ensemble fini `ens` d'entiers et une opération d'ajout d'un entier à cet ensemble. Un raffinement de cette machine consiste à vouloir représenter cet ensemble par un tableau, en introduisant un index pour la dernière case occupée dans le tableau.

La machine `Ens1` s'écrit donc :

MACHINE $\texttt{Ens1}(\texttt{maxe})$
CONSTRAINTS $\texttt{maxe} \in \mathbb{N}_1$
VARIABLES $\texttt{ens}$
INVARIANT $\texttt{ens} \subseteq \mathbb{N} \wedge \mathrm{card}(\texttt{ens}) \leq \texttt{maxe}$
INITIALISATION $\texttt{ens} := \emptyset$
OPERATIONS
$\texttt{ajout}(n) \hat{=}$
PRE $n \in \mathbb{N} - \texttt{ens} \wedge \mathrm{card}(\texttt{ens}) < \texttt{maxe}$
THEN $\texttt{ens} := \texttt{ens} \cup \{n\}$ END
END

Une machine effectuant les mêmes tâches que `Ens1` mais utilisant un tableau aura la forme suivante :

MACHINE $\texttt{Ens2}(\texttt{maxe})$
CONSTRAINTS $\texttt{maxe} \in \mathbb{N}_1$
VARIABLES $\texttt{tab}, \texttt{index}$
INVARIANT $\texttt{index} \in 0..\texttt{maxe} \wedge \texttt{tab} \in 1..\texttt{index} \rightarrowtail \mathbb{N}$
INITIALISATION $\texttt{index}, \texttt{tab} := 0, \emptyset$
OPERATIONS
$\texttt{ajout}(n) \hat{=}$
PRE $n \in \mathbb{N} - \mathrm{ran}(\texttt{tab}) \wedge \texttt{index} < \texttt{maxe}$
THEN $\texttt{index} := \texttt{index}+1 \parallel \texttt{tab}(\texttt{index}+1) := n$ END
END

On rappelle que la formule $\texttt{tab} \in 1..\texttt{index} \rightarrowtail \mathbb{N}$ signifie que `tab` est une injection (cf A.4.2).
Pour montrer que `Ens2` est un raffinement de `Ens1` il faut établir un lien entre les variables d'état des deux machines. Ici on va demander que l'image de la fonction `tab` soit égale à l'ensemble `ens`.
En pratique, la machine `Ens2` va explicitement être introduite comme un raffinement de `Ens1` avec un invariant qui contraint les variables de `Ens2` et qui de plus établit le lien entre les deux états.

REFINEMENT $\texttt{Ens2}$
REFINES $\texttt{Ens1}$
VARIABLES $\texttt{tab}, \texttt{index}$
INVARIANT $\texttt{index} \in 0..\texttt{maxe} \wedge \texttt{tab} \in 1..\texttt{index} \rightarrowtail \mathbb{N} \wedge \ \mathrm{ran}(\texttt{tab}) = \texttt{ens}$
INITIALISATION $\texttt{index}, \texttt{tab} := 0, \emptyset$
OPERATIONS
$\texttt{ajout}(n) \hat{=}$
PRE $n \in \mathbb{N} - \mathrm{ran}(\texttt{tab}) \wedge \texttt{index} < \texttt{maxe}$
THEN $\texttt{index} := \texttt{index}+1 \parallel \texttt{tab}(\texttt{index}+1) := n$ END
END

La machine raffinée `Ens2` récupère les paramètres, ensembles, constantes contraintes et propriétés de la machine qui est raffinée. Toutes les variables de la machine `Ens1` sont visibles dans l'invariant de `Ens2` mais ne peuvent être utilisées dans les substitutions.

Les obligations de preuves Les obligations de preuve doivent établir que l'initialisation et les opérations de `Ens2` préservent l'invariant en ce qui concerne la partie concernant les variables de `Ens2` et que si on fait les opérations qui se correspondent dans `Ens1` et `Ens2` alors les résultats obtenus par l'opération `Ens2` seront un cas particulier des résultats attendus dans `Ens1`.

Pour l'initialisation, on doit assurer que l'initialisation I_2 de `Ens2` correspond à un cas particulier d'initialisation I_1 de `Ens1` et établit l'invariant de `Ens2`.
Comme il peut y avoir plus d'initialisation dans `Ens1` que dans `Ens2` au lieu d'établir $[I_2]([I_1]Inv_2)$ qui dirait que toutes les exécutions de I_1 établissent Inv_2 on établit qu'il existe une exécution de I_1 qui établit Inv_2, ou encore qu'il n'est pas possible que toutes les exécutions de I_1 établissent $\neg Inv_2$. La condition est donc :

$$[I_2](\neg[I_1]\neg Inv_2)$$

Dans le cas de l'opération `ajout` on peut supposer qu'elle va être appelée dans les conditions requises par `Ens1` et que de plus les invariants de `Ens1` et `Ens2` seront vérifiés. Il faut vérifier qu'alors la précondition de `ajout` dans `Ens2` est vérifiée et que, après l'exécution de la substitution associée dans `Ens2`, on peut trouver une exécution de `ajout` dans `Ens1` qui validera Inv_2.
En supposant que `ajout` s'écrit PRE P_1 THEN S_1 END dans `Ens1` et PRE P_2 THEN S_2 END dans `Ens2`, on obtient l'obligation de preuve

$$Inv_1 \wedge Inv_2 \wedge P_1 \Rightarrow (P_2 \wedge [S_2](\neg[S_1]\neg Inv_2))$$

4.4.2 Cas général

On suppose que M_1 a une variable d'état e_1, un invariant Inv_1, une initialisation I_1 et contient une opération op_1 de la forme :

$$y_1 \longleftarrow op(x_1) \hat{=} \mathsf{PRE}\ P_1\ \mathsf{THEN}\ S_1\ \mathsf{END}$$

Dans l'opération de raffinement, on introduit une machine M_2 avec une nouvelle variable d'état e_2 (distincte de e_1), un invariant Inv_2, une initialisation I_2 et contient une opération op_2 de la forme :

$$y_2 \longleftarrow op(x_2) \hat{=} \mathsf{PRE}\ P_2\ \mathsf{THEN}\ S_2\ \mathsf{END}$$

Quitte à renommer les paramètres, on peut toujours supposer qu'ils sont différents dans les deux opérations.
On veut montrer que le comportement de op_2 est un cas particulier de celui attendu de op_1.
On demande que, à chaque état de M_2 corresponde au moins un état de M_1.

Invariant combiné Ceci nous amène à introduire un nouvel invariant portant sur les variables de M_2 :

$$Inv \equiv \exists e_1.Inv_1 \wedge Inv_2$$

Nous allons établir les propriétés portant sur Inv_1 et Inv_2 pour que Inv soit un invariant de la machine M_2 et que l'opération op_2 soit un raffinement de op_1.

