Test de programme à partir de spécifications B: les problèmes

Stéphane Alnet
Laboratoire de Recherche en Informatique,
Université Paris Sud, Centre d'Orsay,
91405 Orsay cedex, France.
Université de Picardie,
alnet@u-picardie.fr

10 septembre 1996

Abstract: We study software "black box" testing for programs designed with the B method. We provide semantics for such programs, observationnal semantics based on execution traces.

Keywords: software testing, B method, formal specifications, test oracle, program correctness.

Résumé : Nous étudions le test boîte-noire de programmes écrits à partir de la méthode B. Nous proposons de plus une sémantique pour de tels programmes, sémantique observationnelle basée sur des traces.

Mots clés : test de logiciel, méthode B, spécifications formelles, oracle de test, correction de programme.

Ce travail a pour objet une étude préliminaire du test de programmes écrits à l'aide de la méthode B. Cette méthode est décrite par Jean-Raymond Abrial dans le *B-Book* [1]. La méthode B est utilisée en particulier pour le développement de logiciels sécuritaire dans les transports [8, 4, 11], ... Outre la base théorique que constitue le *B-Book*, il existe des ensembles d'outils (un tel ensemble est appelé un "atelier"), destinés à aider le développeur dans sa démarche d'utilisation de la méthode. "L'atelier B" développé par Digilog/Stéria [9] en est un exemple français.

Il existe des différences entre le *B-Book* et les outils de l'Atelier. Dans le cadre du test, nous considérerons qu'*en cas de désaccord, c'est l'Atelier qui a raison.* Ce choix s'explique simplement: les programmes que nous serons amenés à tester auront été le plus souvent développés avec un tel atelier.

Notons également que le but cherché est de réaliser un test, c'est-à-dire *de détecter les discordances entre une spécification, donnée, et un programme censé implémenter cette spécification.* Nous montrerons quels sont les problèmes qui se posent au cours de cette démarche, en nous appuyant sur la méthode B, mais il est important de rappeler dès maintenant, comme dans [6], que notre but est de mettre à jour ces discordances.

1 Introduction

La méthode B est une méthode de spécification formelle de développement dont chaque étape du raffinement fait l'objet de *preuves.* Ces preuves assurent en particulier que la sémantique des opérations après raffinement est incluse dans celle des opérations avant raffinement. Le programme résultant est généré de façon automatique à partir de la toute dernière étape de raffinement (voir section 2.3).

On peut donc se demander quelle est l'utilité du test dans ce cadre.

Dans un article fondateur de la théorie du test [7], Gourlay précisait déjà l'importance de l'utilisation des spécification formelles pour la construction et la réalisation de tests de programmes, affirmant qu'un programme est correct si et seulement si il vérifie sa spécification. Il montrait aussi que le test à partir de spécification formelles est plus puissant que d'autres méthodes classiques; il montre en particulier que les méthodes "white box" (basées sur le programme) posent des conditions impraticables sur le jeu de test.

Symétriquement, dans une étude générale sur les méthodes de spécification, incluant B [4, vol. 1, page 25], Susan Gerhart, Dan Craigen et Ted Ralston rappellent l'importance du test dans le cadre des méthodes de spécification formelles. D'ailleurs, dans la pratique, l'utilisation de B est couplée avec des méthodes de test classique du domaine concerné [8, pp. 188 et seq.], mais la notion de test à partir de la spécification est inexistante.

Nous nous proposons donc ici d'étudier le test « boîte-noire » de programmes écrits à partir de la méthode B. Notre approche se distingue des

tests habituels en B par le fait que nous nous testons directement à partir de la spécification (approche boîte-noire) sans nous préoccuper du domaine d'application.

Elle se distingue aussi d'une approche telle que celle d'Hélène Waeselynck et Jean-Louis Boulanger [11]. Dans cet article, les auteurs montrent la nécessité du test dans le cadre de la méthode B contre les tenants d'un relâchement de ces pratiques; ils rappellent que la non-automatisation de la vérification des obligations de preuves, et le fait qu'en pratique toutes les démonstrations nécessaires ne soient pas systématiquement effectuées plaident en faveur du test; ils insistent en particulier sur la nécessité de tester les obligations de preuves (voir section 2.2) qui n'auraient pas pu être prouvées au cours du développement. Nous adoptons ici une approche plus systématique du test; nous expliquons pourquoi le test ne peut pas être basé sur le raffinement. Par contre, la connaissance des obligations de preuves non démontrées pourra être utile dans la phase de sélection du jeu de test.

Enfin, nous verrons que la méthode B ne fournit pas de sémantique pour les programmes qu'elle génère. Nous proposerons une sémantique observationnelle, basée sur des traces.

2 Machines abstraites et substitutions généralisées

Toute spécification B est décrite par un ensemble de *machines abstraites* [1, Chap. 4, page 2].

Chaque machine comporte une partie *statique* (qui caractérise l'état de la machine) et une partie *dynamique* (les opérations sur la machine).

Nous décrivons succintement ici la syntaxe d'une telle machine, les obligations de preuve dont elle fait l'objet, les possibilités de réutilisation proposées par la méthode, ainsi que le domaine sur lequel la sémantique d'une machine est définie. Enfin, nous précisons les définitions des opérations qui en constituent la partie dynamique.

2.1 Syntaxe d'une machine abstraite

Schématiquement, une machine a la forme suivante, décrite dans [1, Chap. 5, page 12][1].

[1] Un exemple détaillé est décrit à la fin de cette section.

MACHINE	$M(X, x)$
CONSTRAINTS	C
SETS	$S; T = \{a, b\}$
CONSTANTS	c
PROPERTIES	P
VARIABLES	v
INVARIANT	I
INITIALIZATION	U
OPERATIONS	$u \leftarrow \mathbf{O}(w) \hat{=}$
	PRE Q THEN V END ;
END	

La machine M peut posséder en paramètres formels des ensembles (désignés ici par la lettre X) ou des valeurs (désignées ici par la lettre x). Les *paramètres formels* d'une machine B permettent d'introduire une forme de généricité dans l'écriture des machines abstraites.

Les *contraintes* C portent sur ces paramètres formels et contrôlent le degré de généricité introduit. La vérification de ces contraintes fait partie des *obligations de preuve* de B (voir section 2.2).

Les *ensembles* définis dans la machine peuvent l'être de façon intentionnelle (cas des ensembles dénotés ici par S), ou extensionnelle (ainsi, l'ensemble T défini en extension comporte les éléments a et b). Les ensembles définis de façon intentionnelle sont appelés « ensembles différés » (*deffered sets*) dans la terminologie B.

Les *constantes* c sont définies pour la machine M. Ensembles et constantes doivent vérifier les *propriétés* P. La vérification de P fait partie des obligations de preuve de B.

Les *variables* v sont définies pour cette machine. On peut dire de façon intuitive, d'une part qu'une affectation de ces variables caractérise l'état de la machine; d'autre part, que les variables doivent vérifier l'*invariant* I tout au long de l'exécution de la machine. La préservation de l'invariant est l'un des aspects critiques des obligations de preuves [1, Chap. 5, page 13] [11, page 66].

Exemple Cette machine, définie dans [1, Chap. 8, page 4], contient en particulier un ensemble différé ($CLIENT$). Les éléments de $CLIENT$ ne sont pas connus à cette étape du raffinement; ils seront précisés dans un raffinement ultérieur.

