Atelier B

Langage B

Manuel Utilisateur

version 1.2

ATELIER B
Langage B Manuel Utilisateur
version 1.2

Document établi par CLEARSY.

CLEARSY
Maintenance ATELIER B
Europarc de PICHAURY
1330 Av. J.R. Guilibert Gauthier de la Lauzière - Bât C2
13856 Aix-en-Provence Cedex 3
France

Tél 33 (0)4 42 37 12 99
Fax 33 (0)4 42 37 12 71
email : maintenance.atelierb@clearsy.com

Table des matières

Table des figures

Chapitre 1

Introduction

Le langage et la méthode B fournissent un moyen de produire des logiciels ou des systèmes prouvés mathématiquement, ce qui permet de garantir que le système produit répond au besoin. L'emploi de B implique donc l'utilisation de raisonnements mathématiques rigoureux. En dehors de ceci, B n'impose pas la manière de conduire le développement d'un système, pas plus que l'utilisation de C++ ou ADA imposent la manière de spécifier et d'analyser.

Nous pensons qu'il est nécessaire de recueillir les solutions connues aux divers problèmes d'utilisation de B, de recueillir les divers styles d'emploi et modes d'usage, pour que chaque équipe de développement puisse définir *sa* méthode de conduite adaptée à son projet. C'est le but de cet ouvrage. Il ne s'agit donc pas ici de décrire une méthode de conduite de projet couvrant tout le cycle de développement du logiciel. Nous nous contenterons d'examiner un certain nombre de problèmes typiques avec leurs solutions possibles.

Pour lire cet ouvrage avec profit, il est nécessaire de connaître les principes du langage B (machine abstraite, raffinement, implantation, constructions mathématiques principales) et d'avoir déjà modélisé en B pour comprendre les problèmes qui sont abordés. Cela correspond à la situation de ceux qui viennent de se former à B et qui démarrent un projet.

Chapitre 2

Questions fréquemment posées

Nous avons essayé durant toute la rédaction de ce manuel d'imaginer dans quelle situation chaque élément décrit est utile. Nous en avons déduit une liste de questions, que nous exposons ci-après avec les renvois aux parties du manuel qui y répondent. Les questions que nous avons sélectionnées ne correspondent pas à des titres de chapitres ou de paragraphes. En pratique, cela signifie que vous devez vérifier que votre question ne figure pas dans la table des matières du document avant d'utiliser cette liste de questions.

2.1 Catégorie Entrées / sorties

- "Comment puis-je atteindre les périphériques de ma machine cible à partir du programme B ?" : voir section 7.2, page 85.
- "Est-il possible de s'interfacer sur des librairies système à partir d'un programme B ?" : voir section 7.2, page 85.
- "Comment un programme B peut-il réutiliser un programme existant non développé en B ?" : voir section 7.2, page 85.
- "Une machine de base peut-elle être instanciée ?" : voir section 7.2, page 85.
- "Comment faire une maquette qui fasse des `printf` en B ?" : voir section 7.4, page 88.
- "Comment valider une machine de base ? Les machines de base sont-elles sécuritaires ?" : voir section 7.6, page 89.
- "La preuve garantit-elle que les ressources systèmes sont bien utilisées ?" : voir section 7.6.1, page 90.
- "Faut-il écrire tout le code des machines de base ? Quelles conventions de programmation utiliser ?" : voir section 7.6.2, page 92.

2.2 Catégorie Implantation / Programmation

- "Où s'arrête l'arbre d'implantation d'un projet B ?" : voir section 7.3, page 86.
- "En B0, les tableaux dont la taille est spécifiée par paramètre de la machine sont refusés. Comment faire ?" : voir section 7.4, page 88.
- "Comment créer des sous procédures en cours de programmation ?" : voir section 4.4, page 38.
- "Y a t-il un mécanisme d'exceptions en B ?" : voir section 3.4.1, page 17.

– “Peut-on mettre un système de traces d’erreurs dans un programme fait avec B ?” : voir section 3.4.1, page 17.
– “Pourquoi n’y a t-il pas de type flottant en B comme dans les autres langages de programmation ?” : voir section 9.1, page 106.
– “Un programme classique peut-il être traduit tel quel en B0 ? Pourra t-on facilement le prouver par rapport à une spécification ajoutée ?” : voir section 11, page 129.
– “Le B0 checker refuse les tableaux dont la taille est un paramètre formel de la machine. Comment réaliser des modules paramétrables ?” : voir section 12.4, page 140.

2.3 Catégorie Modélisation / Principes

– “Comment faire un tri ou une recherche dans un tableau en B ?” : voir section 7.5, page 89.
– “Peut-on définir des ensembles correspondants à des types différents de données numériques (vitesse, distance, etc.) ?” : voir section 8.4, page 102.
– “Peut-on prouver des propriétés générales sur des spécifications très compliquées ?” : voir section 4.2, page 28.
– “En dehors de tout découpage de programme, comment peut-on décomposer une spécification ?” : voir section 4.2, page 28.
– “Quelles sont les phases d’un projet B ? Y a t-il une méthode générale de développement ?” : voir section 3.1, page 7.
– “La réexpression du besoin est-elle nécessaire ? qui doit la faire ?” : voir section 3.1, page 7.
– “Comment dégager les propriétés essentielles du produit à construire ?” : voir section 3.3, page 9.
– “Certaines variables ont des propriétés dans le contexte du logiciel qui ne sont pas démontrables quand on les met dans le composant qui définit la variable. Est-ce normal ?” : voir section 3.3.3, page 13.
– “À quoi servent les préconditions ?” : voir section 3.4.4, page 19.
– “Quel sens donne t-on à la bonne correspondance entre un programme et sa spécification en B ?” : voir section 5, page 49.

2.4 Catégorie Preuve

– “Est-il normal d’avoir `skip` partout dans les machines de base ?” : voir section 7.6.1, page 90.
– “Peut-on déclarer librement des variables dans une machine de base ? Peut-on faire en sorte que ces variables soient lisibles comme des **CONCRETE_VARIABLES** ?” : voir section 7.6.3, page 93.
– “Que faire quand les PO ont un trop grand nombre d’hypothèses ?” : voir section 8.1, page 97.
– “Il y a des obligations de preuves qui ne sont pas démontrables car la valeur des constantes n’est pas indiquée en hypothèse. Que faire ?” : voir section 8.2.1, page 98.
– “Comment se fait-il qu’il faille écrire les **VARIANT**/**INVARIANT** des boucles pour les démontrer alors que ce qu’elles font peut se déduire de leur codage ?” : voir section 6.1, page 63. Et aussi : “Si la construction automatique des invariants de boucle n’est pas

possible, comment mettre à profit l'outil de preuve pour cette tâche ?" : voir section 6.2.3, page 71.
- "Une boucle juste peut-elle être démontrée sans son invariant ?" : voir section 6.1, page 63.
- "Dans la preuve d'une boucle, il y a des variables dans le but sur lesquels il n'y a aucune hypothèse. Que faire ?" : voir section 6.2, page 64.
- "Comment référencer des états initiaux dans un invariant de boucle ?" : voir section 6.5, page 78.
- "Y a t-il des astuces pour simplifier la preuve d'une boucle après construction du variant et de l'invariant ?" : voir section 6.2.4, page 71.
- "Il y a des hypothèses manquantes dans les PO de raffinement et d'implantation. Que faire ?" : voir section 5.2, page 54.
- "Des composants de tests, avec des spécifications vides, utilisant `BASIC_IO` pour imprimer des messages produisent trop de PO. Que faire ?" : voir section 11.3, page 134.
- "Peut-on utiliser des séquences complexes de **IF** en B0 ? Imbriquer des traitements complexes ?" : voir section 11, page 129.
- "Suffit-il d'écrire du B juste pour pouvoir faire la preuve ?" : voir section 3.5.2, page 23.

2.5 Catégorie Architecture

- "Quel est l'équivalent de `#include` ?" : voir section 8, page 97.
- "Comment définir des entités connues de tout un projet ?" : voir section 8, page 97.
- "Pourquoi y il plusieurs machines de contexte séparées ?" : voir section 8.1, page 97.
- "Où importer les machines de contexte ?" : voir section 8.3, page 100.
- "un **SEES** voit il une machine incluse ou importée ?" : voir section 3, page 28.
- "Est-ce une bonne idée d'utiliser **INCLUDES** pour construire graduellement une spécification ?" : voir section 4.2.1, page 31.
- "Avec un découpage par **IMPORTS**, au plus haut niveau il n'y a pas tous les détails. Comment faire ?" : voir section 4.2, page 28.
- "Comment réaliser les parties incluses d'une spécification ?" : voir section 4.3, page 32.
- "D'où viennent toutes les règles spéciales concernant le **SEES** ?" : voir section 4.5, page 42
- "À quoi sert le bouton *Project Check* ?" : voir section 4.5, page 42.
- "Pourquoi le découpage d'un projet B est-il considéré comme si important ?" : voir section 3.5, page 22.
- "Que deviennent les variables concrètes au raffinement ou à l'implantation ?" : voir section 5.1, page 50.
- "Que deviennent les paramètres de retour des opérations au raffinement ?" : voir section 5.4, page 58.

Chapitre 3

Notions et conseils généraux

réexpression Dans ce chapitre, nous allons présenter quelques notions fondamentales, accompagnées de conseils pour la mise en œuvre de la méthode B. Nous allons ainsi examiner les notions de *modélisation mathématique*, de *spécification formelle*, de *programmation offensive*, ...Tout ceci permet de situer la méthode B et comment elle intervient dans un projet.

3.1 Les projets B

Commençons par donner le schéma général d'un projet utilisant B, avec ses différentes étapes. En particulier nous allons introduire la notion de spécification formelle.

La méthode B sert à produire les logiciels ou des systèmes (matériel + logiciel) prouvés. Elle est constituée du langage B, notation mathématique et structurée qui permet d'exprimer les spécifications et le programme, et des règles qui définissent ce qu'est la preuve d'un tel projet B. La manière de conduire l'analyse du produit à réaliser, l'organisation des composants, etc. n'est pas imposée par B qui se situe dans une optique plus large. Néanmoins, dans tout projet B les caractéristiques même du langage B font se dégager les étapes suivantes :

L'analyse du système à construire : avant toute modélisation. Il n'est pas possible de démarrer un projet B sans avoir une compréhension sûre de ce qu'il faut obtenir en fin de compte.

La réexpression du besoin : une fois le problème bien compris, il est nécessaire de produire une nouvelle formulation du besoin initial, structurée avec la rigueur qu'exige l'usage d'une méthode formelle. Attention, il n'y a pas encore de formulation mathématique à ce niveau, mais simplement une réécriture non ambiguë. Ce n'est pas un exercice facile : en fait généralement seules les personnes qui doivent faire la modélisation mathématique par la suite ont la motivation nécessaire, car ils savent l'importance que cette phase aura pour leur prochain travail.

La modélisation mathématique : cette phase sert à produire la spécification en B du système. Il y a plusieurs méthodes :

- Formulation "à plat" des exigences les unes après les autres, puis répartition dans une structure B. La difficulté est de trouver des formulations qui ne seront pas incompatibles lors de leur structuration en B.

– Formulation en partant des exigences essentielles et en introduisant les détails par raffinements / importations successifs. La difficulté est alors d'orienter judicieusement ce parcours pour obtenir simplement la spécification complète.

C'est cette spécification B finalement obtenue qui constitue la *spécification formelle* à partir de laquelle le produit sera réalisé.

La conception et la réalisation du produit : accompagnés par la preuve de l'ensemble. Le processus de réalisation est terminé quand tous les détails de conception sont introduits, arrivant ainsi à un programme écrit dans le sous-ensemble implantable du langage B, le B0. L'Atelier B permet alors une traduction directe dans un langage de programmation traditionnel.

Par rapport au cycle classique d'un développement logiciel, les phases d'analyse amont sont plus importantes car la nécessité de formaliser le besoin initial augmente la durée de ces phases. Cette augmentation est globalement bénéfique car les erreurs étant découvertes très tôt, elles n'ont pas les conséquences graves causées par une découverte tardive.

Notons que les phases de spécification, conception et réalisation *semblent* moins séparées que dans un développement classique, car le langage employé est toujours le même : l'opérateur fait ces phases en écrivant des composants B. Les spécifications B s'arrêtent quand tout le document informel a été reporté, la conception formelle s'arrête au niveau au delà duquel l'opérateur estime qu'apparaissent les détails d'implantation.

Comme nous l'avons déjà dit, B ne restreint pas les méthodes qui peuvent être employées dans ces différentes phases. Par contre, nous allons présenter un certain nombre de points de repère concernant la modélisation, le style de programmation offensive qui convient à B et l'organisation du projet B de telle manière à ne pas avoir de problèmes de preuve.

3.2 Terminologie

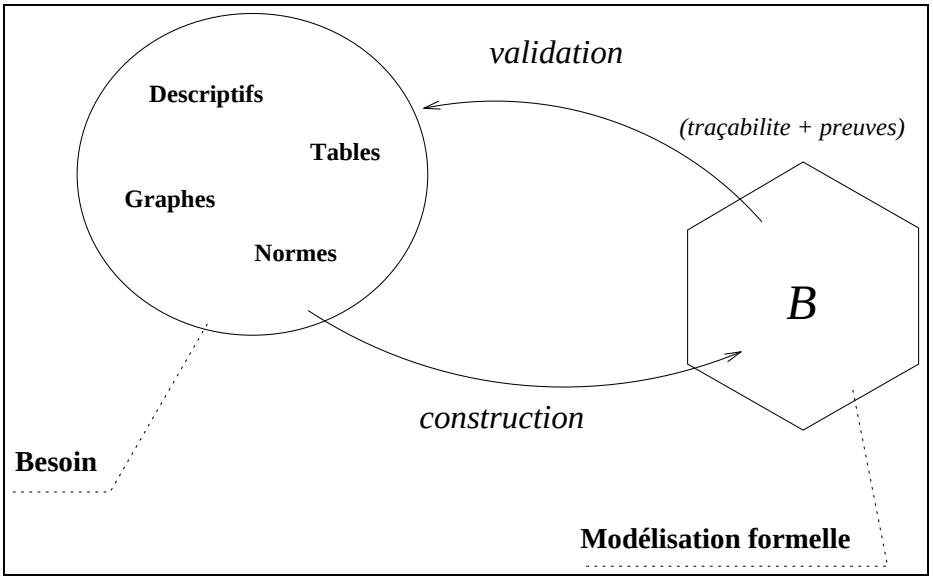
Après avoir décrit le schéma général de l'utilisation de B, nous allons maintenant préciser un certain nombre d'expressions qui sont couramment employées.

Projet B : un projet B est l'ensemble des activités utilisant B qui partant d'un besoin, aboutit au système qui satisfait ce besoin. Cette terminologie peut être employée pour des systèmes entièrement développés en B aussi bien que pour ceux pour lesquels certaines parties ont été faites par des méthodes traditionnelles.

Développement B : idem projet B.

Modèle B : un modèle B est une description mathématique qui représente certaines entités réelles du système à construire ou de son contexte. Dans un projet B, il y a souvent plusieurs modélisations mathématiques, soit pour représenter divers aspects du système, soit pour représenter un aspect à différents niveaux de précision. Les modèles B sont écrits dans des composants B (machines abstraites, raffinements), mais ils constituent des entités indépendantes. Il est d'ailleurs fréquent d'écrire des modèles B sous forme d'un ensemble de prédicats mathématiques avant de se préoccuper de leur répartition dans les machines abstraites. Les phases de réexpression du besoin et de modélisation dont nous parlons plus haut ont pour but la production de ces modèles.

Modèle abstrait : dans notre cadre, idem modèle B.



Modélisation B : activité de création de modèles B.

Modélisation mathématique : dans le cadre d'un projet B, idem modélisation B. Cette terminologie insiste sur le fait que B est basé sur la théorie des ensembles, classique en mathématiques.

Modélisation des propriétés du système : idem modélisation B, avec en plus une évocation de l'approche "par propriété" préconisée dans la méthode B pour l'analyse du système.

Spécification B : la spécification B est l'ensemble des composants B et de leurs liaisons qui constituent l'équivalent formel du besoin exprimé initialement. Tout ce qui est de la conception ne fait pas partie de la spécification B.

3.3 Modélisation des propriétés du système

Un développement B débute par la *construction d'un modèle*, abstrait, qui est une spécification de ce que devra réaliser le composant logiciel.

La figure illustre ce procédé de modélisation. Le besoin est généralement exprimé sous forme de descriptifs informels , de tables, de normes et de graphes. Le travail global de spécification formelle, qui comprend la phase de réexpression du besoin et la phase de formalisation proprement dite a pour but l'obtention d'une modélisation formelle à partir du besoin. Cette modélisation doit faire l'objet d'une validation par rapport à l'expression initiale du besoin, d'où l'importance de la traçabilité entre ces éléments.

3.3.1 Étude de cas

Afin d'illustrer notre propos, dans la suite de cette section nous traitons un exemple simple de cahier des charges d'un logiciel de commutation de batteries.

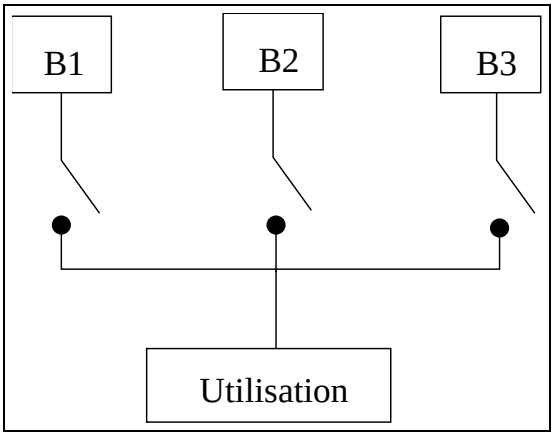


FIG. 3.1 – Schéma du commutateur

Présentation du cahier des charges

Généralement, le cahier des charges[1] d'un composant logiciel contient des schémas, des descriptions en français de son environnement et de sa mission, des tables logiques, des automates, etc.

Nous donnons ci-dessous l'exemple du cahier des charges d'un commutateur de batteries extrêmement simplifié.

> Un appareil est alimenté par trois batteries suivant le schéma 3.1.
>
> Les trois interrupteurs sont télécommandés par le système. Le logiciel *commutateur*, appelé à intervalles réguliers, commutera d'une batterie sur l'autre pour répartir la charge et éviter ainsi qu'une même batterie ne débite trop longtemps. Le système connaît en entrée l'état des trois interrupteurs, il doit calculer le nouvel état à leur donner.

Analyse du cahier des charges

Ce cahier des charges relativement court n'est cependant pas dépourvu de complexité. Afin d'illustrer l'étape d'analyse préliminaire préalable à la construction du modèle B nous présentons ci-dessous quels sont les résultats d'une telle analyse préliminaire :

> L'état de chaque interrupteur est représenté dans le modèle B par une variable d'état de type booléen.
>
> Les trois propriétés essentielles que ce système doit avoir sont dans l'ordre d'importance :
>
> P1 *pas de court-circuit :* il ne doit jamais y avoir deux interrupteurs fermés ensembles, car les batteries ne délivrent jamais exactement la même tension, donc la plus chargée se déchargerait dans les autres.
>
> P2 *continuité de l'alimentation :* les trois interrupteurs ne doivent pas être tous ouverts ensembles.

[1] On parle également de *spécification* ou de STBL (pour Spécification Technique du Besoin Logiciel).

P3 *changement de batterie :* la batterie qui alimente dans le nouvel état doit être différente de celle qui alimentait dans l'état précédent. [2]

Exigence fonctionnelle : le programme devra posséder une opération calculant un nouvel état des booléens représentant les positions des interrupteurs, de telle manière
EF1 qu'il n'y ait jamais deux interrupteurs à VRAI en sortie simultanément,
EF2 qu'il y ait toujours au moins un interrupteur à VRAI en sortie,
EF3 et que si un interrupteur est mis à VRAI il n'était pas à VRAI auparavant.

Cette analyse du cahier des charges conduit à la production d'un document constituant la réexpression plus détaillée du besoin. Elle correspond à la phase de réexpression présentée plus haut. Notons qu'au niveau de cette analyse certains choix de représentation sont faits, comme le choix d'employer des booléens pour représenter les interrupteurs. Bien que ces choix puissent sembler être de la conception, il est difficile de les éviter : ils sont nécessaires pour formaliser le problème.

En principe, ces choix devraient rester indépendants de toute considération de conception, c'est-à-dire que sur cet exemple nous devrions pouvoir réaliser un logiciel dans lequel l'état des batteries soit représenté par autre chose de des variables booléennes. En fait ce n'est pas le cas, il n'y a pas une telle indépendance.

Formalisation

Le cahier des charges, une fois analysé peut être formalisé en un ensemble d'expressions qui seront, dans un deuxième temps, insérées dans un modèle B. Afin de formaliser les exigences précédentes il est nécessaire de choisir un ensemble de variables :

Soit B1 un booléen représentant l'état du premier interrupteur. Lorsque B1 est égal à TRUE cela signifie que le courant passe (l'interrupteur est fermé). Nous introduisons de même les variables booléennes B2 et B3.

Ensuite, il est nécessaire d'identifier et de nommer les services (ou opérations) du système. Ici nous n'aurons qu'une seule opération chargée d'effectuer la commutation entre batteries, appelée « commute ». Cette opération représentera l'évolution de l'état du système, c'est-à-dire que tout changement se fait à l'occasion de l'appel de cette opération. Nous supposerons qu'elle est appelée à intervalles fixes, toutes les secondes par exemple.

Les choix de formalisation que nous faisons à ce niveau devraient *en principe* être indépendants des choix d'implantation. En réalité ce n'est pas le cas : le découpage en opérations, ainsi que les paramètres d'entrée et de sortie se retrouvent dans le programme final. Le choix n'est donc pas dicté uniquement par les seules considérations de modélisation.

Propriétés en invariant

Les propriétés devant être assurées *en permanence* par le système sont modélisées dans les invariants B (cf. § 3.3.2).

[2] Il est clair que le programme sera meilleur s'il fait en sorte que la charge soit également répartie entre les trois batteries (notion d'*équité*), ce qui pourrait constituer un prérequis supplémentaire. Nous allons décider assez arbitrairement que cette dernière propriété, quoique souhaitable, ne fait pas partie des prérequis essentiels (elle n'est pas spécifiée dans le cahier des charges).

Dans l'exemple du commutateur de batteries, les propriétés P1 et P2 doivent être maintenues à tout moment. L'expression logique de ces deux propriétés est effectuée dans l'invariant :

```
INVARIANT
    /* propriété P1 : pas de court-circuit */
    ¬ (B1=TRUE ∧ B2=TRUE) ∧
    ¬ (B1=TRUE ∧ B3=TRUE) ∧
    ¬ (B2=TRUE ∧ B3=TRUE) ∧

    /* propriété P2 : continuité de l'alimentation */
    ¬ (B1=FALSE ∧ B2=FALSE ∧ B3=FALSE)
```

Propriétés en postcondition

Les propriétés devant être établies par un service donné sont des Postconditions (cf. § 3.3.2). Ces propriétés ne sont pas nécessairement vraies avant l'appel au service, mais le développement B doit garantir qu'à la suite de cet appel les propriétés sont établies.

Dans l'exemple du commutateur de batteries, la propriété P3 ne peut être exprimée dans l'invariant puisqu'elle fait intervenir l'état des batteries avant et après l'opération de commutation c'est pourquoi P3 est modélisée en postcondition. EF1, EF2 et EF3 sont des propriétés d'établissement de l'invariant par l'opération, c'est pourquoi elles sont également des post-conditions.

```
OPERATIONS
    commute =
        BEGIN
            ANY nB1,nB2,nB3 WHERE
                /* propriété P3 = EF3 */
                (B1=TRUE ⇒ nB1=FALSE) ∧
                (B2=TRUE ⇒ nB2=FALSE) ∧
                (B3=TRUE ⇒ nB3=FALSE) ∧

                /* propriété EF1 pour assurer P1 en sortie */
                ¬ (nB1=TRUE ∧ nB2=TRUE) ∧
                ¬ (nB1=TRUE ∧ nB3=TRUE) ∧
                ¬ (nB2=TRUE ∧ nB3=TRUE) ∧

                /* propriété EF2 pour assurer P2 en sortie */
                (nB1=TRUE ∨ nB2=TRUE ∨ nB3=TRUE)
            THEN
                /* affectation de la nouvelle valeur pour chaque interrupteur */
                B1,B2,B3 := nB1,nB2,nB3
            END
        END
```

3.3.2 Propriétés statiques et dynamiques

Suivant les deux types de propriétés mises en évidence dans l'étude de cas nous distinguons : *les propriétés statiques* et *les propriétés dynamiques* :
- les *propriétés statiques* représentent le cadre *global* du composant et doivent apparaître en invariant.
- les *propriétés dynamiques* caractérisent la loi d'évolution des données pour un service ; elles mettent donc en corrélation, les anciennes et nouvelles valeurs des données pour chaque état atteint. Ces propriétés sont *locales* à une opération et sont formalisées dans des post-conditions.

Il est nécessaire d'introduire très tôt ces invariants et ces post-conditions afin de détecter rapidement des inconsistances ou des erreurs de spécification, a priori dès l'isolation de ces propriétés pendant la phase de réexpression du besoin. Dans les paragraphes suivants nous donnons des exemples plus développés de modélisation de propriétés statiques et dynamiques.

3.3.3 Modélisation des propriétés statiques

Les propriétés statiques des données sont exprimées dans les invariants. Nous distinguons deux cas :
- Lorsque la propriété statique à exprimer fait intervenir des variables définies dans un composant B donné et qui sont modifiées par les mêmes opérations. Elle doit être introduite au niveau de la clause `INVARIANT` de ce composant. Ainsi, l'initialisation et les opérations de ce composant doivent respecter cette propriété.
- Lorsque la propriété statique à exprimer lie des variables provenant de composants B distincts ou affectées par des opérations différentes. Elle doit apparaître dans la clause `INVARIANT` du composant liant ces modules B entre eux.

Les propriétés statiques ne posent pas d'autres problèmes en principe que celui de l'expression mathématique. Néanmoins il arrive fréquemment que l'opérateur puisse être surpris par un phénomène particulier : la disparition des propriétés globales à partir d'un certain niveau. En effet, certaines propriétés sont réalisées par une combinaison d'éléments du système. On peut donc les exiger dans les machines abstraites de regroupement de ces éléments, mais pas dans la spécification des éléments séparés. Nous allons illustrer ceci sur un exemple.

Soit un système informatique traitant la donnée de niveau rendue par un capteur de niveau d'eau dans une chaudière. Le système est sensé calculer les bornes de l'intervalle de niveau dans lequel se situe le niveau d'eau réel, en fonction d'un certain nombre d'imprécisions connues. Nommons :

MesureBasse : la borne minimale calculée,

MesureHaute : la borne maximale.

Au plus haut niveau, nous allons exiger que la borne minimale `MesureBasse` soit plus petite que la borne maximale `MesureHaute`. Cette propriété ne pourra pas s'exprimer dans les fonctions plus bas qui calculent indépendamment chaque mesure, puisqu'elle n'est vraie qu'une fois les deux mesures calculées.

La propriété `MesureBasse < MesureHaute`, introduite dans l'exemple ci-dessous, est vérifiée par la machine abstraite `EstimeNiveau` ; Puisqu'elle lie deux variables de ce composant qui sont modifiées dans la même opération, elle apparaît dans la clause `INVARIANT`.

L'opération `CalculerValeurs` est réalisée dans l'implantation nommée `EstimeNiveau_imp`, par l'appel aux opérations `CalculerMB`[3] et `CalculerMH`[4] du composant importé `ServicesMesure`. Les quantités `MesureBasse` et `MesureHaute` sont évaluées dans des opérations distinctes ; il n'est donc pas possible de lier ces variables par un invariant dans le composant `ServicesMesure`.

La propriété `MesureBasse < MesureHaute` est satisfaite si l'opération `CalculerValeurs` est correctement implantée ; les opérations `CalculerMB` et `CalculerMH` doivent être appelées suivant un ordre particulier (la variable `MesureHaute` doit être évaluée avant `MesureBasse`). Dans le cas contraire, la propriété de spécification `MesureBasse < MesureHaute` n'est pas respectée par l'implantation `EstimeNiveau_imp`.

La propriété est donc satisfaite par construction du modèle, mais elle ne peut s'exprimer qu'au plus haut niveau : dans l'invariant de la machine EstimeNiveau.

Présentons tout d'abord la machine abstraite `EstimeNiveau`, dans laquelle la propriété liant les deux mesures peut être exprimée :

MACHINE
EstimeNiveau
CONCRETE_VARIABLES
MesureBasse, MesureHaute
INVARIANT
MesureBasse $\in \mathbb{N} \wedge$
MesureHaute $\in \mathbb{N}_1 \wedge$
/* Propriété statique à respecter */
MesureBasse $<$ MesureHaute
INITIALISATION
MesureHaute := 1 ||
MesureBasse := 0
OPERATIONS
CalculerValeurs =
ANY mb, mh **WHERE**
mh $\in \mathbb{N}_1 \wedge$
mb $\in \mathbb{N} \wedge$
mb $<$ mh
THEN
MesureBasse := mb ||
MesureHaute := mh
END
END

L'implantation de cette machine utilise les opérations du composant `EstimeNiveau`. Il doit appeler les opérations dans le bon ordre pour que le lien entre les deux mesures puisse être respecté :

[3] Opération de calcul de la nouvelle valeur de la variable `MesureBasse`.
[4] Opération de calcul de la nouvelle valeur de la variable `MesureHaute`.

```
IMPLEMENTATION
    EstimeNiveau_imp
REFINES
    EstimeNiveau
IMPORTS
    ServicesMesure
OPERATIONS
    CalculerValeurs =
    BEGIN
        CalculerMH ;
        CalculerMB
    END
END
```

Dans la machine `ServicesMesure` qui permet la mise à jour *séparée* des mesures, la propriété liant ces deux mesures n'est pas toujours vraie car l'ordre dans lequel ces opérations seront employées n'est pas présumé. Ainsi, l'opération `CalculerMH` pourrait être appelée *après* `CalculerMB` et produire une valeur de la mesure haute inférieure à la mesure basse.

```
MACHINE
    ServicesMesure
CONCRETE_VARIABLES
    MesureBasse, MesureHaute
INVARIANT
    MesureBasse ∈ ℕ∧
    MesureHaute ∈ ℕ1∧
    /*********************************
    Propriété statique ne pouvant être
    vérifiée dans ce composant
    *********************************/
    /* MesureBasse < MesureHaute */
INITIALISATION
    MesureHaute := 1 ||
    MesureBasse := 0
OPERATIONS
    CalculerMH =
    BEGIN
        MesureHaute :∈ ℕ1
    END
    ;
    CalculerMB =
    BEGIN
        MesureBasse := MesureHaute - 1
    END
END
```

Dans cet exemple, la propriété statique est une propriété *globale* apparaissant dans la clause `INVARIANT` d'une machine abstraite. Elle est vérifiée à la fois par la machine abstraite (l'initialisation et les opérations de la machine satisfont l'invariant) et par son implantation

(grâce à la construction obtenue par le lien IMPORTS, et l'appel en séquence des deux opérations). Un séquencement inverse sera inévitablement sanctionné par l'échec de la démonstration de certaines obligations de preuve.

Il arrive fréquemment que les propriétés statiques sur les variables apparaissent ainsi uniquement à un certain niveau, et non pas sur les composants séparés. Ceci peut surprendre le concepteur qui ne considère ces composants séparés que dans le contexte où il va les employer. Pourtant, en écrivant une machine B pour chaque composant il a implicitement indiqué toutes les conditions d'emploi de ces composants, leur donnant un caractère naturellement réutilisable qu'il doit prendre en compte.

3.3.4 Modélisation des propriétés dynamiques

Les propriétés dynamiques sont introduites dans les post-conditions d'opération ; elles ne peuvent pas être exprimées dans des clauses `INVARIANT`. Ces propriétés sont satisfaites après déroulement du corps de l'opération.

L'exemple suivant (composant `DynamicProperty`) gère une structure de données de type *fifo* (variable `fifo`). Celle-ci mémorise, grâce à l'opération `UpdateFifo` l'historique des trois dernières commandes `ON,OFF` de l'ensemble `CMD`.

MACHINE
DynamicProperty
ABSTRACT_VARIABLES
fifo, cmd
INVARIANT
fifo $\in \{T\_0, T\_MINUS\_1, T\_MINUS\_2\} \rightarrow \{ON, OFF\} \wedge$
cmd $\in \{ON, OFF\}$
INITIALISATION
fifo $:= \varnothing ||$
cmd $:= OFF$
OPERATIONS
SetCmd(cc) = **PRE** cc $\in \{ON, OFF\}$ **THEN**
cmd := cc
END ;
UpdateFifo =
ANY local_fifo **WHERE**
local_fifo $\in \{T\_0, T\_MINUS\_1, T\_MINUS\_2\} \rightarrow \{ON, OFF\} \wedge$
local_fifo[{T_MINUS_2}] = fifo[{T_MINUS_1}] $\wedge$
local_fifo[{T_MINUS_1}] = fifo[{T_0}] $\wedge$
local_fifo[{T_0}] = cmd
THEN
fifo := local_fifo
END
END

Notons simplement que la postcondition est un prédicat qui devra être satisfait après le déroulement du corps de l'opération.

3.3.5 Conclusion

Nous avons étudié dans cette section les différents types de propriétés modélisées en B. Nous avons ainsi distingué des propriétés statiques et des propriétés dynamiques. Notons qu'il existe également des propriétés d'ordonnancement temporel qui sont modélisées en invariant ou en post-conditions. Nous étudions ce type de propriétés dans la section 10.

3.4 La programmation offensive et la méthode B

3.4.1 Introduction

Afin d'atteindre un niveau de confiance acceptable dans un système logiciel trois stratégies complémentaires sont généralement reconnues et utilisées (voir par exemple "Software Engineering" par Ian Sommerville) :

1. *Éviter les fautes*
   Cette stratégie est la plus intéressante et elle est applicable à tout type de système : les processus de conception et d'implantation doivent être organisés avec l'objectif de produire des systèmes « zéro-défaut ».
2. *Tolérer les fautes*
   Cette stratégie suppose qu'il reste des pannes résiduelles dans le système. Des fonctions sont ajoutées au logiciel afin de continuer lorsque ces fautes causent des pannes du système.
3. *Détecter les fautes*
   Les fautes sont détectées avant que le logiciel soit utilisé de façon opérationnelle. Le processus de validation du logiciel utilise des techniques statiques et dynamiques pour découvrir toute faute résiduelle après l'implantation du système.

Dans le cadre du développement logiciel, la méthode B répond à la première de ces stratégies de façon très pertinente. En effet, en utilisant le langage B il est possible de prouver que le logiciel implanté est exempt de fautes *par rapport à sa spécification*. Aucune faute résiduelle ne peut donc rester dans le logiciel puisque la correction est prouvée. Bien sûr la spécification peut ne pas refléter exactement le besoin du client, c'est pourquoi le « zéro-défaut » ne signifie pas que le logiciel répond au besoin du client.

Cette capacité à produire des logiciels exempts de défauts peut avoir un impact sur le style de programmation. Nous allons maintenant étudier les différents styles de programmation qui peuvent être employés pour prendre en compte les cas d'erreur.

Comme nous allons le voir, la garantie particulière obtenue par la démonstration mathématique du programme peut être mise à profit pour alléger le code.

3.4.2 Programmation défensive

Dans le cadre du développement de logiciels tolérants aux fautes, la *programmation défensive* est une approche couramment utilisée pour se prémunir des erreurs résiduelles du logiciel. C'est donc une technique *logicielle* de prévention des erreurs *logicielles*.

Du code redondant est introduit afin de vérifier l'état du système après modifications et afin de garantir la consistance du changement d'état. Si des inconsistances sont détectées, le changement d'état est annulé et le système est restauré dans un état correct.