Initialisation On retrouve la formule :

$$\forall e_1, e_2.[I_2](\neg[I_1]\neg Inv_2)$$

Opération Contrairement au cas précédent, nos opérations ont un paramètre de sortie, il faut donc établir qu'à entrées égales, la sortie de op_2 est un cas particulier de sortie attendue par op_1 ce qui donne :

$$\forall e_1, e_2.Inv_1 \wedge Inv_2 \Rightarrow (\forall x_1, x_2.x_1 = x_2 \wedge P_1 \Rightarrow (P_2 \wedge [S_2](\neg[S_1]\neg(y_1 = y_2 \wedge Inv_2))))$$

5 Implantation

L'implantation est un cas particulier de raffinement en une machine qui ne pourra plus être raffinée. La forme générale d'une machine d'implantation est :

IMPLEMENTATION *nomI*	/* nom de la machine d'implantation */
REFINES *nom*	/* nom de la machine raffinée */
⋮	
END	

5.1 Principes

Les implantations respectent certaines règles :
- Une implantation n'a pas de variables propres.
- Une implantation utilise les opérations de machines qu'elle importe (primitives ou implantées par ailleurs) et dont elle ne peut voir les variables sauf pour exprimer les invariants.
- Certaines restrictions sont appliquées aux substitutions apparaissant dans une implantation afin d'assurer qu'elle pourra être traduite en un programme exécutable :
  - pas de choix non déterministe ou de composition parallèle,
  - les conditions dans les substitutions IF-THEN-ELSE sont limitées à des expressions booléennes simples,
  - les opérations ne s'appliquent qu'à des variables locales ou paramètres,
  - les affectations mettent en jeu des expressions simples,
  - ...
- une implantation peut introduire des substitutions correspondant à des boucles WHILE, il faudra par contre indiquer pour chaque boucle WHILE, un invariant et un variant (quantité qui décroit à chaque passage dans la boucle).
- Au moment de l'implantation, toutes les constantes déclarées ou les ensembles non énumérés doivent être instanciés (explicitement dans une clause **VALUES** ou dans une des machines importées).

5.2 Restrictions

5.2.1 Substitutions autorisées

Les substitutions autorisées dans une implantation sont :
- L'introduction de variables locales par la substitution VAR IN END. Les variables introduites doivent être typées dans la substitution en apparaissant dans une substitution d'initialisation (affectation ou comme variable de sortie d'une opération).
- La substitution BEGIN END qui permet de grouper une séquence de substitutions.
- La substitution IF THEN ELSE END et ses variantes :
  (IF THEN END, IF THEN ELSEIF THEN ELSE END ...)
  La condition doit correspondre à une expression booléenne calculable.
- La substitution CASE qui est une forme de SELECT mais dont les prédicats testent l'appartenance à un ensemble. La substitution s'écrit :

  $$\mathsf{CASE}\ e\ \mathsf{OF\ EITHER}\ E_1\ \mathsf{THEN}\ S_1\ \mathsf{OR}\ E_2\ \mathsf{THEN}\ S_2\ \mathsf{END\ END}$$

  Dans cette substitution, e est une expression simple, E_i un ensemble et S_i une substitution, le nombre de branches peut être quelconque et une clause ELSE peut également apparaître.
- La séquence de deux substitutions.
- Une boucle WHILE précisant un variant et un invariant.

– L'affectation d'une expression simple à une variable qui peut être une variable locale ou une variable de sortie de l'opération.
– Une opération d'une machine vue ou importée. La sortie de l'opération utilisée doit être formée de variables locales ou de variables de sortie de l'opération dans laquelle apparaît la substitution.

Toutes les autres substitutions ANY, SELECT, LET ... ne sont pas autorisées dans les implantations.

5.2.2 Expressions autorisées

Dans les affectations ou dans les boucles, les expressions qui peuvent être utilisées sont restreintes.
Une *expression simple* est une variable locale, ou une variable d'entrée ou bien une constante (des machines raffinées, importées ou vues) ou un littéral (constante entière, valeur booléenne `TRUE` ou `FALSE`).
Les *expressions générales* qui peuvent être utilisées dans une implantation peuvent être soit une expression simple ou bien une expression arithmétique construite à partir des opérateurs d'addition, de soustration, de multiplication, de division, de reste modulo ou d'exponentiation.
Dans les conditions des substitutions IF ou WHILE, les expressions peuvent être formées à partir des connecteurs $\neg, \wedge, \vee$ and $\Rightarrow$ et des formules de comparaison d'expressions simples.

5.3 La clause IMPORTS

Si une implantation ne peut plus être raffinée, elle peut par contre importer des machines qui, elles, seront raffinées.
Une implantation va être construite en combinant d'autre machines plus élémentaires suivant un principe de développement en niveaux. En général on utilisera les descriptions abstraites des machines élémentaires qui elles-mêmes seront raffinées et implantées. On pourra utiliser les machines des bibliothèques présentes dans les outils de développement.
La clause **IMPORTS** se comporte dans les implémentations de manière analogue à la clause **INCLUDES** utilisée dans les spécifications. Cependant, les variables de la machine importée seront complètement cachées dans les opérations de la machine qui importe.

5.4 La clause VALUES

La clause **VALUES** apparait dans les machines d'implantation pour donner une valeur aux constantes concrètes de la clause **CONSTANTS** et aux ensembles abstraits de la clause **SETS** de la machine raffinée.
La clause **VALUES** est suivie d'un ensemble de déclarations *id*=*valeur* séparées par des points virgules. *id* est le nom de la donnée à valuer et *valeur* sa valeur. L'ordre des valuations est important. Une constante ne peut être utilisée dans la partie *valeur* que si elle a été préalablement définie.
Lorsqu'une constante ou un ensemble abstrait d'une implantation M apparaît avec le même nom dans une machine M' importée ou bien vue par M, alors la donnée de M est implicitement valuée par l'objet de même nom de la machine M'.

5.4.1 Valuation d'un ensemble abstrait

Un ensemble abstrait doit toujours être valué par un ensemble non vide d'entiers. Ce peut-être :
– un intervalle d'entiers de la forme $n_1..n_2$
– un ensemble abstrait d'une machine vue ou importée
– un ensemble abstrait de la machine implantée qui a déjà été valué

5.4.2 Valuation d'une constante scalaire

Une constante scalaire a pour type `INT`, ou `BOOL` ou bien un ensemble abstrait ou énuméré. Elle doit être valuée par une expression appartenant au type correspondant.

5.4.3 Valuation d'une constante de tableau

Une constante de tableau peut être valuée
- par une constante de machine vue ou importée,
- par un tableau dont tous les éléments ont la même valeur, exprimé sous la forme $D \times \{e\}$ avec D le domaine du tableau et e la valeur des éléments du tableau, par exemple un tableau `tab` $\in (0..2) \longrightarrow \mathbb{N}$ pourra être implanté par la clause :
  VALUES tab $= (0..2) \times \{0\}$
- par un ensemble de *maplets* qui permettent de représenter des n-uplets, les $n-1$ premiers éléments représentent les indices du tableau et le dernier la valeur en ce point, par exemple un tableau `tab` $\in (0..1) \times (0..1) \longrightarrow \mathbb{N}$ pourra être implanté par la clause
  VALUES tab $= \{0 \mapsto 0 \mapsto 0, 0 \mapsto 1 \mapsto 1, 1 \mapsto 0 \mapsto 2, 1 \mapsto 1 \mapsto 3\}$

5.5 Substitution WHILE

Afin de pouvoir exprimer des programmes qui effectuent un nombre non borné d'itérations, le langage B autorise l'utilisation d'une substitution WHILE. La syntaxe d'une substitution WHILE est :

$$\mathsf{WHILE}\ P\ \mathsf{DO}\ S\ \mathsf{INVARIANT}\ I\ \mathsf{VARIANT}\ V\ \mathsf{END}$$