MACHINE	$Client(max\_allowance, max\_client\_number)$
SETS	$CLIENT$;
	$CATEGORY = \{friend, dubious, normal\}$
CONSTANTS	$discount, client\_number\_to\_client$
PROPERTIES	$discount \in CATEGORY \rightarrow (0..100) \wedge$
	$discount =$
	$\{friend \mapsto 80, dubious \mapsto 100$
	$normal \mapsto 100\} \wedge$
	$client\_number\_to\_client \in$
	$(0..max\_client\_number) \rightarrowtail\!\!\!\!\!\rightarrow CLIENT$
VARIABLES	$client, category, allowance$
INVARIANT	$client \subset CLIENT \wedge$
	$category \in client \rightarrow CATEGORY \wedge$
	$allowance \in client \rightarrow (0..max\_allowance)$
DEFINITIONS	$client\_not\_dubious(c) \mathrel{\hat{=}} category(c) \neq dubious$
...	
END	

□

Les *opérations* correspondent à des changements d'états de la machine. Les opérations sont *toutes* définies en B comme des *substitutions généralisées*, et peuvent être ramenées à la forme présentée ici, où l'opération O a pour paramètre d'entrée formel w, pour paramètre de retour formel u, et où la partie exécutable V est préconditionnée par le prédicat Q. L'initialisation U de la machine est elle-même une substitution généralisée [1, Chap. 4, page 13]; elle peut être non-déterministe.

Une substitution généralisée est essentiellement un *transformateur de prédicat*. On notera $[V]P$ le transformé du prédicat P après application de la substitution généralisée V. Si $[V]P$ est vrai, on peut dire que la substitution V établit la postcondition P. Les différentes substitutions exprimables en B, ainsi que leurs effets sur les post-conditions, sont décrits dans la section 2.5.

B ne possède pas de types au sens des langages de programmation. La vérification de cohérence sur les arguments passés à une opération fait en général partie de cette opération[2] mais n'est pas explicitée dans le profil de l'opération.

En général, les propriétés P, l'invariant I et les préconditions Q contiennent au moins une information permettant de typer respectivement les

[2]Cette vérification est alors exprimée, par le biais des obligations de preuves, dans les préconditions de l'opération.

constantes, les variables de la machine, et les arguments des opérations. Ces règles sont données comme impératives dans [9, pages 32–33], bien que rien dans le *B-Book* ne l'impose.

Exemple Nous considérerons dans cet exposé la machine abstraite suivante [1, Chap. 11, page 25], qui permet d'entrer des entiers et d'obtenir le maximum de la suite d'entiers qui lui a été fournie jusqu'alors:

MACHINE	$LittleExample\_1$
VARIABLES	y
INVARIANT	$y \in F(\mathbf{nat}_1)$
INITIALISATION	$y := \emptyset$
OPERATIONS	$\mathbf{enter}(n) \mathrel{\hat{=}}$
	PRE $n \in \mathbf{nat}_1$ THEN $y := y \cup \{n\}$;
	$m \longleftarrow \mathbf{maximum} \mathrel{\hat{=}}$
	PRE $y \neq \emptyset$ THEN $m := \mathbf{max}(y)$;
END	

□

2.2 Obligations de preuve

Des obligations de preuve sont associées à la spécification d'une machine abstraite. Elles assurent la cohérence de la machine, c'est-à-dire la préservation de l'invariant par l'initialisation et les différentes opérations.

Leur vérification, qui est indispensable à l'application de la méthode, est assistée par les outils de l'Atelier.

Les obligations de preuve que nous allons considérer ici sont celles d'une machine non raffinée (données dans [1, Chap. 5, page 13])[3].

- La première partie de l'obligation de preuve assure que l'invariant est valide après l'initialisation de la machine.

  On utilise les notations suivantes:

  - Un prédicat A qui assure que les *ensembles paramètres* d'une machine abstraite doivent être finis et non vides, et qui s'écrit

$$\mathbf{finite}(X) \wedge X \neq \emptyset$$

[3]Les obligations de preuves du raffinement sont différentes, mais ne nous servirons pas (voir section 3.2). Pour alléger l'exposé, elles ne sont donc pas données ici.

– Un prédicat B qui assure que les *ensembles définis* dans la machine abstraite, que ce soit en intention ou en extension, sont isomorphes à des sous-ensembles non vides des entiers naturels[4]:

$$S \in F_1(\mathbf{int}) \wedge T \in F_1(\mathbf{int}) \wedge T = \{a, b\} \wedge a \neq b$$

On notera que la sémantique des ensembles définis en extension est celle attendue.

– C est la partie CONSTRAINTS d'une machine abstraite;

– P est la partie PROPERTIES d'une machine abstraite.

La première partie de l'obligation de preuve s'écrit alors

$$A \wedge B \wedge C \wedge P \implies [U]I$$

où $[U]I$ est le prédicat obtenu à partir de l'invariant I après application de la substitution généralisée U (*i.e.* après l'initialisation de la machine).

- La deuxième partie de l'obligation de preuve assure que l'invariant est conservé par les opérations. Elle est définie pour chaque opération comme

$$A \wedge B \wedge C \wedge P \wedge Q \wedge I \implies [V]I$$

où

– A, B, C et P sont comme précédemment,

– Q est la précondition et V la partie exécutable d'une même opération **O** (voir 2.1), $[V]I$ désignant donc le prédicat obtenu à partir de l'invariant par application de la substitution V.

2.3 Description d'un projet B

La Machine Abstraite est le composant de base pour la construction d'un projet B. Toutefois, les structures utilisées dans une machine, et en particulier le non-déterminisme, ne sont pas toutes directement implémentables. Réciproquement, certaines structures comme la mise en séquence, caractéristique des langages de programmation, posent des problèmes importants pour la démonstration des obligations de preuves qui assurent la correction d'une Machine Abstraite [1, Chap. 9, page 1].

[4]Plus précisément, **int** est défini comme l'intervalle **minint...maxint** où **minint** et **maxint** sont des constantes. D'autre part, F_1 désigne les sous-ensembles finis non vides d'un ensemble donné.

2.3.1 Nécessité du raffinement

Il est donc nécessaire, en B comme dans les autres méthodes de spécification, de réaliser un raffinement de la spécification d'origine vers une structure plus proche de la machine. On distingue dans une Machine Abstraite:

- la spécification d'origine, distinguée par l'appellation MACHINE ;
- un ou plusieurs raffinements intermédiaires, distingués par l'appellation REFINEMENT ;
- une machine d'implantation, dont la forme des opérations est proche de celle des langages impératifs de haut niveau, et distinguée par l'appellation IMPLEMENTATION .

Le raffinement entraîne de nouvelles obligations de preuves afin de prouver la conservation des propriétés de la Machine Abstraite d'origine jusque dans la Machine d'Implantation. Toutefois, comme nous le verrons section 3.2, nous ne nous intéresserons pas au raffinement et aux preuves qu'il engendre; c'est pourquoi nous ne les présentons pas ici.

2.3.2 Réutilisation de composants

D'autre part, B permet la réutilisation de composants (en l'occurrence, de Machines Abstraites) au sein d'autres Machines Abstraites. Le langage B fournit différents types de visibilité à travers la clause INCLUDES et celles qui lui sont liées: PROMOTES , EXTENDS , USES [1, Chap. 7] [9, pp. 16 et seq.].

Puisque nous nous intéressons au test d'un programme, les composants (même génériques) qui auraient pu être utilisés dans la Machine Abstraite ont nécessairement été instanciés. De plus, la réutilisation s'effectue "à plat" (l'inclusion d'une machine correspond en effet à la création d'une instance abstraite de cette machine) [9, page 16]. Les obligations de preuve qui découlent de la réutilisation de composants ne poseront donc pas de problèmes spécifiques, puisqu'elles seront partie intégrante des obligations de preuves de la spécification d'origine (non raffinée).