Par exemple la procédure Ada suivante calcule le quotient et le reste d'une division entière de deux entiers. Si le dénominateur de la division est nul, alors l'appel à cette procédure est un échec ; la variable booléenne div0 est positionnée à TRUE dans ce cas :

```
procedure Divise(a,b : in NATURAL;
                 q,r : out NATURAL; div0 : out BOOLEAN) is
    begin
        if b = 0 then
            div0 := TRUE;
            q := 0;
            r := 0;
        else
            div0 := FALSE;
            q := a / b;
            r := a rem b;
        end if;
    end Divise;
```

Dans le cas d'une telle programmation la procédure appelant la procédure `Divise` doit faire un test sur le résultat de div0 avant d'utiliser éventuellement le quotient et le reste calculés. Par exemple :

```
declare
    res, quo : NATURAL;
    bool : BOOLEAN;
begin
    Divise(312,21,res,quo,bool);
    if bool then
        ...
    else
        ...
    end if;
end
```

L'algorithme de traitement des erreurs qui est employé ci-dessus consiste donc à faire plusieurs cas à chaque endroit où une opération potentiellement dangereuse doit être effectuée. Nous avons donc fait deux cas pour protéger cette division. Une erreur détectée doit ensuite être remontée jusqu'aux couches supérieures chargées de leur prise en compte.

Remarquons que dans le cas du langage Ada l'utilisation du mécanisme de propagation et de rattrapage des erreurs par les exceptions est une façon différente de présenter le même algorithme ; le concept clef de la programmation défensive est que l'on tente de se prémunir des erreurs logicielles, par exemple en vérifiant les paramètres en entrée des sous-programmes.

En résumé, la programmation défensive consiste à protéger chaque partie du programme des erreurs produites par les autres parties. Si une erreur se produit, il n'est pas forcément possible de poursuivre l'exécution, mais le logiciel sait qu'il y a eu problème. Il peut donc s'arrêter (mise dans un état sécuritaire) et tracer l'erreur (facilité pour corriger le programme).

3.4.3 Programmation offensive

Par opposition à la technique défensive, on appelle programmation offensive la technique de programmation consistant à sciemment ne pas se prémunir des inconsistances de changement d'état. L'hypothèse fondamentale pour utiliser ce type de programmation est que les différents comportements dynamiques possibles du logiciel le maintiendront toujours dans un état consistant. En particulier, pour utiliser ce style de programmation il faut disposer d'un moyen de garantie que les conditions d'appel des opérations sont vérifiées ; c'est-à-dire que les paramètres d'entrée sont dans leur domaine de définition et respectent les contraintes imposées.

Ce style de programmation allège le travail du programmeur et rend le code plus efficace. En effet, il n'est pas nécessaire, dans de telles conditions, de surcharger le code par des vérifications systématiques de la validité des données ; il n'est pas non plus nécessaire de formaliser des algorithmes de reprise sur erreur.

La programmation offensive nécessite de formaliser dès la spécification, les états dynamiques atteignables par le logiciel, et les préconditions d'appel des opérations. Ceci peut être dangereux si *une méthode formelle de développement* n'est pas utilisée ; le passage de la spécification à la conception pouvant mettre en échec les préconditions voire même les occulter.

3.4.4 Programmation offensive avec B

En B, seules les opérations peuvent modifier l'état dynamique du logiciel. Chaque opération est définie dans une machine abstraite et débute par une précondition (substitution `PRE`) éventuellement vide. Les obligations de preuve auxquelles participent ces préconditions conduisent aux preuves suivantes :

- Pour chaque opération, prouver que quels que soient les paramètres d'entrée (vérifiant les préconditions) et l'état dynamique actuel, l'état atteint après exécution du service est correct.
- Pour chaque appel d'opération, prouver que le contexte d'appel satisfait la précondition de l'opération.

Nous pouvons en déduire que dans un développement ainsi démontré le test des paramètres d'entrée des opérations *internes*, c'est-à-dire appelées par d'autres opérations, est inutile. Les principaux avantages liés à l'utilisation de la méthode B pour la programmation offensive sont :

- sécurité : les cas d'erreur sont pris en compte au niveau des modules appelant et la génération des obligations de preuve assure que toutes les combinaisons concernant les conditions d'appel des opérations ont été contrôlées.
- efficacité : les cas d'erreurs sont pris en compte dès la spécification et interviennent également dans la construction de l'architecture. Il n'y a pas de code supplémentaire pour tester la consistance de l'état interne ou les conditions d'appel des opérations.
- maintenabilité : les préconditions d'appel des opérations sont formalisées dès la spécification ; les modifications de ces préconditions entraînent la modification de la machine abstraite définissant l'opération. Le mécanisme des obligations de preuve garantit alors le respect de ces préconditions par les modules appelant et la cohérence de l'opération.

Nous allons présenter un exemple du parti que l'on peut tirer de la programmation offensive. On suppose disposer d'un équipement de mesure possédant deux “fils” de sortie :

l'un pour indiquer que l'équipement est en marche, l'autre pour indiquer qu'il est à l'arrêt. Si l'équipement est en marche, on peut lancer la mesure, sinon le résultat est zéro par convention.

Modélisons ceci de la manière suivante :
– *InfoOn* modélisera le fil "marche". C'est un booléen qui vaut TRUE si le fil est sous tension, FALSE sinon.
– De la même manière, *InfoOff* modélisera le fil "arrêt".
– *Measure* représentera l'opération de mesure qui doit retourner 0 ou la mesure faite.

Clairement, une seule variable booléenne suffirait pour indiquer la marche. Nous avons modélisé les deux fils pour représenter fidèlement la réalité, mais aussi parce qu'ils nous apportent une information supplémentaire : si *InfoOff* n'est pas l'inverse de *InfoOn* c'est que l'appareil est incohérent. Nous supposerons que ce type d'erreur doit être traité au niveau supérieur. La spécification du module est alors la suivante :

```
MACHINE
    Acq
CONCRETE_VARIABLES
    InfoOn, InfoOff
INVARIANT
    InfoOn ∈ BOOL ∧
    InfoOff ∈ BOOL
INITIALISATION
    InfoOff, InfoOn := TRUE, FALSE
OPERATIONS
    res ← Measure = PRE
        InfoOn ≠ InfoOff
    THEN
        ANY nres WHERE
            nres ∈ INT ∧
            (InfoOff = TRUE ⇒ nres = 0)
        THEN
            res := nres
        END
    END ;
    GetInfo = BEGIN
        InfoOn :∈ BOOL ||
        InfoOff :∈ BOOL
    END
END
```

La précondition de la première opération oblige le module utilisateur à tester la cohérence des informations, ce qui correspond à notre idée : si l'équipement peut tomber en panne et que nous voulons que le logiciel prenne ce cas en compte, il faut tester cet équipement avant de l'utiliser. Le précondition de l'opération `Measure` oblige le module appelant à faire ce test.

Nous sommes donc certains que l'opération `Measure` n'est appelée que si l'équipement fonctionne : nous pouvons donc faire une programmation offensive de l'opération de mesure.

```
IMPLEMENTATION
    Acq_imp
REFINES
    Acq
IMPORTS
    LowLevel
INITIALISATION
    InfoOn := FALSE;
    InfoOff := TRUE
OPERATIONS
    res ← Measure = IF InfoOff = TRUE THEN
        res := 0
    ELSE
        res ← PhysMeas
    END;
    GetInfo = BEGIN
        InfoOn ← Get1;
        InfoOff ← Get2
    END
END
```

La machine *LowLevel* représente le niveau physique, nous ne la montrerons pas ici. Grâce à l'assurance que nous avons que l'opération de mesure ne peut être appelée que si les informations sont cohérentes, nous avons simplifié la programmation de cette opération de mesure : elle ne teste plus que le booléen *InfoOff*.

La fiabilité de ce programme n'est absolument pas diminuée par cette simplification. Notons qu'il est possible que l'équipement tombe en panne pendant l'exécution de l'opération `Measure`, dans ce cas si nous avions fait une nouvelle acquisition des variables `InfoOn` et `InfoOff` après le début de l'opération nous aurions détecté un état incohérent. Garder un style de programmation défensif aurait-il permis de tenir compte de ce cas ? Sûrement pas : ce n'est pas parceque l'équipement change d'état que les variables évoluent, il faut pour cela faire une acquisition. Le test supplémentaire n'aurait donc rien détecté car il aurait porté sur des variables représentant le dernier état acquis, qui était correct.

Si nous voulons augmenter encore la fiabilité du système, il faudrait être sûr de détecter tout changement d'état de l'équipement physique. Ceci ne peut être obtenu que si cet équipement est capable de déclencher une interruption quand il change d'état. Dans beaucoup de logiciels cycliques on se contente en fait de faire des acquisitions suffisamment fréquentes et on se fie à leur valeur pour toute la durée du cycle.

Clairement, la capacité d'un système à prendre immédiatement en compte toute modification de ses entrées physiques pour éviter toute incohérence n'est pas liée au style de programmation employé, mais au mode d'entrée employé. On peut donc employer avec profit la programmation offensive pour de telles entrées (à condition bien sûr de faire un développement formel), les hypothèses de cohérence apparaissent alors clairement dans les préconditions.

3.4.5 Conclusion

La programmation offensive, nuisible dans un développement classique, est parfaitement justifiée avec B. Elle permet d'obtenir un programme plus compact et plus rapide sans rien sacrifier à la sécurité. En effet, nous avons prouvé que les tests de paramètres supprimés ne seraient jamais passés dans le cas d'erreur. Il reste la possibilité d'erreurs produites par des phénomène physiques (bits fluctuants). Si de telles erreurs sont envisageables, le test des paramètres d'entrée est insuffisant : il ne protège que des débordements en entrée. Considérons l'exemple suivant, dans lequel une fonction consulte un tableau à l'index qui lui est fourni en entrée.

```
int fnc(e)
int e;
{
    if ((e>=0) && (e<MAXTAB)) {
        return(tab[e]);
    } else {
        /* erreur */
        return(-1);
    }
}
```

Si une perturbation électrique modifie la valeur de `e` en cours d'exécution, alors :
- Soit la perturbation a lieu avant l'entrée dans la fonction, et la valeur devenue fausse n'est plus entre 0 et MAXTAB ; dans ce cas l'erreur est détectée.
- Soit la perturbation a lieu avant l'entrée, mais en restant entre 0 et MAXTAB. Le comportement du programme devient incohérent.
- Soit la perturbation a lieu après le test d'entrée. Dans ce cas le comportement devient incohérent ou bien il y a un débordement de tableau.

La sécurité obtenue par les tests d'entrée est très incomplète. Dans le cas ou les perturbations électriques peuvent modifier les données, on utilisera donc un système redondé ou un processeur à vérification dynamique de données.

3.5 Comment structurer un projet B

Les objectifs d'une "bonne" modélisation formelle sont multiples :
- exprimer de façon claire et précise l'ensemble des propriétés du système,
- faciliter le travail de preuve en proposant une découpe en machines abstraites et implantations raisonnée,
- assurer la maintenabilité en limitant les effets d'une modification sur l'ensemble du modèle.

Nous allons maintenant examiner le second problème : comment architecturer un projet pour faciliter la preuve et comment mesurer la qualité du découpage choisi.

3.5.1 Architecture du Projet

Les performances du prouveur en mode automatique dépendent fortement de l'architecture adoptée : un modèle peu décomposé peut produire des obligations de preuve peu nombreuses, mais compliquées.

La lisibilité et la maintenabilité du modèle B dépendent également de l'architecture du projet : les liens architecturaux superflus ne doivent pas être introduits.

Pour cela, les règles de construction suivantes sont conseillées :
- les liens de type `SEES` sont peu nombreux, ceci afin d'améliorer la lisibilité et la maintenabilité des composants.
  Ces liens sont créés pour consulter les instances des composants de contexte (définissant les ensembles et les constantes).
- les liens de type `INCLUDES` sont utilisés essentiellement sur des composants de contexte : les données statiques d'un composant sont ainsi définies à part, dans un composant de contexte.
- la décomposition par les liens `IMPORTS` se fait dés que la réalisation d'une spécification nécessite l'utilisation de ressources ou services identifiés. L'intérêt apparaît surtout lors de la génération des obligations de preuve : elles sont plus simples et moins nombreuses.
- un composant B comprend une à deux étapes de raffinement. En introduire plus révèle la difficulté rencontrée pour préciser soit un raffinement algorithmique soit un raffinement de données ; il est préférable de décomposer, lorsque cela est possible, en plusieurs machines abstraites. La lisibilité et la maintenabilité du modèle sont moindres lorsque plusieurs étapes de raffinement existent.

En résumé, le découpage doit être fait essentiellement par `IMPORTS`, comme cela est également expliqué au paragraphe 4.2.

3.5.2 Taux de couverture de preuve

Le taux de couverture de preuve correspond au pourcentage d'obligations de preuve démontrées par rapport au nombre total d'obligations de preuve (non triviales) produites. Le nombre total d'obligations de preuve dépend de la manière dont l'outil utilisé découpe les formules théoriques utilisées pour leur construction.

Un certain pourcentage de ces preuves est alors démontré par le prouveur automatique de théorème sans lequel la mise en œuvre de la méthode B n'est pas réaliste. On estime qu'au dessous de 60% de preuves automatiques, le projet B nécessite d'être remanié soit en décomposant les modules B, soit en introduisant des raffinements, soit encore en exprimant le besoin autrement.

La décomposition obtenue par les liens `IMPORTS` permet non seulement de découper le modèle B en activités spécifiques, mais aussi de découper le travail de preuve. Les obligations de preuve produites sont, de ce fait, plus simples et moins nombreuses (apparition d'obligations de preuve triviales).

Notons bien que *tout projet B juste n'est pas forcément raisonnablement démontrable.* Autrement dit, il est possible d'écrire un projet B entièrement juste, mais qui ne pourra pas être démontré car à cause de sa strucure même et de la manière dont les exigences sont modélisées, sa preuve est très compliquée. Utiliser la méthode B, c'est non seulement écrire des projets justes mais également démontrables.

3.5.3 Métriques d'un composant B

La constitution d'un composant B influe sur le travail de preuve. Ainsi, des règles de "bonne programmation B" sont définies au-delà desquelles le travail de preuve risque d'être très

fastidieux.
Ces règles concernent :
– le nombre de données (variables, ensembles et constantes) définies pour un module,
– le nombre d'opérations introduites,
– le nombre de liens architecturaux pour un module,
– le nombre d'instructions de même type en séquence ou imbriquées utilisées dans le corps des opérations.

Bien entendu, la complexité des obligations de preuve produites mesure entre autres ces grandeurs. Il est néanmoins utile de définir des valeurs limites à ne pas dépasser pour obtenir des bons résultats dès la première écriture du composant. L'atelier de génie logiciel utilisé pourra intégrer un vérificateur de limites effectuant ce genre de contrôles.

3.6 Ce que nous avons appris

- La *réexpression du besoin* consiste à redéfinir de manière non ambiguë le produit à obtenir. Cette définition est nécessaire préalablement à la spécification B.
- La *modélisation mathématique* à partir du besoin réexprimé se fait soit à plat, soit en descendant des exigences essentielles.
- La spécification B est la partie de l'arbre des composants du projet qui contient l'équivalent de la spécification informelle.
- De la spécification formelle jusqu'au programme final, tout est constitué de composants B.
- Il faut valider la spécification formelle pour contrôler qu'elle représente bien le besoin initial.
- L'analyse du cahier des charges permet d'isoler les exigences essentielles du produit à construire et d'en faire une liste structurée.
- Les propriétés statiques sont celles qui doivent être conservées en permanence. On les retrouve dans les invariants.
- Les propriétés dynamiques sont celles qui sont établies après certaines opérations.
- Il y a des propriétés statiques globales qui ne peuvent apparaître que dans les machines de regroupement.
- La programmation offensive consiste à éliminer du programme les tests de cohérence des variables et paramètres d'entrée.
- La programmation offensive allège et optimise le programme, mais ne doit être employée que pour des programmes prouvés, garantis sans erreurs internes.
- La programmation défensive n'élimine pas les erreurs internes, mais permet d'arrêter le programme s'il s'en produit.
- La programmation défensive est inutile pour des programmes démontrés : les cas d'erreurs détectés ne peuvent se produire.
- La programmation offensive *n'élimine pas* les tests des informations provenant de l'extérieur du programme.
- La programmation défensive n'est pas une protection suffisante contre les défaillances matérielles.
- Si le produit informatique doit résister à des défaillances matérielles, il convient d'utiliser des processeurs redondants ou à vérification dynamique d'intégrité de données.
- L'architecture des composants B doit faciliter la preuve : utiliser essentiellement **IMPORTS**, éliminer tout lien inutile.
- Un projet B ne peut être fait sans un atelier muni d'un démonstrateur automatique. Si le taux de preuve automatique est inférieur à 60%, on estime généralement que la modélisation doit être retouchée.
- L'utilisation de métriques sur les sources B permet de contrôler la complexité au moment même de leur écriture. Il est souhaitable de disposer d'un tel outil de métriques.

Chapitre 4

Les clauses d'architecture

Nous allons, dans ce chapitre, rappeler les clauses d'architecture utilisables en B. Nous justifierons alors de leur emploi, en présentant, de façon non exhaustive, les différentes méthodes de décomposition d'un projet B. Enfin, nous rappellerons et justifierons les règles d'utilisation de ces clauses d'architecture.

4.1 Rappels

1. La clause `IMPORTS` permet de créer des instances de machines abstraites, afin de pouvoir utiliser dans l'implantation les données et les services de ces instances de machines. En important une machine abstraite, l'implantation demande la création d'une instance concrète de cette machine et devient alors le père de cette instance ; seule cette implantation a le pouvoir de modifier les variables de l'instance, via des opérations de modifications.

   La clause `IMPORTS` permet donc à une implantation de réaliser sa spécification en utilisant les données et services d'autres programmes vus par leur spécification. La décomposition de problèmes en sous-problèmes comme la factorisation de services utilisables par plusieurs opérations sont réalisées par l'utilisation du lien `IMPORTS`. Cet aspect est développé dans la suite de ce chapitre.
2. La clause `INCLUDES` permet de regrouper dans une machine abstraite les variables, les constantes, les ensembles avec leurs propriétés, provenant d'autres machines abstraites. La clause `INCLUDES` permet donc de décomposer une machine abstraite complexe en plusieurs machines abstraites, tout en facilitant le travail de preuve associé. En effet, la preuve d'un composant et des ses machines "incluses"est globalement plus simple que la preuve du composant équivalent non décomposé. L'interprétation d'une inclusion est simple : les variables et constantes de la machine incluse deviennent des entités de la machine incluante, et les opérations incluses deviennent des "morceaux de spécification" utilisables.

   Il est possible d'inclure plusieurs instances différentes d'une même machine abstraite : dans ce cas les instances créés sont des instances abstraites, non atteignables par `SEES`.

   Lorsqu'un composant `M` inclut une machine abstraite `N`, plusieurs possibilités sont envisageables lors de l'écriture de l'implantation `M_imp` qui réalise la spécification de `M`.

L'implantation `M_imp` peut importer la machine abstraite `N` ; de cette façon, les variables, ensembles et constantes de la machine abstraite incluse en spécification seront réalisés par les variables, ensembles et constantes de l'instance importée.

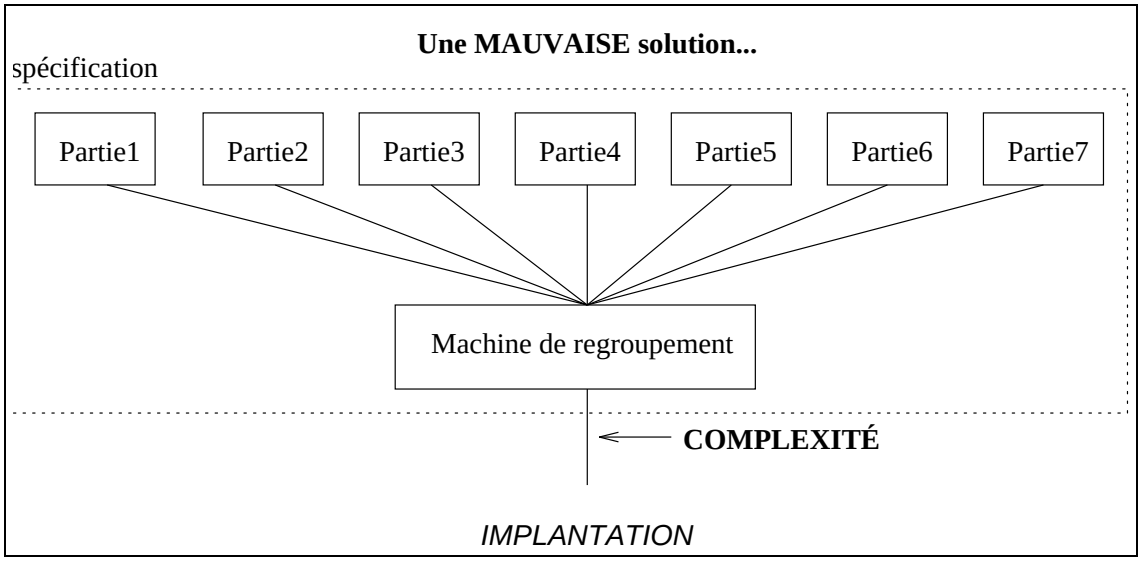
L'implantation `M_imp` peut ne pas importer la machine abstraite `N`. Dans ce cas, les ensembles, variables et constantes concrètes définies dans la machine abstraite `N` devront être réalisés par l'implantation `M_imp` soit en local, soit en créant une instance d'un autre composant. Le choix de cette décomposition reste à la charge de l'utilisateur ; l'intérêt de la combinaison des liens `INCLUDES/IMPORTS` est développé dans ce chapitre.

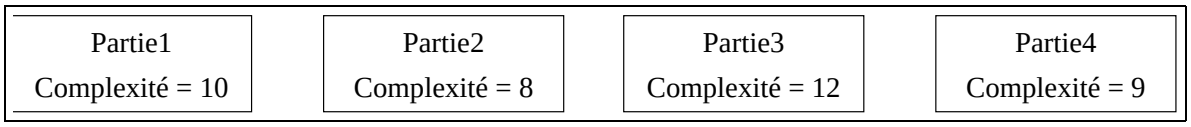
3. La clause `SEES` permet d'introduire dans un composant une liste d'instances de machines dont les constituants (ensembles, constantes et variables) sont consultables dans le composant mais non modifiables. Un composant (machine abstraite, raffinement ou implantation) peut "voir" une instance d'une machine, pour cela, cette instance de la machine abstraite a été créée, au préalable, par un lien `IMPORTS` quelque part dans le projet. Cette clause peut provoquer un problème connu sous le nom d'aliasing (voir 4.5.3) si le projet n'est pas vérifié par l'Atelier B.
4. La clause `USES` est une clause de spécification. Lorsqu'un composant *inclut* plusieurs machines, les machines *incluses* peuvent partager les données (ensembles, constantes et variables) d'une des machines incluses en utilisant la clause `USES`. Ces données sont alors consultables et non modifiables. Cette clause est rarement employée.
5. La clause `PROMOTES` n'est pas une clause d'architecture, mais elle permet à une machine abstraite (respectivement une implantation) de promouvoir des opérations appartenant à des machines incluses (respectivement importées), c'est-à-dire que les opérations concernées deviennent des opérations de la machine d'accueil, proposées à l'extérieur.
6. La clause `EXTENDS` est définie par les équivalences suivantes :
   – Dans une machine ou un raffinement : **EXTENDS** M signifie **INCLUDES** M **PROMOTES** < toutes les opérations de M >
   – Dans une implantation : **EXTENDS** M signifie **IMPORTS** M **PROMOTES** < toutes les opérations de M >

4.2 La décomposition en B

Avant tout problème de conception, l'un des premiers problèmes que doit affronter celui qui fait des spécifications B après la formalisation de ses propriétés est la décomposition de ses spécifications. La spécification complète d'un logiciel est quelque chose de complexe, contenant beaucoup de parties différentes. Même si le spécificateur a pu traduire toutes ces parties par des formules mathématiques, ces formules sont trop nombreuses pour être groupées dans une seule machine abstraite.

Une mauvaise idée consiste à seulement découper ces formules en morceaux indépendants, dans des machines abstraites séparées. En effet, pour tirer parti de la modélisation, il faudra prouver des propriétés qui découlent de la juxtaposition de l'ensemble de ces formules, c'est-à-dire qu'il faudra regrouper les machines abstraites à un certain moment. On perd alors l'intérêt du découpage. C'est exactement ce qui se produit si on tente de découper une grosse spécification par **INCLUDES** : pour implanter la spécification, il faut écrire





une machine de regroupement qui inclut toutes les parties. La preuve de ce raffinement pose alors des problèmes de taille.

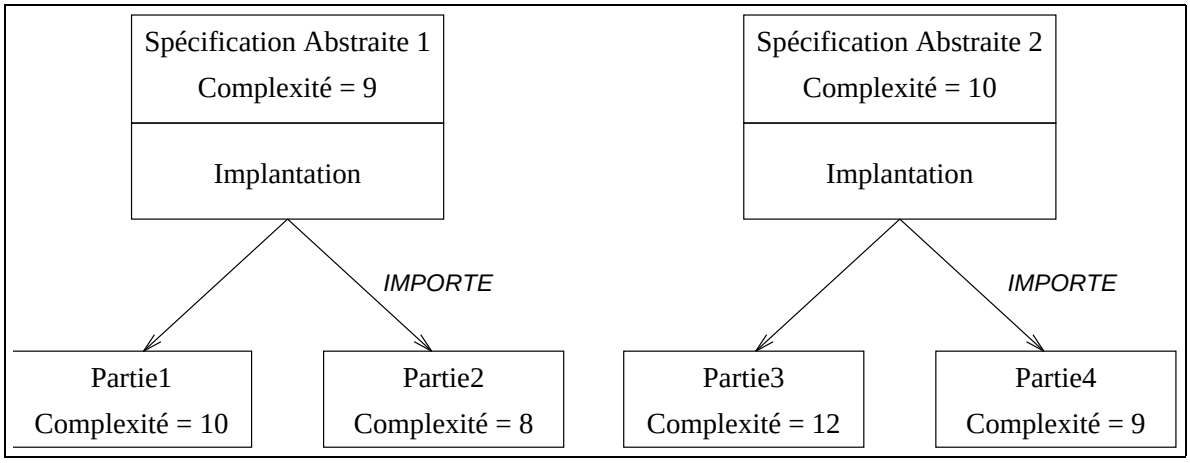
En B, ce problème se résout en utilisant des spécifications plus abstraites. Si notre spécification complète se décompose en un certain nombre de parties, supposons que nous puissions mesurer la complexité de ces parties et leur affecter un nombre :

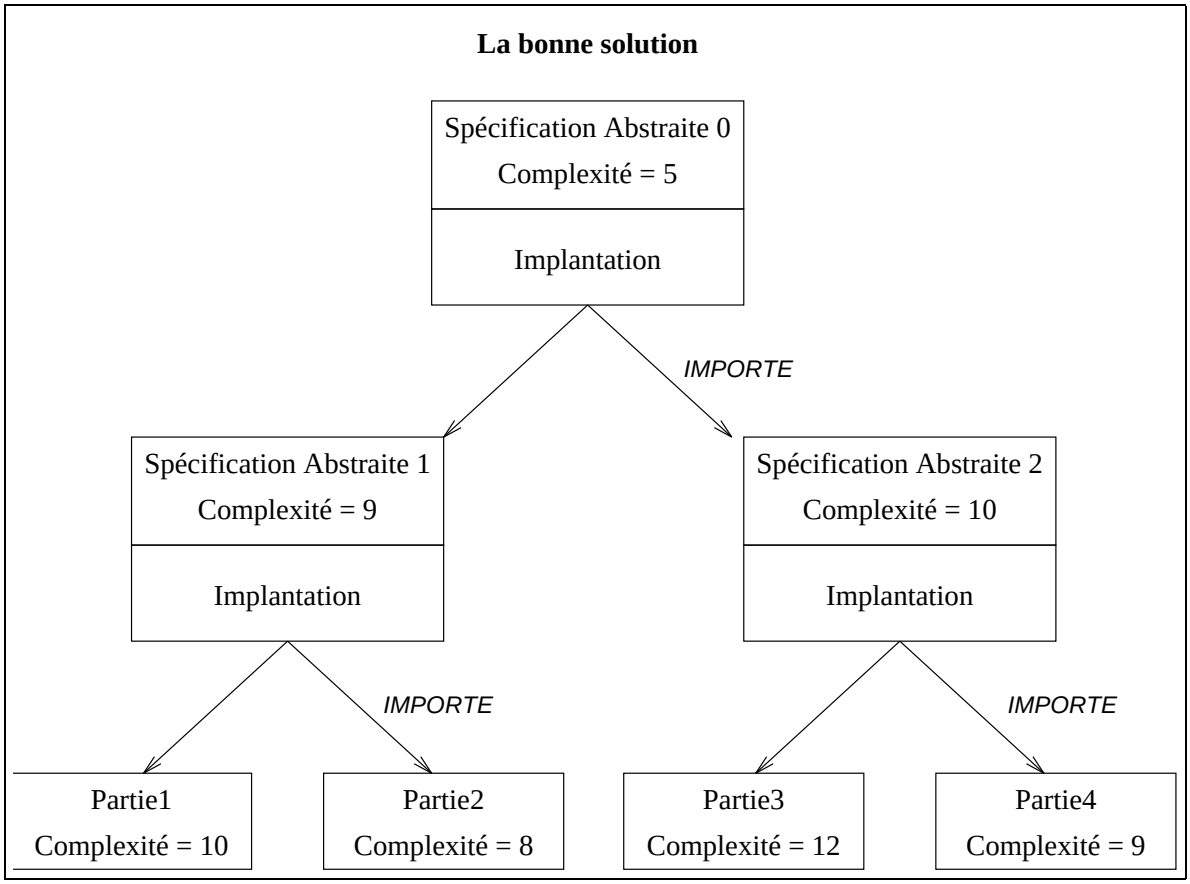
Nous devons trouver comment rassembler ces spécifications sans dépasser un certain niveau de complexité, disons *Complexite = 15* dans notre système de mesure imaginaire. Dans la pratique bien entendu, nous n'aurons jamais un tel système de mesures quantifiables ! Ces chiffres ne servent qu'à illustrer notre propos.

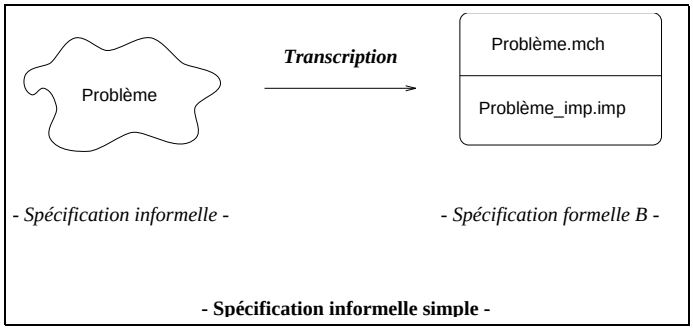
Nous allons regrouper ces spécifications en les représentant par groupes représentés chacun par une spécification plus abstraite, *qui ne peut pas contenir tous les détails du groupe.* Cette représentation est faite en *implantant* (clause **IMPORTS**) la spécification plus abstraite sur les éléments du groupe :

La complexité des spécifications abstraites est inférieure à la somme des complexités des spécifications détaillés couvertes car *il n'y a pas tous les détails* de ces dernières. Par contre, dans ces spécifications plus abstraites il est possible de prouver des propriétés qui découlent des propriétés combinées des spécifications recouvertes. Ceci se généralise jusqu'au plus haut niveau :

L'une des conséquences de cette décomposition est que la machine de plus haut niveau est simple, même s'il s'agit d'un logiciel très compliqué. La machine de plus haut niveau d'un système de contrôle de trafic aérien ne devrait pas être plus compliquée que celle d'un logiciel de gestion d'un panneau de voyants : dans les deux cas cette machine ne contient que la contrainte la plus essentielle. Par exemple, pour un panneau de voyant ce pourrait être "Toute alarme doit être affichée sur l'un des voyants", pour un système de contrôle de trafic ce serait peut être "la distance entre deux avions doit rester supérieure à une


Spécification Abstraite 1
Complexité = 9
Implantation
Spécification Abstraite 2
Complexité = 10
Implantation
IMPORTE
IMPORTE
Partie1
Complexité = 10
Partie2
Complexité = 8
Partie3
Complexité = 12
Partie4
Complexité = 9


La bonne solution
Spécification Abstraite 0
Complexité = 5
Implantation
IMPORTE
Spécification Abstraite 1
Complexité = 9
Implantation
Spécification Abstraite 2
Complexité = 10
Implantation
IMPORTE
IMPORTE
Partie1
Complexité = 10
Partie2
Complexité = 8
Partie3
Complexité = 12
Partie4
Complexité = 9



certaine valeur". Ces propriétés sont toutes deux très simples, même s'il s'agit de systèmes de complexités très différentes.

Ce type de démarche impose que l'on connaisse les propriétés qu'il faut prouver sur l'ensemble du système : ce sont les seules qui figurent dans les spécifications les plus abstraites. Les autres détails ne doivent pas encore apparaître. Nous pensons que la détermination précoce de ces propriétés essentielles est le moyen fondamental pour maîtriser le processus de développement d'un système complexe : savoir ce que l'on veut avant de le faire.

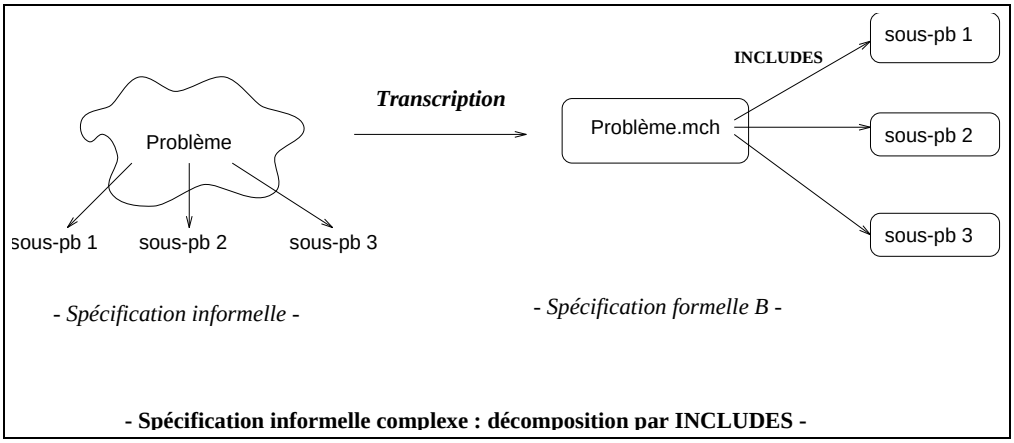
Ces propriétés forment donc *ce que l'on veut* au départ : la vraie spécification de plus haut niveau de notre logiciel. C'est en partant de ces propriétés et en faisant apparaître *graduellement* les détails, en construisant l'arbre de haut en bas que l'on développe la spécification complète. Cette spécification complète est donc un arbre contenant des implantations. Le même style de démarche d'introduction graduelle de détails est également utilisée dans les raffinements.

Une spécification sans description simple et exacte, dont la seule description est une accumulation de détails, doit être réétudiée avant l'écriture des composants pour pouvoir être exprimée en B. Cette phase courante dans les projet B s'appelle la réexpression du besoin.