P est un prédicat (la condition de la boucle, S est une substitution (le corps de la boucle), I est un prédicat, l'invariant et V est une expression, le variant qui représente une grandeur qui diminue à chaque passage dans la boucle mais reste positive.
L'action de cette substitution sur un prédicat R est défini comme la conjonction de plusieurs propriétés :

1	L'invariant I est satisfait à l'entrée de la boucle	I
2	L'invariant I est préservé par le passage dans la boucle	$I \wedge P \Rightarrow [S]I$
3	Le variant est un entier	$I \Rightarrow V \in \mathbb{N}$
4	Le variant décroit dans le passage dans la boucle	$I \wedge P \Rightarrow [n := V][S](V < n)$
5	En sortie de boucle, la propriété R découle de l'invariant et de la non-satisfaction de la condition	$I \wedge \neg P \Rightarrow R$

Dans la condition (4), n est une variable qui n'apparaît pas dans la substitution WHILE. La propriété $[S](V < n)$ assure que la valeur de V après l'exécution de S est inférieure à n. La substitution $[n := V]$ remplace alors n par la valeur de V avant l'exécution de S.
Supposons que l'on veuille implanter une opération qui prend en argument un ensemble non vide c d'entiers positifs et renvoie le minimum de cet ensemble. Dans la machine de spécification, l'opération peut apparaître sous la forme :

$$r \longleftarrow \texttt{MinSet}(c) \mathrel{\hat=} \mathsf{PRE}\ c \in \mathbb{P}_1(\mathbb{N})\ \mathsf{THEN}\ r := \texttt{min}(c)\ \mathsf{END}$$

Si on cherche maintenant à en faire une implantation concrète, on peut choisir une méthode où on teste successivement les entiers à partir de 0, le premier qui apparaît dans l'ensemble est le minimum (cette méthode n'est intéressante d'un point de vue algorithmique que si le test pour savoir si un objet appartient à un ensemble est efficace ...)

Dans le raffinement, l'opération `MinSet` pourra être

```
r ⟵ MinSet(c) ≙
    PRE c ∈ ℙ1(ℕ)
    THEN r := 0;
        WHILE r ∉ c DO r := r+1
        INVARIANT r ∈ 0..min(c) VARIANT min(c)−r
        END
    END
```

5.6 Exemple

Nous prenons un exemple de machine représentant un ensemble fini (ici d'entiers non nuls), qui permet d'ajouter des éléments mais également de calculer le maximum de l'ensemble. La spécification de la machine est la suivante :

```
MACHINE MaxEns
VARIABLE ens
INVARIANT ens ∈ 𝔽(ℕ1)
INITIALISATION ens := ∅
OPERATIONS
  enter(n) ≙ PRE n ∈ ℕ1 THEN ens := ens∪{n} END;
  m ⟵ maximum ≙ PRE ens ≠ ∅ THEN m := max(ens) END
END
```

Comme la seule opération à réaliser sur l'ensemble stocké dans cette machine est de trouver le maximum, il n'est pas utile de garder un ensemble arbitraire, on peut se contenter d'une variable entière. On obtient donc le raffinement :

```
REFINEMENT MaxEns1
REFINES MaxEns
VARIABLES mEns
INVARIANT mEns = max(ens∪{0})
INITIALISATION mEns := 0
OPERATIONS
  enter(n) ≙ PRE n ∈ ℕ1 THEN mEns := max({mEns, n}) END;
  m ⟵ maximum ≙ PRE z ≠ 0 THEN m := mEns END
END
```

Exercice Énoncer les obligations de preuve
Il est possible de faire plusieurs raffinement successifs, par exemple le corps de l'opération `ajout` peut être modifié pour faire disparaître l'appel à la fonction `max`.

```
REFINEMENT MaxEns2
REFINES MaxEns1
VARIABLES mEns
INITIALISATION mEns := 0
OPERATIONS
  enter(n) ≙ IF n ≥ mEns THEN mEns := n END;
  m ⟵ maximum ≙ m := mEns END
END
```

La variable de la machine `MaxEns2` ayant le même nom que celle de la machine `MaxEns1`, un invariant implicite disant que ces deux variables sont égales est ajouté.
Pour implanter notre exemple, nous avons besoin d'une variable entière qui sera obtenue en important une machine implantant une variable entière :

```
MACHINE Scalar(initval)
CONSTRAINTS initval ∈ ℕ
VARIABLES z
INVARIANT z ∈ ℕ
INITIALISATION z := initval
OPERATIONS
  v ⟵ valeur ≙ v := z;
  modifie(v) ≙ PRE v ∈ ℕ THEN z := v END
END
```

Nous pouvons alors écrire l'implémentation de la machine `MaxEns`.

```
IMPLEMENTATION MaxEns3
REFINES MaxEns2
IMPORTS Scalar(0)
INVARIANT z = mEns
OPERATIONS
  enter(n) ≙
     VAR v
     IN v ⟵ value; IF n ≥ v THEN modify(n) END
     END;
  m ⟵ maximum ≙ BEGIN m ⟵ value END
END
```

6 Conclusion

La méthode B introduit un langage logique rigoureux pour représenter les programmes (substitutions généralisées) et les propriétés de ces programmes. Elle utilise une notion simple de *Machine Abstraite* qui est assez voisine de la notion d'objet en programmation mais qui intègre de plus la notion d'*Invariant* qui assure la préservation de propriétés des variables d'état de la machine quelles que soient les opérations appliquées.
Les gros développements logiciel se construisent en combinant des machines abstraites élémentaires ou en utilisant la relation de *raffinement* entre machines.
Des outils permettent de développer des systèmes en utilisant la méthode B. Ces outils offrent différentes fonctionnalités :

- Analyse syntaxique des machines.
- Vérification de la bonne formation des propriétés et programmes (chaque variable est associée à un univers).
- Génération des obligations de preuve (préservation des invariants, respect de la sémantique des programmes lors du raffinement, ...).
- Preuves automatiques ou interactives.
- Animation des machines afin de vérifier la pertinence des spécifications.
- Génération de code à partir des machines d'implantations.
- Documentation des développements.

Cette méthode a pu être appliquée avec succès dans des applications industrielles critiques.

A Eléments du langage de la méthode *B*

Dans cette annexe, nous récapitulons le constructions avancées de la méthode B qui seront utilisées dans ce cours. Nous indiquons dans la première colonne la notation mathématique et dans la seconde la notation ASCII utilisée dans les outils.