2.4 Les objets de B

2.4.1 Les objets de base

Comme nous l'avons déjà dit, la méthode B ne possède pas de types de données au sens des langages de programmation. La sémantique de B est définie dans la logique du premier ordre, étendue aux ensembles finis.

En particulier, tous les ensembles que l'on peut être amené à définir dans une machine abstraite de B doivent être des ensembles finis, et plus précisément isomorphes à des parties finies et non-vides des entiers naturels. Il

existe toutefois une légère différence d'obligations de preuve entre une machine non-raffinée (ou un de ses raffinements) et une machine complètement raffinée.

Au niveau d'une spécification non-raffinée les ensembles définis dans la machine doivent être des sous-ensembles non vides des entiers naturels. Toutefois, pour les ensembles *paramètres* de la machine, l'obligation de preuve indique simplement qu'ils doivent être finis et non vides.

Plus précisément, au niveau d'une spécification non-raffinée, l'obligation de preuve comporte les deux parties suivantes (données en 2.2):

$$\begin{aligned} A &= \mathbf{finite}(X) \wedge X \neq \emptyset \\ B &= S \in F_1(\mathbf{int}) \wedge T \in F_1(\mathbf{int}) \wedge T = \{a, b\} \wedge a \neq b \end{aligned}$$

Au niveau d'une spécification complètement raffinée
autrement dit au niveau d'une *machine abstraite d'implantation*, les obligations de preuve sont légèrement différentes:

$$\begin{aligned} A &= X \in F_1(\mathbf{int}) \\ B &= S \in F_1(\mathbf{int}) \wedge T \in F_1(\mathbf{int}) \wedge T = \{a, b\} \wedge a \neq b \end{aligned}$$

Au niveau d'une implantation B, *tous* les ensembles considérés, y compris les ensembles paramètres de la machine, doivent donc être des sous-ensembles non-vides de **int**.

De plus, l'égalité est définie pour les booléens et les ensembles, et donc, par construction, pour tous les types du langage B.

2.4.2 Les objets mathématiques

Abrial décrit dans [1, Chap. 3] un ensemble de constructions mathématiques qui peuvent être utilisées comme des objets de première classe dans une machine abstraite. En dehors de la notion de sous-ensemble, la plupart de ces objets sont construits à l'aide de l'opérateur point-fixe: nombres naturels, ainsi que l'arithmétique entière et la notion de cardinal d'un ensemble; séquences finies; arbres finis, arbres étiquettés finis et arbres binaires.

2.4.3 La machine prédéfinie

La machine suivante est vue implicitement par toute autre machine B [1, Chap. 5, page 15]:

MACHINE *pre_defined*
SETS $\mathbf{BOOL} = \{\mathbf{true}, \mathbf{false}\}$
CONSTANTS $\mathbf{minint}, \mathbf{maxint}, \mathbf{int}, \mathbf{nat}, \mathbf{nat}_1, \mathbf{char}, \mathbf{string}$
PROPERTIES $\mathbf{minint} \in \mathbf{Z} \wedge$
$\mathbf{maxint} \in \mathbf{N} \wedge$
$\mathbf{minint} < 0 \wedge$
$\mathbf{maxint} > 0 \wedge$
$\mathbf{int} = \mathbf{minint}..\mathbf{maxint} \wedge$
$\mathbf{nat} = 0..\mathbf{maxint} \wedge$
$\mathbf{nat}_1 = 1..\mathbf{maxint} \wedge$
$\mathbf{char} = 0..255 \wedge$
$\mathbf{string} = \mathbf{seq}(\mathbf{char})$
END

Dans le cadre de l'Atelier B [9, page 69], les *paramètres d'entrée des opérations* sont soit des variables concrètes, soit des chaînes de caractères [9, page 73]. Les variables dites "concrètes" doivent être d'un des types suivants:

- un ensemble simple,
- une fonction totale, une bijection, une injection totale, ou une surjection totale d'un produit cartésien d'ensembles simples vers un ensemble simple[5],
- ou un ensemble énuméré.

Les ensembles simples étant les ensembles **BOOL**, **int**, **nat**, $\mathbf{nat}_1$ et **char**, définis dans la machine *pre_defined*.

Par conséquent, et bien que le *B-Book* autorise n'importe quel type d'arguments pour les opérations de n'importe quelle machine abstraite, les types des paramètres des opérations des machines abstraites *raffinables* seront donc *en pratique* ceux que nous venons de décrire[6].

2.4.4 Les machines de librairies

A côté des objets propres au langage B lui-même, que nous venons de décrire, il existe un certain nombre de machines abstraites (appelées "machines de base" dans [9, page 4]), dont le but premier est de proposer des implantations pour les objets du langage.

[5] On notera que ces types sont tous implémentables par des tableaux finis (à indices entiers) contenant des entiers.

[6] Les arguments des opérations d'un "module abstrait" peuvent être d'un type quelconque [9, page 32]. Mais un tel module est, par définition [9, page 4], une machine abstraite qui n'est pas destinée à être implantée, et nous n'aurons donc pas à la tester (voir section 3.3.1).

Ces machines abstraites forment donc des librairies, qui font partie de l'Atelier, et qui sont réutilisables dans un projet de l'utilisateur à l'aide des structures du langage B.

2.4.5 Le cas des *deffered sets*

Les *ensembles différés* (voir section 2.1) n'étant définis qu'au cours du raffinement (éventuellement lors de la dernière étape du raffinement), nous avons besoin d'un moyen pour en désigner les éléments.

Nous savons déjà que les ensembles considérés seront nécessairement finis, et même plus précisément, qu'ils seront isomorphes à des parties finies et non-vides d'un intervalle fini et non vide des entiers naturels, noté **minint . . maxint**.

Rien ne nous empêche donc de considérer, pour le test, des éléments symboliques indexés par des entiers naturels (écrits par exemple $S.1$, $S.2$, ..., si S est le nom de l'ensemble différé considéré).

Exemple Les éléments de l'ensemble différé $CLIENT$ (défini dans la machine *Client* de la section 2.1) seront désignés par $CLIENT.1$, $CLIENT.2$, etc. □

2.5 Substitution généralisée

Les opérations d'une machine abstraite sont définies comme des substitutions généralisées.

Toute substitution généralisée est écrite par composition des opérations élémentaires que nous allons présenter.

Pour la plupart de ces opérations, Abrial fournit une sémantique sous forme de vérification de postcondition [1, Chap. 4].

Il leur correspond une sémantique sous forme de « transformateurs de prédicats » (*à la* Hoare) que nous présentons également [1, Chap. 6, page 10 et seq.]. Cette sémantique s'appuie sur les prédicats **trm** et $\mathbf{prd}_x$, **trm** traduisant la terminaison d'une substitution généralisée

$$\mathbf{trm}(S) \mathrel{\hat{=}} \exists R\,[S]R$$

et $\mathbf{prd}_x$ étant le « transformateur de prédicat » qui nous intéresse,

$$\mathbf{prd}_x(S) \mathrel{\hat{=}} \neg[S](x' \neq x)$$

où x' est une nouvelle variable qui correspond à l'effet de la substitution S sur la variable x.

2.5.1 Substitution simple

C'est la substitution simple classique [1, Chap. 4, page 6].