4.2.1 Revue des différent découpages

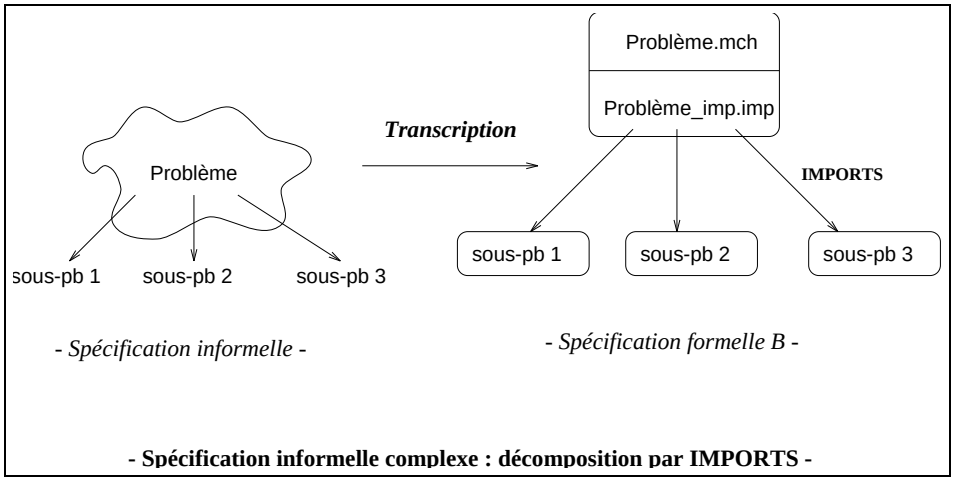
La transcription en B d'une spécification informelle suit le schéma suivant :
– si la spécification informelle est simple, elle est décrite entièrement dans une seule machine abstraite. L'implantation se fait alors très simplement puisque toutes les informations nécessaires à l'écriture de l'implantation sont présentes dans la machine abstraite.
– si la spécification informelle est trop complexe pour être transcrite dans une seule machine abstraite, trois possibilités sont envisageables.

1. décomposer le problème en sous-problèmes et décomposer la machine abstraite en plusieurs machines abstraites par des liens de type `INCLUDES`. L'inconvénient majeur de cette décomposition est que le spécifieur obtient en fait une "grosse"machine abstraite ; l'implantation associée doit réaliser l'ensemble de la spécification. Il existe, cependant, des cas où l'utilisation de la clause `INCLUDES` est fortement recommandée. Le lecteur se reportera alors au paragraphe 4.3.
2. spécifier le minimum nécessaire (les propriétés "importantes") pour prouver les implantations qui utilisent cette machine abstraite ; c'est en fait la spécification du module pour les modules qui l'utilisent. L'implantation découpe alors le problème en sous-problèmes et importe un ensemble de machines abstraites qui réalisent ces



- Spécification informelle complexe : décomposition par INCLUDES -

sous-problèmes.



- Spécification informelle complexe : décomposition par IMPORTS -

3. spécifier le minimum nécessaire pour prouver les implantations qui utilisent cette machine abstraite puis créer un raffinement qui décrit complètement la spécification informelle. Cette technique est utilisée lorsque la spécification informelle est complexe mais non “volumineuse”.

4.3 Combinaison INCLUDES/IMPORTS

Le lien `INCLUDES` ne permet pas de structurer un projet, il permet simplement de décomposer une machine abstraite en plusieurs machines abstraites. Les données des machines incluses (ensembles, constantes et variables) sont regroupées dans la machine qui a demandé l'inclusion.

Lorsqu'une telle décomposition est faite, la question se pose souvent de savoir comment réaliser les entités incluses. On peut alors utiliser la méthode INCLUDES/IMPORTS que nous allons présenter dans cette section.

Rappelons que le fait d'inclure une machine abstraite à partir d'une autre machine abstraite revient à intégrer à cette dernière toutes les notions, variables, ensembles, ..., de la machine incluse. C'est un peu comme si nous avions recopié le texte de la machine incluse dans la machine incluante, sauf que nous n'avons pas le droit de modifier directement ses

variables sans passer par les opérations. Cette dernière restriction permet de considérer que l'invariant inclus reste vrai sans avoir à le prouver, car la machine incluse est prouvée par ailleurs.

Il se peut que la machine incluse ait été implantée par ailleurs. En incluant le texte de sa spécification à notre machine incluante, nous n'avons absolument pas intégré cette implantation. Comment tirer néanmoins partie de cette dernière ?

L'implantation de la machine "incluante" doit réaliser[1] *elle-même* les données des machines abstraites incluses et ses propres données. Pour ce faire, deux possibilités s'offrent à l'utilisateur :

– l'implantation réalise *en local* ces données. Prenons l'exemple de la valuation des éléments constants : des valeurs effectives sont données aux ensembles abstraits et constantes concrètes dans la clause `VALUES`. Il s'agit alors d'une valuation dite *explicite*.

```
MACHINE
  mm
INCLUDES
  mi
SETS
  mm_ens1 = {elem1, elem2, elem3} ;
  mm_ens2
CONCRETE_CONSTANTS
  mm_cst
PROPERTIES
  mm_cst ∈ NAT
...
END
```

```
MACHINE
  mi
SETS
  mi_ens1 = {un, deux, trois} ;
  mi_ens2
CONCRETE_CONSTANTS
  mi_cst
PROPERTIES
  mi_cst ∈ mi_ens1
...
```

```
IMPLEMENTATION
  mm_i
REFINES
  mm
VALUES
  /* valuation des ensembles abstraits */
  mm_ens2 = 1..100 ;
  mi_ens2 = 10..100 ;
  /* valuation des constantes concrètes */
  mm_cst = 10 ;
  mi_cst = elem1
...
END
```

L'exemple ci-dessus montre la valuation *en local* de données statiques. La clause `VALUES` de l'implantation `mm_i` donne une valeur effective aux ensembles abstraits et aux constantes concrètes ; ce sont à la fois les ensembles abstraits et constantes définis dans la machine abstraite `mm`, mais aussi ceux provenant de la machine abstraite "incluse" `mi`. La réalisation des données dynamiques (variables) suit le même principe.

– l'implantation *délègue* la réalisation de ces données en en important d'autres machines abstraites. Rappelons que les variables d'une machine peuvent être réalisées par liaison avec des variables importées (invariant de liaison ou homonymie). Pour la partie statique, la clause `VALUES` permet de donner une valeur aux constantes et ensembles. Le nom de

[1] Les ensembles abstraits et constantes des machines abstraites incluses, comme les ensembles abstraits et constantes concrètes de toute machine abstraite, doivent être valués ; ceci afin de pouvoir les utiliser dans une implantation.

la clause `VALUES` est suivi d'une liste de valuations.
Chaque valuation permet de donner explicitement une valeur à une constante ou ensemble. Si une donnée à valuer possède le même nom qu'un ensemble ou qu'une constante d'une machine *vue* ou *importée*, alors elle ne doit pas être valuée explicitement. La donnée prend implicitement la valeur de sa donnée homonyme. Sinon, elle doit être valuée explicitement.
On est dans le schéma INCLUDES/IMPORTS quand les machines importées pour réaliser les données provenant des machine incluses sont justement ces mêmes machines. On inclut donc ces machines dans l'abstraction, et on les importe dans l'implantation. Les instances abstraites ("morceaux" de spécification) créées par l'inclusion n'ont aucun rapport avec les instances concrètes créées par importation, mais le mécanisme de liaison homonyme s'applique pour la réalisation de toutes les données de ces machines.
Si nous reprenons l'exemple précédent, l'implantation `mm_i` devient avec cette technique :

```
IMPLEMENTATION
  mm_i
REFINES
  mm
IMPORTS
  mi
VALUES
  /* valuation des ensembles abstraits */
  mm_ens2 = 1..100 ;
  /* valuation implicite des ensembles abstraits de mi
  mi_ens2 = 10..100 ; */
  /* valuation des constantes concrètes */
  mm_cst = 10
  /* valuation implicite des constantes concrètes de mi
  mi_cst = elem1 */
...
END
```

L'implantation délègue ici la valuation de l'ensemble abstrait `mi_ens2` et la constante concrète `mi_cst` à l'instance de la machine abstraite importée `mi`.

Un exemple de décomposition suivant le schéma `INCLUDES/IMPORTS` est présenté ci-après. Il s'agit de modéliser un `Agenda` qui stocke les noms, adresses et numéros de téléphone de personnes. On suppose que toute personne présente dans l'agenda possède au moins un numéro de téléphone et une adresse. Les fonctionnalités offertes sont :

– *Ajouter* le triplet (nom, numéro de téléphone, adresse) à l'agenda,
– *Retirer* une personne de l'agenda,
– *Connaître un numéro de téléphone* d'une personne,
– *Connaître une adresse* d'une personne.

Les fonctionnalités décrites précédemment sont définies dans une machine abstraite de plus haut niveau `Agenda`.

La modélisation de l'agenda fait intervenir deux relations. La première associe à une personne, une ou plusieurs adresses. La seconde associe à une personne, un ou plusieurs numéros de téléphone. Les opérations qui agissent sur ces relations sont :
– Ajouter un couple à la relation,
– Retirer tous les couples de la relation dont le premier élément est le nom de la personne à retirer de l'agenda,
– Connaître une des images d'un nom par la relation.

Ces opérations sont identiques pour les deux relations, il est donc préférable d'utiliser une machine abstraite réalisant ces opérations élémentaires sur une relation. Deux instances de cette machine abstraite sont ensuite créées : une pour gérer les adresses, la seconde pour gérer les numéros de téléphone.

La représentation formelle de ce problème est donnée par les machines abstraites suivantes :

MACHINE
Agenda
SETS
NOM ; ADRESSE ; TEL
INCLUDES
adresse.Relation(NOM, ADRESSE),
telephone.Relation(NOM, TEL)
PROMOTES
adresse.Image_de, telephone.Image_de
INVARIANT
dom(telephone. relation) = dom(adresse.relation)
OPERATIONS
Ajout(nom, n_tel, adress)=
PRE
nom ∈ NOM ∧
n_tel ∈ TEL ∧
adress ∈ ADRESSE
THEN
telephone.Ajout_elem(nom, n_tel) ||
adresse.Ajout_elem(nom, adress)
END ;
Retrait(nom) =
PRE
nom ∈ NOM
THEN
telephone.Retrait_elem(nom) ||
adresse.Retrait_elem(nom)
END
END

La machine abstraite `Agenda` inclut deux instances de la machine abstraite `Relation`. Comme ces deux instances sont renommées, les données de la machine abstraite `Relation` sont prefixées par le nom de l'instance qu'elles référencent. Vues de la machine `Agenda`, toutes les variables sont préfixées par le nom de l'instance concernée.

L'invariant `dom(telephone.Relation) = dom(adresse.Relation)` précise que toute personne gérée dans l'agenda possède au moins une adresse et un numéro de téléphone. Pour conserver cet invariant, nous devons entrer en même temps une adresse et un numéro de téléphone pour tout nouvel abonné.

Deux opérations sont promues (cf. clauses `PROMOTES`) pour satisfaire les fonctionnalités désirées : ce sont les opérations de consultation qui permettent de consulter l'agenda.

Notons qu'il est possible d'appeler plusieurs fois de suite l'opération `Ajout` pour le même nom. Dans ce cas les nouveaux numéros de téléphone et adresses seront mémorisés sans effacer les anciens, puisque les objets relation utilisés peuvent avoir plusieurs images pour

le même élément. Lors d'une opération de consultation, nous verrons que le numéro et l'adresse retournés sont choisis de manière non déterministe parmi ceux enregistrés à ce nom : on suppose que l'utilisateur a seulement besoin de connaître l'une des adresses enregistrées.

Lors de l'opération de destruction d'un enregistrement, tous les numéros de téléphone et toutes les adresses associés au nom concerné sont éliminés.

Dans cette machine `Agenda`, l'usage d'un lien `INCLUDES` nous permet de ne définir qu'une fois un modèle d'encapsulation d'une donnée de type relation. Si ce modèle a été implanté, il convient de réutiliser cette implantation. Voici la machine abstraite décrivant ce modèle :

MACHINE
Relation(DOMAINE, CODOMAINE)
ABSTRACT_VARIABLES
relation
INVARIANT
relation ∈ DOMAINE ↔ CODOMAINE
INITIALISATION
relation := {}
OPERATIONS
Ajout_elem(elem1, elem2)=
PRE
elem1 ∈ DOMAINE ∧
elem2 ∈ CODOMAINE
THEN
/* ajout du couple (elem1, elem2) à la relation */
relation := relation ∪ { elem1 ↦ elem2}
END ;
Retrait_elem(elem1) =
PRE
elem1 ∈ DOMAINE
THEN
/* destruction des couples dont le premier
élément est *elem1* */
relation := {elem1} ⩤ relation
END ;
elem2 ← Image_de(elem1) =
PRE
elem1 ∈ dom(relation)
THEN
/* retourne n'importe quelle image */
elem2 ∈ relation[{elem1}]
END
END

Cette machine abstraite encapsule une donnée de type relation, utilisable pour tous les cas où il s'agit de retenir un ensemble de liens entre deux types de données, sachant que chaque donnée du premier type peut être liée à un nombre quelconque de données du second type. Elle propose des services permettant d'ajouter ou de retirer une liaison (écriture) ou de consulter les liaisons existantes.

Notons que le service de consultation `Image_de` retourne l'un des éléments liés à la donnée,

sans préciser lequel : ceci n'est utile que s'il suffit d'une réponse qui convient, par exemple si ces images représentent les canaux disponibles d'un appareil.

```
IMPLEMENTATION
    Agenda_imp
REFINES
    Agenda
VALUES
    /* la valuation des ensembles
    abstraits n'est pas représentative*/
    NOM = 1..100 ;
    TEL = 1..1000 ;
    ADRESSE = 1..100
IMPORTS
    adresse.Relation(NOM, ADRESSE), telephone.Relation(NOM, TEL)
PROMOTES
    adresse.Image_de, telephone.Image_de
OPERATIONS
    Ajout( nom, n_tel, adress )=
    BEGIN
        telephone.Ajout_elem(nom, n_tel) ;
        adresse.Ajout_elem(nom, adress)
    END ;
    Retrait( nom ) =
    BEGIN
        telephone.Retrait_elem(nom)
        adresse.Retrait_elem(nom)
    END
END
```

Les liens `INCLUDES` de spécification ont été remplacés par des liens `IMPORTS` dans l'implantation. L'implantation crée deux instances distinctes de la machine abstraite `Relation`. Les opérations promues en spécification le sont également dans l'implantation. L'utilisation des liens `INCLUDES/IMPORTS` a permis la factorisation du modèle B.

Le schéma type d'utilisation des liens `IMPORTS/INCLUDES/PROMOTES` intervient lorsque une spécification informelle peut être décomposée en activités spécifiques. La rédaction de la machine abstraite associée suit, dans ce cas, les étapes suivantes :

1. Créer des liens `INCLUDES` sur chaque machine abstraite spécifiant une activité spécifique,
2. Ajouter une clause `PROMOTES` si les opérations des machines "incluses" correspondent à des opérations à définir dans la machine abstraite,
3. Ajouter les ensembles, constantes et variables propres à la machine abstraite.
4. Ajouter, si cela est nécessaire, un `INVARIANT` liant les variables de la machine abstraite à celles des machines abstraites "incluses" ou liant les variables issues des machines abstraites "incluses",
5. Spécifier le corps des opérations en faisant appel, lorsque cela est nécessaire, aux services des machines abstraites "incluses",

La rédaction de l'implantation associée suit les étapes suivantes :

1. Créer des liens `IMPORTS` sur chaque machine abstraite spécifiant une activité spécifique,

2. Répéter la clause `PROMOTES` si celle-ci est présente dans la machine abstraite. L'utilisateur peut, toutefois, réaliser *en local* ces opérations dans la clause `OPERATIONS`,

3. Réaliser les données propres à la machine abstraite.

4.4 Combinaison IMPORTS/PROMOTES

Souvent, lors de l'écriture d'une implantation, l'utilisateur identifie de nouveaux services qui sont appelés dans le corps des opérations de l'implantation. Ces services, non spécifiés en tant qu'opération dans la machine abstraite associée et qui pourtant utilisent (ou modifient) les mêmes données ne peuvent être définis dans l'implantation. En effet, les opérations sont définies dans une machine abstraite et il n'est pas possible d'en introduire de nouvelles dans un raffinement ou dans une implantation. En fait, ceci correspond au besoin de créer des procédures locales.

Il y a deux cas possibles :
– Soit les services désirés travaillent sur des données non séparables, il s'agit alors de véritables procédures locales, actuellement non réalisables en B ;
– Soit les services désirés concernent et encapsulent un groupe de données séparées, dans ce cas on va pouvoir regrouper ces données dans une machine abstraite séparée et utiliser la combinaison IMPORTS/PROMOTES ci-après.

La combinaison IMPORTS/PROMOTES consiste à regrouper les services recherchés avec les donnés concernées dans une machine abstraite séparée, qui est importée dans l'implantation. Très souvent, certains services proposés dans la machine de départ concernent uniquement les données ainsi déportées. On peut alors déporter la réalisation de ces services dans l'implantation de la nouvelle machine et promouvoir ces opérations en tant que réalisation des services initialement proposés. Il s'agit en fait d'une technique de programmation "Top-Down".

Illustrons cette technique par un exemple de gestion de stock. Ce stock est composé d'une liste d'articles non précisés pour l'instant. Le module principal doit fournir les services suivants :
– donner l'état du stock c'est-à-dire faire un inventaire,
– rajouter au stock les articles provenant d'un approvisionnement,
– honorer (lorsque cela est possible) les commandes,
– re-initialiser le système : aucun article n'est présent dans le stock

Le choix de modélisation du *stock* est une fonction totale qui associe à chaque article géré, la quantité présente dans le stock. La machine abstraite `GESTION_STOCK`, qui réalise la spécification présentée ci-dessus, définit une variable abstraite *stock* et quatre opérations : *inventorier, approvisionner, commander et re-initialiser-stock*.

```
MACHINE
    GESTION_STOCK
INCLUDES
    Ctx
ABSTRACT_VARIABLES
    stock
INVARIANT
    /* le stock est modélisé comme une fonction
    totale (tous les éléments de l'ensemble de départ ont une image)
    qui associe à un article, la quantité de cet article présente en stock.*/
    stock ∈ ARTICLES → NAT
INITIALISATION
    stock := ARTICLES*{0}
OPERATIONS

    approvisionner(stock_en_plus) =
        PRE
            stock_en_plus ∈ ARTICLES → NAT
        THEN
            ANY nouvo_stock WHERE
                nouvo_stock ∈ ARTICLES → NAT ∧
                ∀(article).(article ∈ dom(stock_en_plus)
                ⇒ nouvo_stock(article) = stock(article)+stock_en_plus(article))
            THEN
                stock := stock ⩤ nouvo_stock
            END
        END ;

    commander(stock_en_moins) =
        PRE
            stock_en_moins ∈ ARTICLES → NAT
        THEN
            ANY nouvo_stock WHERE
                nouvo_stock ∈ ARTICLES → NAT ∧
                ∀(article).(article ∈ dom(stock_en_moins)
                ⇒ nouvo_stock(article) = max({0, stock(article)-stock_en_moins(article)}))
            THEN
                stock := stock ⩤ nouvo_stock
            END
        END ;

    inventorier =
        BEGIN skip END ;

    re_initialiser_stock =
        BEGIN
        /* il n'y a aucun article dans le stock */
            stock := ARTICLES*{0}
        END
END
```

Nous allons réaliser L'implantation de `GESTION_STOCK` en s'appuyant sur deux opérations

intermédiaires, `Ajouter` et `Retirer` :

```
IMPLEMENTATION
    GESTION_STOCK_imp
REFINES
    GESTION_STOCK
IMPORTS
    GESTION_ARTICLE, Ctx
    /* Il n'y a pas de clause INVARIANT puisque
    l'invariant de liaison est implicite */
PROMOTES
    re_initialiser_stock
OPERATIONS

    approvisionner(stock_en_plus) = VAR article IN
        article := MIN_ARTICLES -1 ;
        WHILE (article ≤ MAXI_ARTICLES) DO
            VAR nb_article IN
                article := article + 1 ;
                nb_article := stock_en_plus(article) ;
                Ajouter(article, nb_article)
            END
        INVARIANT
            article ∈ ARTICLES ∪ {MIN_ARTICLES-1} ∧
            ∀(art).(art ∈ MIN_ARTICLES..(article-1)
                ⇒ stock(art) = (stock(art) + stock_en_plus(art))
        VARIANT
                MAXI_ARTICLES - article
        END
    END ;

    commander(stock_en_moins) = VAR article IN
        article := MIN_ARTICLES -1 ;
        WHILE (article ≤ MAXI_ARTICLES) DO
            VAR nb_article IN
                article := article + 1 ;
                nb_article := stock_en_moins(article) ;
                Retirer(article, nb_article)
            END
        INVARIANT
            article ∈ ARTICLES ∪ {MIN_ARTICLES-1} ∧
            ∀(art).(art ∈ MIN_ARTICLES..(article-1)
                ⇒ stock(art) = (stock(art) + stock_en_moins(art))
        VARIANT
                MAXI_ARTICLES - article
        END
    END ;

    inventorier = skip
END
```

La spécification de la machine abstraite GESTION_ARTICLE est la suivante :

```
MACHINE
    GESTION_ARTICLE
SEES
    Ctx
ABSTRACT_VARIABLES
    stock
INVARIANT
    stock ∈ ARTICLES → NAT
INITIALISATION
    stock := ARTICLES*{0}
OPERATIONS

    Ajouter(article, nb_article) =
        PRE
            article ∈ ARTICLES ∧ nb_article ∈ NAT
        THEN
            ANY qte WHERE
                qte ∈ NAT ∧
                ((nb_article + stock(article)) > MAXINT ⇒ qte = MAXINT) ∧
                ((nb_article + stock(article)) ≤ MAXINT ⇒ qte = nb_article + stock(article))
            THEN
                stock(article) := qte
            END
        END ;

    Retirer(article, nb_article) =
    PRE
        article ∈ ARTICLES ∧ nb_article ∈ NAT
    THEN
        stock(article) := max({0, (stock(article) - nb_article)})
    END ;

    re_initialiser_stock =
        BEGIN
        /* il n'y a aucun article dans le stock */
            stock := ARTICLES*{0}
        END
END
```

L'implantation GESTION_STOCK_imp présente quelques particularités, liées à cette technique de décomposition, qui sont respectivement :

– l'absence de la clause INVARIANT : la variable *stock* de la machine abstraite GESTION_STOCK est identique à la variable *stock* de la machine abstraite GESTION_ARTICLE. Grâce au mécanisme de l'importation, l'invariant de liaison entre ces deux variables est implicite du fait de leur homonymie.

– l'opération *re_initialiser_stock* est promue (dans la clause PROMOTES) dans l'implantation GESTION_STOCK_imp. Cette opération de la machine importée devient ainsi la réalisation

de l'opération prévue dans la spécification GESTION_STOCK.
– les opérations *commander* et *approvisionner* utilisent les services définis dans le composant GESTION_ARTICLE. Le corps de ces opérations est, de ce fait, allégé.
– l'opération *inventorier* est réalisée dans l'implantation GESTION_STOCK_imp. Comme sa spécification ne présente pas d'utilisation de la variable *stock*, il n'est pas nécessaire de la définir comme opération de la machine abstraite GESTION_ARTICLE.

Cette méthode de factorisation de code pour une implantation M_imp suit les étapes suivantes :

1. Identifier les opérations de service qui facilitent l'écriture du corps des opérations de M_imp et qui utilisent les variables définies dans la machine abstraite M,
2. Créer une nouvelle machine abstraite N qui possède les mêmes variables que M et définit les opérations de service identifiées lors de l'étape précédente. Les opérations de M qui modifient ses variables, sans pour autant utiliser les services de N, sont redéfinies dans N en recopiant leur définition dans M. Ces opérations sont ensuite promues dans l'implantation M_imp.
3. Rajouter le lien IMPORTS vers la machine abstraite N dans l'implantation M_imp. Si les variables de M ne portent pas le même nom que les variables de N, rajouter les invariants de liaison entre ces variables.
4. Promouvoir toutes les opérations de M qui sont redéfinies dans N.

Ce processus de factorisation peut s'appliquer, de nouveau, pour l'implantation N_imp. Si les opérations de N sont réalisées en local dans l'implantation N_imp, ce processus s'arrête.

4.5 Règles d'utilisation

Nous présentons et justifions ici les règles d'utilisation des clauses d'architecture. Le contrôle du respect de ces règles est assuré par les outils de l'Atelier B. En particulier, nous allons préciser et justifier les règles d'utilisation de la clause **SEES**.

4.5.1 Une machine abstraite ne peut être importée qu'une seule fois dans un projet

En effet, une importation correspond à une création d'instance. Si une machine abstraite était importée deux fois dans un même projet B, il y aurait deux instances portant le même nom, et on ne saurait pas laquelle utiliser dans les machines qui la voient.

Le mécanisme de renommage resoud le problème d'utilisation de plusieurs instances d'une même machine abstraite : *les instances sont distinguées par leur nom.*

Il est donc possible d'importer plusieurs fois une même machine abstraite en la renommant, à condition que le préfixe de renommage ne soit pas utilisé par ailleurs dans le projet.

Le contrôle de cette règle est réalisé par l'outil "Project Check" de l'Atelier B. Cet outil est appelé systématiquement avant la traduction globale d'un projet. L'utilisateur peut fait appel à cet outil de façon asynchrone (utilisation du bouton dédié à cet outil dans l'IHM de l'Atelier B). Un contrôle systématique à chaque étape de construction du projet est fortement déconseillé puisque tous les modules B n'ont pas été développés et donc, des "incohérences temporaires" peuvent exister.

4.5.2 Une machine vue doit être importée dans une implantation du projet

La clause SEES est prévue pour donner une visibilité sur l'instance physique d'une machine, c'est à dire pour avoir accès en lecture aux données d'un module concrètement intégré au programme final.

Cette instance physique doit exister au niveau du projet : pour cela, il faut qu'une machine du projet importe cette machine avec le nom sous lequel elle est vue.

Le contrôle de cette règle, comme pour la précédente, est réalisé par l'outil "Project Check" et est global au projet.

4.5.3 Un composant B ne peut être vu et importé dans la même branche

Quand une machine abstraite fait référence, via un **SEES**, à une instance physique, cela vaut dire qu'elle veut connaître l'existence des variables de ce module, supposées stables pendant l'exécution des opérations de la machine "voyante". Ceci se concrétise par le fait que dans les obligations de preuve de la machine qui fait le **SEES**, aucune substitution n'est appliquée aux variables de la machine vue. Si dans l'implantation de l'une de ses opérations, la machine qui fait le **SEES** avait le droit d'utiliser l'une des opérations modifiant ces variables vues, les obligations de preuve seraient fausses.

L'exemple suivant montre comment le non respect de cette règle brise un invariant.

```
MACHINE
  mm
SEES
  mv
VARIABLES
  aa
INVARIANT
  aa :∈ NAT
INITIALISATION
  aa := bb
END
```

```
IMPLEMENTATION
  mm_i
REFINES
  mm
IMPORTS
  mv
VARIABLES
  aa
INITIALISATION
  aa ← get_b ;
  set_bb(5)
```

```
MACHINE
  mv
VARIABLES
  bb
INVARIANT
  bb ∈ NAT
INITIALISATION
  bb := 0
OPERATIONS
  vv ← get_bb =
   BEGIN vv := bb END ;
  set_bb(vi) =
   PRE   vv ∈ NAT
   THEN bb := vv
   END
END
```

Ici, le problème vient du fait que l'implémentation de mm réalise effectivement l'initialisation de *aa* comme spécifié, mais change la valeur de *bb* en même temps. Or la présence de la clause SEES dans mm impose la stabilité de la valeur de *bb* au cours d'une opération de mm. L'obligation de preuve démontrant que l'initialisation implantée respecte sa spécification va être juste ! En effet, vu de la spécification, *bb* est stable. Nous devons donc prouver que *aa* de l'implantation, devenu *bb* avant modification (implantation), correspond bien à *aa* dans la spécification, devenu égal à *bb* supposé stable (spécification). Ceci est vrai bien entendu, et pourtant l'initialisation implantée ne réalise pas sa spécification puisqu'à la sortie, *aa* et *bb* n'ont pas la même valeur.

Ce type de problème où les variables d'une machine vue ne sont pas stables pendant les opérations de la machine "voyante" est appelé *aliasing*. La particularité des aliasings

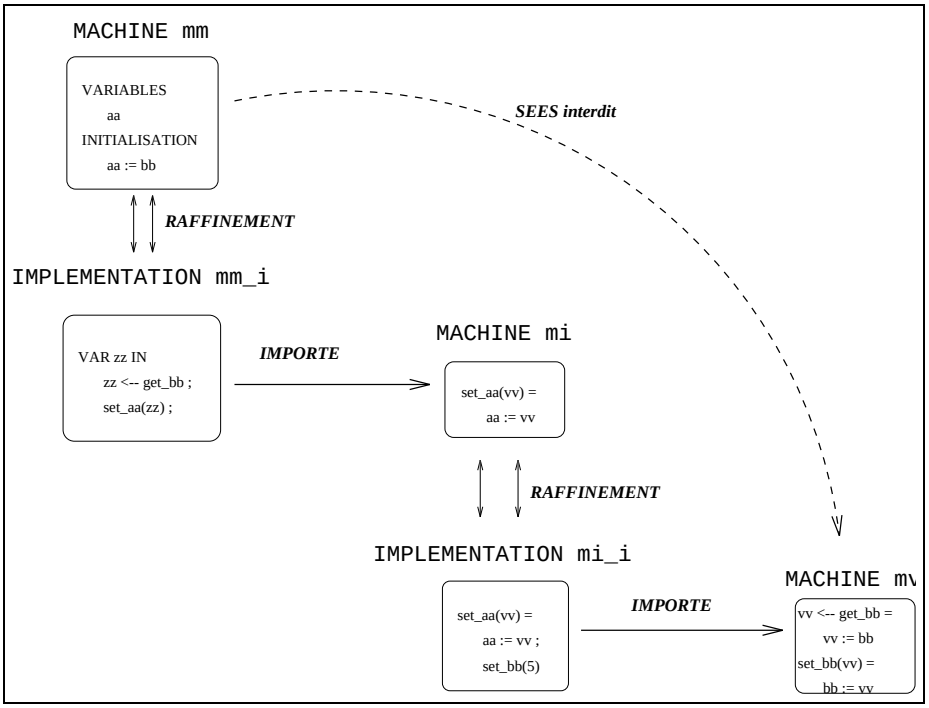


FIG. 4.1 – Un ancêtre ne peut pas faire de SEES

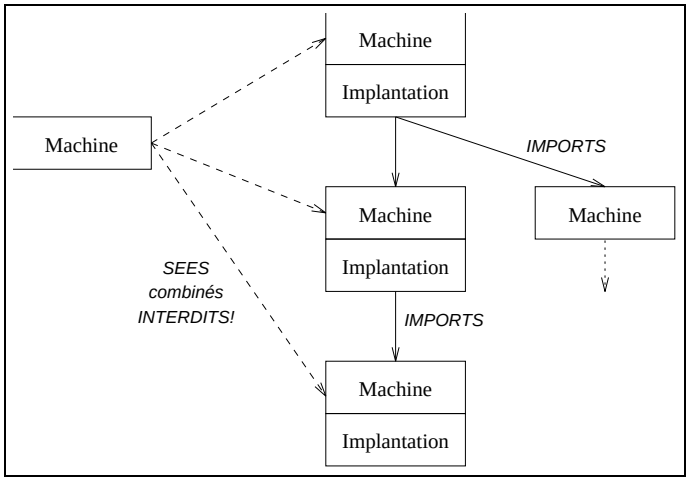
est que la preuve ne les détecte pas : il est nécessaire de faire un contrôle statique sur l'ensemble de l'architecture du projet concerné.

Dans l'Atelier B, comme nous l'avons déjà dit c'est le bouton *Project Check* qui doit être utilisé. Il est conseillé de faire cette vérification dès la création du squelette des composants, en particulier si l'architecture choisie est complexe : il ne faut pas développer développer de manière coûteuse des modélisations basées une architecture incorrecte !

4.5.4 La clause SEES est interdite sur un ancêtre

Cette règle est une généralisation de la précédente. En effet, si un ancêtre d'une machine M fait un `SEES` sur cette machine, alors on suppose dans la réalisation des opérations qu'aucune opération ne modifiant les variables de la machine vue ne sera appelée. Cela peut se produire, il suffit pour cela de cacher une opération de modification d'une variable de la machine vue dans une opération d'un raffinement, comme le montre la figure 4.1

Sur cette figure, nous voyons une machine `mm` dont la spécification est d'initialiser aa à la valeur de bb. Cette initialisation est obtenue en lisant la valeur de bb (opération `get_bb`) et en positionnant aa. Si dans l'implantation de l'opération `set_aa` il nous est possible de modifier la valeur de bb, alors la spécification initiale n'aura pas été satisfaite car en positionnant aa à la dernière valeur de bb, comme bb a changé nous n'avons pas établi $aa = bb$. Cette possibilité existe seulement si la machine `mv` est importée dans la descendance de `mm` : dans ce cas `mm` ne doit pas être autorisé à voir `mv`.



4.5.5 SEES sur la même branche interdits

Un autre cas d'aliasing se produit si un composant fait plusieurs **SEES** à différents niveaux d'une même branche d'implantation :

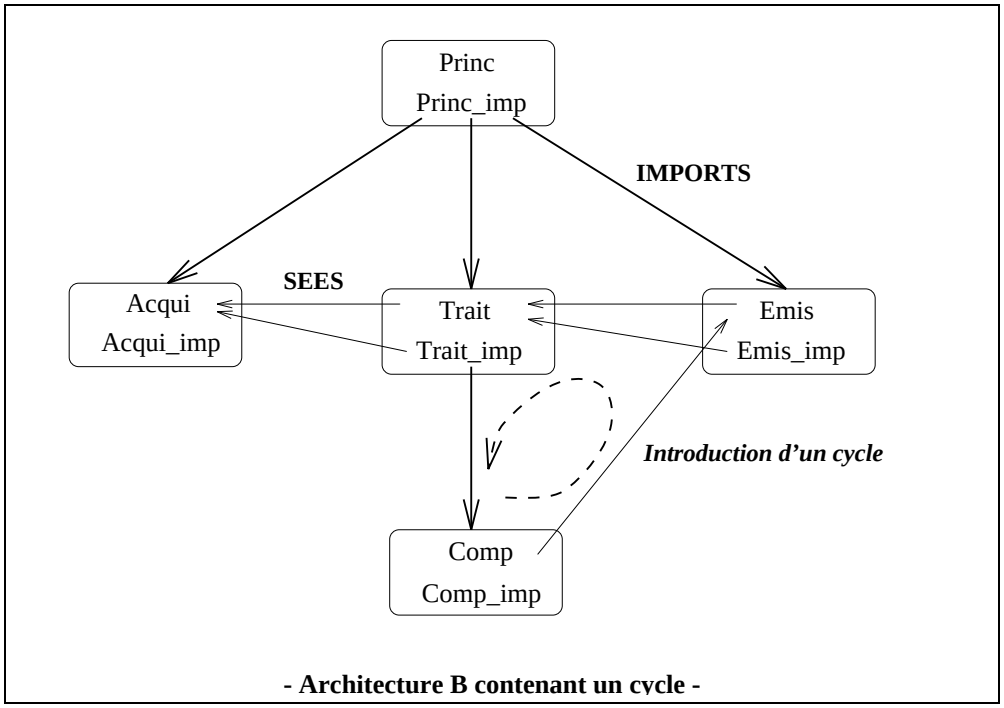
En effet, les données des différentes machines vues sont supposées stables pendant les opérations de la machine voyante. Or, ces opérations peuvent utiliser les opérations de consultation des machines vues. Si l'une des machines vues en importe une autre, alors il se peut qu'elle modifie les données de cette deuxième machine vue dans ses opérations. Ceci est vrai même si les opérations en question sont de simple consultations. Là encore, nous n'avons donc plus le droit de considérer que les variables vues sont stables. Une telle architecture est donc interdite.

Les règles que nous avons données dans ces trois derniers paragraphes décrivent les cas les plus courants d'*aliasing*, ces règles ne sont néanmoins pas exhaustives. En pratique, elles permettent d'éviter ces problèmes lors des choix d'architecture, la vérification complète étant faite par l'Atelier B.

4.5.6 La clause SEES doit être transversale à un composant

La clause `SEES` doit être transversale à un composant, i.e. si un composant voit une spécification, alors tous ses raffinements doivent aussi la voir. Cette règle a une justification très simple : si une spécification (machine abstraite) a été faite en fonction d'un module concret (**SEES** sur ce module) alors il est impossible que ses raffinements respectent cette spécification sans voir le même module concret. Par exemple, si la machine vue contient une constante et que la spécification d'une opération est d'initialiser une variable à une valeur inférieure à cette constante, il est impossible de réaliser cette opération sans accéder à la constante. En fait, s'il est possible de réaliser la spécification "voyante" sans voir la même machine, c'est qu'aucune des entités de la machine vue n'était utilisée : le **SEES** initial était superflu.