A.1 Opérations arithmétiques

math	ASCII	
e_1+e_2	e_1`+`e_2	Addition
e_1-e_2	e_1`-`e_2	Soustraction
$-e$	`-`e	Moins unaire
$e_1\times e_2$	e_1`*`e_2	Multiplication
e_1/e_2	e_1`/`e_2	Division entière
$e_1^{e_2}$	e_1`**`e_2	Puissance
$e_1 \bmod e_2$	e_1 `mod` e_2	Modulo

A.2 Les ensembles

A.2.1 Construction d'ensembles

math	ASCII	
$\emptyset$	`{}`	Ensemble vide
$\mathbb{N}$	`NAT`	Ensemble des entiers
$\mathbb{N}_1$	`NAT1`	Ensemble des entiers non nuls
$\{e_1,\ldots,e_n\}$	`{`e_1`,`$\ldots$`,`e_n`}`	Ensemble composé des éléments $e_1,\ldots,e_n$.
$n_1..n_2$	`(`n_1`..`n_2`)`	ensemble composé des entiers compris entre n_1 et n_2
$\mathbb{P}(E)$	`POW(`E`)`	L'ensemble des parties de E
$\mathbb{P}_1(E)$	`POW1(`E`)`	L'ensemble des parties non vides de E
$\mathbb{F}(E)$	`FIN(`E`)`	L'ensemble des parties finies de E
$E_1\times E_2$	E_1`*`E_2	Ensemble des couples (e_1,e_2) formés de e_1 dans l'ensemble E_1 et de e_2 dans l'ensemble E_2

A.2.2 Composition d'ensembles

math	ASCII	
$E_1\cup E_2$	E_1`\/`E_2	L'union des ensembles E_1 et E_2
$E_1\cap E_2$	E_1`/\`E_2	L'intersection des ensembles E_1 et E_2
E_1-E_2	E_1`-`E_2	E_1 auxquel on a soustrait les éléments de E_2

A.2.3 Autres constructions

math	ASCII	
$\max(E)$	`max(`E`)`	L'entier maximal d'un ensemble fini non vide d'entiers E.
$\min(E)$	`min(`E`)`	L'entier minimal d'un ensemble fini non vide d'entiers E.

A.3 Les formules

A.3.1 Formules de base

e_1 et e_2 représentent des expressions quelconques (entiers, ensembles, paires ...), n_1 et n_2 sont des expressions qui représentent des entiers.

math&ASCII
$e_1 = e_2$
$n_1 > n_2$
$n_1 < n_2$

math	ASCII
$e_1 \neq e_2$	e_1`/=`e_2
$n_1 \geq n_2$	n_1`>=`n_2
$n_1 \leq n_2$	n_1`<=`n_2

A.3.2 Formules ensemblistes

Dans le tableau suivant, e représente une expression tandis que E, E_1 et E_2 représentent des ensembles.

math	ASCII	
$e \in E$	e `:` E	L'expression e est un objet de l'ensemble E
$e \notin E$	e`/:`E	L'expression e n'est pas un objet de l'ensemble E.
$E_1 \subseteq E_2$	E_1`<:`E_2	E_1 est un sous-ensemble de E_2.
$E_1 \not\subseteq E_2$	E_1`/<:`E_2	E_1 n'est pas un sous-ensemble de E_2.
$E_1 \subset E_2$	E_1`<<:`E_2	E_1 est un sous-ensemble strict de E_2 (différent de E_2).
$E_1 \not\subset E_2$	E_1`/<<:`E_2	E_1 n'est pas un sous-ensemble strict de E_2.

A.3.3 Formules composées

Dans le tableau suivant, F, F_1 et F_2 représentent des formules.

math	ASCII	
$\neg F$	`not`(F)	négation
$F_1 \wedge F_2$	F_1 & F_2	conjonction
$F_1 \vee F_2$	F_1 `or` F_2	disjonction
$F_1 \Rightarrow F_2$	F_1 `=>` F_2	implication
$F_1 \Leftrightarrow F_2$	F_1 `<=>` F_2	équivalence
$\exists$ var . F	`#` var . F	quantification existentielle
$\forall$ var . F	`!` var . F	quantification universelle

A.4 Relations et fonctions

A.4.1 Relations

E, E_1 et E_2 représentent des ensembles tandis que R, R_1 et R_2 représentent des relations.

math	ASCII	
$E_1 \leftrightarrow E_2$	E_1`<->`E_2	L'ensemble des relations entre éléments de E_1 et de E_2
		$\mathbb{P}(E_1 \times E_2)$
$R_1 ; R_2$	R_1`;`R_2	Composition des relations R_1 et R_2
		$(x, y) \in (R_1; R_2) \Leftrightarrow \exists z.((x, z) \in R_1 \wedge (z, y) \in R_2)$
$\mathrm{id}(E)$	`id(`E`)`	la relation identité sur l'ensemble E
R^{-1}	R`~`	relation inverse de R
		$(x, y) \in R^{-1} \Leftrightarrow (y, x) \in R$
$E \lhd R$	E`<|`R	la restriction de la relation R au domaine E
		$(x, y) \in (E \lhd R) \Leftrightarrow (x \in E \wedge (x, y) \in R)$
$R \rhd E$	R`|>`E	la restriction de la relation R à l'image E
		$(x, y) \in (R \rhd E) \Leftrightarrow ((x, y) \in R \wedge y \in E)$
$E \ntriangleleft R$	E`<<|`R	l'anti-restriction de la relation R au domaine E
		$(x, y) \in (E \ntriangleleft R) \Leftrightarrow (x \notin E \wedge (x, y) \in R)$
$R \ntriangleright E$	R`|>>`E	l'anti-restriction de la relation R à l'image E
		$(x, y) \in (R \ntriangleright E) \Leftrightarrow ((x, y) \in R \wedge y \notin E)$
$R^{\cdot}$	`closure(`R`)`	la fermeture transitive de R
		$(x, y) \in R^{\cdot} \Leftrightarrow \exists(x_0, \ldots, x_n).x = x_0 \wedge y = x_n \wedge \forall i.(x_i, x_{i+1}) \in R$

Ensembles définis à partir de relations

math	ASCII	
$\mathrm{dom}(R)$	`dom(`R`)`	le domaine de la relation R
		$x \in \mathrm{dom}(R) \Leftrightarrow (\exists y.(x, y) \in R)$
$\mathrm{ran}(R)$	`ran(`R`)`	l'image de la relation R
		$y \in \mathrm{ran}(R) \Leftrightarrow (\exists x.(x, y) \in R)$
$R[E]$	R`[`E`]`	l'image de l'ensemble E par la relation R
		$y \in R[E] \Leftrightarrow (\exists x.x \in E \wedge (x, y) \in R)$

A.4.2 Fonctions

math	ASCII	
$E_1 \nrightarrow E_2$	E_1`+->`E_2	l'ensemble des fonctions partielles de E_1 dans E_2
		$\forall(x, y, z).(x \mapsto y) \in f \wedge (x \mapsto z) \in f \Rightarrow y = z$
$E_1 \longrightarrow E_2$	E_1`-->`E_2	l'ensemble des fonctions totales de E_1 dans E_2
		$f \in E_1 \nrightarrow E_2 \wedge \forall x.x \in E_1 \Rightarrow \exists y.(x \mapsto y) \in f)$
$E_1 \rightarrowtail\!\!\!\!\nrightarrow E_2$	E_1`>+>`E_2	l'ensemble des fonctions partielles injectives de E_1 dans E_2
		$f \in E_1 \nrightarrow E_2 \wedge (\forall(x, y, z).(x \mapsto z) \in f \wedge (y \mapsto z) \in f \Rightarrow x = y)$
$E_1 \rightarrowtail E_2$	E_1`>->`E_2	l'ensemble des fonctions totales injectives de E_1 dans E_2
		$f \in E_1 \longrightarrow E_2 \wedge (\forall(x, y, z).(x \mapsto z) \in f \wedge (y \mapsto z) \in f \Rightarrow x = y)$
$E_1 \twoheadrightarrow\!\!\!\!\!\nrightarrow E_2$	E_1`+->>`E_2	l'ensemble des fonctions partielles surjectives de E_1 dans E_2
		$f \in E_1 \nrightarrow E_2 \wedge (\forall y.\exists x.x \mapsto y \in f)$
$E_1 \twoheadrightarrow E_2$	E_1`-->>`E_2	l'ensemble des fonctions totales surjectives de E_1 dans E_2
		$f \in E_1 \longrightarrow E_2 \wedge (\forall y.\exists x.x \mapsto y \in f)$
$E_1 \rightarrowtail\!\!\!\!\twoheadrightarrow E_2$	E_1`>->>`E_2	l'ensemble des fonctions bijectives de E_1 dans E_2
		$f \in E_1 \rightarrowtail E_2 \wedge f \in E_1 \twoheadrightarrow E_2$