Syntaxe

$$\boxed{x := E}$$

Intuitivement, la valeur de x est « remplacée par » la valeur de E.

Ecriture postconditionnelle

$$[x := E]R \iff [x \mapsto E]R$$

où $\mapsto$ est la substitution sur les prédicats (R étant la postcondition à vérifier): $[x \mapsto E]R$ est le prédicat construit sur R dans lequel toutes les occurrences libres de la variable x ont été remplacées par l'expression E.

Transformateur de prédicat La substitution simple termine toujours:

$$\mathbf{trm}(x := E)$$

Elle vérifie

$$\begin{aligned} \mathbf{prd}_x(x := E) &\iff x' = E \\ \mathbf{prd}_{x,y}(x := E) &\iff x', y' = E, y \end{aligned}$$

Autrement dit, la valeur de la variable x est modifiée après la réalisation de la substitution, tandis que les valeurs des autres variables sont inchangées.

2.5.2 Substitution vide

C'est l'« opération qui ne fait rien » [1, Chap. 4, page 25].

Syntaxe

$$\boxed{\mathbf{skip}}$$

Ecriture postconditionnelle

$$[\mathbf{skip}]R \iff R$$

Transformateur de prédicat La substitution vide termine toujours:

$$\mathbf{trm}(\mathbf{skip})$$

Elle vérifie

$$\mathbf{prd}_x(\mathbf{skip}) \iff x' = x$$

Autrement dit, quelle que soit la variable x que l'on considère, sa valeur n'est pas modifiée par la réalisation de la substitution.

2.5.3 Substitution préconditionnée

Elle est présentée dans [1, Chap. 4, page 10]. Intuitivement, la substitution S est réalisée si la précondition P est vérifiée. Si la précondition n'est pas vérifiée, l'état de la machine est indéterminé.

Syntaxe

$$\boxed{P \mid S}$$

La barre verticale correspond à l'opérateur préconditionnel.

Ecriture postconditionnelle

$$[P \mid S]R \iff P \wedge [S]R$$

Transformateur de prédicat

$$\begin{aligned} \mathbf{trm}(P \mid S) &\iff P \wedge \mathbf{trm}(S) \\ \mathbf{prd}_x(P \mid S) &\iff P \implies \mathbf{prd}_x(S) \end{aligned}$$

Autre forme syntaxique La forme syntaxique suivante correspond à l'opération qui vient d'être définie.

$$\boxed{\text{PRE } P \text{ THEN } S \text{ END}}$$

2.5.4 Choix borné

Cette substitution est présentée dans [1, Chap. 4, page 21]. Son utilisation première est de permettre de différer les choix d'implémentation: la spécification propose plusieurs choix, qui sont cohérents entre eux (cette cohérence étant assurée par les obligations de preuve).

Syntaxe

$$\boxed{S \,[]\, T}$$

Ecriture postconditionnelle

$$[S \,[]\, T]R \iff [S]R \wedge [T]R$$

Transformateur de prédicat

$$\begin{aligned} \mathbf{trm}(S \,[]\, T) &\iff \mathbf{trm}(S) \wedge \mathbf{trm}(T) \\ \mathbf{prd}_x(S \,[]\, T) &\iff \mathbf{prd}_x(S) \wedge \mathbf{prd}_x(T) \end{aligned}$$

Autres formes syntaxiques

$$\boxed{S \,[]\, T \,[]\, \ldots \,[]\, U}$$

est noté

$$\boxed{\text{CHOICE } S \text{ OR } T \text{ OR } \ldots \text{ OR } U \text{ END}}$$

2.5.5 Substitution gardée

Elle est présentée dans [1, Chap. 4, page 24]. Elle permet de conditionner l'application d'une substitution.

Syntaxe

$$\boxed{P \Longrightarrow S}$$

où P est un prédicat et S un substitution généralisée.

Ecriture postconditionnelle

$$[P \Longrightarrow S]R \iff P \implies [S]R$$

Transformateur de prédicat

$$\begin{aligned} \mathbf{trm}(P \Longrightarrow S) &\iff P \implies \mathbf{trm}(S) \\ \mathbf{prd}_x(P \Longrightarrow S) &\iff P \wedge \mathbf{prd}_x(S) \end{aligned}$$

Autres formes syntaxiques On montre que l'écriture suivante:

$$\boxed{\text{IF } P \text{ THEN } S \text{ ELSE } T \text{ END}}$$

qui a pour définition

$$(P \Longrightarrow S) \,[]\, (\neg P \Longrightarrow T)$$

a pour sémantique

$$[\text{ IF } P \text{ THEN } S \text{ ELSE } T \text{ END }]R \iff (P \implies [S]R) \wedge (\neg P \implies [T]R)$$

De même, on montre que l'écriture

$$\boxed{\text{IF } P \text{ THEN } S \text{ END}}$$

définie par

$$\text{IF } P \text{ THEN } S \text{ ELSE } \mathbf{skip} \text{ END}$$

a pour sémantique

$$[\text{ IF } P \text{ THEN } S \text{ END }]R \iff (P \implies [S]R) \wedge (\neg P \implies R)$$

2.5.6 Choix non borné

Il est présenté dans [1, Chap. 4, page 30]. Cet opérateur introduit une forme de non-déterminisme dans une machine abstraite, en précisant les conditions que doit remplir le résultat. Du point de vue du test, cela revient à dire que la valeur cherchée n'est plus unique mais est prise parmi un ensemble de possibles.

Il est à noter que cet opérateur ne peut pas apparaître dans une machine abstraite qui décrit une implantation (autrement dit, le non-déterminisme doit disparaître au cours du raffinement).

Syntaxe

$$\boxed{@z.S}$$

Ecriture postconditionnelle

$$[@z.S]R \iff \forall z.[S]R$$

R ne devant pas contenir d'occurrence libre de z.

Transformateur de prédicat

$$\begin{aligned}
\mathbf{trm}(@z.S) &\iff \forall z.\mathbf{trm}(S) \\
\mathbf{prd}_x(@z.S) &\iff \exists z.\mathbf{prd}_x(S) \\
\mathbf{prd}_x(@y.T) &\iff \exists (y, y').\mathbf{prd}_{x,y}(T)
\end{aligned}$$

où T modifie les variables x et y, S ne modifie que la variable x.

Autres formes syntaxiques La forme syntaxique

$$\boxed{\text{VAR } z \text{ IN } S}$$

est équivalente à l'opération qui vient d'être définie.

La forme syntaxique

$$\boxed{\text{ANY } z \text{ WHERE } P \text{ THEN } S \text{ END}}$$

a pour définition [1, Chap. 4, page 29]

$$@z.(P \implies S)$$

La forme syntaxique

$$\boxed{x :\in E}$$

a pour définition [1, Chap. 4, page 30]

$$\text{ANY } z \text{ WHERE } z \in E \text{ THEN } x := z \text{ END}$$

à condition que $z \backslash E$, *i.e.* que la variable z n'apparaisse pas dans l'expression E.

Cette dernière forme a pour particularité de toujours terminer [1, Chap. 6, page 9]:

$$\mathbf{trm}(x :\in E)$$

et d'assurer que la valeur trouvée appartient à l'ensemble E

$$\mathbf{prd}_x(x :\in E) \iff x' \in E$$

2.5.7 Substitution multiple, mise en parallèle

Elle est définie dans [1, Chap. 4, page 20]. Intuitivement, les opérations S et T sont effectuées concurremment. Les variables sur lesquelles elles portent doivent être différentes (ce qui assure de façon simple la faisabilité de l'opération).