C'est le *Vérificateur de syntaxe et de type* ("Type Checker"), outil intégré à l'Atelier B, qui contrôle que l'ensemble des modules d'un projet respectent cette règle. Cette vérification est donc réalisée à chaque étape de construction du projet B.



- Architecture B contenant un cycle -

4.5.7 Le graphe des dépendances ne doit pas contenir de cycle

Le graphe des dépendances d'un projet est obtenu en reliant chaque machine du projet aux machines qu'elle voit ou importe.

Si une machine M dépend d'une machine N, alors N doit être initialisée avant M car M peut utiliser des variables de N lors de son initialisation.

L'initialisation correcte des éléments d'un projet est donc conditionnée par l'existence d'un ordre valide d'initialisation de ces éléments.

On voit que si le graphe de dépendance contient un cycle, alors un tel ordre ne peut être obtenu.

Ce contrôle est effectué automatiquement avant chaque action sur un projet ou sur un composant, de façon à détecter au plus tôt ce type de problème (qui remet en question l'architecture B du projet).

Tout doit donc être fait lors de la conception de l'architecture B pour qu'aucun cycle n'apparaisse. Notons qu'un cycle subsistant dans un développement B est le signe d'une définition mutuellement récursive de plusieurs entités (`M` dépend de `N`,`N` dépend de `O`, `O` dépend de `M`). Or, l'utilisation de définitions récursives est interdite en B afin de faciliter le processus de preuve et d'augmenter la sécurité du code produit. Les cycles doivent alors être éliminés par restructuration des entités mutuellement dépendantes.

L'exemple suivant illustre comment depuis une architecture B contenant un cycle, il est possible d'obtenir une architecture B "saine".

Il s'agit d'un processus cyclique dont l'architecture, spécifiée *a priori*, a dû être remise en cause *a posteriori*, lors du développement des divers composants B. `Princ` est le module principal, `Acqui` est le module d'acquisition de données, `Trait` est le module de traitement des données et `Emiss` est le module d'emission des résultats.

Dans cet exemple, le traitement des données est réalisé en fonction d'un état interne et des données reçues, et conduit au calcul d'un nouvel état interne et à l'émission de données calculées. Or, au cours du développement du module `Comp`, il s'est avéré nécessaire de connaître la valeur de cet état interne au cycle précédent. Cette valeur est connue par le module `Emis`, d'où le réflexe naturel d'effectuer un `SEES` sur ce module, ce qui introduit un cycle. Deux solutions n'introduisant pas de cycle sont possibles :

1. L'état interne étant connu dans le module `Trait`, il est possible de le transporter comme paramètre supplémentaire des opérations de `Comp`. Cette solution est cependant difficilement envisageable si plusieurs modules sont insérés entre `Comp` et `Trait`.
2. Le module `Princ` spécifie la trame des données émises et des données reçues, ainsi que le calcul, par l'intermédiaire des opérations du module `Trait`, permettant de lier ces données. Le nouvel état est calculé en fonction des données acquises, mais aussi de l'état précédent. Il est judicieux de voir l'état précédent comme une donnée acquise particulière, ce qui est possible en transmettant l'état précédent de `Princ` vers `Acqui`. Du point de vue de l'architecture B, cela oblige le module `Comp` à effectuer un `SEES` sur le module `Acqui`, ce qui n'introduit pas de cycle.

4.6 Ce que nous avons appris

- **IMPORTS** permet de réaliser une spécification à partir de sous spécifications.
- **INCLUDES** permet de décomposer une machine abstraite.
- **SEES** permet des spécifier en fonction d'une instance concrète d'un module.
- **PROMOTES** permet de déclarer les opérations de machines importées ou incluses qui doivent être des opérations de la machine utilisatrice.
- **EXTENDS** est une importation ou inclusion avec promotion de toutes les opérations.
- **USES** permet d'établir des liens entre deux machines incluses. Il est rarement employé.
- Découper la spécification complète d'un logiciel ne suffit pas pour pouvoir l'exprimer en B. Les propriétés à prouver devraient toujours voir l'ensemble des morceaux.
- En B, décomposer un problème c'est créer des spécifications plus abstraites que la spécification en langage naturel, pour dégager les pré-requis essentiels.
- La machine de plus haut niveau de la spécification d'un très gros logiciel n'est pas plus compliquée que celle d'un petit logiciel.
- Pour faire une telle décomposition en niveaux de détails, il faut souvent *réexprimer le besoin*.
- Le découpage d'une spécification par **INCLUDES** suppose la présence d'une machine de regroupement, de complexité équivalente à la spécification complète.
- Lors d'un découpage par **IMPORTS**, il est recommandé de faire remonter le minimum d'informations nécessaire pour prouver les propriétés principales demandée au niveau supérieur.
- Les éléments inclus dans une spécification (**INCLUDES**) peuvent être réalisés soit directement, soit par importation de la machine incluse (**INCLUDES**/**IMPORTS**).
- Lors de la programmation, on peut différer la réalisation de certains éléments groupables en les rejetant dans une machine importée (technique **IMPORTS**/**PROMOTES**).
- Une machine abstraite ne peut être importée qu'une seule fois dans un projet.
- Une machine vue doit être importée dans une implantation du projet.
- Un composant B ne peut être vu et importé dans la même branche.
- La clause SEES est interdite sur un ancêtre.
- Il est interdit de faire plusieurs SEES sur la même branche.
- La clause SEES doit être transversale à un composant.
- Le graphe des dépendances ne doit pas contenir de cycle.

Chapitre 5

Invariant de liaison

L'invariant d'un raffinement et l'invariant d'une implantation contiennent des prédicats de liaison entre les nouvelles variables (introduites dans le raffinement et dans l'implantation) et les variables abstraites du niveau précédent ; ces prédicats de liaison sont appelés *invariants de liaison*. Ils définissent la relation d'implantation entre les variables de la machine abstraite et les variables de l'implantation, ou entre les variables de la machine abstraite et les variables des instances des machines importées. Ce sont ces prédicats qui assurent la correction de développement[1] d'un module B.

MACHINE
Liaison
ABSTRACT_VARIABLES
ensemble
INVARIANT
$ensemble \subseteq NAT$
INITIALISATION
$ensemble := \varnothing$
END

REFINEMENT
Liaison_1
REFINES
Liaison
ABSTRACT_VARIABLES
maximum
INVARIANT
$maximum \in NAT \wedge$
/* invariant de liaison */
$maximum = max(ensemble \cap \{0\})$
INITIALISATION
$maximum := 0$

[1] La correction de développement d'un raffinement (respectivement d'une implantation) vis à vis du composant raffiné est vérifiée lorsque l'initialisation et les opérations du raffinement (respectivement de l'implantation) précisent (raffinent correctement) les opérations du composant raffiné.

Dans l'exemple précédent, l'invariant de liaison lie la variable du raffinement `maximum` avec la variable de la machine abstraite `ensemble`. L'objet de la vérification formelle est alors de démontrer que l'initialisation et les opérations du raffinement assurent la correction de développement du module.

Nous allons dans ce chapitre définir, tout d'abord, les notions d'invariants de liaison implicites ou explicites. Nous étudions, ensuite, les effets d'un oubli d'invariant de liaison. Enfin, nous proposons une méthode systématique de détection de ces problèmes de conception.

5.1 Invariant de liaison implicite ou explicite

Dans l'exemple précédent l'invariant de liaison est dit *explicite* puisqu'il lie deux variables portant des noms distincts. Dans le cas où les deux variables sont homonymes et de même type, l'invariant de liaison est *implicite* et n'apparaît donc pas dans la clause `INVARIANT`.

Dans l'exemple suivant nous introduisons une machine avec deux variables concrètes et deux variables abstraites, pour étudier des différentes combinaisons possibles d'invariant implicites ou explicites.

MACHINE
Invariant_Liaison
ABSTRACT_VARIABLES
vrb1, *vrb2*
CONCRETE_VARIABLES
vrb3, *vrb4*
INVARIANT
$vrb1 \in NAT \wedge$
$vrb2 \in BOOL \wedge$
$vrb3 \in INT \wedge$
$vrb4 \in INT$
INITIALISATION
$vrb1 :\in NAT \;||$
$vrb2 :\in BOOL \;||$
$vrb3 :\in INT \;||$
$vrb4 :\in INT$
END

Nous allons implanter ces variables de la manière suivante :
– *vrb1* est explicitement liée à une variable concrète de l'implantation ;
– *vrb2* est transformée en variable concrète ;
– *vrb3* déjà concrète, est conservée dans l'implantation ;
– *vrb4* elle aussi déjà concrète, est conservée également.
L'implantation est la suivante :

IMPLEMENTATION
Invariant_Liaison_imp
REFINES
Invariant_Liaison
CONCRETE_VARIABLES
vrb2, *vrb5*, *vrb6*
INVARIANT
$vrb2 \in BOOL \wedge$
$vrb5 \in NAT \wedge$
/* vrb6 est une variable locale à l'implémentation */
$vrb6 \in NAT \wedge$
/* invariant de liaison explicite */
$vrb5 = vrb1$
/* invariant de liaison implicite : transformation de vrb2 en variable concrète */
/* $vrb2 = vrb2$ */
/* invariant de liaison implicite : héritage des variables concrètes */
/* $vrb3 = vrb3 \ \wedge \ vrb4 = vrb4$ */
INITIALISATION
$vrb2 := TRUE$;
$vrb3 := 0$;
$vrb4 := 0$;
$vrb5 := 0$;
$vrb6 := 0$
END

L'exemple précédent montre des invariants de liaison entre les variables d'une implantation et les variables de la machine abstraite associée. Trois types d'invariant de liaison y sont présentés :

- *un invariant de liaison explicite* : (vrb5 = vrb1). Ce prédicat précise qu'il y a eu un changement de variable entre la machine abstraite et son implantation.
- *un invariant de liaison implicite* : (vrb2 = vrb2). La variable de la machine abstraite et celle de l'implantation sont homonymes et de même type. L'implantation précise seulement qu'il s'agit d'une variable concrète : elle peut être manipulée directement, en lecture et écriture, dans le corps de l'implantation. En fait, vrb2 à été transformée en variable concrète.
- *un invariant de liaison implicite pour l'héritage de variables* concrètes définies dans la machine abstraite : (vrb3 = vrb3 ∧ vrb4 = vrb4). Ces variables peuvent être manipulées, en lecture et en écriture, dans le corps de l'implantation. On est d'ailleurs obligé de les initialiser dans l'implantation.

Notons que la variable concrète `vrb5` est une variable locale à l'implantation ; aucun lien n'existe alors entre cette variable et une variable de la machine abstraite.

L'exemple précédent montre les différents invariants de liaison entre les variables d'une machine abstraite et les variables de son raffinement. Les invariants de liaison permettent également de lier les variables provenant de modules différents, et ce, grâce au mécanisme de l'importation.

Le principe est le suivant : il est toujours possible de réaliser l'une des variables de la spécification initiale en indiquant dans l'implantation que cette variable correspond à l'une des variables importées. S'il y a homonymie, cette correspondance est implicitement sup-

posée, sinon le lien doit être écrit dans l'invariant de l'implantation.

En ce qui concerne le type concret ou abstrait des variables, la liaison est autorisée seulement si la variable qui réalise est au moins aussi concrète que la variable réalisée. Ainsi il est possible de réaliser une variable abstraite par une concrète, mais pas de transformer une variable concrète en variable abstraite. En effet, les composants qui utilisent la machine sont autorisés à lire directement la valeur des variables concrètes sans passer par une opération, service qui ne pourrait plus être maintenu si ces variables concrètes étaient réalisées par des variables abstraites.

Signalons enfin que si une variable concrète apparaît dans l'initialisation de l'implantation, elle est considérée comme implantée directement : lors de la traduction de cette implantation, une variable informatique est créée. Cette variable ne doit donc pas correspondre à une variable importée.

Nous allons maintenant examiner sur un exemple ces liaisons avec des variables provenant de modules importés. Reprenons une machine avec quatre variables abstraites et concrètes comme précédemment :

MACHINE
Impl_Explicite
ABSTRACT_VARIABLES
$vrb1, vrb2$
CONCRETE_VARIABLES
$vrb3, vrb4$
INVARIANT
$vrb1 \in NAT \land$
$vrb2 \in BOOL \land$
$vrb3 \in INT \land$
$vrb4 \in INT$
INITIALISATION
$vrb1 :\in NAT \;||$
$vrb2 :\in BOOL \;||$
$vrb3 :\in INT \;||$
$vrb4 :\in INT$
END

Nous allons implanter ces variables de la manière suivante :
- *vrb1* est explicitement liée à une variable de la machine importée ;
- *vrb2* abstraite, est liée par homonymie avec la variable abstraite de même nom dans la machine importée ;
- *vrb3* concrète, est liée par homonymie avec la variable concrète de même nom dans la machine importée ; notons qu'il n'aurait pas été possible de la lier à une variable importée *abstraite*.
- *vrb4* déjà concrète, est conservée telle quelle dans l'implantation.

L'implantation est la suivante :

IMPLEMENTATION
Impl_Explicite_imp
REFINES
Impl_Explicite
IMPORTS
Composant
INVARIANT
/* invariant de liaison explicite */
vrb1 = imported_vrb
/* invariants implicites avec Composant */
/* *vrb2 = vrb2 ∧ vrb3 = vrb3* */
/* invariant implicite : héritage d'une variable concrète */
/* *vrb4 = vrb4* */
INITIALISATION
vrb4 := 0
END

Le composant importé possède une variable abstraite non homonyme, et deux variables (concrète et abstraite) reprenant des noms déjà utilisés :

MACHINE
Composant
ABSTRACT_VARIABLES
imported_vrb, vrb2
CONCRETE_VARIABLES
vrb3
INVARIANT
imported_vrb $\in NAT \wedge$
vrb2 $\in BOOL \wedge$
vrb3 $\in INT$
INITIALISATION
imported_vrb $:\in NAT \;||$
vrb2 $:\in BOOL \;||$
vrb3 $:\in INT$
END

L'exemple précédent montre les liens entre les variables d'une machine abstraite et les variables d'une machine abstraite importée.

Deux types d'invariant de liaison sont présentés :
– *un invariant de liaison explicite* (vrb1 = imported_vrb). Il y a un changement de variable entre la machine abstraite et son implantation, via la machine abstraite importée.
– *un invariant de liaison implicite* (vrb2 = vrb2 ∧ vrb3 = vrb3). Ces variables de la machine abstraite et les variables de la machine abstraite importée sont de même type et sont homonymes.
– *un invariant de liaison implicite* (vrb4 = vrb4). C'est l'héritage de la variable concrète vrb4, implantée automatiquement puisqu'elle ne correspond à aucune variable importée.

5.2 Défaut d'invariant sur un changement de variable

Les exemples suivants montrent comment l'absence d'invariant de liaison influe sur la véracité des obligations de preuve à démontrer.

Changement de variable par mécanisme d'importation Il s'agit de réaliser une variable de la machine abstraite par une variable d'une machine abstraite importée.

```
MACHINE
    Clash
ABSTRACT_VARIABLES
    my_var
INVARIANT
    my_var ∈ ℕ
INITIALISATION
    my_var := 0
OPERATIONS
    Set(vv) = PRE
        vv ∈ NAT
    THEN
        my_var := vv
    END ;
    uu ← Get = PRE
        my_var ≠ 0
    THEN
        uu := my_var
    END
END
```

```
IMPLEMENTATION
    Clash_imp
REFINES
    Clash
IMPORTS
    ImportedComposant
    /* liaison manquante */
OPERATIONS
    Set(vv) = BEGIN
        ImportedSET(vv)
    END ;
    uu ← Get = BEGIN
        uu ← ImportedGET
    END
END
```

MACHINE
ImportedComposant
ABSTRACT_VARIABLES
Other_var
INVARIANT
$Other\_var \in \mathbb{N}$
INITIALISATION
$Other\_var := 0$
OPERATIONS
ImportedSET(aa) = **PRE**
$aa \in NAT$
THEN
$Other\_var := aa$
END ;
$bb \leftarrow$ ImportedGET = **PRE**
$Other\_var \neq 0$
THEN
$bb := Other\_var$
END
END

La machine `Clash` qui encapsule la variable `my_var`, est ici implantée sur une machine équivalente `ImportedComponent`. Cela revient, en quelque sorte, à faire un changement de variable.

Pour le composant `Clash_imp`, une des obligations de preuve produite pour vérifier la correction de l'opération `Get` est la suivante :

`Proof obligation 2 of operation Get`

"'Included,imported and extended machines invariants'"
Other_var$1 ∈ ℕ∧
"'Previous components invariants'"
my_var ∈ ℕ∧
"'Get preconditions in previous components'"
not(my_var = 0) ∧
"'Get preconditions in this component'"
not(my_var = 0) ∧
"'Check that the invariant (uu$1 = uu) is preserved by the operation - ref 4.4, 5.5'"
"'Check operation refinement - ref 4.4, 5.5'"
⇒
Other_var$1 = my_var

L'objet de cette obligation de preuve est de vérifier que le corps de l'opération `Get` de l'implantation raffine le corps de l'opération de la machine abstraite. En particulier, les paramètres retournés doivent être égaux. Or, comme les paramètres retournés sont affectés par des variables, la vérification consiste à montrer l'égalité de ces variables ; c'est le sens du prédicat `Other_var$1 = my_var`.

Notons simplement que les hypothèses liées à ce but sont formées des invariants des machines abstraites `ImportedComponent` et `Clash`, de la précondition de l'opération `Get`.

Cette obligation de preuve est fausse puisqu'aucune hypothèse ne permet de déduire l'égalité recherchée.

En effet, bien qu'il soit clair ici, pour le concepteur, que l'implantation `Clash_imp` raffine la machine abstraite `Clash`, dans le cas général il faut indiquer au démonstrateur quel est le lien entre les variables de la machine et les variables de la machine importée.

On peut pour cela se fier aux initialisations : *l'état initial de la machine importée doit correspondre à celui de la machine, via le lien entre les variables*.

Ici, pour éliminer ces PO fausses il suffit de rajouter l'invariant de liaison `Other_var = my_var` dans l'implantation `Clash_imp`.

Remarque Si la variable de `ImportedComponent` s'appelle `my_var` au lieu de `Other_var`, alors l'égalité des noms des deux variables génère un invariant de liaison implicite qui résout, dans ce cas, ces problèmes.

Changement de variable dans un raffinement Il s'agit de raffiner une variable de la machine abstraite par une variable du raffinement. Dans l'exemple suivant, nous avons une variable `my_var`accessible par une opération de lecture et une opération d'écriture :

MACHINE
OtherClash
ABSTRACT_VARIABLES
my_var
INVARIANT
$my\_var \in \mathbb{N}$
INITIALISATION
$my\_var := 0$
OPERATIONS
Other_Set(aa) = **PRE**
$aa \in NAT$
THEN
$my\_var := aa$
END ;
$bb \leftarrow$ Other_Get = **PRE**
$my\_var \neq 0$
THEN
$bb := my\_var$
END
END

Dans le raffinement de cette machine, nous allons arbitrairement décider de changer le nom de la variable en la renommant `ref_var`. Cette nouvelle variable est sensée représenter la même chose que `my_var`.

Il est clair qu'un invariant de liaison manque dans l'invariant de ce raffinement :

REFINEMENT
OtherClash_ref
REFINES
OtherClash
ABSTRACT_VARIABLES
ref_var
INVARIANT
ref_var $\in \mathbb{N}$
/* liaison manquante */
INITIALISATION
ref_var := *0*
OPERATIONS
Other_Set(*aa*) = **PRE**
aa $\in$ *NAT*
THEN
ref_var := *aa*
END ;
bb $\leftarrow$ Other_Get = **PRE**
ref_var $\neq$ *0*
THEN
bb := *ref_var*
END
END

Une des obligations de preuve générée est :

ref_var$1 $\in N$
ref_var$1 <= 2147483646
myvar $\in N$
myvar <= 2147483646
not(myvar = 0)
$\Rightarrow$
myvar = ref_var$1

Cette obligation de preuve n'est pas démontrable car il manque l'invariant de liaison liant `myvar` et `ref_var` : `myvar = ref_var`.

De façon générale, lorsqu'on utilise dans un raffinement une variable ayant strictement le même rôle qu'une des variables de la machine, alors on utilise le même nom de variable entre la machine et son raffinement, et l'invariant de liaison est, de ce fait, implicite.

5.3 Dégradation de la preuve par défaut d'invariant de liaison

Un défaut d'invariant de liaison ne se détecte pas toujours par une obligation de preuve fausse. Il est même possible, à cause d'un défaut d'invariant de liaison, de prouver des raffinements clairement insuffisants.

Considérons l'exemple suivant :

```
MACHINE
    BlindClash
ABSTRACT_VARIABLES
    var_clash
INVARIANT
    var_clash ∈ BOOL
INITIALISATION
    var_clash := FALSE
OPERATIONS
    SET_TRUE = BEGIN
        var_clash := TRUE
    END
END
```

```
IMPLEMENTATION
    BlindClash_imp
REFINES
    BlindClash
CONCRETE_VARIABLES
    var
INVARIANT
    var ∈ BOOL
    /* liaison manquante */
INITIALISATION
    var := FALSE
OPERATIONS
    SET_TRUE = BEGIN
        var := FALSE
    END
END
```

Deux obligations de preuve triviales, c'est-à-dire déjà acquittées automatiquement par l'outil, sont produites pour vérifier la correction de l'opération de l'implantation. En effet, le corps de l'opération respecte effectivement l'invariant de l'implantation : la constante booléenne `FALSE` appartient à l'ensemble `BOOL`. Il ne s'agit pas d'une erreur de la méthode B ; cette dernière ne peut pas vérifier des propriétés qui ne sont pas exprimées. C'est une erreur de conception ; la pose des invariants de liaison est une activité suffisamment critique pour mériter toute l'attention de l'utilisateur.

Pour éviter de telles erreurs de conception, une solution est d'introduire *une opération de lecture*[2]des variables.

5.4 Détection des défauts d'invariant de liaison

Un défaut d'invariant de liaison peut ne pas être détecté par la phase de preuve. Pour éviter de tels problèmes, l'utilisateur introduira dés la machine abstraite des opérations

[2]Une opération de lecture est une opération avec des paramètres de retour constitués à partir des variables du composant.

de lecture des variables.

Une opération de lecture comporte des paramètres de retour, lesquels doivent être constitués à partir des variables de la machine abstraite. Lors du développement du module, les opérations des raffinements doivent être correctes ; en particulier, les paramètres de retour doivent être égaux (ils sont de même type et sont élaborés de la même façon) aux paramètres de retour de l'opération raffinée.

Il faut cependant rester très vigilant pour l'écriture du corps de l'opération de lecture dans la machine abstraite. En effet, dans l'exemple suivant, bien qu'une opération de lecture soit présente, le défaut d'invariant n'est pas détecté. Dans cet exemple, la machine possède une variable *vrb* qui peut être affectée par une opération `SET_TRUE` et "apparemment" accédée par une opération de lecture :

MACHINE
Exemple
ABSTRACT_VARIABLES
vrb
INVARIANT
$vrb \in BOOL$
INITIALISATION
$vrb := FALSE$
OPERATIONS
SET_TRUE = **BEGIN**
$vrb := TRUE$
END ;
$res \leftarrow$ OpLecture = **BEGIN**
$res :\in \{vrb\} \cup BOOL$
END
END

À première vue, l'opération de lecture qui porte sur *vrb* devrait nous protéger de tout défaut d'invariant de liaison sur cette variable. En fait, la présence textuelle de *vrb* dans le corps de l'opération de lecture ne suffit pas pour garantir que la variable est bien lue. Dans cet exemple le corps de l'opération `OpLecture` n'introduit en fait aucun lien entre `res` et la variable abstraite `vrb`, car on remarque l'égalité suivante :

$$\{\text{vrb}\} \cup \text{BOOL} = \text{BOOL}$$

Le corps de notre opération est en fait une expression mathématique non simplifiée. Après simplification, l'opération `OpLecture` exige simplement que la variable de retour soit un booléen, sans lien avec *vrb*.

Dans ce cas, l'implantation suivante est valide injustement.

```
IMPLEMENTATION
    Exemple_imp
REFINES
    Exemple
CONCRETE_VARIABLES
    vrb_imp
INVARIANT
    vrb_imp ∈ BOOL
    /* défaut d'invariant de liaison */
    /* il faudrait vrb = vrb_imp */
INITIALISATION
    vrb_imp := FALSE
OPERATIONS
    SET_TRUE = BEGIN
        vrb_imp := FALSE /* au lieu de TRUE */
    END ;
    res ← OpLecture = BEGIN
        res := vrb_imp
    END
END
```

Le corps de l'opération de lecture de la spécification est trop lâche ; il ne permet pas de détecter le défaut d'invariant de liaison. L'initialisation et l'opération de lecture de l'implantation sont prouvées.

5.5 Ce que nous avons appris

- Les invariants de liaison définissent les liens entre les variables de niveaux différents :
  - Lien entre les variables abstraites et les variables du premier raffinement ;
  - Lien entre les variables d'un raffinement et celles du raffinement suivant ;
  - Lien entre les variables du dernier raffinement et les variables importées dans une implantation.
- Les invariants de liaison peuvent être :
  - Explicites : écrits par l'opérateur dans le composant inférieur ;
  - Implicites : si les variables raffinées portent le même nom.
- En l'absence d'invariant de liaison :
  - S'il y a des opérations qui retournent des résultats liés aux variables, cela produit des PO indémontrables ;
  - Sinon : la preuve est juste mais dégradée : elle ne garantit pas la correspondance avec la spécification, car cette correspondance n'a pas été définie.
- Pour se protéger des défauts d'invariant de liaison : munir les machines abstraites d'opérations de lecture permettant d'avoir une visibilité sur les variables abstraites. Le raffinement de ces opérations sera faux si l'invariant de liaison est insuffisant.

Chapitre 6

Construction de boucles en B

Parmi les substitutions généralisées du langage B, la boucle **WHILE** est la plus difficile à modéliser. C'est pourquoi il est nécessaire de donner des techniques particulières de modélisation, de façon à assister le concepteur B.

6.1 Introduction

Que ce soit en B ou dans un langage comme C, Pascal ou Ada, une boucle est toujours précédée d'une phase d'initialisation des variables manipulées dans le corps de la boucle. Puis, la condition d'arrêt est indiquée, avant ou après (`repeat .. until` de Pascal) le corps de la boucle.

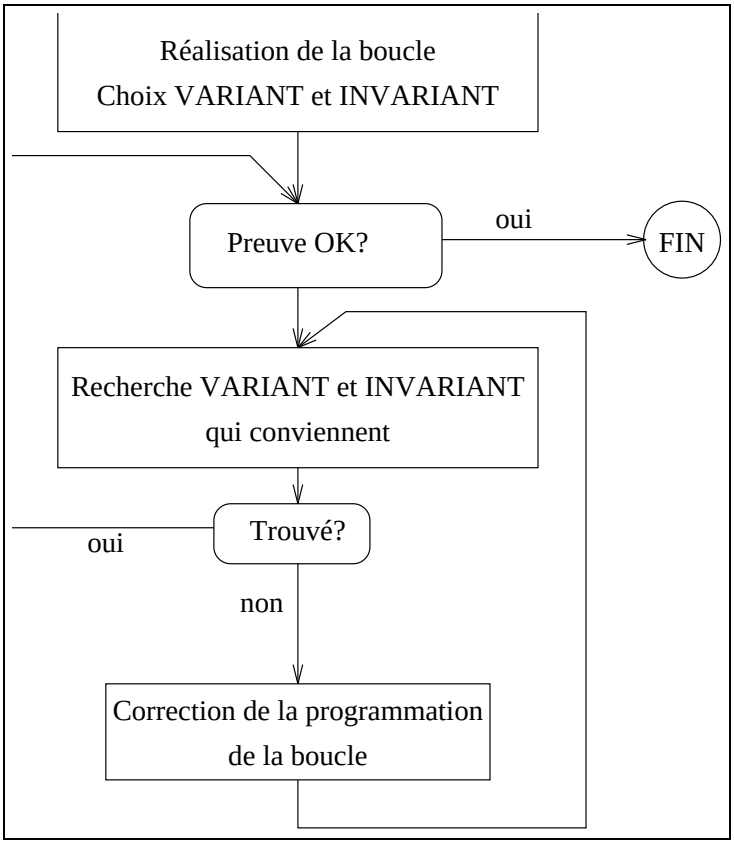
La partie **VARIANT**/**INVARIANT** de la boucle **WHILE** en B n'existe pas dans les langages de programmation. Elle est utilisée pour raisonner sur la boucle. En particulier, cette partie sert à démontrer :

- la convergence (terminaison) de la boucle : l'expression contenue dans le **VARIANT** doit décroître et être toujours positive, ce qui montre que la boucle se termine en un nombre d'itérations fini.
- la validité des substitutions utilisées dans le corps de la boucle : ces substitutions doivent être applicables à chaque itération de boucle.
- le bon comportement de la boucle : le résultat obtenu à la sortie de la boucle est celui attendu.

Réaliser une boucle en B, c'est donc non seulement programmer cette boucle mais également trouver un variant qui permettra de démontrer que la boucle se termine et un invariant qui permettra de démontrer qu'elle réalise bien la spécification.

Remarquons que le choix du variant et de l'invariant constitue un problème de deuxième niveau : une boucle peut être parfaitement juste et réaliser sa spécification mais produire des obligations de preuve fausses parce que l'invariant ou le variant ont été mal choisis. C'est là la difficulté de la boucle en B : les PO fausses ne détectent pas forcément une erreur dans la boucle. En fait, la mise au point d'une boucle se fait de la manière suivante :

Il est clair qu'il faut préparer chaque étape, ne pas aller trop vite pour éviter de "boucler" sur la réalisation d'une boucle. La recherche du variant est souvent simple, la plupart des problèmes sont rencontrés lors de la recherche de l'invariant. Nous allons examiner en détail ces recherches, en commençant par la plus difficile : l'invariant.



Attention : lors de la recherche des variant et invariant d'une boucle, toujours supposer que la boucle peut être fausse. En effet, il n'est pas rare de perdre beaucoup de temps sur la recherche d'un invariant pour s'apercevoir finalement que la boucle est fausse : il faut alors reprendre le processus au début.

Rappel : les boucles ne sont autorisées que dans les implantations. Ceci évite d'avoir des boucles spécifiant d'autres boucles, ce qui poserait des problèmes de preuves. D'autre part, il est déconseillé de faire de boucles imbriquées. La boucle interne doit être rejetée dans une machine importée.

6.2 Construction de l'invariant de boucle

L'invariant d'une boucle est une proposition logique que l'opérateur doit fournir avec la boucle, choisie de manière à vérifier certaines propriétés. Pour exposer ces propriétés de manière plus compréhensible, nous allons les voir sur un exemple. Considérons la boucle suivante, qui copie la variable *v1* dans *v2* puis l'incrémente trois fois pour fabriquer $v2 = v1 + 3$: cette boucle réalise donc la spécification $v2 := v1 + 3$.

```
op = ...
    index := 0;
    v2 := v1;
    WHILE index < 3 DO
        v2 := v2 + 1;
        index := index + 1
    INVARIANT
        index : 0 .. 3 ∧
        v2 = v1 + index
    ...
```

Les propriétés qu'un invariant de boucle doit satisfaire sont les suivantes :

1. L'invariant doit être vrai à l'entrée de la boucle, c'est-à-dire juste avant le premier passage (dans notre exemple : avec $index = 0$ et $v2 = v1$, l'invariant doit être établi).

2. Si l'invariant est vérifié au pas précédent, et si la condition de continuation de la boucle est vraie, alors le corps de boucle doit conserver cet invariant (si $index$ appartient à $0 \,..\, 3$ et $v2 = v1 + index$, et si $index < 3$ alors l'invariant doit encore être vrai si on remplace $v2$ par $v2 + 1$ et $index$ par $index + 1$).

3. Si l'invariant est vrai, et si la condition de sortie de boucle est vraie, alors on doit pouvoir en déduire la propriété voulue en spécification (de l'invariant et de la condition $index \geq 3$ on doit pouvoir déduire $v2 = v1 + 3$).

La première propriété doit être vraie dans le contexte du composant, mais les deux dernières doivent être vraies *pour toute variable vérifiant l'invariant* de la boucle. En effet dès que la boucle a commencé on ne sait plus rien sur les variables en cours de modification, sauf l'invariant de boucle qui est vrai à chaque pas. Ceci explique la présence de variables différenciées (généralement indicées par un dollar et un 2) dans les obligations de preuve de boucle.

Pour vérifier tout ceci, un invariant de boucle doit être composé :
– des conjoints[1] de typage des variables affectées par le corps de la boucle,
– des conjoints reprenant une partie de l'invariant du composant. Cette partie est satisfaite à chaque itération de boucle (c'est le cas particulier des invariants de liaison, par exemple).
– des conjoints contenant les préconditions des opérations appelées dans le corps de la boucle (ou des prédicats à partir desquels il est possible de déduire ces préconditions).
– des conjoints exprimant les propriétés respectées à chaque itération de la boucle ; il s'agit de l'algorithme de la boucle.

Plusieurs méthodes peuvent être employées pour faciliter la construction de l'invariant de la boucle. Dans tous les cas, l'important est de ne jamais perdre de vue les trois conditions fondamentales que doit vérifier cet invariant.

[1] Un *conjoint* est un élément d'une conjonction ; par exemple A est l'un des conjoints de l'expression $A \wedge B$.

6.2.1 Construction avec ajout de raffinement

Cette méthode consiste à répartir la difficulté en ajoutant un raffinement intermédiaire entre la boucle et sa spécification. L'hypothèse de départ est que la machine abstraite modélise un algorithme de boucle de très haut niveau. Si la substitution décrite dans la machine abstraite est trop synthétique, un raffinement doit être introduit pour modéliser le comportement de la boucle sur l'ensemble des itérations de manière plus explicite, par une substitution contenant une formule quantifiée.

Cette substitution est alors reprise telle quelle dans l'invariant de boucle, en remplaçant les constantes de la substitution du raffinement par l'indice de boucle de l'implantation. La plus grande partie de l'**INVARIANT** de boucle est formalisée, la preuve de la boucle **WHILE** validera cet invariant.

Trois étapes doivent être distinguées :
– Tout d'abord, exprimer l'algorithme de la boucle (introduire une étape de raffinement),
– puis, reprendre cet algorithme et remplacer certaines constantes par l'indice de boucle,
– enfin, la preuve permet de modifier si nécessaire l'**INVARIANT** de boucle.

Exemple 1 : déterminer le maximum d'un tableau

Soit à écrire un algorithme de calcul du maximum des éléments d'un tableau. La machine abstraite spécifiant l'opération de calcul de ce maximum peut être modélisée de la façon suivante :

MACHINE
Exemple1
CONCRETE_VARIABLES
tableau
INVARIANT
$tableau \in 0 \mathinner{..} N \rightarrow \mathsf{NAT}$
INITIALISATION
$tableau := (0 \mathinner{..} N) \times \{0\}$
OPERATIONS
$maxi \leftarrow$ DonneMaximum =
BEGIN
$maxi := \max(ran(tableau))$
END
END

N étant un nombre strictement positif, non précisé ici. Pour être plus rigoureux cet identifiant devrait être déclaré comme une constante de la machine ou un paramètre formel.

L'expression `ran(tableau)` représente l'ensemble des valeurs manipulées par le tableau (codomaine), l'expression `max(ran(tableau))` détermine le maximum de cet ensemble. Notons que cet ensemble de valeurs manipulées ne peut pas être vide : comme il s'agit d'une fonction totale, tout élément du tableau a une valeur. Il y a donc au moins une valeur manipulée.

Bien entendu, comme nous avons initialisé notre fonction totale à zéro et qu'il n'y a pas d'opération pour la modifier, le résultat de l'opération `DonneMaximum` sera toujours zéro.

Ceci n'est qu'un exemple, dans une véritable machine la fonction serait modifiable.