Image d'une fonction Si f est une fonction et $e_1 \in \mathrm{dom}(f)$ alors $f(e_1)$ représente l'unique expression e_2 telle que $e_1 \mapsto e_2 \in f$. Si E est un ensemble alors $f(E)$ représente l'image de E par la relation f, c'est-à-dire l'ensemble des y pour lesquels il existe x tel que $x \mapsto y \in f$ ou encore l'ensemble des $f(x)$ pour $x \in E \cup \mathtt{dom}(f)$.

A.4.3 Séquences

Dans le tableau suivant, s_1 et s_2 représentent des séquences, e, $e_1, \ldots, e_n$ des expressions quelconques.

math	ASCII	
$\mathrm{seq}(E)$	`seq`(E)	L'ensemble des séquences finies d'objets de E
$[]$	`<>`	la séquence vide
$[e]$	`[`e`]`	la séquence réduite à un élément e
$[e_1, \ldots, e_n]$	`[`$e_1, \ldots, e_n$`]`	la séquence formée des n éléments $e_1, \ldots, e_n$
$s_1 \hat{} s_2$	s_1`^`s_2	La concaténation des séquences s_1 et s_2
$e \rightarrow s$	e`->`s	L'ajout de e au début de la séquence s
$s \leftarrow e$	s`<-`e	L'ajout de e à la fin de la séquence s
$\mathrm{first}(s)$	`first`(s)	le premier élément de la séquence non-vide s
$\mathrm{tail}(s)$	`tail`(s)	la séquence non-vide s privée de son premier élément
$\mathrm{size}(s)$	`size`(s)	la taille de la séquence s

Chaînes de caractères Les constantes correspondant à des chaînes de caractères sont écrites entre guillemets comme `"Hello"`.

B Exemple : un carnet d'anniversaire

B.1 Modélisation

Nous allons étudier comment modéliser un carnet d'anniversaires.
Le carnet doit contenir pour un ensemble de personnes, leur nom et la date de leur anniversaire. Les opérations possibles sur le carnet sont :

- ajouter un nom et une date anniversaire
- trouver la date anniversaire d'une personne
- trouver toutes les personnes dont l'anniversaire est un certain jour.

On suppose de plus que le nombre maximal de personnes pouvant être entrées dans le carnet est fixé.
On choisit de modéliser le carnet par une machine paramétrée par deux ensembles `NOM` et `DATE` représentant respectivement les univers des noms des personnes et des dates, ainsi qu'un entier `max` qui représente le nombre maximal d'entrées dans le carnet.
Une modélisation possible est la suivante :

```
MACHINE CarnetAnniversaires(NOM,DATE,maxn)
CONSTRAINTS maxn : ℕ1
VARIABLES connus,anniversaire
INVARIANT
  connus ⊆ NOM ∧ card(connus) ≤ maxn ∧ anniversaire ∈ connus ↣ DATE
INITIALISATION connus,anniversaire := ∅,∅
OPERATIONS
  ajout(n,a) ≙
     PRE n ∈ NOM−connus ∧ a ∈ DATE ∧ card(connus) < maxn
     THEN connus := connus∪{n}||anniversaire(n) := a END
  a ⟵ trouve(n) ≙
     PRE n ∈ connus THEN a := anniversaire(n) END
  ns ⟵ a_feter(j) ≙
     PRE j ∈ DATE THEN ns := anniversaire⁻¹[{j}] END
END
```

On peut vérifier que les différentes préconditions des opérations permettent d'assurer que l'invariant de la machine est bien préservé.

B.2 Machine défensive

Cependant les opérations de cette machine ont des conditions d'utilisation trop restrictives: ce n'est pas à l'utilisateur de s'assurer que le nom qu'il ajoute n'est pas dans la base, que la base n'est pas pleine ou que le nom qu'il cherche est bien dans la base.
Pour remédier à ce problème, on modifie les opérations de manière à ce que chaque opération renvoie un indicateur si l'opération a pu se dérouler normalement et un autre indicateur si une erreur s'est produite. On utilisera un indicateur différent pour chaque type d'erreur. On peut faire une analogie avec les codes de retour des fonctions unix, ou un mécanisme paramétré d'erreurs.
On choisit d'introduire une nouvelle machine `CarnetAnniversairesI` qui aura les mêmes variables que la machine `CarnetAnniversaires` mais qui contiendra un nouvel ensemble `RAPPORT` ainsi que de nouvelles opérations pour l'ajout ou la consultation renvoyant les rapports apropriés. Comme cette nouvelle machine a les mêmes variables d'état que l'ancienne machine, il est possible d'utiliser une déclaration `INCLUDES`.

MACHINE `CarnetAnniversairesI(NOM,DATE,max)`
CONSTRAINTS $\texttt{maxn} : \mathbb{N}_1$
INCLUDES `CarnetAnniversaires(NOM,DATE,max)`
SETS $\texttt{RAPPORT} = \{\texttt{ok},\texttt{deja\_connu},\texttt{inconnu},\texttt{plein}\}$
OPERATIONS
$r \longleftarrow \texttt{ajoutI}(n, a) \mathrel{\hat{=}}$
  PRE $n \in \texttt{NOM} \wedge a \in \texttt{DATE}$ THEN
   IF $n \notin \texttt{connus}$ THEN
    IF $\mathrm{card}(\texttt{connus}) < \texttt{maxn}$ THEN $\texttt{ajout}(n, a) \| r := \texttt{ok}$ ELSE $r := \texttt{plein}$ END
   ELSE $r := \texttt{deja\_connu}$ END
  END;
$r, a \longleftarrow \texttt{trouveI}(n) \mathrel{\hat{=}}$
  PRE $n \in \texttt{NOM}$ THEN
  IF $n \in \texttt{connus}$ THEN $(r, a) := (\texttt{ok}, \texttt{trouve}(n))$
  ELSE $r := \texttt{inconnu} \| a :\in \texttt{DATE}$ END
  END;
$r, ns \longleftarrow \texttt{a\_feterI}(j) \mathrel{\hat{=}}$ PRE $j \in \texttt{DATE}$ THEN $ns \longleftarrow \texttt{a\_feter}(j) \| r := \texttt{ok}$ END
END

B.3 Raffinement

Le raffinement transforme les ensembles de noms et la relation d'anniversaire en deux tableaux partageant les mêmes indices.