Syntaxe

$$\boxed{S \parallel T}$$

Ecriture postconditionnelle La forme générale de cet opérateur n'a pas de définition par post-condition. Toutefois, la substitution multiple, notée

$$\boxed{y := E \parallel z := F}$$

et définie [1, Chap. 4, page 20] comme

$$y, z := E, F$$

vérifie la propriété

$$[y, z := E, F]P \iff [w := F][y := E][z := w]P$$

(La substitution multiple décrite ici est la substitution classique avec plusieurs variables; elle s'écrirait de façon simple

$$\{x \mapsto E, y \mapsto F\}$$

avec, comme nous l'avons dit, $x \neq y$.)

Transformateur de prédicat

$$\begin{aligned} \mathbf{trm}(S \parallel T) &\iff \mathbf{trm}(S) \wedge \mathbf{trm}(T) \\ \mathbf{prd}_x(S \parallel T) &\iff \mathbf{prd}_x(S) \wedge \mathbf{prd}_x(T) \end{aligned}$$

2.5.8 Mise en séquence

Elle est définie dans [1, Chap. 9, page 2]. Elle correspond à la mise en séquence des langages de programmation impératifs.

Symétriquement au choix non borné, cette opération n'est pas autorisée dans une machine abstraite de plus haut niveau (non raffinée). Elle est par contre autorisée dans une machine d'implémentation.

Syntaxe

$$\boxed{S;T}$$

Ecriture postconditionnelle

$$[S;T]R \iff [S][T]R$$

Transformateur de prédicat La terminaison de la mise en séquence nécessite la terminaison de la première opération, ainsi que celle de la deuxième opération à la suite de la première.

$$\mathbf{trm}(S;T) \iff \big(\mathbf{trm}(S) \wedge \forall x'.(\mathbf{prd}_x(S) \implies [x := x']\mathbf{trm}(T))\big)$$

Le « transformateur de prédicats » traduit la même idée sur les prédicats de chacune des opérations.

$$\begin{aligned} &\mathbf{prd}_x(S;T) \iff \\ &\quad \big(\mathbf{trm}(S) \implies \exists x''.([x' := x'']\mathbf{prd}_x(S) \wedge [x := x'']\mathbf{prd}_x(T))\big) \end{aligned}$$

3 Proposition d'une notion de test pour une machine spécifiée en B

Intuitivement, un comportement d'une machine abstraite B peut être décrit en utilisant une *trace*, c'est-à-dire *une suite finie d'opérations de cette machine commençant par son initialisation.*

Toutefois, parmi toutes ces traces, certaines ne nous intéressent pas beaucoup car leur comportement peut être quelconque à partir d'un certain rang. C'est le cas en particulier des traces dont une opération comporte une précondition qui n'est pas vérifiée: en effet, dans ce cas nous ne pouvons rien

affirmer sur le comportement de la machine à partir de cette opération (section 2.5.3). Nous allons donc être obligé dans un premier temps de définir une notion de *trace licite*.

D'autre part, et puisque nous nous plaçons dans l'optique du test, il est important que nous puissions, en soumettant une trace donnée à un programme censé implémenter la machine spécifiée, déterminer si les résultats obtenus sont en accord avec ceux souhaités. Ce problème nous amènera à décrire une procédure de vérification des sorties du programme.

Dans l'ensemble de cette discussion, pour nous placer dans un cadre intéressant, nous considérerons que la Machine Abstraite initiale, qui nous sert pour la définition du test, est *cohérente*, c'est-à-dire qu'elle satisfait les obligations de preuve de B.

3.1 Trace et trace licite

Nous allons tout d'abord préciser ce que nous entendons par une *trace* sur une machine spécifiée en B. Nous définirons ensuite ce qu'est une trace licite, autrement dit une *trace qui doit être exécutable*.

3.1.1 Trace

Une trace est une suite d'*invocations d'opérations* d'une machine B, qui commence par l'initialisation de la machine, et dont chaque invocation comporte le même nombre d'arguments que l'opération qui lui correspond.

De plus, puisque les résultats des opérations de la trace peuvent être éventuellement réutilisés comme des données dans des opérations ultérieures (ou des étapes ultérieures du test), nous utiliserons des variables pour désigner les arguments effectifs et les valeurs résultats des opérations.

De façon symbolique, une trace peut donc s'écrire sous la forme d'une succession d'invocations d'opérations écrites

$$op_i(a_{i,1} \ldots a_{i,k_i}) \updownarrow r_{i,1} \ldots r_{i,k'_i}$$

où

- op_i désigne la $i^{\text{ème}}$ invocation d'opération de la machine de la trace considérée;
- les $a_{i,j}$ sont les variables représentant les entrées soumises aux opérations (*au niveau de la spécification*); ce peuvent être également des valeurs retournées par des opérations antérieures (autrement dit des $r_{i',j'}$ avec $i' < i$);
- la double flèche $\updownarrow$ indique la terminaison de l'invocation;
- les $r_{i,j}$ sont les variables correspondants aux valeurs retournées par les opérations (*au niveau de la spécification*).

D'autre part, une trace débutant toujours par l'initialisation de la machine abstraite, nous définissons une opération fictive *init* qui correspond à l'initialisation de la machine et permet de préciser que l'on démarre de l'état initial.

Une trace est donc finalement de la forme suivante:

$$
\begin{aligned}
& init. \\
& op_1(a_{1,1} \ldots a_{1,k_1}) \updownarrow r_{1,1} \ldots r_{1,k'_1}. \\
& op_2(a_{2,1} \ldots a_{2,k_2}) \updownarrow r_{2,1} \ldots r_{2,k'_2}. \\
& \ldots . \\
& op_n(a_{n,1} \ldots a_{n,k_n}) \updownarrow r_{n,1} \ldots r_{n,k'_n}
\end{aligned}
$$

D'autre part, si l'opération dont op_i est l'invocation comporte k_i paramètres formels en entrée et k'_i paramètres de retour, alors op_i doit également comporter k_i arguments $a_{i,1} \ldots a_{i,k_i}$ et k'_i valeurs de retour $r_{i,1} \ldots r_{i,k'_i}$.

Exemple Les formes syntaxiques suivantes sont des traces pour la machine *Little_Example_*1 décrite section 2.1:

- $init.\mathbf{enter}(3).\mathbf{enter}(4).\mathbf{enter}(1).\mathbf{maximum} \updownarrow r_{4,1}$
- $init.\mathbf{maximum} \updownarrow r_{1,1}$
- $init.\mathbf{enter}(5).\mathbf{enter}(4).\mathbf{maximum} \updownarrow r_{3,1}.\mathbf{enter}(r_{3,1}).\mathbf{maximum} \updownarrow r_{5,1}.\mathbf{enter}(r_{3,1})$

□

L'invocation d'opération

$$op_i(a_{i,1} \ldots a_{i,k_i}) \updownarrow r_{i,1} \ldots r_{i,k'_i}$$

se traduit sous la forme d'une substitution:

$$V_i = [w_{i,1} \ldots w_{i,k_i} := a_{i,1} \ldots a_{i,k_i}];[op_i];[r_{i,1} \ldots r_{i,k'_i} := u_{i,1} \ldots u_{i,k'_i}]$$

où

les $w_{i,j}$ sont les *paramètres formels d'entrée* de l'opération op_i (*i.e.* les noms des variables d'entrée de op_i, tels que donnés dans la clause OPÉRATIONS de la machine abstraite).

les $u_{i,j}$ sont les *paramètres formels de retour* de op_i.