L'algorithme de la boucle n'est pas modélisé dans la machine abstraite ; pour cela nous écrivons un raffinement algorithmique (seule l'opération est raffinée). Le raffinement associé est le suivant :

REFINEMENT
Exemple1_ref
REFINES
Exemple1
INITIALISATION
tableau := $(0\,..\,N) \times \{0\}$
OPERATIONS
maxi ← DonneMaximum =
BEGIN
maxi :(maxi ∈ ran(tableau) ∧
∀ indice.(indice ∈ 0..N
⇒ tableau(indice) ≤ maxi))
END
END

Ce raffinement précise l'algorithme qui sera finalement implanté dans la boucle. L'algorithme consiste bien entendu à parcourir tous les éléments du tableau, en conservant la valeur visitée en tant que maximum provisoire si elle est supérieure à la valeur actuelle de ce maximum provisoire. Ce parcours est déjà annoncé par la formule quantifiée du raffinement.

Dans ce raffinement, la substitution utilisée est du type « devient tel que » dont le prédicat est la conjonction de deux expressions logiques :
– le premier prédicat précise que le maximum retourné par l'opération correspond à une valeur du tableau,
– le second prédicat précise que quelque soit l'indice du tableau, la valeur associée à cet indice est inférieure ou égale au maximum retourné.

Nous pouvons donner une lecture en français de l'opération raffinée : la variable *maxi* retournée est telle que *maxi* soit dans le domaine d'arrivée du tableau, et que pour tout indice compris entre 0 et N la valeur du tableau à cet indice soit inférieure à *maxi*. Il s'agit en fait d'une expansion de la notion de maximum.

L'invariant de boucle est obtenu en remplaçant la constante N contenue dans la substitution du raffinement par l'indice de boucle. C'est la transformation de l'algorithme de boucle, car à chaque étape la variable *maxi* en cours de calcul est bien le maximum de la partie déjà parcourue du tableau. L'invariant de boucle comporte donc les deux prédicats suivants :

pas ∈ 0 .. N ∧

∀ indice.(indice ∈ 0..pas ⇒ tableau(indice) ≤ maxi)

où `pas` est l'incrément de boucle.

L'implantation associée à l'exemple précédent est de la forme :

```
IMPLEMENTATION
    Exemple1_imp
REFINES
    Exemple1_ref
INITIALISATION
    tableau := (0 .. N) × {0}
OPERATIONS
    maxi ← DonneMaximum =
    VAR pas, maxi_loc IN
        /* pas : indice de la boucle max_loc : maximum déterminé à chaque pas
        val : valeur du tableau pour un indice */
        /* Initialisation des variables locales : elles doivent établir l'invariant de la boucle
        */
        pas := 0 ;
        maxi_loc := tableau(pas) ;
        WHILE (pas < N) DO
            VAR var_loc IN
                pas := pas + 1 ;
                var_loc := tableau(pas)
                /* Recherche du maximum */
                IF (maxi_loc ≤ var_loc) THEN
                    maxi_loc := var_loc
                END
            END
        INVARIANT
            /* Prédicats de typage */
            pas ∈ 0 .. N∧
            maxi_loc ∈ ran(tableau) ∧
            /* Propriété sur le maximum local */
            ∀indice.(indice ∈ 0..pas
                ⇒ tableau(indice) ≤ maxi_loc)
        VARIANT
            N - pas
        END ;
        maxi := max_loc
    END
END
```

Dans cette implantation `pas` est l'incrément de boucle, `var_loc` est la valeur du tableau à la position `pas`, `max_loc` représente le maximum obtenu en parcourant le tableau de `0` à `pas`. Notons que `var_loc` est volontairement déclarée à l'intérieur de la boucle pour éviter d'avoir à la caractériser dans l'invariant.

Cet invariant vérifie bien les trois propriétés fondamentales : il est vérifié à l'entrée de la boucle, il est conservé par le corps de boucle et lorsque la condition de sortie est vraie, il implique la propriété voulue en spécification.

L'invariant de boucle pourrait également être formalisé de la façon suivante :

INVARIANT
pas $\in 0..$ pas $\wedge$
maxi_loc = max(ran((0.. pas) $\lhd$ tableau))

C'est-à-dire que *maxi_loc* est le maximum de l'ensemble des valeurs de la restriction du tableau à *0 .. pas*. Nous avons simplement éliminé la formule quantifiée en utilisant une restriction. Les deux formes sont mathématiquement équivalentes ; dans ce cas précis la forme quantifiée est sans doutes préférable pour faciliter le travail des démonstrateurs automatiques car elle est plus proche des instructions du corps de la boucle.

D'autre part, le raffinement que nous avons écrit sert principalement à introduire cette formule quantifiée. Il devient ainsi très facile de démontrer qu'à la sortie de la boucle, la condition demandée dans le raffinement est établie : cette condition coïncide alors textuellement avec l'invariant de boucle.

Nous voyons que le choix du degré de simplification des expressions mathématiques conditionne la difficulté de preuve. D'une manière générale, il n'est pas utile de chercher les formes les plus synthétiques pour les formules qui apparaissent dans des raffinements ou des invariants de boucle : ces formules ne sont pas lues par les utilisateurs de la machine.

Exemple 2 : copie de tableaux

Il s'agit de modéliser une opération qui copie un tableau (paramètre d'entrée de l'opération) dans une variable de type tableau. Il faut, pour cela, que les deux tableaux manipulent des données de même type ; nous supposons, pour cet exemple, que le tableau à copier n'affecte que la moitié des indices de la variable `tableau`.

La machine abstraite de cette modélisation est :

MACHINE
Exemple2
CONCRETE_VARIABLES
tableau
INVARIANT
tableau $\in 0..N \rightarrow$ NAT
INITIALISATION
tableau := $(0..N)\times\{0\}$
OPERATIONS
DonneMaximum(tab_a_copier) =
PRE
tab_a_copier $\in N/2..N \rightarrow$ NAT
THEN
tableau := tableau $\lhd\!\!\!-$ tab_a_copier
END
END

La substitution décrite dans le corps de l'opération formalise la copie des tableaux. Un raffinement est introduit pour décrire plus précisément l'algorithme qui sera finalement implanté :

REFINEMENT
Exemple2_ref
REFINES
Exemple2
INITIALISATION
tableau := $(0\,..\,N)\times\{0\}$
OPERATIONS
DonneMaximum(tab_a_copier) =
LET tableau_0 **BE**
tableau_0 = tableau
IN
tableau :(tableau $\in 0\,..\,N \rightarrow \mathsf{NAT}\ \wedge$
$(0..(N/2-1)) \lhd tableau = (0\,..\,(N/2-1)) \lhd tableau\_0\ \wedge$
$(N/2\,..\,N) \lhd tableau = tab\_a\_copier$)
END
END

La substitution `LET...BE...IN...END` précise le comportement de l'opération en isolant la partie du tableau qui n'est pas modifiée et celle qui est remplacée par le tableau à copier. Elle évite l'utilisation d'une variable en $0 à ce niveau.

Il suffit maintenant de remplacer l'indice maximum `N` du tableau par la valeur de l'itération de la boucle ; en effet, c'est à partir de l'indice `N/2` et jusqu'à l'indice `N` que la variable `tableau` est modifiée par le paramètre d'entrée `tab_a_copier`.

Une implantation possible de ce composant est la suivante :

IMPLEMENTATION
Exemple2_imp
REFINES
Exemple2_ref
INITIALISATION
tableau := $(0\,..\,N)\times\{0\}$
OPERATIONS
DonneMaximum(tab_a_copier) =
VAR pas **IN**
pas := $N/2-1$;
WHILE (pas $< N$) **DO**
pas := pas +1;
tableau(pas) := tab_a_copier(pas)
INVARIANT
pas $\in N/2-1\,..\,N\wedge$
tableau $\in 0\,..\,N \rightarrow \mathsf{NAT}\ \wedge$
$(0\,..\,(N/2-1)) \lhd tableau = (0\,..\,(N/2-1)) \lhd tableau\$0\ \wedge$
$(N/2..\ \text{pas}\ ) \lhd tableau = (N/2\,..\,pas) \lhd tab\_a\_copier$
VARIANT
$N-$ pas
END
END

L'expression `tableau$0` représente la variable `tableau` avant l'opération ; ceci permet

d'exprimer la loi d'évolution du tableau manipulé.

6.2.2 Construction itérative

La grosse difficulté dans la recherche d'un invariant de boucle, c'est de trouver la *propriété de récurrence* qui la caractérise suffisamment. Cette propriété est telle que l'on doit pouvoir démontrer facilement qu'elle est conservée à chaque itération et qu'elle implique la propriété voulue en spécification à la sortie de la boucle. Nous avons appris comment isoler cette phase dans un raffinement intermédiaire si besoin est, nous allons maintenant creuser ce problème. La méthode proposée est la suivante :

1. Écrire la spécification de la boucle ;
2. Programmer la boucle, c'est-à-dire trouver la séquence d'itérations permettant d'obtenir le résultat ;
3. De cette programmation, exprimer les variables qui varient dans la boucle sous forme d'une suite définie par récurrence :

   $$var_n : var_0 = \ldots \text{ (état initial)}$$
   $$var_{n+1} = f(var_n)$$

4. De cette formulation, chercher à exprimer mathématiquement l'état des variables à n'importe quel pas : $var_n = f(n)$. De cette formule, on déduit la formule de récurrence cherchée.

Cette méthode ne prétend pas résoudre la difficulté dans tous les cas, mais donner un cadre de réflexion qui facilite cette résolution.

6.2.3 Construction avec le prouveur

Cette méthode de fabrication consiste à utiliser le prouveur pour voir les prédicats qui manquent dans l'invariant d'une boucle. On commence avec un invariant minimal (par exemple : `btrue`), et on examine les obligations de preuve. Un certain nombre de ces obligations seront fausses par manque d'hypothèses. Ces hypothèses manquantes sont alors ajoutées dans l'invariant, et le cycle recommence. Toute la difficulté et que ces prédicats ajoutés dans l'invariant doivent être tels que le corps de boucle les conserve, il faut donc les choisir dans ce sens lors de l'examen des hypothèses manquantes.

Avec ce procédé la présence du prouveur est mise à profit, et les invariants constitués se prouvent souvent facilement car ils sont construits à partir du prouveur. Par contre, ceci ne résoud pas le problème du choix de la formule de récurrence, qui apparaît avec la recherche des hypothèses manquantes dans les PO de sortie de boucle. Le but est alors la propriété voulue en spécification. Si une hypothèse proche de ce but est ajoutée, le problème se déplace sur la PO de conservation de cette hypothèse par le corps de boucle. On ajoute alors une autre hypothèse plus proche de la boucle, dont on devra démontrer la conservation, et ainsi de suite.

6.2.4 Conseils et pièges

Nous allons maintenant voir quelques pièges qu'il faut éviter lors de l'écriture d'un invariant de boucle, ainsi que quelques conseils généraux, classés par ordre d'importance.

Utiliser des variables locales à la boucle : si des variables intermédiaires sont nécessaires pour conduire les calculs à l'intérieur de la boucle, il est conseillé de les déclarer *à l'intérieur de la boucle*. Ainsi localisées, elles n'existent pas au niveau de l'invariant ce qui simplifie la preuve. Nous avons employé cette technique dans l'exemple *Exemple1_imp* précédent, éliminant ainsi deux prédicats dans l'invariant.

Faire des essais d'ordre : les problèmes de commutativité et d'associativité sont fréquents dans les preuves de boucle, à cause des manipulations d'indice. Le fait d'écrire les formules contenant des additions et des soustractions dans un ordre ou dans un autre peut changer les performances du prouveur du simple au double sur une boucle ! Il n'y a pas de règle pour trouver l'ordre optimal, nous conseillons de faire plusieurs essais. L'expérience montre que c'est préférable plutôt que de passer du temps en preuve interactive.

Invariant implicatif si la boucle contient des cas : si le corps de la boucle contient des IF, l'évolution sera différente suivant les branches sélectionnées. Il est alors souvent nécessaire d'utiliser des implications dans l'invariant de boucle. Nous citons ce cas car l'opérateur peut être bloqué s'il n'a pas l'idée d'utiliser ces implications.

Cas où la boucle n'a pas lieu : ce cas se produit pour des boucles dont le nombre d'itérations est variable, par exemple en fonction d'un paramètre d'entrée. Pour certaines valeurs, la boucle n'a aucune itération ; par exemple une boucle de *0* à *param* n'a pas lieu si *param* est négatif.

La difficulté est alors de ne pas oublier que l'invariant doit être vrai à l'entrée de la boucle, *même si celle-ci n'a pas lieu*. En reprenant l'exemple précédent, l'invariant $i \in 0 \,..\, param$ (i étant l'indice de boucle) n'est pas correct : il faut mettre $i \in 0 \,..\, param \cup \{param\}$ pour le cas où *param* est négatif.

Débordements de dernier tour : il n'est pas rare d'écrire des boucles dont l'indice peut potentiellement atteindre des valeurs proches de **MAXINT** en fonction des paramètres. Il peut alors y avoir un débordement lors de la dernière incrémentation de l'indice si la boucle incrémente à la sortie. Il faut donc que le paramétrage qui conditionne la boucle soit tel que l'indice de boucle s'arrête avant de dépasser **MAXINT**. Par exemple, pour réaliser une boucle simple qui copie une valeur dans un tableau de taille paramétrable, on est conduit à limiter cette taille à **MAXINT**-1.

6.3 Construction du variant de boucle

Le **VARIANT** de la boucle est plus simple à formaliser que l'**INVARIANT**. Le **VARIANT** permet de démontrer que la boucle converge, c'est-à-dire qu'elle se termine en un nombre d'itérations fini. Le **VARIANT** est choisi en fonction de deux critères :
- il doit être *strictement décroissant* à chaque itération de boucle,
- il doit rester *positif ou nul* même après le dernier pas de boucle, lorsque la condition du **WHILE** est devenue fausse.

Notons que le VARIANT n'est pas une grandeur informatique et ne doit donc pas rester inférieur à **MAXINT** : si V est un variant correct pour la boucle, alors $V + N$ où N est positif l'est aussi. Donc si l'opérateur a trouvé une quantité strictement décroissante, mais qui ne reste pas positive, il lui suffit de rajouter une constante pour obtenir un variant valide.

Dans les exemples précédents, le **VARIANT** est très simple ; l'algorithme de la boucle consistant uniquement à parcourir un intervalle d'entiers, il est constitué de l'indice de boucle et de la valeur maximale de l'indice de boucle.

Parfois, l'algorithme de la boucle ne nécessite pas un parcours de l'ensemble des indices, mais une condition de sortie (plus complexe) indique la terminaison des itérations. Dans ces cas également le **VARIANT** doit décroître strictement entre deux itérations, et l'itération de sortie doit vérifier cette condition. Il faut alors faire intervenir d'autres quantités que l'indice dans le variant. Ce cas sera illustré par l'exemple du paragraphe 6.4.3.

Pour construire efficacement un **VARIANT** de boucle, nous conseillons la méthode suivante :

- Rechercher ce qui varie à chaque itération, dans les différents cas ;
- Vérifier que les quantités choisies sont bien connues avec des hypothèses suffisamment fortes (en particulier dans l'invariant de la boucle) ;
- Rassembler ces cas en créant une quantité qui varie dans *tous* les cas ;
- Ajouter des nombres suffisamment grands pour que le variant ainsi construit reste positif.

6.4 Différents styles

Considérons un composant de gestion d'un tableau et une opération testant la présence d'un élément particulier dans le tableau (cette opération retourne une variable booléenne). Nous allons étudier différents styles de réalisation d'une telle boucle.

La machine abstraite modélisant cet exemple est de la forme :

MACHINE
Exemple3
CONCRETE_VARIABLES
tableau
INVARIANT
$tableau \in 0 \,..\, 100 \rightarrow \mathsf{NAT}$
INITIALISATION
$tableau := (0 \,..\, 100) \times \{0\}$
OPERATIONS
present ← est_present(element) =
PRE
$element \in \mathsf{NAT}$
THEN
present := bool(element ∈ ran(tableau))
END
END

6.4.1 Pré-incrémentation avec indicateur de fin

Une solution possible est d'écrire une boucle où l'incrémentation de l'indice est à l'entrée de la boucle (pré-incrémentation), avec un indicateur booléen indiquant si l'élément est trouvé. La boucle se poursuit tant que l'index est strictement inférieur à la borne et que

l'indicateur n'est pas positionné.

```
IMPLEMENTATION
    Exemple3_imp
REFINES
    Exemple3
INITIALISATION
    tableau := (0 .. 100) × {0}
OPERATIONS
    present ← est_present(element) = VAR trouve, pas IN
        trouve := FALSE;
        pas := −1;
        WHILE pas ≤ 99 ∧ trouve = FALSE DO
            pas := pas + 1;
            VAR vloc IN
                vloc := tableau(pas);
                IF element = vloc THEN
                    trouve := TRUE
                END
            END
        INVARIANT
            pas ∈ −1 .. 100 ∧
            trouve = bool(element ∈ ran(0 .. pas ◁ tableau))
        VARIANT
            100 − pas
        END;
        present := trouve
    END
END
```

Ainsi écrite, cette boucle se démontre assez facilement. Notez en particulier la déclaration de *var_loc* qui permet de ne pas avoir cette variable dans l'invariant. La formule de récurrence est donnée par l'égalité qui définit *trouve* dans la boucle. La similitude entre cette formule et la spécification facilite la preuve.

6.4.2 Optimisation : élimination de l'indicateur

Le test de continuation de la boucle précédente est $pas \leq 99 \ \wedge \ trouve = FALSE$. Il contient deux éléments, ce qui n'est pas en faveur de la rapidité de la boucle. En fait, il vaut mieux terminer la boucle en mettant artificiellement l'indice de boucle à sa valeur maximale : ainsi le test ne contiendra plus qu'un élément. La boucle est la suivante :

```
IMPLEMENTATION
    Exemple3_imp
REFINES
    Exemple3
INITIALISATION
    tableau := (0 .. 100) × {0}
OPERATIONS
    present ← est_present(element) = VAR pas, trouve IN
        trouve := FALSE;
        pas := 0;
        WHILE pas ≤ 100 DO
            VAR vloc IN
                vloc := tableau(pas);
                IF element ≠ vloc THEN
                    pas := 1 + pas
                ELSE
                    trouve := TRUE;
                    pas := 101
                END
            END
        INVARIANT
            pas ∈ 0 .. 101 ∧
            trouve ∈ BOOL ∧
            (trouve = TRUE ⇒ pas = 101  ∧  trouve = bool(element ∈ ran(tableau))) ∧
            (trouve = FALSE ⇒ trouve = bool(element ∈ ran(0 .. pas − 1 ◁ tableau)))
        VARIANT
            101 − pas
        END;
        present := trouve
    END
END
```

De plus nous avons remplacé le test d'égalité dans la boucle par un test de différence, plus rapide. Le nombre d'instructions dans le niveau le plus interne est réduit au minimum : deux tests, un accès au tableau et une incrémentation. Notons que la déclaration interne de la variable *var_loc* ne devrait pas empêcher le compilateur d'utiliser un registre du processeur pour cette variable.

L'invariant est plus compliqué parce que la boucle n'a plus un comportement homogène à chaque itération : il faut alors écrire un invariant implicatif, comme nous l'avons déjà vu.

Nous avons également transformé cette boucle en une post-incrémentation : cela ne fait pas beaucoup de différence au niveau de la preuve, mais l'expression de l'invariant fait intervenir $pas - 1$.

Le variant reste très simple, car *pas* augmente toujours à chaque itération, même la dernière. Cette variable suffit donc à définir le variant.

6.4.3 Boucle sans index toujours croissant

Dans certains cas, il se peut que la boucle n'ait pas d'index matérialisé par une seule variable. Par exemple, une boucle de parcours d'un arbre pour rechercher un élément : la

variable qui varie est la position dans l'arbre, mais ce n'est pas une quantité numériquement croissante ou décroissante. Le variant de la boucle est alors une expression composée de plusieurs variables. Nous allons examiner ce cas sur un exemple simple : celui d'une boucle de recherche dans un tableau qui n'incrémente pas l'index quand l'élément recherché est atteint. Ceci pourrait être utile si l'index de l'élément constitue le résultat à retourner.

La spécification de la boucle serait :

```
MACHINE
    ExempleV
CONCRETE_VARIABLES
    tableau
INVARIANT
    tableau ∈ 0 .. 100 → NAT
INITIALISATION
    tableau := (0 .. 100) × {0}
OPERATIONS
    rang ← find(element) = PRE
        element ∈ NAT
    THEN
        SELECT tableau^-1[{element}] = ∅ THEN
            rang := 101
        ELSE
            rang :∈ tableau^-1[{element}]
        END
    END
END
```

Pour simplifier, nous supposons que la boucle doit retourner l'index 101 si l'élément demandé n'est pas dans le tableau. La boucle est alors simple : il suffit de parcourir le tableau et de s'arrêter dès que l'élément est atteint, ou que l'index est 101. Nous avons donc besoin d'un indicateur d'élément atteint. Nous allons utiliser une variable numérique pour pouvoir l'utiliser dans le variant :

```
IMPLEMENTATION
    ExempleV_imp
REFINES
    ExempleV
INITIALISATION
    tableau := (0 .. 100) × {0}
OPERATIONS
    rang ← find(element) = VAR trouve, pas IN
        pas := 0;
        trouve := 0;
        WHILE pas ≤ 100 ∧ trouve = 0 DO
            VAR var_loc IN
                var_loc := tableau(pas);
                IF var_loc ≠ element THEN
                    pas := pas + 1
                ELSE
                    trouve := 1
                END
            END
        INVARIANT
            pas ∈ 0 .. 101 ∧
            trouve ∈ {0, 1} ∧
            (trouve = 0 ⇒ tableau⁻¹[{element}] ∩ 0 .. (pas − 1) = ∅) ∧
            (trouve = 1 ⇒ pas ∈ tableau⁻¹[{element}])
        VARIANT
            200 − pas − trouve
        END;
        rang := pas
    END
END
```

Notez le variant $200 - pas - trouve$: tant que l'élément recherché n'est pas atteint, la variable *pas* augmente ; sinon c'est *trouve* qui passe de 0 à 1. Le variant est donc bien strictement décroissant. Pour être sûr de le garder positif, on ajoute 200. On peut aussi utiliser un indicateur *trouve* booléen, mais il faut alors déclarer une fonction constante pour transformer un booléen en nombre pour pouvoir l'utiliser dans le variant.

L'exemple que nous avons présenté ici est assez artificiel dans un tel cas, il serait préférable d'initialiser le résultat *rang* dans la branche du **IF** sélectionnée quand l'élément est atteint. Parallèlement, *pas* serait mis à 101 pour faire terminer la boucle : le programme obtenu est alors semblable à l'exemple du paragraphe précédent. Assez fréquemment, le programmeur essaie de conserver l'indice de boucle comme résultat alors qu'il vaudrait mieux avoir une variable de résultat séparée et arrêter la boucle par affectation de l'indice : non seulement le variant sera plus simple, mais la boucle sera plus rapide.

Avec ce dernier exemple, nous avons terminé l'étude de base des boucles. Après les problèmes liés à la constitution théorique des **VARIANT** et **INVARIANT**, l'opérateur se trouve souvent confronté à des problèmes de résolution de portée pour savoir ce que désigne chaque nom de variable dans la boucle. Nous allons maintenant examiner ce problème.

6.5 La résolution de portée des variables dans une boucle

Comme nous l'avons déjà vu, le générateur d'obligation de preuve crée de nouveaux identifiants (`var$2`) pour différencier les variables en cours d'évolution dans la boucle de leurs valeurs avant celle-ci. Il arrive assez fréquemment que dans le variant ou l'invariant l'opérateur doive référencer cette valeur initiale : par exemple pour exprimer qu'une variable est égale à sa valeur initiale plus l'indice de boucle à chaque pas.

Le même problème se pose pour les variables importées quand la boucle contient des appels aux opérations importées qui les modifient. Rappelons qu'une boucle étant toujours écrite dans une implantation, il peut y avoir des machines importées ; les variables importées sont accessibles via les opérations importées, mais sont référençables directement dans l'invariant de liaison. De la même manière, les variables importées peuvent être référencées dans un invariant de boucle.

Les règles de résolution de portée des variables dans une boucle sont les suivantes :

1. Les identifiants des variables modifiées dans le corps de boucle, s'ils sont utilisés dans le variant ou l'invariant de la boucle, désignent les variables en cours de modification et apparaissent décorrélés du contexte extérieur dans les PO. Ceci est vrai pour les variables modifiées directement (variables concrètes de l'implantation) comme pour les variables importées modifiées par l'appel des opérations importées.
2. Les autres identifiants (variables inutilisées ou en lecture seule dans le corps de boucle) référencent l'état de la variable *avant l'opération.*
3. Les identifiants des variables modifiées dans le corps de boucle concaténés avec $0 référencent la variable de même nom dans la spécification de la boucle, si cette variable existe. Dans tous les autres cas, les $0 dans une boucle sont illicites.

En particulier si certaines variables sont positionnées dans un état initial *juste avant la boucle*, par une initialisation locale, il n'y a pas moyen de référencer cet état car on peut seulement référencer des états *avant l'opération.* La solution consiste alors à déclarer une variable locale qui sert à "photographier" l'état initial voulu.

Nous allons illustrer ces règles sur des exemples.

6.5.1 Boucle avec une variable importée non homonyme

Considérons l'exemple suivant :

```
MACHINE
    Boucle
VARIABLES
    MyVar
INVARIANT
    MyVar ∈ 0 .. 100
INITIALISATION
    MyVar := 0
OPERATIONS
    Op = BEGIN
        skip
    END
END
```

Examinons l'implémentation suivante de cette spécification, dont l'invariant de boucle est volontairement incomplet :

```
IMPLEMENTATION
    Boucle_imp
REFINES
    Boucle
IMPORTS
    ImportedComponent
INVARIANT
    MyVar = ImportedVar
OPERATIONS
    Op = VAR var_loc IN
        ImportedInc;
        var_loc := 0;
        WHILE var_loc ≠ 5 DO
            ImportedDec;
            var_loc := var_loc + 1
        INVARIANT
            var_loc ∈ NAT ∧
            var_loc ≤ 5
            /* invariant incomplet ! */
        VARIANT
            10 − var_loc
        END
    END
END
```

MACHINE
ImportedComponent
VARIABLES
ImportedVar
INVARIANT
$ImportedVar \in NAT$
INITIALISATION
$ImportedVar := 0$
OPERATIONS
ImportedInc = **PRE**
$ImportedVar + 5 \in NAT$
THEN
$ImportedVar := ImportedVar + 5$
END ;
ImportedDec = **PRE**
$ImportedVar - 1 \in NAT$
THEN
$ImportedVar := ImportedVar - 1$
END
END

Nous avons créé ici une machine `Boucle` possédant une variable `MyVar`, et une opération `op = skip` qui ne doit donc pas modifier `MyVar`.

Insistons sur le fait que `op = skip` n'est pas une spécification vide, elle impose une condition sur l'opération `op` : ne pas modifier `MyVar`, condition que la boucle devra réaliser et qui donnera lieu à des obligations de preuve.

L'implantation `Boucle_imp` utilise le composant importé `ImportedComponent`. Ce composant encapsule une variable `ImportedVar` qui implante la variable de spécification `MyVar` ; l'invariant de liaison `MyVar = ImportedVar` précise ce lien.

L'opération `op` du composant `ImportedComponent` incrémente de 5 la variable `ImportedVar`, puis celle-ci est décrémentée 5 fois ; cette variable a donc été modifiée localement dans le corps de la boucle (elle est modifiée à chaque itération) mais, elle n'est pas modifiée globalement.

L'invariant de la boucle est clairement insuffisant, car il ne caractérise pas l'effet de l'opération `ImportedDec` sur la variable `ImportedVar` à chaque itération de boucle.

Les obligations de preuve générées par l'Atelier B montrent cette erreur :

```
"‘Local hypotheses’" &
var_loc$0: NATURAL &
var_loc$0<=2147483646 &
var_loc$0<=5 &
not(var_loc$0 = 5) &
"‘Check preconditions of called operation, or While loop construction,
or Assert predicates, ref 9’"
    =>
       ImportedVar$2-1: NATURAL
```

(Conservation de l'invariant de la boucle, premier prédicat).

```
"‘Local hypotheses’" &
var_loc$0: NATURAL &
var_loc$0<=2147483646 &
var_loc$0<=5 &
not(var_loc$0 = 5) &
"‘Check preconditions of called operation, or While loop construction,
or Assert predicates, ref 9’"
    =>
        ImportedVar$2-1<=2147483646
```

(Conservation de l'invariant de la boucle, premier prédicat).

Dans ces obligations de preuve, l'identificateur `ImportedVar$2` désigne la variable importée `ImportedVar` utilisée dans le corps de la boucle (seules les hypothèses locales sont présentées ici). Aucune hypothèse sur `ImportedVar$2` n'apparaît dans ces trois obligations de preuve, elles ne pourront être démontrées. Il manque un prédicat sur `ImportedVar$2` qui caractérise son évolution au cours de la boucle (à chaque itération la variable est décrémentée) et qui permette de vérifier qu'elle a bien la propriété voulue à la sortie (globalement la variable n'a pas été modifiée).

Explication :

> Comme nous l'avons déjà dit, à l'intérieur d'une boucle, les variables transformées ne vérifient plus nécessairement les hypothèses les concernant qui ont été déduites du contexte (invariants, préconditions, et substitutions préalables). Par exemple, la variable importée `ImportedVar` ne vérifie pas l'invariant de liaison `ImportedVar=MyVar` au cours de la boucle. Il faut donc pour raisonner sur ces variables, introduire de nouveaux identifiants, sur lesquels les seules hypothèses sont celles de l'invariant de la boucle ; dans l'exemple précédent, la variable importée `ImportedVar$2` a ainsi été introduite.

Le sens du prédicat à ajouter dans l'invariant doit être : « la variable `ImportedVar` pendant la boucle est égale à `ImportedVar` juste avant la boucle moins la valeur de `var_loc` ». La seule méthode pour représenter « `ImportedVar` juste avant la boucle » consiste à utiliser la variable abstraite, ici `MyVar`. Le prédicat à ajouter dans l'**INVARIANT** de boucle est :

```
ImportedVar = 5 + MyVar - var_loc
```

où `MyVar` représente alors la valeur de la variable avant utilisation de l'opération `op`. Ici il n' y a pas de problème de résolution de portée car `MyVar` est une variable externe à la boucle.

Les obligations de preuve générées sont alors les suivantes :

```
"‘Local hypotheses’" &
var_loc$0: NATURAL &
var_loc$0<=2147483646 &
var_loc$0<=5 &
ImportedVar$2 = MyVar+5-var_loc$0 &
not(var_loc$0 = 5) &
"‘Check preconditions of called operation, or While loop construction,
or Assert predicates, ref 9’"
```

```
=>
  ImportedVar$2-1<=2147483646
```

Puis, sous les mêmes hypothèses :

```
  ImportedVar$2-1 = MyVar+5-(var_loc$0+1)
```

et encore :

```
  ImportedVar$2-1<=2147483646
```

La variable `ImportedVar$2` apparaît désormais dans les hypothèses de ces trois obligations de preuve : elles sont maintenant démontrables.

Remarque L'expression $ImportedVar\$2 = 5 + MyVar - var\_loc\$0$ représente dans les obligations de preuve le prédicat que nous avons ajouté à l'invariant de la boucle. On voit que $ImportedVar\$2$ est différencié du contexte car il représente la variable en cours d'évolution.

6.5.2 Boucle avec une variable importée homonyme à la variable abstraite

Nous avons vu que l'invariant des boucles agissant sur des variables importées doit référencer à la fois ces variables importées et leurs modèles abstraits pour caractériser la boucle (voir paragraphe précédent). Ceci pose un problème si une variable importée a le même nom que son modèle abstrait. Nous utilisons alors une fonctionnalité du langage B spécialement prévue pour ce cas : la notation \$0. Reprenons l'exemple précédent d'une boucle qui ne doit pas modifier une variable `MyVar`, en y introduisant une telle homonymie. Cette variable `MyVar` est réalisée par une variable importée également appelée `MyVar`.

MACHINE
Loop
VARIABLES
MyVar
INVARIANT
$MyVar \in 0 \,..\, 100$
INITIALISATION
$MyVar := 0$
OPERATIONS
Op = **BEGIN**
skip
END
END

```
IMPLEMENTATION
    Loop_imp
REFINES
    Loop
IMPORTS
    ImportedComponentP
OPERATIONS
    Op = VAR var_loc IN
        ImportedInc;
        var_loc := 0;
        WHILE var_loc ≠ 5 DO
            ImportedDec;
            var_loc := var_loc + 1
        INVARIANT
            var_loc ∈ NAT ∧
            var_loc ≤ 5 ∧
            MyVar = MyVar$0 − var_loc + 5
        VARIANT
            10 − var_loc
        END
    END
END
```

Remarquez l'utilisation de $MyVar\$0$ dans l'invariant de la boucle pour référencer $MyVar$ de la spécification, avant l'opération. Le composant importé est le suivant :

```
MACHINE
    ImportedComponentP
VARIABLES
    MyVar
INVARIANT
    MyVar ∈ NAT
INITIALISATION
    MyVar := 0
OPERATIONS
    ImportedInc = PRE
        MyVar + 5 ∈ NAT
    THEN
        MyVar := MyVar + 5
    END;
    ImportedDec = PRE
        MyVar − 1 ∈ NAT
    THEN
        MyVar := MyVar − 1
    END
END
```

Ce cas où une variable de la spécification est réalisée par importation d'une variable homonyme est très fréquent, c'est pourquoi la technique que nous avons montrée ici est souvent utilisée.

6.6 Ce que nous avons appris

- Les clauses **VARIANT** et **INVARIANT** d'une boucle **WHILE** en B servent à démontrer la boucle. Elles ne sont jamais traduites en code.
- Le **VARIANT** sert à démontrer que la boucle n'est jamais infinie.
- L'**INVARIANT** sert à démontrer que la boucle réalise sa spécification.
- Le **VARIANT** et l'**INVARIANT** d'une boucle sont écrits par l'opérateur.
- Une boucle programmée correctement peut produire des PO fausses parce que ses **VARIANT** ou **INVARIANT** sont faux.
- Les boucles ne sont autorisées que dans les implantations.
- Il ne faut pas faire de boucles imbriquées.
- Les trois propriétés qui définissent un invariant valide sont :
  - A l'entrée de la boucle, l'invariant doit être établi,
  - Pendant la boucle, l'invariant doit être conservé,
  - La condition de sortie et l'invariant doivent impliquer la propriété spécifiée.
- Le variant est une expression numérique calculée à partir des variables qui doit être strictement décroissante à chaque itération et rester positive.
- Les preuves internes d'une boucles sont faites sur des variables décorrélées du contexte de la machine : renommage en $2 dans les PO.
- Pour vérifier les trois propriétés, l'invariant doit contenir :
  - le typage des variables de boucle,
  - les prédicats nécessaire pour assurer entre autres les préconditions des opérations appelées,
  - les relations de récurrence vraies à chaque itération qui permettent de démontrer que la boucle réalise sa spécification en sortie.
- La méthode d'introduction d'un raffinement intermédiaire permet de séparer la recherche des formules de récurrence pour la construction de l'invariant.
- La méthode de construction de l'invariant par définition d'une suite par récurrence facilite la recherche des formules de récurrence à partir du corps de la boucle.
- La méthode de construction de l'invariant avec le prouveur utilise l'examen des PO fausses pour compléter l'invariant.
- Les variables locales doivent si possible être déclarées à l'intérieur de la boucle pour ne pas intervenir dans son invariant.
- Faire des essais sur l'ordre des opérateurs arithmétiques utilisés sur les indices de boucles pour simplifier la preuve.
- Si la boucle contient des **IF**, l'invariant devra sans doutes contenir des implications.
- Attention aux cas où la boucle fait zéro itération : l'invariant doit quand même être établi.
- Attention aux débordements numériques au dernier tour de boucle.
- Si vous avez trouvé une expression strictement décroissante, ajoutez n'importe quelle constante positive suffisamment grande pour que le résultat reste positif pour obtenir un **VARIANT** valide.
- Si l'indice de boucle ne varie pas sur certaines itérations, le variant doit contenir d'autres variables que cet indice.
- Utilisez des indicateurs numériques plutôt que booléens : ils peuvent ainsi être employés dans le variant.
- Pour terminer une boucle, forcez l'indice à la valeur de sortie : c'est plus optimisé qu'un indicateur séparé et cela facilite l'écriture du **VARIANT**.
- Les identifiants des variables modifiées dans le corps de boucle désignent les variables locales à la boucle.
- Les identifiants des variables modifiées dans le corps de boucle, augmentés de $0 désignent les variables de spécification avant l'opération.
- Si l'écriture de l'invariant nécessite l'accès à un état initial difficile à référencer, utiliser une variable locale pour "noter" celui-ci.