REFINEMENT `CarnetAnniversairesRI`
REFINES `CarnetAnniversairesI`
VARIABLES `noms, dates, index`
INVARIANT
$\texttt{index} \in 0..\texttt{maxn} \wedge \texttt{noms} \in 1..\texttt{index} \rightarrowtail \texttt{NOM} \wedge \texttt{dates} \in 1..\texttt{index} \rightarrowtail \texttt{DATE}$
$\wedge \texttt{connus} = \mathrm{ran}(\texttt{noms})$
$\wedge \forall x, y.(x \mapsto y) \in \texttt{anniversaire} \Leftrightarrow \exists i.i \in 1..\texttt{index} \wedge \texttt{noms}(i) = x \wedge \texttt{dates}(i) = y$
INITIALISATION $\texttt{index}, \texttt{noms}, \texttt{dates} := 0, \emptyset, \emptyset$
OPERATIONS
$r \longleftarrow \texttt{ajoutI}(n, a) \mathrel{\hat{=}}$
  PRE $n \in \texttt{NOM} \wedge a \in \texttt{DATE}$ THEN
   IF $n \notin \mathrm{ran}(\texttt{noms})$ THEN
    IF $\texttt{index} < \texttt{maxn}$
     THEN $\texttt{noms}(\texttt{index}+1) := n \,\|\texttt{dates}(\texttt{index}+1) := a \,\|$
       $\texttt{index} := \texttt{index}+1 \| r := \texttt{ok}$
     ELSE $r := \texttt{plein}$ END
   ELSE $r := \texttt{deja\_connu}$ END
  END;
$r, a \longleftarrow \texttt{trouveI}(n) \mathrel{\hat{=}}$
  PRE $n \in \texttt{NOM}$
  THEN IF $n \in \mathrm{ran}(\texttt{noms})$ THEN $(r, a) := (\texttt{ok}, \texttt{dates}(\texttt{noms}^{-1}(n)))$
     ELSE $r := \texttt{inconnu} \,\| a :\in \texttt{DATE}$ END
  END;
$r, ns \longleftarrow \texttt{a\_feterI}(j) \mathrel{\hat{=}}$
  PRE $j \in \texttt{DATE}$ THEN $ns \longleftarrow ns := \texttt{noms}[\texttt{dates}^{-1}[\{j\}]] \| r := \texttt{ok}$ END
END

B.4 Implantation

Pour traiter notre exemple, nous pouvons en particulier utiliser une machine `Arr` qui encapsule un tableau (variable `arr` avec des opérations d'accès et de mise à jour) et une machine `Nvar` pour implanter une variable entière `var`. On utilise deux tableaux (l'un pour les noms, l'autre pour les dates), lors de l'import, un préfixe permet de distinguer les deux instances.

B.4.1 Machine auxilliaire : variable

Pour la machine qui implante une variable entière, nous utilisons des opérations de lecture et d'incrémentation.
La spécification de la machine `Nvar` est la suivante :

```
MACHINE Nvar(maxn)
CONSTRAINTS maxn : ℕ1
VARIABLES nvar
INVARIANT nvar ∈ 0..maxn
INITIALISATION nvar := 0
OPERATIONS
  v ⟵ val_nvar ≙ v := nvar;
  inc_nvar ≙
    PRE nvar < maxn THEN nvar := nvar+1 END;
END
```

B.4.2 Tableaux

Pour les tableaux, nous aurons besoin des opérations d'accès à une valeur, de mise à jour ainsi que d'une fonction de recherche d'un élément dans un sous-ensemble du tableau. Cette opération renvoie un booléen qui dit si l'élément a été trouvé et dans le cas positif renvoie le plus petit indice où l'élément figure.

```
MACHINE Arr(TYPE,maxn)
CONSTRAINTS maxn : ℕ1
SEES Bool_TYPE
VARIABLES arr
INVARIANT arr ∈ 1..maxn ⇸ TYPE
INITIALISATION arr := ∅
OPERATIONS
  v ⟵ val_arr(i) ≙ PRE i ∈ dom(arr) THEN v := arr(i) END;
  sto_arr(i,v) ≙
   PRE i ∈ 1..maxn ∧ v ∈ TYPE THEN arr(i) := v END;
  b,k ⟵ ch_eql_arr(i,j,v) ≙
    PRE i..j ⊆ dom(arr) ∧ i ≤ j ∧ v ∈ TYPE THEN
       IF v ∈ arr[i..j]
       THEN b := TRUE||
              k : (k ∈ i..j ∧ arr(k) = v ∧ ∀l.l ∈ i..(k−1) ⇒ arr(l) ≠ v)
       ELSE b := FALSE||k :∈ TYPE
       END
    END
END
```

B.4.3 Implantation du carnet

Nous pouvons alors écrire l'implémentation de la machine des carnets d'adresse.

```
IMPLEMENTATION CarnetAnniversairesRII
REFINES CarnetAnniversairesRI
IMPORTS noms_Arr(NOM,max),dates_Arr(DATE,max),index_Nvar(max)
SEES Bool_TYPE
INVARIANT
 index = index_nvar
∧noms = 1..index ◁ noms_arr
∧dates = 1..index ◁ dates_arr
OPERATIONS
 r ⟵ ajoutI(n,a) ≙ VAR b,i,j IN
    j ⟵ index_val_nvar;
    IF j ≠ 0 THEN b,i ⟵ noms_ch_eql_arr(1,j,n) END;
    IF j = 0 ∨ b = FALSE THEN
      b ⟵ index_neq_nvar(maxn);
      IF b = TRUE
      THEN noms_sto_arr(j+1,n);dates_sto_arr(j+1,a);
           index_inc_nvar;r := ok
      ELSE r := plein  END
    ELSE r := deja_connu  END
   END;
 r,a ⟵ trouveI(n) ≙
    VAR b,i,j
    IN j ⟵ index_val_nvar;
      IF j ≠ 0 THEN b,i ⟵ noms_ch_eql_arr(1,j,n)
      ELSE b := FALSE;i := maxn END;
      IF b = TRUE THEN r := ok
      ELSE r := inconnu END
      a ⟵ dates_val_arr(i)
    END;
```

L'opération de recherche des fêtes à souhaiter ne peut pas être implantée directement car c'est une opération qui nécessite de calculer un ensemble de valeurs, ce qui n'est pas possible directement en B. Une solution est de stocker une séquence des noms des personnes dont c'est l'anniversaire puis d'avoir une opération de lecture de cette séquence nom par nom. Nous appelons `collect` cette nouvelle opération qui sera spécifiée à l'aide d'une variable abstraite `fetes` et implantée à l'aide d'une séquence `fetes_Seq`. L'invariant nous assure l'égalité des deux objets.

```
collect(jo) ≙
    VAR b, i, j, k
    IN j ⟵ index_val_nvar; fetes_clr_seq; i := 0
      WHILE i ≠ j DO
          i := i+1; (b, i) ⟵ dates_ch_eql_arr(i, j, jo);
          IF b = TRUE THEN k := noms_val_arr(i); fetes_psh_seq(k)
          ELSE i := j
          END
      VARIANT j−i
      INVARIANT i ∈ ℕ ∧ j ∈ ℕ ∧ 0 ≤ i ∧ i ≤ j ∧ j = index
      ∧fetes_seq[1..i] = 1..i ◁ nom[dates⁻¹[{jo}]] ∧ size(fetes_seq) ≤ i
      END
    END;
```

Références

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

[2] Site web de l'atelier B. http://www.atelierb.societe.com.

[3] Site web de B Tool. http://www.b-core.com/btool.html.

[4] J.-F. Monin. *Comprendre les méthodes formelles.* Collection technique et Scientifique des Télécommunications. Masson, 1996.