Une trace consiste en la mise en séquence de plusieurs invocations d'opérations. Elle peut donc aussi s'écrire:

$$V_0;V_1;V_2;\ldots;V_n$$

On notera que V_0 désigne l'initialisation de la machine abstraite[7].

Exemple Les traces de l'exemple précédent se traduisent par les substitutions suivantes:

- *init*;

  $n := 3$;**enter**(n);

  $n := 4$;**enter**(n);

  $n := 1$;**enter**(n);

  $m \longleftarrow$ **maximum**;$r_{4,1} := m$

- *init*;

  $m \longleftarrow$ **maximum**;$r_{1,1} := m$

- *init*;

  $n := 5$;**enter**(n);

  $n := 4$;**enter**(n);

  $m \longleftarrow$ **maximum**;$r_{3,1} := m$;

  $n := r_{3,1}$;**enter**(n);

  $m \longleftarrow$ **maximum**;$r_{5,1} := m$;

  $n := r_{3,1}$;**enter**(n)

□

Etant donnée une spécification B, autrement dit une Machine Abstraite, SP, l'ensemble des traces finies sur SP se note

$$\mathfrak{T}_{SP} = \{\mathrm{t} \mid \exists n > 0, \mathrm{t} = [V_0];\ldots;[V_n]\}$$

On notera en particulier que $[V_0]$ (la substitution généralisée qui ne comporte que l'initialisation) *n'est pas* une trace. Du point de vue du test du programme, on ne peut pas distinguer le succès ou l'échec de l'initialisation en n'effectuant *aucune* opération après; d'autre part, sa terminaison sera nécessairement testée par une autre trace (toutes les traces commençant par l'initialisation).

[7]Cette substitution est notée U dans la section 2.1.

3.1.2 Trace licite

Comme nous l'avons vu lors de l'introduction de la substitution préconditionnée, si la précondition d'une opération n'est pas vérifiée, l'on ne peut rien affirmer sur l'état de la machine après cette opération.

Il est évident que des traces qui comporteraient de tels cas seraient inintéressantes, puisque nous ne pouvons rien affirmer sur l'état de la machine à partir de la première opération dont la précondition n'est pas vérifiée.

Définition Une *trace licite* est une trace dont l'enchaînement des invocations d'opérations vérifie:

$$\mathbf{trm}([V_0;V_1;V_2;\ldots;V_n])$$

autrement dit, une trace dont la terminaison est requise (du point de vue de la spécification).

Notons cependant que la vérification de ce prédicat est semi-décidable puisqu'il est exprimé dans la logique du premier ordre et les ensembles finis.

Exemple Clairement, la trace

$$init.maximum \updownarrow r_{1,1}$$

à laquelle correspond la substitution

$$init;m \longleftarrow \mathbf{maximum};r_{1,1} := m$$

(où $V_0 = [init]$ et $V_1 = [m \longleftarrow \mathbf{maximum};r_{1,1} := m]$) n'est pas une trace licite (la précondition de l'opération **maximum** n'est pas remplie).

Vérifions ceci. Afin de simplifier la démonstration, nous allons tout d'abord calculer les prédicats **trm** et prd_X de **enter**, **maximum**, ainsi que de l'initialisation **init**:

- $\mathbf{trm}(\mathbf{enter}(n))$

  $\iff \quad \mathbf{trm}(n \in \mathbf{nat}_1 | \, y := y \cup \{n\})$

  $\iff \quad n \in \mathbf{nat}_1 \wedge \mathbf{trm}(y := y \cup \{n\})$

  $\iff \quad n \in \mathbf{nat}_1$

- $\mathbf{trm}(m \longleftarrow \mathbf{maximum})$

  $\iff \quad \mathbf{trm}(y \neq \emptyset | m := \mathbf{max}(y))$

  $\iff \quad y \neq \emptyset \wedge \mathbf{trm}(m := \mathbf{max}(y))$

  $\iff \quad y \neq \emptyset$

- L'initialisation termine toujours:

$$\mathbf{trm}(init) \iff \mathbf{trm}(y := \emptyset)$$

- $\mathbf{prd}_X(\mathbf{enter}(n))$ La notation SI ... ALORS ... SINON ... FIN introduite ici est un raccourci pour les deux cas suivants:

  $\iff \mathbf{prd}_X(n \in \mathbf{nat}_1 \mid y := y \cup \{n\})$

  $\iff n \in \mathbf{nat}_1 \implies \mathbf{prd}_X(y := y \cup \{n\})$

  $\iff n \in \mathbf{nat}_1 \implies$

  SI X EST y ALORS $X' = y \cup \{n\}$ SINON $X' = X$ FIN

  - soit la variable que nous considérons est la variable y, auquel cas nous calculons

    $$\mathbf{prd}_y(\mathbf{enter}(n))$$

    qui vaut donc

    $$n \in \mathbf{nat}_1 \implies y' = y \cup \{n\}$$

  - soit la variable que nous considérons n'est pas y, auquel cas

    $$\mathbf{prd}_X(\mathbf{enter}(n)) \iff n \in \mathbf{nat}_1 \implies X' = X$$

- $\mathbf{prd}_X(m \longleftarrow \mathbf{maximum})$

  $\iff \mathbf{prd}_X(y \neq \emptyset \mid m := \mathbf{max}(y))$

  $\iff y \neq \emptyset \implies \mathbf{prd}_X(m := \mathbf{max}(y))$

  $\iff y \neq \emptyset \implies$

  SI X EST m ALORS $X' = \mathbf{max}(y)$ SINON $X' = X$ FIN

- $\mathbf{prd}_X(init)$

  $\iff \mathbf{prd}_X(y := \emptyset)$

  $\iff$ SI X EST y ALORS $X' = \emptyset$ SINON $X' = X$ FIN

Si la trace $init.\mathbf{maximum} \updownarrow r_{1,1}$ était licite, l'on aurait alors

$$\mathbf{trm}([init; m \longleftarrow \mathbf{maximum}; r_{1,1} := m])$$

$$\Longleftrightarrow$$

$$\mathbf{trm}([init]) \wedge \forall X_1'.\big(\mathbf{prd}_{X_1}(init) \implies$$
$$[X_1 := X_1']\mathbf{trm}([m \longleftarrow \mathbf{maximum}; r_{1,1} := m])\big)$$

$$\Longleftrightarrow$$

$$\forall X_1'.\big(\mathbf{prd}_{X_1}(init) \implies$$
$$[X_1 := X_1']\mathbf{trm}([m \longleftarrow \mathbf{maximum}; r_{1,1} := m])\big)$$

$$\Longleftrightarrow$$

$$\forall X_1'.\big(\mathbf{prd}_{X_1}(init) \implies$$
$$[X_1 := X_1']\big(\mathbf{trm}(m \longleftarrow \mathbf{maximum}) \wedge$$
$$\forall X_2'.\big(\mathbf{prd}_{X_2}(m \longleftarrow \mathbf{maximum}) \implies$$
$$[X_2 := X_2']\mathbf{trm}(r_{1,1} := m)\big)\big)\big)$$

Comme $\mathbf{trm}(r_{1,1} := m)$ est toujours vrai, ceci se réduit à:

$$\forall X_1'.\mathbf{prd}_{X_1}(init)$$
$$\implies [X_1 := X_1']\big(\mathbf{trm}(m \longleftarrow \mathbf{maximum})\big)$$

$$\Longleftrightarrow$$

$$\forall X_1'.\mathbf{prd}_{X_1}(init) \implies [X_1 := X_1'](y \neq \emptyset)$$

$$\Longleftrightarrow$$

$$\forall X_1'.\big(\text{SI } X_1 \text{ EST } y \text{ ALORS } X_1' = \emptyset \text{ SINON } X_1' = X_1 \text{ FIN}\big)$$
$$\implies [X_1 := X_1'](y \neq \emptyset)$$

Pour $X_1 = y$, le prédicat quantifié universellement s'écrit $(y' = \emptyset \implies y' \neq \emptyset)$ La quantification universelle est donc fausse, et nous avons donc montré que la trace $init.\mathbf{maximum} \updownarrow r_{1,1}$ n'est pas une trace licite.