Chapitre 7

Développement de Machines de Base

7.1 Définition

Une machine de base est un module constitué d'une spécification B et d'un code informatique écrit dans un langage classique.

Les machines de base ont donc la caractéristique de ne pas posséder d'implantation écrite en langage B. Les fonctions spécifiées dans les machines de base sont directement réalisées dans un langage de programmation classique.

7.2 Rôle des machines de base

Revenons aux principes même de la programmation : un appareil que nous pouvons programmer possède forcément deux choses :
- Une capacité de calcul ;
- Un certain nombre de fonctionnalités pour interagir avec son environnement, que nous appellerons entrées / sorties.

Le deuxième point est très important : sans lui, l'appareil n'aurait aucun intérêt puisqu'il n'y aurait aucun moyen d'échanger des informations avec l'extérieur. Les traducteurs de l'Atelier B ne produisent que le code qui correspond au premier point. En effet les entrées / sorties ne peuvent pas être décrites universellement, à l'opposé des fonctions de calcul communes à tous les calculateurs, ces dernières étant les seules sur lesquelles le B0 s'appuie. Les entrées / sorties doivent donc être atteintes autrement.

Il y a deux méthodes envisageables pour réaliser la communication entre un programme développé avec B et son environnement :
- Utiliser B pour définir des procédures qui ont des paramètres d'entrée / sortie. L'environnement qui appellera ces procédures fournira alors les paramètres.
- Utiliser des machines de base.

La première solution n'est pas très satisfaisante car étant donné que l'environnement d'appel voit le projet B par les machines abstraites de plus haut niveau, on est obligé de faire figurer ces entrées / sorties très concrètes dans les machines abstraites de plus haut niveau. Il vaut donc mieux utiliser des machines de base qui permettent de faire

apparaître les détails concrets dans les couches plus basses. Par exemple, si nous disposons d'un afficheur capable d'afficher des nombres entre 0 et 99, nous définirons la machine de base suivante :

```
MACHINE
    Display
VARIABLES
    val
INVARIANT
    val ∈ 0..99
INITIALISATION
    val :∈ 0..99
OPERATIONS
    output(vv) = PRE vv ∈ 0..99 THEN
        val := vv
    END
END
```

L'implantation de cette machine sera faite directement, en utilisant les moyens qui existent sur l'appareil considéré pour piloter cet afficheur. Le plus souvent, ces moyens sont constitués de bibliothèques d'entrée / sortie fournies par le constructeur et écrites dans un langage de programmation donné. La présence de ce langage influe bien entendu sur le choix du langage de traduction employé pour le projet B, afin d'obtenir une utilisation facile des bibliothèques d'entrée / sortie existantes.

Les machines de base constituent donc l'interface entre le programme B et son environnement. Notons que la définition de cet environnement est arbitraire : on peut par exemple décider d'utiliser des bibliothèques graphiques non réalisées en B, dans ce cas elles feront partie de l'environnement extérieur et seront vues comme des machines de base depuis le programme B.

indexmachine de base!instanciation Lors de la définition d'une machine de base, il y a un point très important qu'il faut considérer : *l'instanciation*. Dans un projet B, une machine abstraite peut être importées plusieurs fois avec des noms d'instances différents (**IMPORTS** i1.machine, i2.machine). Ces machines définissent alors des données séparées. Dans le cas de machines de base, il est souvent impossible de créer de telles instances car les variables représentent des éléments physiques. Dans notre exemple précédent, la variable *val* représente l'afficheur, cela n'a donc pas de sens de créer plusieurs instances de la machine. Lors de la définition d'une machine de base, il faut donc choisir l'une des options suivantes :

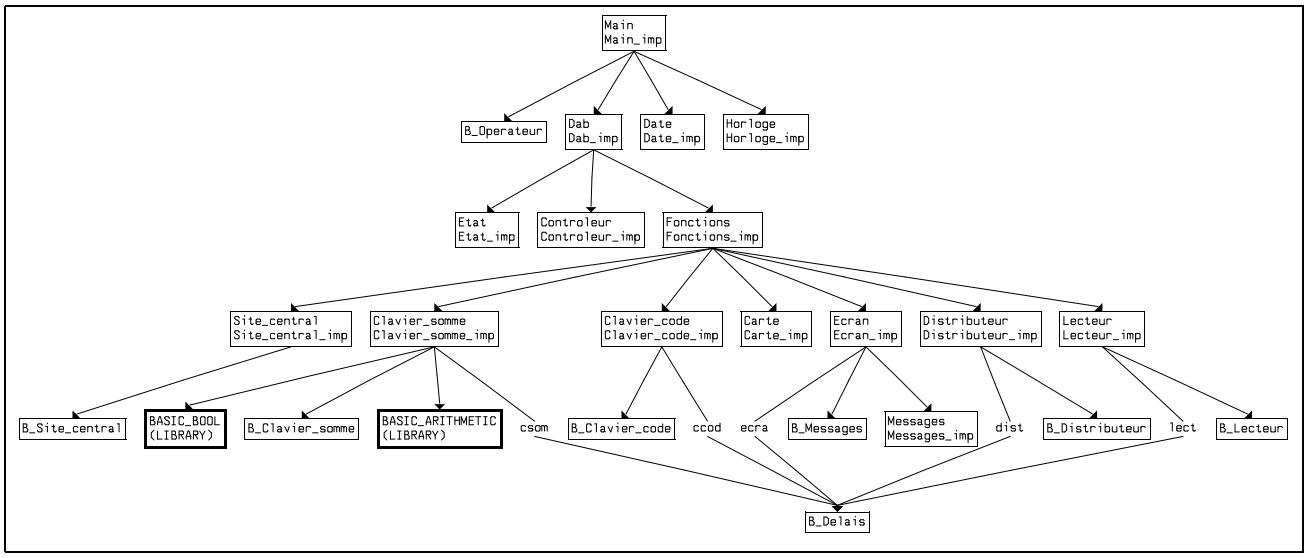
- L'instanciation de la machine de base sera autorisée, il faut alors définir le sens de cette instanciation ;
- L'instanciation sera interdite (cas le plus fréquent).

7.3 Quelques exemples

L'exemple suivant montre l'utilisation d'une machine de base pour imprimer "bonjour" sur un terminal.

```
MACHINE
    Bonjour
OPERATIONS
    main = skip
END
```

```
IMPLEMENTATION
    Bonjour_1
REFINES
    Bonjour
IMPORTS
    /* Utilisation de la machine de base d'entrées/sorties */
    BASIC_IO
OPERATIONS
    main =
    BEGIN
        /* impression d'une chaîne de caractères à l'écran */
        STRING_WRITE("bonjour \n")
    END
END
```



Dans un projet B, les machines de base sont des *feuilles de l'arbre d'importation*[1] ; le schéma suivant présente un arbre d'importation où les flèches représentent des liens de type `IMPORTS` entre les différents modules B.

Le graphe d'importation précédent possède dix machines de base. Ce sont :
– les machines de base `BASIC_BOOL` et `BASIC_ARITHMETIC`. Elles proviennent, comme cela est indiqué, du projet `LIBRARY` fourni avec l'Atelier B.
– les machines de base `B_Site_Central, B_Clavier_Somme, B_Clavier_Code, B_Delais, B_Messages, B_Distributeur, B_Lecteur`. Ces machines sont spécifiques au projet.

La machine de base `B_Delais` est importée cinq fois, les liens de type `IMPORTS` sont renommés ; il existe donc cinq instances de cette machine de base qui sont `csom, ccod, ecra, dist` et `lect`.

7.4 Machines de base fournies avec l'Atelier B

L'Atelier B ne propose que très peu de machines de base, car l'interface physique des programmes B est extrêmement variable. Il aurait été possible de proposer une interface Unix standard, mais les programmes sécuritaires développés en B sont le plus souvent implantés sur des systèmes dédiés dont l'interface est spécifique.

Les machines de base livrées avec l'Atelier B permettent soit de réaliser certains types de tableaux actuellement non gérés par le B0, soit de produire des maquettes utilisant des entrées/sorties style `vt100`. Ce sont :
– `BASIC_ARRAY_VAR` : implantation d'un tableau à une dimension,
– `BASIC_ARRAY_RANGE` : implantation d'un tableau à deux dimensions,
– `BASIC_IO` : entrées/sorties style `vt100`.

Les machines de base concernant les tableaux doivent être utilisées quand les tailles ne sont pas directement fixées : par exemple, pour réaliser des tableaux dont la taille est précisée par un paramètre formel de la machine. Ceci n'est actuellement pas faisable en

[1] L'arbre d'importation d'une machine est composé de l'ensemble des machines importées par son implantation et de l'union des arbres d'importation des machines qu'elle importe

B0.

7.5 Machines de librairie fournies avec l'Atelier B

Les machines de librairies sont des machines abstraites écrites en langage B. Elles modélisent généralement un type d'objet mathématique (séquence, fonction, ensemble, etc...) et offrent les opérations qui permettent de manipuler ces objets.

Contrairement aux machines de base, les machines de librairie sont concrètement réalisées en B ; c'est-à-dire par raffinement, implantation en B et preuve de l'ensemble. Elles ne concernent donc pas l'interface entre les programmes B et l'extérieur, nous allons néanmoins les examiner ici car elles offrent un moyen de ne pas redévelopper sans cesse les mêmes implantations des structures mathématiques communément utilisées.

Les machines de librairie livrées avec l'Atelier B sont les suivantes :
- `L_ARITHMETIC1` offre les opérations arithmétiques pour des applications plus calculatoires (exponentielle, racine carrée entière, logarithme),
- `L_ARRAY1` réalise par un tableau une variable abstraite de type fonction,
- `L_ARRAY1_RANGE` : réalisation d'une rangée de tableaux de même taille, à index numériques,
- `L_ARRAY1_COLLECTION` : réalisation de tableaux de même taille, à index numériques,
- `L_ARRAY3` : réalisation d'un tableau à valeurs non ordonnées avec opérations maximales (échanges, décalage, inversion, recherche, etc...),
- `L_ARRAY3_RANGE` : réalisation d'une rangée de tableaux de même taille à index numériques à valeurs non ordonnées,
- `L_ARRAY5` : réalisation d'un tableau à valeurs ordonnées avec opérations de tri (échange, décalage, inversion, recherche et tri),
- `L_ARRAY5_RANGE` : réalisation d'une rangée de tableaux de même taille à index numériques à valeurs ordonnées avec opérations de tri,
- `L_ARRAY_COLLECTION` permet la manipulation de tableaux à une dimension et de taille identique,
- `L_PNFC` : réalisation d'une fonction partielle (échange, décalage, inversion, recherche, surcharge et tri),
- `L_SEQUENCE` : réalisation d'une séquence,
- `L_SEQUENCE_RANGE` : réalisation d'une rangée de séquences,
- `L_SET` : réalisation d'un ensemble,

Une description complète des machines de base et des machines de librairie livrées avec l'Atelier B est donnée dans le document *Composants réutilisables : manuel de référence.*

Il est possible d'enrichir ces modèles réutilisables par ses propres machines de base ou de librairie.

7.6 Développement des machines de base avec l'Atelier B

Les machines de base ne sont pas entièrement développées en B, une attention particulière doit donc être portée sur les points suivants :

1. La *spécification B* d'une machine de base doit décrire le plus précisemment possible ses fonctionnalités. Ainsi, le mécanisme de preuve garantit que les conditions d'appel aux services de la machine de base satisfont les contraintes décrites dans l'`INVARIANT`

et/ou dans les préconditions des opérations. De ce fait, le code informatique qui implémente les fonctionnalités de la machine de base peut être offensif.

2. La *vérification de la correction du code informatique* associé à la machine de base par rapport à sa spécification doit se faire de façon traditionnelle ; c'est à dire en appliquant la politique de tests préconisée lors de la vérification des composants logiciels. Nous ne détaillerons pas ce point qui relève de la programmation classique.
3. Le code informatique doit respecter les conventions de traduction utilisées par les traducteurs de l'Atelier B afin de garantir la *validité des interfaces* entre les composants B et non B.

7.6.1 Spécification B pour une machine de base

Supposons par exemple que nous disposons d'un afficheur capable d'afficher des nombres entre 0 et 99. nous pouvons définir la machine de base suivante :

```
MACHINE
    Display
OPERATIONS
    output(vv) = PRE vv ∈ ℕ THEN
        skip
    END
END
```

Cette spécification est suffisante pour que, vu du programme B, l'opération output soit connue et utilisable dans l'implantation qui importe cette machine. La seule preuve qui sera faite lors de l'appel de cette opération est que le paramètre d'entrée est un entier naturel. Si l'afficheur physique fait un court circuit quand on lui présente une valeur supérieure à 99, le programme B peut le détruire. La preuve effectuée sur ce programme n'est d'aucun secours dans ce cas. Pour résoudre ce problème, nous pouvons améliorer la précondition :

```
MACHINE
    Display
OPERATIONS
    output(vv) = PRE vv ∈ 0..99 THEN
        skip
    END
END
```

Nous sommes maintenant sûrs *par la preuve* que l'opération n'est jamais appelée avec des valeurs dangereuses. Par contre, la preuve ne nous garantit sûrement pas que l'opération est appelée : en effet sa spécification est skip, c'est-à-dire qu'elle *n'a aucune influence* du point de vue de la preuve. Nous avons pu spécifier très habilement ce que devait faire notre programme par une modélisation B très complète, mais qu'importe : nous n'avons aucun moyen de lier cette modélisation à l'appel de l'opération d'affichage. Le preuve ne nous garantit donc rien sur cet affichage.

Nous voyons que la spécification de notre machine de base doit contenir une modélisation de ce qu'elle fait, par exemple :

MACHINE
Display
VARIABLES
val
INVARIANT
val ∈ 0..99
INITIALISATION
val :∈ 0..99
OPERATIONS
ouput(vv) = PRE vv ∈ 0..99 THEN
val := vv
END
END

Cette fois, il nous est possible de prouver que l'opération est bien appelée et au bon endroit.

Examinons un autre exemple : la machine de base suivante réalise un *buffer*[2]et les opérations d'écriture et de lecture de ce buffer. Un buffer est constitué de données `DATAS` et est limité physiquement à un nombre maximum de données `MAX_DATAS`. L'ensemble de base `DATAS` et la constante `MAX_DATAS` sont définis dans un module de contexte `Ctx` non présenté ici.

[2]La modélisation du buffer n'est pas exacte puisque l'ordre des données n'est pas pris en compte ; à la place de la variable ensembliste une variable de type séquence serait plus adéquate.

```
MACHINE
    B_Buffer
SEES
    Ctx
ABSTRACT_VARIABLES
    buffer
INVARIANT
    buffer ⊆ DATAS ∧
    card(buffer) ≤ MAX_DATAS
INITIALISATION
    buffer := ∅
OPERATIONS
    ecrire_une_(donnée) =
    PRE
    /* Ces préconditions doivent être satisfaites par
    les modules appelant cette opération. */
        donnée ∈ DATAS ∧
        /* invariant dit de typage */
        card(buffer) < MAX_DATAS
        /* l'écriture est impossible si le résultat
        dépasse la limite physique du buffer */
    THEN
        buffer := buffer ∪ {donnée}
    END ;

    donnée_lue ← lire_donnée =
    BEGIN
        /* retourne n'importe quelle donnée du buffer */
        donnée_lue :∈ buffer
    END
    ...
END
```

7.6.2 Comment garantir la validité des interfaces ?

La validité des interfaces entre code informatique issu des implementations B et code informatique natif est assurée dès lors que le code informatique associé aux machines abstraites est produit par les traducteurs de l'Atelier B à partir de leur implantation. Nous allons reprendre ce principe pour les machines de base.

Pour ce faire, nous proposons que le développement des machines de base suive les étapes suivantes :

1. Écriture de la spécification B de la machine de base,
2. Vérification, par le mécanisme de preuve, de la correction de la spécification B de la machine de base,
3. Vérification, par le mécanisme de preuve, que les composants B utilisant les services de la machine de base respectent les contraintes d'utilisation spécifiées,
4. Écriture d'une implantation *provisoire* de la machine de base. Ce type d'implantation est dite "coquille vide" puisque le corps des opérations ne comprend que les instruc-

tions suffisantes pour respecter les règles du langage B. Il n'y a pas de vérification formelle de la correction de cette implantation par rapport à la spécification B puisque les fonctionnalités de la machine de base ne sont pas encore implémentées,

5. Utilisation de la plateforme de traduction de l'Atelier B pour générer le code informatique associé à la machine de base ; ce code correspond en fait au *squelette* du code informatique qui sera finalement réalisé pour la machine de base,

6. Le code informatique produit automatiquement est enrichi par l'utilisateur de sorte qu'il réalise les fonctionnalités attendues de la machine de base. Cette ultime étape correspond à la phase de codage des machines de base.

7.6.3 Écriture d'une implantation "coquille vide"

L'implantation "coquille vide" doit contenir le minimum d'informations afin de respecter les règles de syntaxe et de grammaire du langage B.

Les clauses de l'implantation qui devront **impérativement** être complétées sont définies ci-dessous.
- Une clause `VALUES` est introduite si des ensembles abstraits (respectivement des constantes concrètes) sont définis dans la clause `SETS` (respectivement `CONCRETE_CONSTANTS`) de la machine abstraite associée.
- Une clause `INITIALISATION` est introduite si des variables concrètes sont définies dans la clause `CONCRETE_VARIABLES` de la machine abstraite.
- La clause `OPERATIONS` est introduite si elle apparaît déjà dans la machine abstraite. Le corps des opérations est minimal :
  - si l'opération ne possède pas de paramètres de retour, le corps de l'opération se réduit à la seule instruction `skip`,
  - sinon le corps de l'opération est uniquement constitué d'instructions d'affectation des paramètres de retour.

En résumé, il faut donner des valeurs à toute la partie concrète issue de la spécification. En traduisant la "coquille vide", le traducteur définit toutes cette partie concrète qu'il ne reste plus qu'à initialiser.

Le programmeur peut alors récupérer le code produit et insérer ses propres variables et ses propres procédures. C'est à cette étape que le choix d'autoriser ou d'interdire l'instanciation de la machine de base (voir 7.2) doit être pris en compte :
- Si l'instanciation est autorisée, le programmeur doit tenir compte de la méthode d'instanciation utilisée par le traducteur de l'Atelier B. Se référer au manuel d'utilisation du traducteur concerné. Le programmeur doit en effet définir ses variables de telle manière que chaque nouvelle instanciation le la machine abstraite crée un nouvel espace de variables.
- Si l'instanciation n'est pas autorisée, il n'y a pas de précaution particulière. Par contre, il faut vérifier dans le projet B que la machine de base n'est pas importée en plusieurs instances. C'est une bonne idée que d'implanter ceci comme un test automatique dans le code de la machine de base, bien que la programmation de ce test nécessite la connaissance du mécanisme d'instanciation utilisé par le traducteur concerné.

Si nous reprenons l'exemple `Buffer` présenté dans ce chapitre, une implantation dite "coquille vide" peut être :

```
IMPLEMENTATION
    B_Buffer_imp
REFINES
    B_Buffer
SEES
    Ctx
CONCRETE_VARIABLES
    buffer
INVARIANT
    buffer ∈ 1..MAX_DATAS → DATAS ∧
    /* il n'y a pas d'invariant de liaison puisque cette
    implantation n'est pas vérifiée formellement.*/
INITIALISATION
    buffer := (1..MAX_DATAS)*{NULL}
OPERATIONS
    ecrire_une_(donnée) =
    BEGIN
        /* cette substitution sera ensuite reprise */
        buffer(1) := donnée
    END ;

    donnée_lue ← lire_donnée =
    BEGIN
        /* cette substitution sera ensuite reprise */
        donnée_lue := buffer(1)
    END
    ...
END
```

La variable concrète *buffer* ne correspond pas à la variable qui sera réellement décrite dans le code informatique. L'intérêt de cette définition est que le source informatique (produit automatiquement depuis cette implantation, par un traducteur de l'Atelier B) comprend déjà la définition d'une variable de type simple, ainsi que les opérations. Il reste alors à l'utilisateur à reprendre la définition de la variable *buffer* et de renseigner le corps des opérations du code informatique produit par l'Atelier B.

Remarques

1. Les variables et constantes abstraites ne pouvant être introduites dans une implantation, les clauses `ABSTRACT_CONSTANTS` et `ABSTRACT_VARIABLES` sont interdites dans toute implantation.
2. Seules les opérations définies dans la machine abstraite sont rappelées en implantation ; aucune opération ne peut être rajoutée.

7.7 Ce que nous avons appris

- Une machine de base est une machine dont l'implantation n'est pas faite en B.
- Les machines de base sont l'interface entre le programme B et l'extérieur.
- On peut aussi faire cette interface par les paramètres des services développés en B, mais cela oblige à avoir des paramètres trop concrets au plus haut niveau.
- "L'extérieur" d'un programme B est la limite que l'on donne au développement fait en B.
- Le langage de traduction choisi est souvent fonction des librairies que l'on veut atteindre sur le calculateur cible.
- Il faut choisir lors de la conception d'une machine de base si elle doit être instanciable ou non.
- Les machines de bases fournies avec l'Atelier B permettent de faire des maquettes ou pallient à certaines limitations B0 concernant les tableaux.
- Les machines de librairies implantent les structures mathématiques communes.
- Pour développer une machine de base, il faut écrire sa spécification B, réaliser et valider son code informatique, et rendre ce code compatible avec le code produit par l'Atelier B.
- La spécification en B d'une machine de base doit contenir :
  - des préconditions pour garantir les paramètres d'entrée,
  - des variables pour modéliser les effets produits.
- Pour produire un squelette de code compatible, le mieux est d'écrire et de traduire une implantation "coquille vide".
- Dans l'implantation "coquille vide" : valuer ou initialiser la partie concrète de la spécification, écrire des accès aux éléments réalisés par le traducteur dans les opérations pour avoir un modèle codé de ces accès.
- Si la machine de base est instanciable : toute variable d'état doit être codée conformément au système d'instanciation utilisé par le traducteur concerné.
- Si la machine n'est pas instanciable : l'absence d'instances doit être vérifié dans le projet B.

Chapitre 8

Développement de Machines de Contexte

Dans une architecture logicielle on souhaite que les variables soient séparées des données statiques qui sont partagées par plusieurs modules. En B, les données statiques sont les constantes et les ensembles abstraits ou énumérés introduits dans les clauses `CONSTANTS`, `SETS` et `PROPERTIES`. Nous proposons de définir les données statiques à partager par plusieurs composants, dans des composants dits de contexte.

8.1 Utilité d'une machine de contexte

Une machine de contexte introduit les ensembles et les constantes nécessaires à la définition des variables et des opérations d'une machine abstraite.

Chaque machine abstraite inclut la machine de contexte qui lui est associée et définit donc, par procuration, ses ensembles et constantes ; le détail de cette technique est précisé dans ce chapitre.

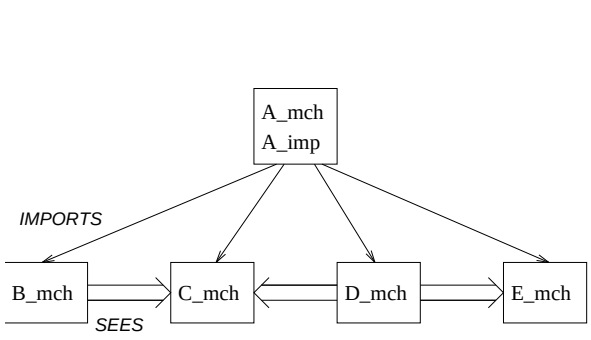
L'utilité d'une machine de contexte est indéniable lorsque plusieurs machines abstraites nécessitent la connaissance des mêmes données statiques. Ainsi, les composants n'utilisent pas la machine abstraite mais directement la machine de contexte ; les liens architecturaux superflus entre les machines abstraites ne sont pas créés.

Pour les raisons exposées précédemment, on préférera à l'architecture de la *figure 1*, celle de la *figure 2*.

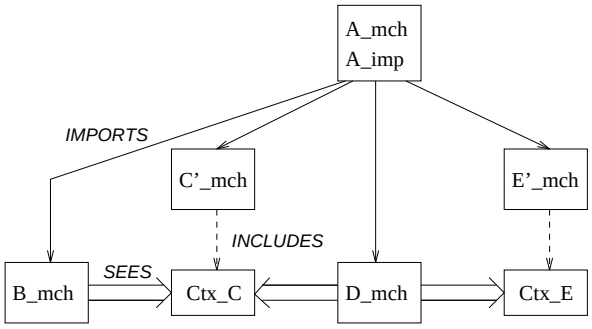
Dans la *figure 2*, la machine `C_mch` est séparée en deux machines : `C'_mch` et `Ctx_C`. `Ctx_C` est la machine de contexte contenant les éléments nécessaires aux composants `B_mch` et `D_mch`.

Les machines de contexte permettent donc de :

1. limiter les liens superflus entre les différents composants B. Dans la *figure 2*, on a évité, par exemple, un lien `SEES` entre les composants `B_mch` et `C_mch` ; on limite ainsi l'utilisation du lien `SEES` à la consultation de données statiques.
2. améliorer la lisibilité et la maintenabilité du modèle B. Dans la *figure 2*, la modification du composant `C'_mch` n'affecte pas les modules `B_mch` et `D_mch`. En particulier, *il n'est pas nécessaire de reprouver ces machines* après une modification de `C'_mch`.



- figure 1 : D utilise les donnees statiques de C et E -
- B utilise les donnees statiques de C -



- figure 2 : D utilise les machines de contexte de C et E -
- B utilise la machine de contexte de C -

Sur un gros projet, ceci est déterminant.

3. limiter le nombre d'hypothèses présentes dans une Obligation de Preuve (du fait que le lien `SEES` porte sur un moins grand nombre de données).

Il est également important de répartir les constantes constituant le contexte dans plusieurs machines de contexte, si les différentes parties du projet utilisent des groupes séparés de constantes. Faire une seule machine de contexte utilisée partout serait s'exposer à devoir reprouver l'ensemble du projet si une seule constante change de nom ou reçoit une propriété supplémentaire !

8.2 Constitution d'une machine de contexte

Dans la plupart des cas, une machine de contexte est implantée de manière autonome. L'implantation de la machine de contexte permet de valuer les ensembles abstraits ainsi que les constantes concrètes introduits au cours du développement du composant. Rappelons que les constantes dites raffinables ne sont pas valuées puisqu'elles ne sont pas utilisées en implantation. Les ensembles énumérés ne sont pas non plus valués : leur définition est exhaustive puisque la liste des éléments de l'énumération est fournie dès la spécification.

8.2.1 Spécification formelle d'une machine de contexte

Une machine de contexte contient la définition des données statiques. Les clauses présentes dans une machine de contexte sont :
– la clause `SETS` pour introduire des ensembles énumérés ou abstraits,
– les clauses `ABSTRACT_CONSTANTS` et `CONCRETE_CONSTANTS` pour introduire des constantes,
– la clause `PROPERTIES` pour définir les propriétés des ensembles et constantes.
Une machine de contexte doit être, dans la mesure du possible, indépendante de toute autre machine abstraite et doit seulement définir des données statiques. De ces deux recommandations, on déduit que les clauses inutilisées d'une machine de contexte sont :
– les clauses de définition de variables `VARIABLES`, `ABSTRACT_VARIABLES`, `CONCRETE_VARIABLES`, `INVARIANT` et `INITIALISATION`,
– les clauses `INCLUDES`, `EXTENDS`, `SEES` et `USES`,

– la clause `OPERATIONS`.

Souvent, les constantes doivent être "valuées" dès la spécification, c'est à dire dans la clause `PROPERTIES` pour permettre la preuve des obligations de preuve des modules B utilisant la machine de contexte.

Comme la machine de contexte ne définit ni variables ni opérations, aucune obligation de preuve ne sera produite pour vérifier la cohérence de la machine abstraite.

La vérification de la validité des propriétés se fera lors de la vérification de la valuation de ces données statiques. Il est, de ce fait, fortement recommandé de vérifier la correction des composants (machines abstraites, implantations) de contexte avant de prouver le reste du modèle B.

L'exemple ci-dessous définit un ensemble de constantes (*Cst1,..,Cst6*) dans la machine abstraite *CstMachine* sans préciser la valeur effective de ces constantes. La machine abstraite *M5* consulte la machine de constantes (lien de type `SEES`) et les variables *vrb1, vrb2* utilisent ces constantes pour préciser leur domaine de définition.

MACHINE
CstMachine
CONCRETE_CONSTANTS
Cst1, Cst2, Cst3, Cst4, Cst5, Cst6
PROPERTIES
Cst1 $\in INT \wedge$ Cst2 $\in INT \wedge$
Cst3 $\in INT \wedge$ Cst4 $\in INT \wedge$
Cst5 $\in INT \wedge$ Cst6 $\in INT \wedge$
/* Propriétés supplémentaires pour valider le modèle */
Cst1 $\leq$ Cst5*Cst3 - Cst6 $\wedge$
Cst5*Cst4 - Cst6 $\leq$ Cst2
END

MACHINE
M5
SEES
CstMachine
CONCRETE_VARIABLES
vrb1, vrb2
INVARIANT
vrb1 $\in$ Cst1..Cst2 $\wedge$
vrb2 $\in$ Cst3..Cst4
INITIALISATION
vrb1 :$\in$ Cst1..Cst2 ||
vrb2 :$\in$ Cst3..Cst4
OPERATIONS
CalculateValues =
BEGIN
vrb1 := Cst5*vrb2 - Cst6
END
END

L'obligation de preuve produite pour l'opération `CalculateValues` permet de vérifier que le corps de l'opération satisfait l'invariant de la machine abstraite ; en particulier, la variable *vrb1* doit appartenir à son domaine de définition soit : $vrb1 \leq Cst2$ et $vrb1 \geq Cst1$.

Nous avons, de la même façon, $vrb2 \leq Cst4$ et $vrb2 \geq Cst3$.
En utilisant ces inégalités dans la substitution *vrb1 := Cst5*vrb2 - Cst6*, nous obtenons les inégalités :
$vrb1 \leq Cst5*Cst4 - Cst6$ et $vrb1 \geq Cst5*Cst3 - Cst6$.
Or, comme la variable *vrb1* appartient à l'intervalle *Cst1..Cst2*, on déduit les inégalités :
$Cst1 \leq Cst5*Cst3 - Cst6$ et $Cst5*Cst4 - Cst6 \leq Cst2$.
Ces inégalités apparaissent comme propriété des constantes de la machine *CstMachine* afin de démontrer la correction de l'opération *CalculateValues*. La valuation des constantes, non présentée ici, doit satisfaire ces inégalités.

Dans l'exemple précédent, les propriétés des constantes à rajouter ne sont pas nombreuses. Dans certains cas, il est préférable de donner directement la valeur de la constante (on utilise pour cela un prédicat d'égalité), plutôt que de lister toutes les propriétés nécessaires pour valider le modèle.

En résumé, on doit indiquer dans la spécification d'une machine de contexte *toutes les propriétés nécessaires* pour que les composants utilisant ce contexte puissent être démontrés. Ces composants ne connaissent en effet les constantes que par leurs propriétés exprimées. Il n'est donc pas étonnant qu'il soit souvent nécessaire d'indiquer ces propriétés de manière très précise, pouvant aller jusqu'à indiquer les valeurs.

8.2.2 Implantation d'une machine de contexte

Seule la clause `VALUES` est présente dans une implantation d'une machine de contexte. Elle permet de donner une valeur effective aux ensembles abstraits et constantes concrètes définis dans la machine abstraite.

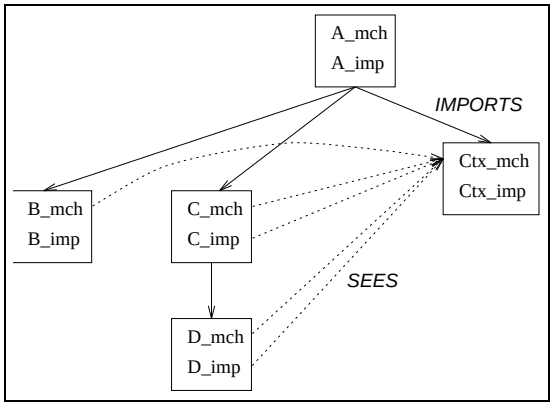
Ce sont les obligations de preuve liées à la vérification de la correction de la valuation qui valident les propriétés introduites dans la machine abstraite.

Comme pour les machines abstraites de contexte, les implantations doivent être indépendantes des autres composants B. Aucun lien de type `IMPORTS`, `EXTENDS` ou `SEES` ne doit être utilisé.

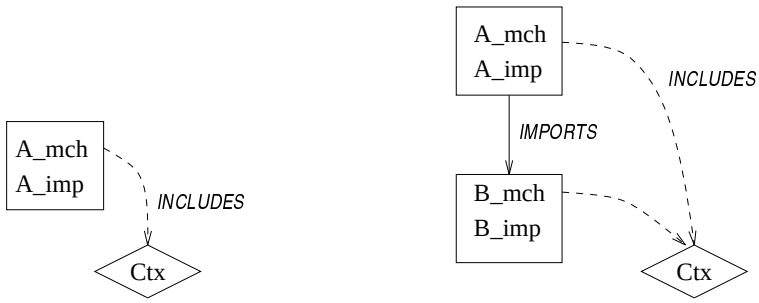
8.3 Les machines de contexte au sein d'un projet B

La plupart des machines de contexte contiennent des constantes concrètes valuées dans l'implantation. Ces constantes doivent être accessibles à différents endroits du projet. La méthode préconisée consiste alors à importer la machine de contexte assez haut dans l'arbre d'implantation, puis à voir (lien SEES) cette machine à partir des composants utilisateurs :

On importe la machine de contexte à un niveau plus haut que son utilisation pour éviter les problèmes "d'aliasing" (voir 4.5.3) : il est interdit d'importer une machine qui est vue au dessus dans la même branche. En fait, ces problèmes ne concernent que la partie dynamique d'une machine et n'ont pas d'influence ici, mais le contrôle statique de l'Atelier B ne peut



- figure 3 : Architecture simple -



- figure 4 : Architecture en cascade -

pas tenir compte de l'absence d'éléments dynamiques. Cette architecture a néanmoins un défaut : le contexte apparaît au niveau de la machine ou `Ctx_mch` est importée, `A_imp` sur notre schéma. Ce contexte est généralement inutile à ce niveau, il surcharge les hypothèses de preuve.

Dans le cas d'une machine de contexte qui ne possède que des ensembles énumérés (automatiquement réalisés) et des constantes abstraites, il est inutile d'écrire une implantation. Dans ce cas la machine de contexte peut être tout simplement incluse dans les composants concernés. Insistons bien sur le fait que cette solutions n'est employable que pour un contexte sans constantes ou ensembles à valuer : sinon cette valuation devrait être faite dans l'implantation de chaque composant utilisateur, et elle n'a aucune raison d'être concordante. Dans un tel schéma, chaque utilisation définirait en fait une constante séparée.

Toutefois, pour une machine de contexte utilisée dans une seule branche de l'arbre d'implantation, il est possible de ne pas écrire d'implantation séparée pour le contexte. On utilise alors le schéma suivant :

Les losanges montrent que les machines abstraites de contexte ne possèdent pas d'implantation.

Architecture simple La machine abstraite `A_mch` inclue la machine de constante `Ctx` ; l'implantation `A_imp` peut valuer directement les données statiques des machines incluses dans la clause `VALUES`. Ces données statiques, par l'utilisation de la clause `INCLUDES`, font partie intégrante de `A_mch`. Les données sont ici valuées *explicitement* dans l'implantation

`A_imp`. En fait, l'implantation du contexte est fait dans `A_imp`.