[5] Site web sur B. http://www-lsr.imag.fr/B.

[6] J. M. Spivey. *Understanding Z. A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.

[7] J.B. Wordsworth. *Software Engineering with B.* Addison-Wesley, 1996.

A Rappels sur la logique de Hoare

A.1 Conditionnelle

La règle pour la conditionnelle en logique de Hoare est :

$$\frac{\{\phi \wedge C\}P\{\psi\} \qquad \{\phi \wedge \neg C\}Q\{\psi\}}{\{\phi\}\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ Q\ \mathsf{end}\{\psi\}}$$

Donner la règle permettant d'établir : $\{\phi\}\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{end}\{\psi\}$

A.2 Affectation

Trouver ϕ tel que $\{\phi\}a := a+b\{a = b\}$

A.3 Échange de valeurs

Soit le programme P suivant d'échange des valeurs des variables x et y :

$$z := x; x := y; y := z$$

1. Écrire une spécification de ce programme exprimant l'échange de valeurs.
2. Prouver la correction de ce programme.

A.4 Boucle

Soit le programme P suivant qui calcule $2n$ dans la variable x :

$$x := 0; y := n; \mathsf{while}\ y > 0\ \mathsf{do}\ y := y-1; x := x+2\ \mathsf{end}$$

1. Écrire la spécification de ce programme.
2. Écrire l'invariant de boucle.
3. Prouver la correction du programme en justifiant si la terminaison est partielle ou totale.

B Langage de la méthode B

B.1 Formules

L'anti-restriction consiste à restreindre le domaine d'une relation R au complémentaire d'un ensemble E. Elle se note en $E \lhd\!\!\!- R$ ($E << |R$ en notation ASCII).
La surcharge droite d'une relation R par une relation S est une relation dont le domaine est l'union des domaines de R et de S. Elle correspond à S sur le domaine de S et à R sur les éléments du domaine de R qui ne sont pas dans le domaine de S. Elle se note $R \ \lhd\!\!- S$ et $R < +S$ en notation ASCII.

1. Donner une formule équivalente à $x \mapsto y \in E \lhd\!\!\!- R$ uniquement en terme d'appartenance à E et R.
2. Donner une formule équivalente à $x \mapsto y \in R \ \lhd\!\!- S$ uniquement en terme d'appartenance à R et S puis en utilisant l'anti-restriction.

B.2 Objets et formules

On souhaite modéliser un système de gestion de cours. Le système comprend des professeurs, des étudiants, des cours. Chaque cours est enseigné par un seul professeur et peut accueillir un nombre maximum d'étudiants. Il ne peut y avoir plus d'étudiants inscrits dans un cours que le nombre maximal. Chaque étudiant est identifié par un numéro unique.
Décrire dans le langage de B la modélisation de ce problème. On prendra comme ensemble abstraits les ensembles `PROFESSEURS`, `ÉTUDIANTS` et `COURS` qui représentent le type des objets que peut manipuler le système. On introduira en particulier les relations `est_suivi_par`, `est_enseigné_par`, `a_pour_professeur`.
Utiliser prioritairement les notations algébriques de B.

B.3 Substitutions

Expliciter la formule $[S]P$ dans les cas suivants :

B.3.1 Substitutions simples

S	P	$[S]P$
$x := 1$	$x \neq y$	
$x := y$	$x = y$	
$x := 0$	$y > 0$	
$x := x{+}1$	$x > 0$	
$x := 1$	$(\forall x.x \in \mathbb{N} \Rightarrow x \geq y) \vee x \geq 0$	

B.3.2 Substitutions multiples

S	P	$[S]P$
$x, y := 1, 2$	$x{+}y \geq z$	
$x, y := 0, 1$	$\forall x.x \in \mathbb{N}. \Rightarrow x \geq y$	
$x, y := y, x$	$\forall x.x \in \mathbb{N}. \Rightarrow x \geq y$	

B.3.3 Substitutions généralisées

S	P	$[S]P$
$x > 0 \vert x := x{-}1$	$x > y$	
$x \in \mathbb{N} \vert xs := xs{\cup}\{x\}$	$xs \subseteq \mathbb{N}$	
$x \in S \Longrightarrow T := T{\cup}\{x\}$ $[]\ x \notin S \Longrightarrow S := S{\cup}\{x\}$	$x \in S{\cap}T$	

B.3.4 Raffinement

On suppose que x et y sont des variables de type $\mathbb{N}$. Soit les substitutions :

$$S_1 \equiv \textsf{PRE } x \neq 0 \textsf{ THEN } y := y/x \textsf{ END} \qquad S_2 \equiv \textsf{IF } x \neq 0 \textsf{ THEN } y := y/x \textsf{ END}$$

1. Pour une formule P quelconque, calculer $[S_1]P$ et $[S_2]P$.
2. A-t-on $S_1 \sqsubseteq S_2$, $S_2 \sqsubseteq S_1$?
3. On introduit maintenant $S_3 \equiv \textsf{PRE } x \neq 0 \textsf{ THEN ANY } z \textsf{ WHERE } z \leq y \textsf{ THEN } y := z$.
   Calculer $[S_3]P$, a-t-on $S_1 \sqsubseteq S_3$, $S_3 \sqsubseteq S_1$?

B.4 Machines

B.4.1 Signalisation

(D'après M. Ducassé & L. Rozé, IRISA)
L'objectif de cet exercice est la modélisation d'un système de feux rouges à un croisement simple de deux routes. La modélisation doit permettre de traiter les conventions de feux de différents pays.
La spécification se fait en considérant d'abord la notion de route. Une route peut avoir plusieurs états : `cross` (les voitures sur la route peuvent traverser le carrefour), `cross_end` (les voitures sur la route ne peuvent plus s'engager dans le carrefour, les voitures déjà dans le carrefour doivent en sortir), `stop` (les voitures sur la route ne sont pas engagées dans le carrefour) `stop_end` (les voitures sur la route

vont bientôt être autorisées à s'engager dans le carrefour) et enfin l'état `off` pour lequel le trafic n'est pas régulé.
Pour modéliser l'état d'une route, on introduit une machine :

$$\begin{array}{|l|}\hline
\textbf{MACHINE}\ \texttt{Route} \\
\textbf{SETS}\ \texttt{STATE} = \{\texttt{cross}, \texttt{cross_end}, \texttt{stop}, \texttt{stop_end}, \texttt{off}\} \\
\textbf{CONSTANTS}\ \texttt{opposé} \\
\textbf{PROPERTIES}\ \texttt{oppposé} \in \texttt{STATE} \rightarrowtail\!\!\!\!\twoheadrightarrow \texttt{STATE} \\
\wedge\ \texttt{oppposé}\ =\ \{\texttt{cross} \mapsto \texttt{stop}, \texttt{cross_end} \mapsto \texttt{stop_end}\ , \\
\qquad\qquad\qquad \texttt{stop} \mapsto \texttt{cross}, \texttt{stop_end}\ \ \mapsto \texttt{cross_end}, \texttt{off} \mapsto \texttt{off}\} \\ \hline
\end{array}$$