□

Etant donnée une spécification B, autrement dit une Machine Abstraite, SP, l'ensemble des traces licites sur SP se note

$$\mathfrak{TL}_{SP} = \{\mathrm{t} \in \mathfrak{T}_{SP} | \mathbf{trm}(\mathrm{t})\}$$

On notera en particulier que si $\mathfrak{TL}_{SP} = \emptyset$, aucune trace de SP n'a de sémantique spécifiée (autrement dit, tout programme qui fournit les opérations de SP, quoi qu'elles fassent, est une implémentation de SP). En cela, la notion de traces licites est plus forte que la notion de cohérence d'une Machine Abstraite telle qu'elle est établie par les obligations de preuve de B.

Ainsi, une machine dont l'invariant serait toujours vrai (par exemple le prédicat $0 = 0$), mais dont les préconditions des opérations seraient toutes fausses (par exemple le prédicat $0 = 1$) serait cohérente au sens des obligations de preuves de B[8] . Par contre, elle ne fournirait aucune trace licite (*i.e.* une telle machine est un cas pour lequel $\mathfrak{TL}_{SP} = \emptyset$). La notion de traces licites, à elle seule, est donc un outil nouveau et important d'analyse d'une spécification B.

3.2 Pourquoi des traces?

Nous justifions ici notre utilisation des traces pour le test.

Nous désirons faire du test, et nous devons donc travailler sur un programme censé implémenter une spécification. Dans ce programme, nous pouvons imaginer qu'il existe un état interne, qui ne correspond pas forcément à l'état défini dans la spécification, puisqu'il a pu être modifié au cours du raffinement de cette spécification.

Nous pouvons alors:

- soit nous autoriser à observer l'état du programme, et nous devons donc considérer toutes les étapes du raffinement pour la conception et la réalisation du test; nous prenons alors le risque que les erreurs éventuelles du raffinement soient reconduites dans la conception du test.

- soit nous interdire d'observer l'état du programme, ce qui, on va le voir, mène à considérer des traces.

Nous considérerons ici comme exemple la spécification donnée par Abrial pour le "maximum", et rappelée dans la section 2.1.

Au niveau de la spécification la plus abstraite (qui est celle qui nous sert à concevoir nos tests), la post-condition (*i.e.* pour ce qui nous intéresse ici, l'invariant de la machine abstraite) est donnée en fonction de la variable y, qui est un ensemble fini contenant des entiers naturels non nuls:

$$y \in F(\mathbf{nat}_1)$$

[8]Les obligations de preuves, données section 2.2, comportent essentiellemnt l'implication $Q \wedge I \implies [V]I$, où Q est la précondition de l'opération, I l'invariant de la machine et $[V]I$ le prédicat formé sur l'invariant I auquel on a appliqué la substitution V. On montre simplement sur la forme de V que $[V]I$ est vrai (V ne contient ni précondition, ni mise en séquence), on sait que Q est faux, et l'implication est donc vraie.

L'implémentation proposée par Abrial est classique. Elle conserve, dans une variable z, le plus grand entier fourni à la machine abstraite jusqu'alors [1, Chap. 12, page 6][9]:

IMPLEMENTATION *Little_Example_3*
REFINES *Little_Example_2*
IMPORTS *Scalar*(0)
INVARIANT $z' = z$
OPERATIONS **enter**$(n) \mathrel{\hat{=}}$
VAR v IN
$v \longleftarrow$ **value**;
IF $n \geq v$ THEN **modify**(n) END
$m \longleftarrow$ **maximum** $\mathrel{\hat{=}}$
BEGIN $m \longleftarrow$ **value** END
END

On voit donc qu'au niveau de l'implantation, la variable y n'existe pas (la seule variable étant la variable z' qui est encapsulée dans la machine *Scalar*). Nous ne pouvons donc pas observer l'état de la Machine Abstraite dans le programme.

3.2.1 Obligations de preuves

Nous avions pensé dans un premier temps tester les obligations de preuve qui n'ont pas été prouvées[10].

Considérons l'obligation de preuve de l'opération **maximum** définie précédemment, qui s'écrit

$$[\ \text{PRE } y \neq \emptyset \text{ THEN } m := \mathbf{max}(y) \text{ END }]\, y \in F(\mathbf{nat}_1)$$

et se simplifie en

$$y \in F(\mathbf{nat}_1)$$

si la précondition $y \neq \emptyset$ est vraie, et est indéfinie dans le cas contraire. Cette obligation de preuve ne fait pas mention des résultats car dans la substitution la variable correspondante a disparu.

[9] La variable z est celle de la machine de raffinement intermédiaire, *Little_Example_2*. D'autre part, la machine *Little_Example_3* décrite ici *utilise* la machine *Scalar*, qui est une machine de librairie qui implémente une variable scalaire. La machine *Scalar* possède une variable (interne), z'; le paramètre de cette machine (la valeur 0) précise la valeur initiale de la variable.

[10] Comme le conseillent H. Waeselynck et J-L. Boulanger [11].

Les obligations de preuve n'apportent donc aucune information sur les résultats (dont il est pourtant clair qu'ils sont importants pour le test).

Bien plus, comme nous l'avons déjà fait remarquer, en ce qui concerne le test, ces obligations de preuve nous obligeraient à prouver des choses sur un état que nous ne savons pas observer (puisqu'il a été modifié lors du raffinement).

3.2.2 Approche par triplets précondition – opération – postcondition "à la Hoare"

Nous aurions pu aussi considérer un test sous la forme de triplets formés d'une précondition, d'une opération, et d'une post-condition.

A chaque opération d'un triplet correspond une opération du programme, et à chaque post-condition définie pour la machine abstraite correspond une certaine post-condition exprimée dans le domaine du programme.

Or, pour pouvoir soumettre un triplet, il nous faudrait nous placer dans un état où la précondition soit satisfaite (d'après la spécification, mais pas nécessairement dans le programme). Le triplet permettrait alors de vérifier la précondition, d'exécuter l'opération, et de vérifier la postcondition.

Malheureusement, cette dernière partie, et plus particulièrement l'opération qui répond à la question: « l'état final (après exécution de l'opération concernée) satisfait-il la post-condition? » est une opération qui pose le même problème que précédemment: on ne peut pas observer l'état de la Machine Abstraite (de spécification) sans être obligé de tenir compte du raffinement, ce que nous voulons éviter afin de ne pas reporter les éventuels erreurs de raffinement dans la conception du test.

Dans l'exemple du **maximum** que nous avons présenté, le problème peut être formulé de la façon suivante: comment pouvons-nous assurer la validité de la post-condition (qui comporte l'invariant de la machine abstraite) alors que les variables de la machine abstraite ne sont pas accessibles? Et, même en imaginant qu'elles le soient[11], il n'est pas possible de déduire la validité de la post-condition portant sur y de la valeur de z', cette dernière variable étant la seule qui existe au niveau de l'implantation[12].