Architecture en cascade Cette architecture est issue de celle présentée ci-dessus. L' implantation `A_imp` ne value pas *explicitement* les données statiques des machines de contexte dans la clause `VALUES` ; cette fois, l'implantation `A_imp` value ces objets *implicitement*. Pour cela :
– l'implantation `A_imp` importe une machine abstraite `B_mch` qui nécessite la connaissance des données statiques définies dans la machine de contexte,
– le composant `B_mch` inclue la machine de contexte `Ctx`.
Le mécanisme de valuation implicite par homonymie s'applique ; les données statiques incluses dans `A_mch` sont valuées par les données statiques incluses dans `B_mch` (elles portent le même nom et sont de même type).

8.4 Ensembles abstraits et typage

Une question qui est souvent liée aux machines de contexte est l'utilisation d'ensembles abstraits définissant des types séparés. Par exemple, on peut vouloir définir un ensemble `Vitesse` et un ensemble `Distance`. Pour un physicien ces deux grandeurs ne doivent pas être mélangées car elles ne sont pas homogènes, au sens physique du terme.

Si dans la machine de contexte nous définissons `Vitesse` et `Distance` comme deux ensembles abstraits (clause **SETS**), alors nous bénéficions d'un typage fort car le contrôle de types détectera une erreur si nous initialisons par exemple une variable de type vitesse par une distance. Par contre, ces ensembles ne sont alors pas considérés comme numériques (bien que l'on sache par ailleurs qu'ils seront finalement implantés comme tel). Il n'est donc pas possible d'affecter une valeur numérique, ou d'additionner, multiplier, etc.

Une autre solution consiste à définir ces ensembles comme des intervalles numériques :

CONSTANTS
Vitesse, Distance
PROPERTIES
Vitesse = 0..1000 ∧
Distance = 0..1000

Dans ce cas les opérations numériques sont possibles, mais le typage est plus faible : il n'y a plus d'erreur détectée quand une variable de type vitesse et initialisée par une distance.

8.5 Ce que nous avons appris

- Une machine de contexte ne contient que des entités constantes, et aucune opération.
- Les machines de contexte servent à écrire une seule fois le contexte.
- Les machines de contexte servent à isoler les constantes qui vont être utilisées partout, afin de ne pas avoir de telles constantes mélangées avec d'autres éléments risquant d'être modifiés (limitation de l'impact des modifications).
- Il vaut mieux répartir les constantes dans de petites machines de contexte que les grouper dans une seule :
  - pour limiter l'impact de la modification d'une constante,
  - pour rendre possible l'utilisation d'une partie du contexte (réduction du nombre d'hypothèses).
- Les constantes concrètes et les ensembles abstraits doivent être implantés une seule fois : c'est là que leur valeur est fixée.
- Les propriétés exprimées sur les constantes doivent suffire pour pouvoir démontrer les composant utilisateurs. En effet, ceux ci ne voient que ces propriétés.
- L'implantation d'une machine de contexte doit être autonome : pas de **SEES** ou **IMPORTS**.
- La méthode préconisée est d'importer les machines de contexte au dessus de leur utilisation, et de les voir par **SEES**.
- Pour un contexte constitué uniquement d'ensembles énumérés, il suffit d'inclure la machine de contexte partout.
- Si le contexte ne concerne qu'un seule branche : on peut inclure la machine dans la branche et l'implanter au bout.
- Les ensembles abstraits permettent un typage fort, mais ne permettent pas les calculs numériques.

Chapitre 9

Algorithmes et données numériques en B

B est une méthode formelle généraliste en ce sens qu'elle peut être utilisée pour toute sorte de logiciels. Les algorithmes de manipulation des réels sont nombreux et, actuellement, B ne permet pas de les formaliser efficacement. Ainsi, il peut être intéressant d'envisager le développement de librairies de manipulation formelle des réels en B, ou bien d'intégrer un nouvel ensemble REAL au langage B.

La première solution consistant à développer des machines de librairie pour la manipulations des réels ne nécessite pas de modification du langage B, mais en contrepartie elle n'autorise pas les preuves de correction des algorithmes réels implantés sur les flottants.

La seconde solution, plus élaborée, consiste à ajouter un ensemble REAL au langage B, avec la sémantique de l'ensemble $\mathbb{R}$ axiomatisée dans la base de règle du prouveur. Cette solution autorise la description complète de ce qu'est un flottant et de l'erreur commise, d'où la possibilité de preuve de correction des algorithmes effectivement implémentés manipulant les flottants.

Dans cette section nous nous focalisons sur la première solution en détail et nous donnons l'ébauche de la bibliothèque de flottants réalisable avec le langage B actuel.

9.1 Introduction aux flottants

De par la nature finie des représentations de données informatiques, les réels mathématiques ne peuvent être rigoureusement interprétés par un système logiciel. Deux possibilités s'offrent alors au concepteur d'un composant logiciel faisant intervenir des calculs numériques : soit les réels sont traduits en flottants informatiques, soit les calculs réels sont effectués de façon symbolique. La seconde solution a cependant l'inconvénient d'être très peu efficace, c'est pourquoi la première solution est généralement choisie.

Traduire les réels mathématiques en flottants entraîne des problèmes de précision. En effet, les flottants représentent partiellement les réels, et sont limités en précision. Par exemple, sur la plupart des ordinateurs actuels (en simple précision) l'expression $(1,5 \times 10^{12} + 2 \times 10^{-2}) - 1,5 \times 10^{12}$ donne comme résultat 0 au lieu de $0,02$.

Généralement la précision des flottants dépend d'un triplet langage de programmation / compilateur / architecture, ce qui rend le portage des applications difficiles. La confiance

que l'on peut accorder à un composant logiciel faisant intervenir des flottants en est affectée.

Afin de gagner en rigueur et en généralité, il est utile de modéliser les flottants dans un formalisme comme B. En effet, la modélisation B des flottants informatiques va nous obliger à nous abstraire d'une architecture particulière, et d'un langage particulier. L'intérêt de ce gain de rigueur est bien sûr la possibilité de garantir que les flottants seront bien utilisés dans le cadre sollicité, c'est-à-dire que la précision voulue ne sera jamais perdue. Le moyen de fournir ce service est de passer par l'intermédiaire d'une librairie de machines de bases qui vont fournir les opérations nécessaires sur les flottants.

Concevoir une librairie B comme un ensemble cohérent de composants B répondant aux besoins attendus des utilisateurs n'est pas chose aisée. Nous tentons de répondre à ce problème dans ce paragraphe.

Rappelons que notre motivation dans la création d'une librairie B de flottants est de fournir un cadre sûr d'utilisation de réels approximés dans un système logiciel.

Nous nous plaçons volontairement dans cette section dans le formalisme B actuel, et nous donnons les éléments nécessaires à la manipulation des flottants.

9.2 Les décimaux

Pour agir sur les flottants, des opérations sont fournies, notamment les quatre opérations arithmétiques et les opérations de conversion depuis et vers les entiers. Afin de créer un nouveau flottant ou bien de donner un argument flottant aux fonctions de la machine BASIC_FLOAT nous avons besoin de représenter un flottant par des entiers. Les réels sont souvent représentés en base dix, comme par exemple

$N = 6,02250 \times 10^{23}$ nombre d'Avogadro
$h = 1,0545 \times 10^{-27}$ constante de Planck

Afin de faciliter la transcription de ces réels vers des flottants, et de limiter les risques d'erreur il est judicieux de proposer des opérations de conversion proches de ces notations. Ainsi, le couple

$$(60225, 24)$$

représente le nombre d'Avogadro vu sous la forme $0,60225 \times 10^{24}$. Le premier élément du couple est appelé la *mantisse* et le second élément est l'*exposant*. Remarquons qu'avec cette modélisation la virgule décimale est toujours positionnée à gauche de la mantisse.

Remarque : caractérisation des entiers

> Avec cette normalisation (voir la définition ci-dessous), l'ensemble des nombres entiers est caractérisé par le fait que l'exposant est plus grand que la taille de la mantisse. Ainsi le couple $(12345, 2)$ n'est pas entier $(= 12,345)$, mais le couple $(123, 5)$ est un entier $(= 12300)$.

L'ensemble $\mathbb{D}$ des décimaux est l'ensemble des réels d de la forme

$$d = m \times 10^e$$

où $m \in \mathbb{Z}$ et $e \in \mathbb{Z}$. Il est possible de modéliser cet ensemble en B par l'ensemble des tableaux de deux entiers relatifs :

DEFINITIONS
$DECIMAL \mathrel{\widehat{=}} (0\,..\,1) \rightarrow \mathbb{Z}$

Par convention si $d \in DECIMAL$, $d(0)$ représente la mantisse de d et $d(1)$ est l'exposant de d.

Normalisation :

Nous appelons *normalisation* l'opération consistant à convertir un décimal quelconque $m \times 10^e$ sous la forme $0, d_1 d_2 \ldots \times 10^{e'}$, avec $d_1 \in [1, 9]$.

m est un entier de $l = \lfloor Log(m) \rfloor + 1$ chiffres. Pour mettre m sous la forme désirée, il suffit de le multiplier par 10^{-l}. Par conséquent, l'équivalent normalisé du décimal $m \times 10^e$ est le couple (m, e+l).

Remarque :

Le logarithme en base 10 entier est utilisé dans l'ensemble de cette section sur les entiers : il nous sert à compter le nombre de décimaux (nombre de chiffres) nécessaires pour écrire un nombre.

9.3 Les flottants

Les flottants sont définis comme une restriction des décimaux ; ce sont les décimaux implémentables en B :

DEFINITIONS
$FLOAT \mathrel{\widehat{=}} (0\,..\,1) \rightarrow \mathsf{INT}$;
$FLOATp(f) \mathrel{\widehat{=}} f \in (0\,..\,1) \rightarrow \mathsf{INT} \wedge$
$f(0) \in MINMAN\,..\,MAXMAN \wedge$
$f(1) \in MINEXP\,..\,MAXEXP$

$FLOATp(f)$ est un prédicat qui sera utilisé pour typer des variables flottantes f. Les valeurs des constantes bornant la mantisse et l'exposant sont données ci-après.

9.4 Précision des flottants B

Un flottant est une quantité à laquelle est associée une précision. La précision du flottant est exprimée en terme de « nombre significatif de chiffres » utilisés pour représenter le flottant. Par exemple, suivant la précision, les représentations de π sont données dans la table suivante :

Précision	**Valeur de π**
1	3
4	3,142
9	3,14159265

Dans notre modélisation B, la précision maximale est fixée à 9 chiffres significatifs du fait de la taille maximale des entiers de mantisse représentables : la mantisse d'un flottant doit pouvoir être exprimée par un entier implémentable donc elle doit être inférieur à MAXINT $= 2^{31} - 1$; or, $\lfloor Log(2^{31} - 1) \rfloor = 9$.

9.5 Précision du matériel

De par la nature finie des représentations des réels sur machine il existe certaines limites intrinsèques de taille et de précision. Le tableau ci-dessous indique les caractéristiques communes des flottants (simple précision) et des doubles (flottants double précision) des principaux constructeurs actuels (Sun, HP, PC). Nous donnons ces valeurs à titre indicatif d'une part, mais également parce que la librairie des flottants est destinée à être implantée, en particulier, sur ces architectures, et il est nécessaire de contraindre l'utilisateur à ne pas dépasser ces limites.

limite	**valeur**
Plus grand flottant	$3,40 \times 10^{38}$
Plus petit flottant	$1,18 \times 10^{-38}$
Nombre de décimales significatives pour les flottants	6
Plus grand double	$1,80 \times 10^{308}$
Plus petit double	$2,23 \times 10^{-308}$
Nombre de décimales significatives pour les doubles	15

Il nous importe particulièrement ici de savoir que les constructeurs fournissent généralement une précision décimale maximale de 15 chiffres, et des bornes (en valeur absolue) de l'ordre de 10^{308} et 10^{-308}. Cette limite est compatible avec la précision offerte par notre implémentation B des flottants : pour obtenir 9 chiffres significatifs nous appuierons nos machines de base sur les flottants double précision.

L'exposant des réels utilisés en B doit être suffisamment contraint pour que ceux-ci restent dans l'intervalle $[2,23 \times 10^{-308}, 1,80 \times 10^{308}]$. L'intervalle le plus large que nous puissions choisir, avec cette normalisation décimale, est [-307,307] puisque

$$2,23 \times 10^{-308} < d_1, \ldots \times 10^{-307}$$

pour tout chiffre décimal non nul d_1 ($d_1 \in [1,9]$), et

$$e_1, \ldots \times 10^{307} < 1,80 \times 10^{308}$$

pour tout chiffre décimal non nul e_1.

Or, afin de simplifier les expressions de flottants par des entiers nous avons choisi d'utiliser la normalisation décimale $0, d_1 d_2 d_3 d_4 d_5 \times 10^{e'}$ où $e' = e - 1$. C'est pourquoi les valeurs maximales et minimales des exposants sont respectivement 306 et -308.

constante B	**valeur**
MAXDIG	9
MAXMAN	$10^9 - 1$
MINMAN	- MAXMAN
MAXEXP	306
MINEXP	-308

9.6 Addition de deux flottants

Nous présentons dans ce paragraphe l'opération d'addition de deux décimaux, telle qu'il est possible de l'implanter en B.

Les seules opérations numériques actuellement utilisables en B sont les opérations arithmétiques entières dans $\mathbb{Z}$. Par conséquent, nous ne pouvons utiliser que ces opérations.

Afin d'effectuer les calculs décimaux avec les opérations arithmétiques de $\mathbb{Z}$, nous multiplions $f1$ ou $f2$ par une puissance de 10 mettant les deux nombres à la même échelle, puis nous faisons la somme, et nous divisons le résultat par une puissance de 10 annulant la multiplication précédente.

Nous présentons pédagogiquement l'addition de deux flottants quelconques $f1 = (m1, e1)$ et $f2 = (m2, e2)$:

$$\begin{aligned} f1 + f2 &= 0, m1 \times 10^{e1} + 0, m2 \times 10^{e2} && (9.1)\\ &= m1 \times 10^{e1-\lfloor Log(m1)\rfloor+1} + m2 \times 10^{e2-\lfloor Log(m2)\rfloor+1} && (9.2) \end{aligned}$$

De façon à calculer une somme dans $\mathbb{Z}$, il faut que nous obtenions des exposants positifs pour les deux opérandes de l'addition 9.2. Pour cela multiplions en haut et en bas par 10^k, pour un k que nous déterminerons ensuite :

$$\begin{aligned} f1 + f2 &= 10^k \times \frac{m1 \times 10^{e1-\lfloor Log(m1)\rfloor+1} + m2 \times 10^{e2-\lfloor Log(m2)\rfloor+1}}{10^k} && (9.3)\\ &= \frac{m1 \times 10^{k+e1-\lfloor Log(m1)\rfloor+1} + m2 \times 10^{k+e2-\lfloor Log(m2)\rfloor+1}}{10^k} && (9.4) \end{aligned}$$

D'après 9.4, pour que les deux exposants soient positifs, k doit vérifier les deux inéquations suivantes :

$$\begin{aligned} k + e1 - \lfloor Log(m1)\rfloor + 1 &\geq 0\\ k + e2 - \lfloor Log(m2)\rfloor + 1 &\geq 0 \end{aligned}$$

Par conséquent,

$$k = max(\{0, \lfloor Log(m1)\rfloor + 1 - e1, \lfloor Log(m2)\rfloor + 1 - e2\})$$

est la solution optimale à notre problème (nous insérons 0 dans l'ensemble afin de ne pas effectuer de calculs superflus lorsque $f1$ et $f2$ sont déjà des entiers – voir la caractérisation des entiers, § 9.2).

Par exemple considérons les deux décimaux suivants :

décimal	**codage mantisse**	**codage exposant**
$f1 = 2,33 \times 10^{-2}$	$f1(0) = 233$	$f1(1) = -1$
$f2 = 52,04 \times 10^{3}$	$f2(0) = 5204$	$f2(1) = 5$

Afin de faire la somme sur une échelle correcte, il faut additionner 233 et 520400000 puis diviser le résultat par 10^4, puis appliquer la normalisation décimale pour coder le résultat sous la forme $0, ddd \times 10^e$. Pour comprendre comment fonctionnent ces décalages considérons la notation tabulaire de la figure 9.1.

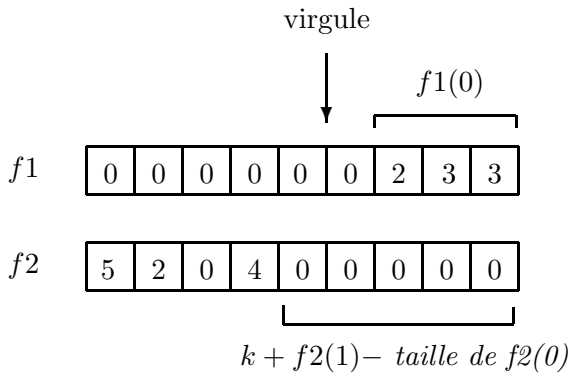


FIG. 9.1 – Notation décimale de $f1$ et $f2$

Pour cet exemple nous avons

$$\begin{aligned} \lfloor Log(f1(0)) \rfloor + 1 &= 3 \\ \lfloor Log(f1(0)) \rfloor + 1 - f1(1) &= 4 \end{aligned}$$

$$\begin{aligned} \lfloor Log(f2(0)) \rfloor + 1 &= 4 \\ \lfloor Log(f2(0)) \rfloor + 1 - f2(1) &= -1 \end{aligned}$$

Donc $k = 4$ et

$$\begin{aligned} f1 + f2 &= \frac{f1(0) \times 10^{k+f1(1)-\lfloor Log(f1(0)) \rfloor+1} + f2(0) \times 10^{k+f2(1)-\lfloor Log(f2(0)) \rfloor+1}}{10^k} \\ &= \frac{233 \times 10^{4+(-1)-3} + 5204 \times 10^{4+5-4}}{10^4} \\ &= \frac{233 + 520400000}{10^4} \end{aligned}$$

Ce dernier numérateur étant bien l'addition entière que nous souhaitions réaliser. Nous obtenons le décimal

$$f1 + f2 = 520400233 \times 10^{-4}$$

Après normalisation nous obtenons le couple $(520400233, -4 + 9) = (520400233, 5)$.

9.7 Le composant BASIC_FLOAT

Nous présentons ci-dessous le composant BASIC_FLOAT de manipulation des flottants en B. En plus des opérations de conversion, seule l'opération arithmétique d'addition est spécifiée afin d'alléger le texte (remarquons que cette opération est déjà volumineuse).

/*————————————————————————————————-
Modélisation des flottants en B
————————————————————————————————-*/

MACHINE
BASIC_FLOAT

DEFINITIONS
/* *Ensemble des décimaux* */
$DECIMAL \mathrel{\widehat{=}} (0\,..\,1) \rightarrow \mathbb{Z}$;

/* *Ensemble des flottants (ou décimaux implémantables)* */
$FLOAT \mathrel{\widehat{=}} (0\,..\,1) \rightarrow \mathsf{INT}$;

/* *Le prédicat FLOATp(f) permet de définir le flottant f, modélisé par un tableau de 2 entiers :*
- la mantisse m
- l'exposant e,
tels que : $f = m \times 10^e$ */
$FLOATp(f) \mathrel{\widehat{=}} f \in FLOAT \wedge$
$f(0) \in MINMAN\,..\,MAXMAN \wedge$
$f(1) \in MINEXP\,..\,MAXEXP$;

/* *L'expression f2i(f) permet de convertir le flottant f en entier* */
$f2i(f) \mathrel{\widehat{=}} f(0) * 10 ** max(\{f(1) - Log10(f(0)), 0\})/10 ** max(\{Log10(f(0)) - f(1), 0\})$

SEES float_ctes

La machine `float_ctes` est une machine de constantes définissant les bornes `MINMAN`, `MAXMAN`, etc. ainsi que le logarithme en base 10. Elle doit être vue par tout composant souhaitant manipuler des flottants :

MACHINE
float_ctes

VISIBLE_CONSTANTS
$MAXDIG$, /* Precision de la mantisse (nombre de chiffres) */
$MINMAN$, /* Valeur minimale de la mantisse d'un flottant */
$MAXMAN$, /* Valeur maximale de la mantisse d'un flottant */
$MINEXP$, /* Valeur minimale de l'exposant d'un flottant */
$MAXEXP$ /* Valeur maximale de l'exposant d'un flottant */

HIDDEN_CONSTANTS
$Log10$ /* Logarithme entier en base 10, c'est-à-dire $\lfloor Log() \rfloor + 1$ */

PROPERTIES
$MAXDIG \in \mathsf{NAT} \ \wedge$
$MINMAN \in \mathsf{INT} \ \wedge$
$MAXMAN \in \mathsf{NAT} \ \wedge$
$MINEXP \in \mathsf{INT} \ \wedge$
$MAXEXP \in \mathsf{NAT} \ \wedge$
$Log10 \in \mathsf{NATURAL1} \rightarrow \mathsf{NATURAL} \ \wedge$

$MAXDIG = 9 \ \wedge$
$MAXMAN = 10 ** MAXDIG - 1 \ \wedge$
$MINMAN = -MAXMAN \ \wedge$
$MINEXP = -308 \ \wedge$
$MAXEXP = 306 \ \wedge$
$\forall x1.(x1 \in \mathsf{NATURAL}$
$\Rightarrow$
$Log10(x1) = min(\{y1 | y1 \in \mathsf{NATURAL} \ \wedge x1 < 10 ** (y1)\}))$

END

L'opération B d'addition de deux flottants $f1$ et $f2$ est définie si la somme $f1+f2$ (calculée dans $\mathbb{Z}$) peut être normalisée. La mantisse du flottant consiste en les 9 chiffres les plus significatifs de $f1+f2$ donc elle est dans l'intervalle $MINMAN..MAXMAN$. L'exposant de la somme est (avec les notations du § 9.6) $-k + \lfloor Log(f1 + f2) \rfloor + 1$. Cet exposant doit être compris dans l'intervalle $MINEXP..MAXEXP$.

DEFINITIONS

$CoefAdd(f1, f2) \mathrel{\widehat{=}} max(\{0, Log10(f1(0)) - f1(1), Log10(f2(0)) - f2(1)\})$;
$Som(f1, f2) \mathrel{\widehat{=}}$
$(f1(0) * 10 **(CoefAdd(f1, f2) + f1(1) - Log10(f1(0))) +$
$f2(0) * 10 **(CoefAdd(f1, f2) + f2(1) - Log10(f2(0))))$

OPERATIONS

/ Operation d'addition de 2 flottants f1 et f2 dans f0 */*
$f0 \longleftarrow$ **AddFloat** $(f1, f2) \mathrel{\widehat{=}}$
PRE
$FLOATp(f1) \wedge$
$FLOATp(f2) \wedge$

/ Non debordement de l'exposant de la somme f1 + f2 */*
$-CoefAdd(f1, f2) + Log10(Som(f1, f2)) \in MINEXP \mathinner{..} MAXEXP$
THEN
ANY
$som0, som1$
WHERE
$som0 \in MINMAN \mathinner{..} MAXMAN \wedge$
$som1 \in MINEXP \mathinner{..} MAXEXP \wedge$
/ Division entière de la mantisse longue */*
$som0 = Som(f1, f2)/(10 **CoefAdd(f1, f2)) \wedge$
$som1 = -CoefAdd(f1, f2) + Log10(Som(f1, f2))$
THEN
$f0 := \{(0 \mapsto som0), \quad (1 \mapsto som1)\}$
END
END
;

```
    /* Operation de conversion d'un flottant f1 en entier i0 */
    i0 ⟵ Float2Int(f1) ≙
        PRE
            FLOATp(f1) ∧

            /* Non debordement de la conversion flottant-entier */
            f2i(f1) ∈ INT
        THEN
            i0 := f2i(f1)
        END
    ;

    /* Operation de conversion d'un entier i1 en flottant f0 (ne deborde jamais)*/
    f0 ⟵ Int2Float(i1) ≙
        PRE
            i1 ∈ INT
        THEN
            f0 : (f0 ∈ FLOAT ∧ f0 = {(0 ↦ i1), (1 ↦ Log10(i1))})
        END
END
```

9.8 Utilisation du composant BASIC_FLOAT

Nous donnons ci-dessous un exemple d'utilisation de la machine des flottants. Notons que la définition `FLOATp` est répétée ici puisque la clause **DEFINITIONS** est locale à chaque composant.

MACHINE
test_float

INCLUDES
float_ctes

DEFINITIONS
$FLOATp(f) \mathrel{\widehat{=}} f \in (0\,..\,1) \rightarrow \mathsf{INT} \wedge$
$f(0) \in MINMAN\,..\,MAXMAN \wedge$
$f(1) \in MINEXP\,..\,MAXEXP$

VISIBLE_VARIABLES
x1, x2, x3, i1

INVARIANT
FLOATp(x1) $\wedge$
FLOATp(x2) $\wedge$
FLOATp(x3) $\wedge$
i1 $\in$ INT

INITIALISATION
x1 :(FLOATp(x1)) ||
x2 :(FLOATp(x2)) ||
x3 :(FLOATp(x3)) ||
i1 :$\in$ INT

END

IMPLEMENTATION
test_float_i

REFINES test_float

IMPORTS float, float_ctes

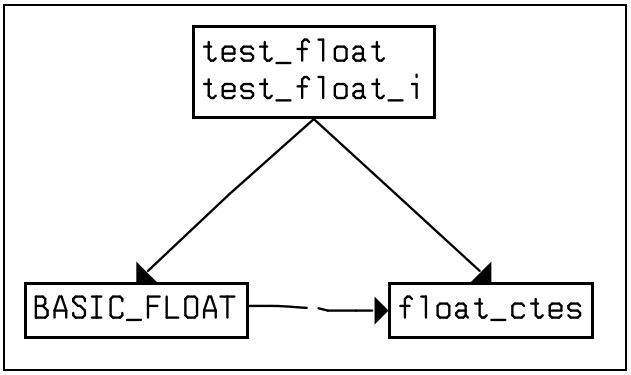
INITIALISATION
$x1 := \{0 \mapsto 1234, 1 \mapsto 2\}$; /* *x1 := 12,34* */
$x2 := \{0 \mapsto 2, 1 \mapsto 4\}$; /* *x2 := 2000* */
$x3 \longleftarrow$ AddFloat$(x1, x2)$; /* *x3 := x1 + x2* */
$i1 \longleftarrow$ Float2Int$(x3)$

END

Nous donnons le graphe du module test_float produit par l'atelier B.

L'initialisation du composant test_float_i engendre 24 obligations de preuve avec l'Atelier B, dont 9 ne se prouvent pas automatiquement. Parmi ces obligations de preuve certaines sont relativement complexes, et cela provient de deux facteurs :

– Tout d'abord la modélisation du logarithme en base 10 sous la forme d'un ensemble en compréhension est difficilement manipulable ; une définition constructive (c'est-à-dire



une définition donnant un moyen de calculer le résultat à partir d'une entrée) aurait facilité les preuves.
– Ensuite, la modélisation des flottants sous la forme d'un tableau contenant la mantisse et l'exposant est intrinsèquement une solution de programmation plus qu'une solution de spécification. Afin de rester plus abstrait tout en fournissant les moyens de décrire des calculs réels, un véritable type REAL dans le langage B nous aurait fourni une solution plus satisfaisante.

9.9 Les réels en B

Dans les paragraphes précédents nous avons présenté une modélisation des flottants en B, à même de constituer les fondements d'une librairie de manipulation des flottants.

Cependant, nous avons vu que les obligations de preuve à prouver pour un petit exemple comme celui du composant test_float_i sont très complexes. La complexité de cette modélisation des flottants peut devenir rédhibitoire pour des exemples plus réalistes.

De plus, la complexité des formules à établir pour les différentes opérations arithmétiques posent le problème de la confiance que l'on peut avoir dans leur exactitude. Malgré tout le soin apporté à la démonstration manuelle du § 9.6 des erreurs peuvent se glisser dans les formules, ainsi que lors de leur retranscription dans le module BASIC_FLOAT.

Nous avons esquissé l'idée d'une solution à ces problèmes à la fin du paragraphe 9.8, c'est-à-dire utiliser en B un type réel tout comme il est actuellement possible de manipuler les types entiers : les éléments de l'ensemble mathématique $\mathbb{Z}$ sont les éléments de l'ensemble B appelé INTEGER. L'idée du type REAL en B serait de manipuler formellement les éléments de $\mathbb{R}$. Bien sûr, ces réels, non implantables, resteront des quantités de spécification, et n'apparaîtront pas en implantation.

Actuellement, de nombreux logiciels utilisent les flottants par nécessité pour les applications traitées. Les calculs d'erreurs, leur propagation et la mesure de leurs impacts sont rarement effectués car il s'agit de problèmes mathématiques non triviaux. La modélisation B des réels permettrait aux concepteurs B d'écrire des spécifications relativement simples et intuitives. La complexité du problème des approximations des réels par les flottants pourrait alors être reportée dans la preuve d'une implantation B important la machine BASIC_FLOAT.

La modélisation d'un processus physique se fait naturellement en utilisant des nombres

réels, puisque la physique continue est basée sur l'emploi de tels nombres. Pour que les modèles B aient une correspondance physique directe, il faut donc intégrer ces nombres au langage, même si les programmes concrets n'utilisent que des flottants.

Par exemple, nous pouvons vouloir modéliser un programme qui calcule une vitesse d'après des mesures de distance et de temps. Nous disposons de la distance et du temps mesurés : d_m et t_m qui sont des flottants informatiques et nous calculons $v_m = d_m/t_m$. Quelle est la spécification de notre programme ? Pour pouvoir faire une spécification significative, il faut avoir modélisé les capteurs de distance et de temps :

$$d_m \in (d - e) \mathinner{..} (d + e)$$

$$t_m \in (t - e) \mathinner{..} (t + e)$$

où d et t sont les modélisations *réelles* de la distance et du temps et e est un terme d'erreur. Nous pouvons alors spécifier :

$$v_m : (v_m \in (d/t - e') \mathinner{..} (d/t + e'))$$

Où e' est l'erreur maximale tolérée. Une telle spécification ne peut s'exprimer qu'avec des nombres réels.

L'ajout d'un type REAL en B doit se faire de façon coordonnée entre la théorie et l'atelier B. Les points à étudier dans l'optique d'un tel enrichissement sont :
- Ajouter $\mathbb{R}$ dans les fondements mathématiques de B ;
- Ajouter le type numérique REAL au langage B ;
- Perfectionner le prouveur et le générateur d'obligations de preuve sur les calculs des réels, et surtout sur les problèmes de précision.

9.10 Ce que nous avons appris

- Pour manipuler des nombres à virgule en B, on peut utiliser des librairies de nombre flottants, puis également ajouter $\mathbb{R}$ dans le langage B.
- Avec les deux solutions ci-dessus, on pourrait écrire des modélisations utilisant des nombres réels (précision infinie) et implanter sur les librairies de flottants, réalisant la spécification à une certaine précision près.
- Les problèmes de précision doivent être pris en compte pour que la preuve soit juste. Avec B, ces questions rarement traitées doivent être prises en compte.
- Les flottants sont les nombres exprimés sous la forme d'une mantisse et d'un exposant de taille fixée. Les opérations arithmétiques sur ces nombres sont difficiles à formaliser à causes des approximations faites.
- Le comportement des nombres flottants dépend du langage de programmation, du compilateur et de l'architecture matérielle employée.
- Les décimaux sont les nombres qui peuvent se mettre sous la forme $0, m \times 10^e$ où m et e sont des entiers relatifs (mantisse et exposant).
- On modélise les décimaux comme une fonction de $\{0, 1\}$ dans $\mathbb{Z}$. L'image de 0 est la mantisse, l'image de 1 est l'exposant.
- La forme $0, m$ pour la mantisse permet d'éviter que plusieurs décimaux ne correspondent au même nombre réel.
- La précision des flottants dépend de la taille des entiers informatiques. Avec $\mathsf{MAXINT} = 2^{31} - 1$, on obtient 9 chiffres significatifs.
- On peut créer une machine de base offrant les opérations arithmétiques de base et les conversion entre flottants et entiers (exemple de $BASIC\_FLOAT$), les flottants étant vus de l'utilisateur comme des tableaux à deux éléments (mantisse, exposant). Il y a les limites suivantes :
  - L'utilisateur peut difficilement spécifier à partir de cette modélisation,
  - Les opérations sont assez complexes. Par exemple, l'addition de deux flottants nécessite leur réduction au même exposant, l'addition, puis la renormalisation.
- L'introduction de $\mathbb{R}$ dans les fondements mathématiques de B permettrait la modélisation par équations physiques.

Chapitre 10

Le contrôle de l'ordre des opérations

Il arrive fréquement que les spécifications informelle d'un logiciel contiennent des prérequis du genre "telle mesure devra être effectuée avant tel calcul...". La question se pose alors naturellement : puisque l'opérateur de séquencement est interdit dans les machines abstraites, comment formaliser ces prérequis ?

10.1 Implantations de spécification

La spécification complète d'un logiciel complexe n'est jamais regroupée dans une seule machine abstraite. Elle est en fait constituée d'un morceau de l'arbre d'implantation du projet complet, c'est-à-dire que les quelques exigences de plus haut niveau sont dans une machine implantée sur quelques machines où apparaissent les exigences de niveau inférieur, et ainsi de suite. Il y a donc des implantations *considérées* comme faisant partie de la spécification du produit. Dans de telles implantations, il est évidemment facile de faire figurer les prérequis de séquentialité.

Un bon exemple de ce procédé est la structure classique pour un logiciel réactif devant à chaque cycle faire des mesures, effectuer des calculs et imposer des consignes. Généralement, la spécification de plus haut niveau ne précise que les conditions de sortie du cycle : par exemple, l'arrêt ne doit se produire que s'il y a une panne ou sur ordre de l'opérateur :

MACHINE
Reactive
VARIABLES
Stop, *Failure*
INVARIANT
$Stop \in BOOL \wedge$
$Failure \in BOOL$
INITIALISATION
$Stop, Failure := FALSE, FALSE$
OPERATIONS
Cycle = **BEGIN**
$Stop, Failure : (Stop \in BOOL \wedge$
$Failure \in BOOL \wedge$
$\neg(Stop = FALSE \ \wedge \ Failure = FALSE))$
END
END

Cette spécification de plus haut niveau dit seulement que l'arrêt ne se produit que sur panne ou ordre de l'opérateur. La spécification informelle précise ensuite les phases de mesure, de calcul et de contrôle à faire :

MACHINE
PhysMach
ABSTRACT_CONSTANTS
ff
PROPERTIES
$ff \in NAT \rightarrow NAT$
CONCRETE_VARIABLES
$StopButton, FailDetect$
VARIABLES
$tt, vv, InfoIn, InfoOut$
INVARIANT
$tt \in NATURAL \wedge$
$StopButton = bool(tt = 0) \wedge$
$FailDetect \in BOOL \wedge$
$vv \in NAT \wedge$
$InfoIn \in NAT \wedge$
$InfoOut \in NAT$
INITIALISATION
$StopButton, FailDetect, vv, InfoIn, InfoOut := FALSE, FALSE, 0, 0, 0 \;||$
$tt :\in NATURAL1$
OPERATIONS
Measure = **PRE**
$tt \neq 0$
THEN
$StopButton := bool(tt = 1) \;||$
$tt := tt - 1 \;||$
$FailDetect :\in BOOL \;||$
$InfoIn :\in NAT$
END;
Calculate = **PRE**
$FailDetect = FALSE$
THEN
$vv := ff(InfoIn)$
END;
Control = **BEGIN**
$InfoOut := vv$
END
END

Pour simplifier, nous avons supposé que mesure, calcul et consigne sont spécifiés dans la même machine. La variable *tt* abstraite représente le nombre de cycles au bout duquel l'opérateur arrête la machine. On ne connaît pas ce nombre, mais il existe car le programme finit forcément par s'arrêter. Cette variable *tt* ne sera jamais implantée, elle va se transmettre jusqu'à la machine de base du bouton d'arrêt, dans laquelle nous devrons supposer l'opérateur appuie sur le bouton au bout d'un temps fini.