La constante `opposé` associe à l'état d'une route, l'état que doit avoir une route qui la croise afin de respecter les règles de sécurité.
Un carrefour de deux routes A et B va être modélisé par deux variables `routeA` et `routeB` représentant chacune l'état de la route correspondante. Un booléen `regulé` indique si le carrefour est régulé. L'invariant assure que les deux routes sont dans des états opposés et que l'état des routes est `off` exactement lorsque le carrefour n'est pas régulé.
Le squelette de la machine de carrefour est donc :

$$\begin{array}{|l|}\hline
\textbf{MACHINE}\ \texttt{Carrefour} \\
\textbf{INCLUDES}\ \texttt{Route} \\
\textbf{VARIABLES}\ \texttt{régulé}, \texttt{routeA}, \texttt{routeB} \\
\textbf{INVARIANT}\ \ldots \\
\textbf{INITIALISATION}\ \ldots \\
\textbf{OPÉRATIONS}\ \texttt{start} \mathrel{\hat{=}} \ldots \\
\textbf{END} \\ \hline
\end{array}$$

1. Compléter la machine de carrefour avec une initialisation en mode non régulé et une opération `start` qui passe le carrefour d'un mode non régulé dans un mode régulé quelconque.
2. Écrire les obligations de preuves qui doivent être vérifiées pour que l'invariant soit établi à l'initialisation et préservé par l'opération.
3. Pour modéliser la suite des changements d'états du carrefour, on introduit de plus dans la machine une constante `suivant` qui fait passer d'un état à l'autre suivant le cycle :

   $$\texttt{cross} \rightarrow \texttt{cross\_end} \rightarrow \texttt{stop} \rightarrow \texttt{stop\_end} \rightarrow \texttt{cross}$$

   Cette fonction n'est pas amenée à être modifiée dans la machine, il n'est donc pas nécessaire de la mettre en variable, par contre elle ne va pas être représentée telle quelle dans l'implantation, qui reposera sur les changements d'états des feux. On l'introduit dans une clause **ABSTRACT_CONSTANTS** qui déclare des constantes qui pourront être raffinées, l'invariant de liaison apparaissant alors dans la clause **PROPERTIES**.
   On écrit donc :

   $$\begin{array}{|l|}\hline
   \textbf{ABSTRACT_CONSTANT}\ \texttt{suivant} \\
   \textbf{PROPERTIES}\ \ldots \\ \hline
   \end{array}$$

   Compléter la clause **PROPERTIES** pour préciser les propriétés de la fonction `suivant` (type et valeurs) et ajouter à la machine une opération `change` qui change l'état des routes en suivant le cycle autorisé.
4. Dans l'objectif de raffiner le carrefour, on introduit une machine correspondant au contrôleur des feux. L'état d'un feu est repéré par un état $\{s_0, s_1, s_2, s_3, s_4\}$ qui correspond à l'état de la route $\{\texttt{off}, \texttt{cross}, \texttt{cross\_end}, \texttt{stop}, \texttt{stop\_end}\}$. Le lien entre les états des feux et des routes est établi

par une constante `convention` qui est une bijection entre l'état d'une route et l'état d'un feu, on introduit également une fonction `feu_suivant` qui définit le cycle de fonctionnement du feu.

Écrire une machine `Contrôleur` correspondant au contrôleur des feux en introduisant les constantes `convention` et `feu_suivant` avec leurs propriétés, deux variables `feuA` et `feuB` correspondant aux deux états des feux des routes A et B, l'initialisation (feu non régulés), une opération `start_feu` qui s'applique à des feux non régulés et prend en argument l'état de la route A qu'il faut réaliser et enfin une opération `feu_suivant`.

5. Écrire une machine qui implante le carrefour en important la machine `Contrôleur`. Expliciter la propriété qui lie la constante `suivant` de la machine `Route` aux constantes `feu_suivant` et `convention` de la machine `Contrôleur` et l'invariant qui lie `routeA` et `routeB` aux variables `feuA` et `feuB`.

6. La machine `Contrôleur` est alors raffinée en une machine `Contrôleur2` pour faire intervenir la couleur du feu. L'ensemble `COULEUR` des couleurs est {`vert`, `orange`, `rouge`, `orange_clignotant`, `noir`, `rouge_et_jaune`} Une fonction `couleur` est introduite qui associe à chaque état du feu une couleur (la fonction dépend du code de la route des différents pays). Dans cette machine on ajoute aux variables `feuA` et `feuB` les variables `couleurA` et `couleurB`.

   (a) Que peut-on dire des propriétés de la fonction `couleur` (partielle, totale, injective, surjective) ?

   (b) Quel invariant lie les variables ?

7. On se donne les machines suivantes pour représenter l'état et la couleur d'un feu :

MACHINE `Etat_feu`
SETS `ETATFEU` $= \{s_0, s_1, s_2, s_3, s_4\}$
VARIABLES `etat`
INVARIANT `etat` $\in$ `ETATFEU`
INITIALISATION `etat` $:= s_0$
OPERATIONS
`set`$(v) \hat{=}$ PRE $v \in$ `ETATFEU` THEN `etat` $:= v$ END;
$r \longleftarrow$ `valeur` $\hat{=}$ BEGIN $r :=$ `etat` END
END

MACHINE `Couleur_feu`
SETS
`COULEUR` = {`vert`, `orange`, `rouge`, `orange_clignotant`, `noir`, `rouge_et_jaune`}
VARIABLES `couleur`
INVARIANT `couleur` $\in$ `COULEUR`
INITIALISATION `couleur` := `noir`
OPERATIONS
`set`$(v) \hat{=}$ PRE $v \in$ `COULEUR` THEN `couleur` $:= v$ END;
$r \longleftarrow$ `valeur` $\hat{=}$ BEGIN $r :=$ `couleur` END
END

Pour réaliser un croisement français, on implante le contrôleur de feux (raffiné) à l'aide d'une variable d'état et de deux variables de couleur. Le squelette de la machine s'écrit :

IMPLEMENTATION `Controleur_francais`
REFINES `Controleur2`
INCLUDES `Route`
IMPORTS `Etat_feu`, A.`Couleur_feu`, B.`Couleur_feu`
VALUES
$\texttt{convention} = \{\texttt{cross} \mapsto s_1, \texttt{cross\_end} \mapsto s_2, \texttt{stop} \mapsto s_3, \texttt{stop\_end} \mapsto s_4, \texttt{off} \mapsto s_0\};$
$\texttt{feu\_suivant} = \{s_1 \mapsto s_2, s_2 \mapsto s_3, s_3 \mapsto s_4, s_4 \mapsto s_1, s_0 \mapsto s_0\};$
$\texttt{color} = \{s_1 \mapsto \texttt{vert}, s_2 \mapsto \texttt{orange}, s_3 \mapsto \texttt{rouge}, s_4 \mapsto \texttt{rouge}, s_0 \mapsto \texttt{orange\_clignotant}\}$
INVARIANT ...
INITIALISATION ...
...
END

Compléter l'initialisation et l'invariant. Écrire la propriété qui doit être vérifiée pour que l'initialisation respecte l'invariant.

8. La machine `Couleur_feu` peut elle-même être implantée en utilisant trois instances d'une machine `Lampe` modélisant l'état d'une lampe qui peut être allumée, éteinte ou clignotante.
   Écrire la spécification de la machine `Lampe` ainsi qu'une implantation de la machine `Couleur_feu` utilisant trois lampes.
9. Dessiner le graphe de dépendance entre les machines de ce système en indiquant les liens de raffinement, d'inclusion et d'importation.