L'ordre des opération peut maintenant être spécifié dans l'implantation de la machine *Reactive* sur la machine *PhysMach*, implantation qui fait partie de la spécification puisque le cahier des charges indique explicitement l'ordre dans lequel les opérations doivent se faire.

```
IMPLEMENTATION
    Reactive_imp
REFINES
    Reactive
IMPORTS
    PhysMach
INVARIANT
    Stop = StopButton ∧
    Failure = FailDetect
OPERATIONS
    Cycle = WHILE StopButton = FALSE ∧ FailDetect = FALSE DO
        Measure;
        IF FailDetect = FALSE THEN
            Calculate;
            Control
        END
    INVARIANT
        tt ∈ NATURAL ∧
        StopButton = bool(tt = 0)
    VARIANT
        tt
    END
END
```

Cette implantation “de spécification” indique que le logiciel doit à chaque cycle acquérir les grandeurs, procéder au calcul et au contrôle sauf si une panne a été détectée, et s’arrêter en cas de panne ou sur ordre extérieur. Il s’agit bien d’exigences issues du cahier des charges.

10.2 Séquencement imposé par le résultat

Une autre manière d’imposer l’ordre dans lequel certains traitements seront faits est de spécifier l’état à atteindre de telle manière que seul l’ordre correct permette de l’atteindre. Considérons la spécification suivante :

MACHINE
Result
CONCRETE_VARIABLES
bb
VARIABLES
vv
DEFINITIONS
$inv(vx, bx) == (vx \in NAT \ \wedge \ bx = bool(vx \leq 100))$;
$Keep(vx, bx) == (vx, bx : inv(vx, bx))$
INVARIANT
$inv(vv, bb)$
INITIALISATION
$Keep(vv, bb)$
OPERATIONS
Move = $Keep(vv, bb)$
END

La spécification nous indique que l'opération *Move* doit faire évoluer *vv* et *bb* de telle manière que ce dernier représente toujours l'ordre de *vv* par rapport à 100. Ceci impose l'ordre d'évaluation : il faut bien sûr faire évoluer *vv* d'abord. Une implantation de cette machine pourrait être la suivante (implantation supposée ne pas faire partie de la spécification, cette fois) :

```
IMPLEMENTATION
    Result_imp
REFINES
    Result
IMPORTS
    ResultI
INVARIANT
    vv = v2
INITIALISATION
    VAR zz IN
        zz ← Getv2 ;
        bb := FALSE;
        IF zz ≤ 100 THEN
            bb := TRUE
        END
    END
OPERATIONS
    Move = VAR zz IN
        GetNext;
        zz ← Getv2 ;
        bb := FALSE;
        IF zz ≤ 100 THEN
            bb := TRUE
        END
    END
END
```

La machine importée *ResultI* possédant la variable *v2* offrirait une opération *GetNext* pour faire évoluer cette variable et une opération *Getv2* pour accéder à sa valeur. Nous ne la représenterons pas ici.

Dans ce cas la forme de la spécification abstraite impose que l'opération *GetNext* soit appelée *avant* le calcul de *bb* : sinon nous ne pouvons pas établir que *bb* représente l'ordre de la variable par rapport à 100, ce qui se traduit par des PO fausses.

10.3 Phasage par préconditions

Il peut arriver que la spécification indique explicitement que les services offerts par une certaine partie du logiciel n'ont de sens que s'ils sont employés dans un certain ordre. Le plus souvent ceci est explicite dans la définition même de ces services. Dans la spécification B, les préconditions traduisent alors cet ordre imposé. Par exemple, voici la spécification d'un module qui crée ou détruit des objets :

MACHINE
  $Object(OBJECTS)$
VARIABLES
  $LivingObj$
INVARIANT
  $LivingObj \subseteq OBJECTS$
INITIALISATION
  $LivingObj := \varnothing$
OPERATIONS
  $rr \leftarrow$ Create = **PRE**
    $LivingObj \neq OBJECTS$
  THEN
    ANY *new* **WHERE**
      $new \in OBJECTS - LivingObj$
    THEN
      $LivingObj := LivingObj \cap \{new\} \parallel$
      $rr := new$
    END
  END ;
  Delete(ee) = **PRE**
    $ee \in LivingObj$
  THEN
    $LivingObj := LivingObj - \{ee\}$
  END ;
  $bb \leftarrow$ Test(ee) = **PRE**
    $ee \in OBJECTS$
  THEN
    $bb := bool(ee \in LivingObj)$
  END
END

Les préconditions imposeent l'usage *dans l'ordre* des opérations de cette machine : il n'est pas possible par exemple, de détruire des objets avant de les avoir créés.

10.4 Variables de phasage

Dans certains cas, la spécification indique qu'un module doit offrir des services à utiliser dans un ordre donné sans que cet ordre puisse découler de la modélisation de ces services. On a alors recours à des *variables de phasage* abstraites qui permettent d'écrire les préconditions imposant l'ordre. Ces variables ne sont jamais implantées, elles ne servent qu'à produire les obligations de preuve qui démontrent que l'ordre d'utilisation est respecté. Voici un exemple avec trois services :

```
MACHINE
    Phase
VARIABLES
    ph
INVARIANT
    ph ∈ {1, 2, 3}
INITIALISATION
    ph := 1
OPERATIONS
    op1 = PRE
        ph = 1
    THEN
        ph := 2
    END;
    op2 = PRE
        ph = 2
    THEN
        ph := 3
    END;
    op3 = PRE
        ph = 3
    THEN
        ph := 1
    END
END
```

Bien sûr, la spécification des trois services devraient en réalité contenir aussi la modélisation de ce qu'ils font. La variable de phase *ph* impose d'utiliser ces services dans l'ordre. Elle ne sera jamais implantée.

10.5 Ce que nous avons appris

– Certaines implantations font partie de la spécification : on peut alors y "spécifier" l'ordre des traitements.
– La modélisation du résultat à obtenir impose souvent l'ordre des opérations.
– L'ordre d'emploi des services spécifiés dans un module peut être imposé par les préconditions de ces services. Si la modélisation naturelle des services ne conduit pas à des préconditions suffisantes, on peut ajouter des variables abstraites jamais implantées, qui permettent d'imposer l'ordre (variables de phase).

Chapitre 11

Explosion combinatoire du nombre de PO : origine et solution

Le développement d'un module B requiert la plus grande vigilance : une formalisation inadaptée peut mener à la génération d'un "trop" grand nombre d'obligations de preuve. En fait, ce n'est pas tant le nombre d'obligations de preuve produites qui est gênant, mais le nombre d'obligations de preuve non démontrées automatiquement ou pour lesquelles la démonstration automatique est trop coûteuse. Il faut donc trouver un compromis entre le nombre d'obligations de preuve produites et leur complexité, pour que le travail de preuve soit le plus simple possible.

Dans ce qui suit, nous nous attacherons à montrer, dans un premier temps, les origines de ces explosions combinatoires du nombre d'obligations de preuve ; nous étudierons alors les techniques préconisées pour limiter ces effets.

Les origines de ce type de problème sont nombreuses :
– séquencement de structures de contrôle,
– raffinement algorithmique trop compliqué,
– décomposition en machines abstraites inadaptée.

11.1 Séquencement de structures de contrôle

Considérons la machine abstraite suivante :

MACHINE
Complex
CONCRETE_CONSTANTS
value_min, value_max
PROPERTIES
value_min ∈ NAT ∧
value_max ∈ NAT ∧
value_min ≤ value_max
CONCRETE_VARIABLES
value
INVARIANT
value ∈ value_min .. value_max
INITIALISATION
value :∈ value_min .. value_max
OPERATIONS
Op(param) =
PRE param ∈ NAT
THEN
value :∈ value_min .. value_max
END
END

L'opération de cette machine abstraite positionne la variable `value` en fonction de la valeur du paramètre `param`.

IMPLEMENTATION
Complex_1
REFINES
Complex
VALUES
value_min = 30 ;
value_max = 50
DEFINITIONS
Tranche1 == 35 ;
Tranche2 == 40 ;
Tranche3 == 45
INITIALISATION
value := value_min
OPERATIONS
Op(param) =
BEGIN
IF (param < value_min) **THEN**
value := value_min **END** ;
IF (param > value_max) **THEN**
value := value_max **END** ;
IF (param ≥ value_min ∧ param ≤ Tranche1) **THEN**
value := Tranche1 **END**;
IF (param > Tranche1 ∧ param ≤ Tranche2) **THEN**
value := Tranche2 **END** ;
IF (param > Tranche2 ∧ param ≤ Tranche3) **THEN**
value := Tranche3 **END**;
IF (param > Tranche3 ∧ param ≤ value_max) **THEN**
value := value_max **END**
END
END

Une vérification visuelle montre que cette implantation raffine bien sa spécification ; en effet, dans tous les cas, la variable `value` est affectée par une valeur appartenant à l'intervalle `value_min`, `value_max`. La génération automatique des obligations de preuve pour cette implantation produit énormément de PO par rapport à la complexité du composant (sur l'Atelier B : 84 POs et 156 POs évidentes, notons que toutes sont néanmoins démontrées automatiquement). En fait, la preuve doit établir que la variable `value` fabriquée par la succession de `IF` appartient à l'intervalle `value_min`..`value_max`. Pour ce faire, le corps de chaque `IF` est traduit par un prédicat. L'effet du premier `IF` se traduit par (param < value_min ⇒ value = value_min). Le prédicat comprend en fait deux cas : (param < value_min) et son contraire. Dans chacun de ces prédicat, l'effet du deuxième `IF` est traduit, ce qui les dédouble de la même façon. Finalement, nous obtenons 2^6 prédicats à démontrer pour vérifier la correction de l'opération.

Nous allons maintenant proposer une autre implantation de notre exemple.

```
IMPLEMENTATION
    Complex_1
REFINES
    Complex
VALUES
    value_min = 30 ;
    value_max = 50
DEFINITIONS
    Tranche1 == 35 ;
    Tranche2 == 40 ;
    Tranche3 == 45
INITIALISATION
    value := value_min
OPERATIONS
    Op(param) =
    BEGIN
    IF ( param < value_min ) THEN
        value := value_min
    ELSIF ( param > value_max ) THEN
        value := value_max
    ELSIF ( param ≤ Tranche1 ) THEN
        value := Tranche2
    ELSIF ( param ≤ Tranche2 ) THEN
        value := Tranche2
    ELSIF ( param ≤ Tranche3 ) THEN
        value := Tranche3
    ELSIF ( param ≤ value_max ) THEN
        value := value_max
    END
END
```

La génération automatique des obligations de preuve produit cette fois un nombre raisonnable de PO (11 POs et 26 POs évidentes, sur l'Atelier B, tout est démontré automatiquement bien plus rapidement).

L'explosion, obtenue lors de la première implantation, est principalement due au séquencement (symbole ;). La construction des prédicats qui caractérisent une quantité modifiée en séquence peut facilement accumuler de multiples cas. C'est pourquoi le séquencement est interdit en spécification, sinon la preuve qu'un séquencement raffine un séquencement produirait un trop grand nombre de cas à vérifier.

11.2 Raffinement algorithmique trop compliqué

Si les obligations de preuve sont peu nombreuses mais mathématiquement très difficiles à démontrer, cela signifie souvent qu'il manque un raffinement pour passer *progressivement* de la spécification abstraite au programme concret.

Il peut aussi se produire une explosion combinatoire entre deux raffinements ou entre un raffinement et l'implantation. Dans ce cas il s'agit souvent d'un raffinement trop "programmé" : le raffinement correspond pratiquement à un programme et le niveau suivant

à un autre programme. On se retrouve alors à démontrer qu'un programme raffine un autre programme, ce qui produit naturellement une explosion combinatoire de cas. C'est pour éviter que ce genre de chose ne se produise au niveau des machines abstraites que les boucles et le séquencement y sont interdits. Les modèles de plus haut niveau sont ainsi contraints à être mathématiques et non programmés. Le problème peut néanmoins apparaître au niveau des raffinement où le séquencement est autorisé.

11.3 Décomposition inadaptée

Considérons l'exemple suivant :

MACHINE
Affich
OPERATIONS
Op(aa,bb,cc) =
PRE
aa $\in NAT \wedge$
bb $\in NAT \wedge$
cc $\in NAT$
THEN
skip
END
END

Cette machine dont la spécification est vide sert à afficher *zéro*, *un* ou *beaucoup* suivant la valeur des trois paramètres. Des messages de fin de traitement sont affichés à chaque étape.

```
IMPLEMENTATION
    Affich_1
REFINES
    Affich
IMPORTS
    BASIC_IO
OPERATIONS
    Op(aa,bb,cc) =
    BEGIN
    CASE aa OF
        EITHER 0 THEN
            STRING_WRITE("zero\n")
        OR 1 THEN
            STRING_WRITE("un\n")
        ELSE
            STRING_WRITE("beaucoup\n")
        END
    END ;
    STRING_WRITE("fin traitement 1\n");
    CASE bb OF
        EITHER 0 THEN
            STRING_WRITE("zero\n")
        OR 1 THEN
            STRING_WRITE("un\n")
        ELSE
            STRING_WRITE("beaucoup\n")
        END
    END ;
    STRING_WRITE("fin traitement 2\n");
    CASE cc OF
        EITHER 0 THEN
            STRING_WRITE("zero\n")
        OR 1 THEN
            STRING_WRITE("un\n")
        ELSE
            STRING_WRITE("beaucoup\n")
        END
    END ;
    STRING_WRITE("fin traitement 3\n")
END
```

On pourrait penser que l'implantation ci-dessus ne va générer que très peu d'obligations de preuves, car il suffit de prouver que tous les arguments de `STRING_WRITE` sont des `STRING` et que l'opération raffine sa spécification (ce qui est évident car la spécification est `skip` et que la machine n'a pas de variables). On s'attendrait donc à trouver autant de PO qu'il y a d'appels à `STRING_WRITE`.

En fait, cette l'implantation génère beaucoup d'obligations de preuve (78 obligations et 31 obligation de preuve triviales sur l'Atelier B). En fait, il y a encore un problème dû au séquencement : le premier `CASE` génère 3 PO, mais le deuxième en génère 3×3 car il faut se placer dans les trois cas concernant le premier `CASE`. En fait, ces cas n'influent pas sur

la suite mais pour le savoir, il faudrait rentrer dans le détail de ce que fait ce premier `CASE` ce qui dépasse la simple génération des PO. En fait, le séquencement produit toujours une *accumulation* de cas et d'informations qui complique et multiplie les PO des dernières instructions.

Comme pour l'exemple présenté dans le paragraphe précédent, il faut éviter de mettre des substitutions en séquence après une substitution conditionnelle.

L'opération décrite dans l'implantation fait intervenir trois `CASE` en séquence, chacun d'eux réalisant des traitements similaires. Nous venons d'identifier un service commun, appelons-le `decrypte`, qui peut être défini dans un module séparé `Util`.

L'implantation de la machine abstraite est alors :

```
IMPLEMENTATION
    Affich_2
REFINES
    Affich
IMPORTS
    BASIC_IO, Util
OPERATIONS
    Op(aa,bb,cc) =
    BEGIN
        decrypte(aa) ;
        STRING_WRITE("fin traitement 1\n");
        decrypte(bb) ;
        STRING_WRITE("fin traitement 2\n");
        decrypte(cc) ;
        STRING_WRITE("fin traitement 3\n")
    END
END
```

```
MACHINE
    Util
OPERATIONS
    decrypte(xx) =
    PRE
        xx ∈ NAT
    THEN
        skip
    END
END
```

```
IMPLEMENTATION
    Util_1
REFINES
    Util
IMPORTS
    BASIC_IO
OPERATIONS
    decrypte(xx) =
    BEGIN
    CASE xx OF
        EITHER 0 THEN
            STRING_WRITE("zero\n")
        OR 1 THEN
            STRING_WRITE("un\n")
        ELSE
            STRING_WRITE("beaucoup\n")
        END
    THEN
        skip
    END
END
```

L'utilisation d'un seul niveau de décomposition ramène à 6 le nombre d'obligations de preuve produites (au lieu de 78) pour les composants `Affich, Affich_2`, `Util` et `Util_1`.

11.4 Ce que nous avons appris

- Suivant la manière dont les composants B sont écrits, il peut se produire des “explosions de PO”, c’est-à-dire que les PO sont très nombreuses alors qu’il suffit d’écrire les choses d’une autre manière équivalente pour avoir très peu de PO.
- Pour éviter les explosions : ne jamais mélanger des séquencements longs avec des instructions conditionnelles.
- Ne pas avoir une modélisation trop “programmée” dans un raffinement (séquencements, conditions imbriquées ...).
- Si les PO sont mathématiquement difficiles : introduire un raffinement intermédiaire pour répartir la difficulté.
- Pour faire des séquencements longs, rejeter les expressions conditionnelles dans des opérations importées.

Chapitre 12

Modélisation de tableaux en B

Un tableau est une structure de données qui associe à un ensemble d'index, des données ; un tableau peut donc être modélisé par une fonction (ou application) dont l'ensemble de départ est constitué des index et l'ensemble d'arrivée est constitué des données.

La fonction utilisée peut être *injective, surjective* ou *bijective* suivant les propriétés du tableau à modéliser.

Le langage B0 permet aujourd'hui l'implantation des tableaux de taille fixe sans passer par l'importation des machines de bases comme `BASIC_ARRAY_VAR` *(machine de base permettant d'implanter un tableau à une dimension)*, `BASIC_ARRAY_RGE` *(machine de base permettant d'implanter un tableau à deux dimensions)* ou `L_SEQUENCE` *(machine de librairie permettant d'implanter un tableau ordonné d'éléments ; cette structure est appelée suite)* .

L'utilisation de ces machines de base est désormais limitée au seul cas de l'implantation de tableaux de taille variable, illustrée au paragraphe 12.4.

12.1 Définition d'un tableau en B0

Dans une spécification, un tableau est modélisé par une fonction *partielle* ou *totale*[1].

Dans une implantation, un tableau est une fonction totale B partant d'un produit cartésien d'une liste non vide d'ensembles index vers un ensemble simple.

Les différents ensembles index utilisables sont les intervalles d'entiers, les booléens, les ensembles énumérés et les ensembles abstraits.

12.2 Initialisation

En B0, l'implantation d'un tableau se fait au moment de son initialisation. Un tableau peut être valué par un autre tableau, par une énumération de maplet ou par l'initialisation de ses différentes plages.

[1] Le domaine d'une *fonction partielle* est inclus dans son ensemble de départ ; le domaine d'une *fonction totale* est exactement son ensemble de départ.

IMPLEMENTATION
Tableaux_1
REFINES
Tableaux
CONCRETE_VARIABLES
T1, T2, T3
INVARIANT
T1 ∈ -5..5 → INT ∧
T2 ∈ -5..5 → INT ∧
T3 ∈ (1..2) ∗ (1..2) → BOOL
INITIALISATION
T1 := (-5..0)*{0} ∪ (1..5)*{1} ;
T2 := T1 ;
T3 := { 1 ↦ 1↦ TRUE, 1↦ 2↦ FALSE, 2↦ 1↦ FALSE, 2↦ 2 ↦ TRUE }
...
END

12.3 Accès

L'accès en lecture ou en écriture d'un tableau se fait directement en indiquant le nom du tableau et le ou les index, comme dans les exemples suivants :

```
xx := T3(1, yy) ;
T2(zz) := 4 ;
```

12.4 Tableaux de taille variable

L'implantation en B0 des tableaux présentée dans les paragraphes précédents n'est valable que pour implanter des tableaux dont les bornes sont connues au sein d'un composant. Autrement dit, un tableau de taille variable (au moins une borne dépend d'un paramètre de la machine) ne peut être implanté de la sorte.

Par exemple, un composant B `FIFO` qui réalise une pile de type "first-in, first-out" dont la taille maximale serait un paramètre *(appelé* `taille` *dans la suite de ce paragraphe)* ne peut pas implanter directement cette pile par un tableau B0 car la taille du tableau dépend de `taille` . Deux solutions sont alors possibles :

– Une solution en B0 natif, où l'on impose au paramètre `taille` de ne pas dépasser une valeur prédéfinie `max_taille` connue "statiquement" dans le composant `FIFO`. On implante alors la pile dans un tableau "plus grand", de taille `max_taille`, et on utilise un invariant de liaison pour exprimer que la restriction du tableau à ses `taille` premiers éléments implante la pile décrite dans la spécification du composant.
La spécification de ce composant[2] est :

[2]Pour les besoins de l'exercice, la pile est modélisée par une fonction totale ; cependant, comme les données sont ordonnées, il serait souhaitable d'utiliser une suite. L'invariant serait alors :
pile ∈ seq(DATAS) ∧ size(pile) ≤ taille

MACHINE
FIFO(taille)
DEFINITIONS
/* taille maximum de la pile */
max_taille == 20
CONSTRAINTS
/* prédicat de typage */
taille ∈ NAT ∧
/* contrainte */
taille ≤ max_taille
SETS
DATAS
ABSTRACT_VARIABLES
pile
INVARIANT
pile ∈ 1..taille → DATAS
...
END

max_taille ne peut pas être une constante ; les paramètres d'une machine abstraite ne peuvent être définis à partir d'autres constituants. Pour plus d'informations, le lecteur se référera au *Manuel de Référence du langage B*.
L'implantation de ce composant est :

IMPLEMENTATION
FIFO_1
REFINES
FIFO
DEFINITIONS
max_taille == 20
VALUES
DATAS = 1..10
CONCRETE_VARIABLES
pile_B0
INVARIANT
pile_B0 ∈ 1..max_taille → DATAS ∧
/* invariant de liaison */
(1..taille) ◁ pile_B0 = pile
...
END

– La seconde solution utilise une machine de base, ici `BASIC_ARRAY_VAR`. On implante alors la pile par collage explicite avec une instance importée de cette machine de base. Bien sûr, l'importation se fait en valuant le paramètre `INDEX` de cette machine avec la valeur `1..taille`.
La spécification de ce composant est :

```
MACHINE
    FIFO(taille)
CONSTRAINTS
    /* prédicat de typage */
    taille ∈ NAT1
SETS
    DATAS
ABSTRACT_VARIABLES
    pile
INVARIANT
    pile ∈ 1..taille → DATAS
...
END
```

Ici le paramètre *taille* ne peut être nul ; en effet, l'instanciation de la machine de base $BASIC\_ARRAY\_VAR$ nécessite que l'ensemble 1..*taille* ne soit pas vide.
L'implantation du composant `FIFO` est alors :

```
IMPLEMENTATION
    FIFO_1
IMPORTS
    BASIC_ARRAY_VAR(1..taille, DATAS)
INVARIANT
    /* invariant de liaison */
    arr_vrb = pile
...
END
```

La variable `pile` définie en spécification est réalisée par la variable `arr_vrb` de l'instance de la machine de base. L'invariant de liaison est nécessaire pour créer le lien entre la variable de la machine abstraite et celle de la machine de base.

12.5 Conclusion

Actuellement, l'utilisation des fonctionnalités du B0 s'impose et il n'est plus nécessaire d'utiliser des machines de base, sauf dans le cas du problème de la taille variable. De telles machines dimensionnées extérieurement sont assez fréquentes, et il est difficile d'admettre qu'il faut leur donner une taille maximum fixe surtout que la place mémoire utilisée est alors toujours maximale. L'utilisation des machines de base résout le problème, mais la place des tableaux ainsi réalisés est allouée dynamiquement dans la phase d'initialisation du logiciel. L'allocation dynamique n'est pas toujours bien vue dans les logiciels sécuritaires, même si elle n'est faite qu'au lancement ...En fait il serait toujours possible de calculer la taille de tous les tableaux lorsque le projet est complet, en phase de compilation par exemple. Il n'y aurait alors aucune allocation dynamique à faire : gageons que les prochains traducteurs posséderont ces fonctionnalités.

12.6 Ce que nous avons appris

– Les tableaux informatiques sont modélisés mathématiquement par des fonctions totales.
– Les fonctions du B0 et des traducteurs associés permettent l'implantation directe de tableaux : en tant que variables concrètes ou paramètres d'opération. Les ensembles de départ et d'arrivée doivent alors être implantables, bien sûr.
– Les traducteurs ne peuvent pas implanter automatiquement des tableaux dont la taille n'est pas indiquée directement dans sa définition. Pour ce cas on peut employer les machines de librairie `BASIC_ARRAY_VAR` ou `BASIC_ARRAY_RGE`.

Chapitre 13

Implantation de variables ensemblistes

Une variable de type ensemble doit être raffinée en données scalaires ou tableaux. Nous allons, dans ce chapitre, décrire comment trouver une implantation des variables de type ensemble directement.

13.1 Introduction

Pour réaliser une variable ensembliste dans une implantation, un raffinement de données est nécessaire. Une solution est de remplacer la variable ensembliste par une variable de type tableau qui associe aux éléments de l'ensemble un booléen (précisant si l'élément fait partie de l'ensemble ou non).

Si un élément est associé par la variable de type tableau au booléen `TRUE`, alors il appartient à la variable ensembliste de spécification.
Si un élément est associé par la variable de type tableau au booléen `FALSE`, alors il n'appartient pas à la variable ensembliste de spécification.

L'implantation de variables ensemblistes par des variables de type tableau nécessite les recommandations suivantes :
- les variables de type tableau sont des *fonctions totales*.
  Seules les fonctions totales sont acceptées pour définir des variables concrètes de type tableau.
- *l'invariant de collage* est obligatoire ; il fait le lien entre la variable abstraite ensembliste et la variable concrète tableau.
  Cet invariant sert à vérifier la correction de développement du module B : l'implantation raffine sa spécification sans la contredire.

L'exemple ci-dessous montre l'implantation de différentes variables ensemblistes.

```
MACHINE
    Ensembles
SETS
    ENS1 = {elem1, elem2, elem3, elem4} ;
    ENS2
ABSTRACT_VARIABLES
    ens1, ens2
INVARIANT
    ens1 ⊆ ENS1 ∧
    ens2 ⊆ ENS2
INITIALISATION
    ens1 := {} ||
    ens2 := ENS2
OPERATIONS
    ...
END
```

```
IMPLEMENTATION
    Ensembles_imp
REFINES
    Ensembles
VALUES
    ENS2 = 10..20
CONCRETE_VARIABLES
    tab1, tab2
INVARIANT
    tab1 ∈ ENS1 → BOOL ∧
    tab2 ∈ ENS2 → BOOL ∧
    /* Prédicats de collage */
    tab1^-1[{TRUE}] = ens1 ∧
    tab2^-1[{TRUE}] = ens2
INITIALISATION
    tab1 := ENS1*{FALSE} ;
    tab2 := ENS2*{TRUE}
OPERATIONS
    ...
END
```

13.2 Initialisation

Un tableau est initialisé à une seule valeur pour tout son domaine. Si l'initialisation de spécification de l'exemple précédent avait été :

```
INITIALISATION
    ens1 := {elem1, elem3} ||
    ens2 := ENS2
```

L'initialisation de l'implantation serait :

INITIALISATION
tab1 := ENS1*{FALSE} ;
tab1(elem1) := TRUE ;
tab1(elem3) := TRUE ;
tab2 := ENS2*{TRUE}

L'initialisation des tableaux se fait, dans un premier temps, sur l'ensemble du domaine de définition ; les affectations suivantes permettent de particulariser certains éléments du tableau.

13.3 Ajout ou retrait d'un élément

Ajouter un élément à une variable ensembliste se traduit en spécification par la substitution généralisée :

$$\text{ens1} := \text{ens1} \cup \{\text{elem1}\}$$

Dans l'implantation, l'ajout d'un élément se traduit par l'instruction :

$$\text{tab1(elem1)} := \text{TRUE}$$

Retirer un élément à une variable ensembliste se taduit en spécification par la substitution généralisée :

ens1 := ens1 - {elem1} /* différence ensembliste */

Dans l'implantation, le retrait d'un élément d'un ensemble est traduit par l'instruction :

$$\text{tab1(elem1)} := \text{FALSE}$$

13.4 Union, intersection ou différence ensembliste

Ces opérations ensemblistes sont réalisées grâce à des boucles ; celles-ci sont présentées ci-après. Le lecteur dispose de ce fait de la méthode d'implantation de ces opérations et la partie `INVARIANT/VARIANT` de la boucle.

Dans les exemples suivants, `ens1, ens2, ens3` sont trois variables ensemblistes inclues dans l'ensemble abstrait `ENS`. La valuation de ce dernier n'est pas connue.
Les trois variables ensemblistes sont réalisées par trois variables de type tableau `tab1, tab2, tab3` (qui associent aux éléments de `ENS` un booléen).
Les invariants de collage entre variables concrètes et variables de spécification sont du type :

$$tab_i^{-1}[\{\text{TRUE}\}] = ens_i$$
$$\text{avec } tab_i \in \text{ENS} \rightarrow \text{BOOL et } ens_i \subseteq \text{ENS}$$

Union ensembliste L'union ensembliste se formalise facilement en spécification.

```
union =
BEGIN
    ens3 := ens1 ∪ ens2
END
```

La réalisation de l'union ensembliste dans une implantation fait intervenir une boucle.

```
union =
VAR indice IN
    cpt := MAX_ENS ;
        WHILE (cpt ≥ MIN_ENS) DO
            tab3(cpt) := bool( tab1(cpt)= TRUE or tab2(cpt) = TRUE);
            cpt := cpt - 1
        INVARIANT
            tab1 ∈ ENS → BOOL ∧
            tab2 ∈ ENS → BOOL ∧
            tab3 ∈ ENS → BOOL ∧
            cpt : ENS ∪ {MIN_ENS - 1} ∧
            ((cpt+1)..MAX_ENS) ∩ tab3^-1[{TRUE}] =
                ((cpt+1)..MAX_ENS) ∩ (ens1 ∪ ens2)
        VARIANT
            cpt
        END
END
```

Intersection La réalisation de cette opération suit le schéma proposé précédemment. L'intersection se formalise facilement en spécification.

```
intersection =
BEGIN
    ens3 := ens1 ∩ ens2
END
```

La réalisation de l'intersection dans une implantation fait intervenir une boucle.

```
intersection =
VAR indice IN
    cpt := MAX_ENS ;
        WHILE (cpt ≥ MIN_ENS) DO
            tab3(cpt) := bool( tab1(cpt)= TRUE ∧ tab2(cpt) = TRUE);
            cpt := cpt - 1
        INVARIANT
            tab1 ∈ ENS → BOOL ∧
            tab2 ∈ ENS → BOOL ∧
            tab3 ∈ ENS → BOOL ∧
            cpt : ENS ∪ {MIN_ENS - 1} ∧
            ((cpt+1)..MAX_ENS) ∩ tab3^-1[{TRUE}] =
                ((cpt+1)..MAX_ENS) ∩ (ens1 ∩ ens2)
        VARIANT
            cpt
        END
END
```

Différence La réalisation de cette opération suit le schéma proposé précédemment. La différence ensembliste se formalise facilement en spécification.

```
difference =
BEGIN
    ens3 := ens1 − ens2
END
```

La réalisation de la différence ensembliste dans une implantation fait intervenir une boucle.

```
difference =
VAR indice IN
    cpt := MAX_ENS ;
        WHILE (cpt ≥ MIN_ENS) DO
            tab3(cpt) := bool( tab1(cpt)= TRUE ∧ tab2(cpt) = FALSE);
            cpt := cpt - 1
        INVARIANT
            tab1 ∈ ENS → BOOL ∧
            tab2 ∈ ENS → BOOL ∧
            tab3 ∈ ENS → BOOL ∧
            cpt : ENS ∪ {MIN_ENS - 1} ∧
            ((cpt+1)..MAX_ENS) ∩ tab3^-1[{TRUE}] =
            ((cpt+1)..MAX_ENS) ∩ (ens1 − ens2)
        VARIANT
            cpt
        END
END
```

13.5 Ce que nous avons appris

- Une variable ensembliste s'implante par une fonction totale de l'ensemble complet dans BOOL, la valeur à chaque position indiquant la présence ou non de l'élément.
- L'invariant de collage est de la forme $tab^{-1}[\{TRUE\}] = ens$ où ens est l'ensemble abstrait et tab est la fonction totale implantée.
- Les différences, union et intersection d'ensembles sont réalisables par des boucles (voir exemples).
- Les fonctionnalités du B0 permettent d'éviter le codage de boucles pour vider l'ensemble ou le remplir complètement.

Chapitre 14

Glossaire

Cahier des Charges : document dans lequel est exprimé le besoin initial que le système à construire doit satisfaire. Normalement rédigé par l'organisme qui utilisera ce système.

Composant B : unité de découpage d'un projet B. Les composants peuvent être des machines abstraites, des raffinements ou des implantations. Un projet B est constitué (pour sa partie purement formelle) de composants B liés entre eux.

Développement B : idem projet B.

Implantation : composant B contenant une partie du programme obtenu dans le projet B.

Machine abstraite : composant B contenant la spécification d'une partie d'un projet B.

Modèle B : un modèle B est une description mathématique qui représente certaines entités réelles du système à construire ou de son contexte. Dans un projet B, il y a souvent plusieurs modélisations mathématiques, soit pour représenter divers aspects du système, soit pour représenter un aspect à différents niveaux de précision. Les modèles B sont écrits dans des composants B (machines abstraites, raffinements), mais ils constituent des entités indépendantes. Il est d'ailleurs fréquent d'écrire des modèles B sous forme d'un ensemble de prédicats mathématiques avant de se préoccuper de leur répartition dans les machines abstraites. Les phases de réexpression du besoin et de modélisation dont nous parlons plus haut ont pour but la production de ces modèles.

Modèle abstrait : dans notre cadre, idem modèle B.

Modélisation B : activité de création de modèles B.

Modélisation des propriétés du système : idem modélisation B, avec en plus une évocation de l'approche "par propriété" préconisée dans la méthode B pour l'analyse du système.

Modélisation mathématique : dans le cadre d'un projet B, idem modélisation B. Cette terminologie insiste sur le fait que B est basé sur la théorie des ensembles, classique en mathématiques.

Obligations de preuve : prédicats mathématiques qu'il faut démontrer pour prouver qu'un composant B est correct. La théorie de B définit les *obligations de preuves brutes* qui sont construites à partir des composant. L'outil de génération les découpe ensuite en prédicats de taille lisible pour la preuve.

PO : obligation de preuve.

Programmation défensive : style de programmation qui consiste à tester les paramètres d'entrée des fonction internes, les indices de tableaux, les résultats intermédiaires de calcul.

Programmation offensive : style de programmation qui consiste à éliminer tous les tests sur des quantités internes au programme pour lesquelles une valeur erronée ne peut provenir que d'une erreur dans ce programme. Ne doit être utilisé que pour un logiciel prouvé.

Programme B : programme écrit en langage B. Ce programme est l'aboutissement d'un projet B comprenant des phases importantes d'études amont (réexpression du besoin), les phases de modélisation et les phases de réalisation accompagnées de preuves.

Projet B : un projet B est l'ensemble des activités utilisant B qui partant d'un besoin, aboutit au système qui satisfait ce besoin. Cette terminologie peut être employée pour des systèmes entièrement développés en B aussi bien que pour ceux pour lesquels certaines parties ont été faites par des méthodes traditionnelles.

Raffinement : composant B contenant une réexpression plus concrète d'une partie d'un projet B.

Réaliser : dans le contexte de B, ce terme signifie écrire un élément de programme qui corresponde à certaines entités provenant de machines abstraites.

Réexpression du besoin : travail souvent nécessaire avant de formaliser les spécifications d'un projet, qui consiste à réordonner et reformuler *en langage naturel*, mais plus rigoureusement, les spécifications initiales.

Spécification B : la spécification B est l'ensemble des composants B et de leurs liaisons qui constituent l'équivalent formel du besoin exprimé initialement. Tout ce qui est de la conception ne fait pas partie de la spécification B.

Spécification formelle : la spécification formelle d'un système est l'ensemble des textes en langage formel qui constituent l'équivalent du besoin exprimé initialement. Tout ce qui est de la conception ne fait pas partie de la spécification formelle.

STBL : spécification technique des besoins logiciels. Correspond au cahier des charges dans le cadre d'un système informatique.