D'autre part, l'approche par triplets nécessite pour la soumission du test de se placer dans un état vérifiant la précondition du triplet à tester. Le seul moyen de se placer dans un tel état est de construire une suite d'invocations d'opérations permettant de l'atteindre. Autrement dit, il s'agirait de construire une *trace* préalablement à la soumission du test!

L'approche par triplets précondition – opération – postcondition pose

[11] Il s'agit d'une hypothèse d'école ...

[12] De façon intuitive, on peut dire qu'il est impossible de reconstruire l'ensemble y à partir de la seule variable z. Pour reconstruire y il faudrait considérer toute l'histoire de la machine (depuis *init*), ce que nous voulons justement éviter dans l'approche par triplets.

donc d'une part un problème évident pour la vérification du test. D'autre part, la soumission d'un tel test nécessitant de construire des traces exécutables, il est plus intéressant d'exploiter directement ces traces, comme nous l'avons proposé dans notre approche.

3.3 Une procédure de vérification de programme

La réalisation d'un test correspond, partant d'une spécification S et d'un programme P,

- d'une part à la sélection d'un jeu de test T;
- d'autre part à l'application d'une procédure de vérification du programme; étant donnés P, S et T, cette dernière doit pouvoir dire "le test a réussi" ou "le test a échoué".

  De façon classique, cette procédure de vérification comporte elle-même deux phases:

  - la soumission du jeu de test T au programme P,
  - la vérification des résultats par rapport à la spécification S.

Toutefois, le cadre de travail de B comporte certaines spécificités que nous mettons tout d'abord en avant. Nous nous intéressons ensuite aux problèmes liés à la soumission d'un jeu de test (jeu de test réalisé pour une machine abstraite écrite en B) à un programme, puis à ceux liés à la vérification des résultats éventuellement obtenus.

3.3.1 Cadre de travail

Le cadre de travail proposé comporte donc trois types d'objets:

- des jeux de tests;
- une spécification écrite en B;
- un programme, écrit dans un langage de programmation donné.

Les jeux de tests sont constitués de traces, c'est-à-dire de suites d'invocations d'opérations de la spécification, opérations instanciées par des arguments.

La spécification est écrite en B, c'est donc une (ou plusieurs) machines abstraites de B.

Dans le cadre de la méthode, un certain nombre d'*objets mathématiques*, qui sont à la base de B, sont disponibles [1, Chap. 3, “Construction of Mathematical Objects”]; c'est le cas des booléens, des entiers, des ensembles.

D'autre part, une spécification donnée utilisera un certain nombre de machines abstraites, regroupées dans des librairies [9, page 4]; c'est le cas des librairies qui implémentent les scalaires, les tableaux.

Ce contexte de librairies est important: il implique que certaines machines ont déjà été prouvées (du point de vue de la sémantique de B), et certainement testées. Ces librairies sont des produits commerciaux, qui sont développés dans le milieu du logiciel sécuritaire, et dont nous considérerons la qualité comme bonne.

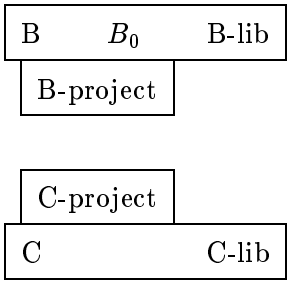
Pour résumer, la seule partie de la spécification qui sera à tester est celle qui est propre aux machines abstraites qui ne font pas parties de ces librairies, et qui est donc spécifique au projet considéré.

Le programme est écrit dans un langage de programmation.

Plus précisément, et dans le contexte d'un Atelier de développement, il a été engendré à partir d'une machine abstraite, dite machine *d'implantation* [1, Chap. 12, page 10] [9, page 4].

Cette machine vérifie un certain nombre de propriétés (en particulier sur la forme des opérations qu'elle comporte) qui la rendent plus aisément transposable dans un langage de programmation. Ainsi, dans l'Atelier B, le langage utilisable dans une machine d'implantation est un sous-ensemble du langage B, le langage B0 [9, pages 64–65]. Dans le cadre de B, la transposition de la machine d'implantation dans le langage de programmation est le plus souvent faite à l'aide d'outils automatisés (ceux de l'Atelier).

Nota Nous schématisons ci-dessous les composants essentiels d'un projet B.



Le projet B (*B-project*) est écrit à l'aide des structures du langage B; la Machine Abstraite d'implantation est écrite à l'aide d'un sous-ensemble du langage B, appelé langage B0 dans [9]. Cette spécification réutilise des composants fournis par des librairies (*B-lib*).

Les outils de génération du code (supposé être ici écrit en C) génèrent un projet C (*C-project*), écrit à l'aide des structures du langage C. Les librairies de B (les *B-lib*s) sont quant à elles fournies sous la forme de librairies écrites en C (*C-lib*).

□

Le programme comporte donc un certain nombre de *constructions spéciales* pour la spécification de départ. *Ce sont ces constructions, et elles seules, que nous chercherons à tester.* De plus, comme pour la spécification, un certain nombre de constructions du programme seront destinées à implémenter les machines abstraites des librairies (utilisées dans la spécification). Ces dernières constructions ne rentrent donc pas dans le cadre de notre test.

Le programme comporte bien entendu aussi des constructions liées au langage de programmation. Ces constructions relèvent typiquement de la compétence des outils qui génèrent une forme exécutable du programme (compilateur, éditeur de lien). Il ne nous appartient pas non plus de les tester. Signalons toutefois, dans le même ordre d'idées, que le système d'exploitation sur lequel est exécuté le programme, ou même le micro-code du processeur, pourraient être autant d'éléments intervenants dans le test.

Pour résumer, nous allons nous intéresser au test des constructions qui, dans le programme, sont liées à la spécification (ou, si l'on préfère, qui ne seraient pas là si cette spécification n'existait pas); ces constructions seront à tester par rapport à celles qui les ont provoquées, qui existent donc dans la spécification, et qui elles sont écrites (et trouvent leur sémantique) dans B. Nous ferons, comme dans [2], l'hypothèse que le reste est implémenté correctement.

3.3.2 Soumission du test

Maintenant que notre cadre de travail est bien établi, nous allons tout d'abord nous intéresser aux problèmes posés par la soumission du test.

Nous choisissons l'approche habituelle suivante: le test est soumis jusqu'à son terme, quoi qu'il arrive, et la vérification prend place après la fin de la soumission.

Activation du programme Un jeu de test comporte un certain nombre de traces (licites). Chaque trace consiste en une succession d'invocations d'opérations. Nous devons donc tout d'abord nous assurer que nous savons effectivement réaliser ces invocations.

Opérations Pour cela, il faut pouvoir établir la relation entre les opérations de la machine abstraite et les opérations du programme (cette relation, étant purement syntaxique, peut être vue comme un morphisme de signature). Autrement dit, à chaque nom d'opération d'une machine abstraite de B doit correspondre un identificateur de procédure du programme.

Exemple Comparaison entre une trace exprimée dans le langage B, et une trace traduite pour un langage de programmation donné (ici, le langage C).

Trace (exprimée en B)

... **enter**(4) ↕ ...

Trace (exprimée en C)

```
...  n = 4; enter(n);  ...
```

□

Les obligations syntaxiques de B nous assurent que les noms des opérations sont conservés tout au long du raffinement. De plus, les outils chargés de générer le programme final (que nous testons) à partir de la spécification la plus raffinée devront eux aussi fournir les éléments permettant d'établir cette relation (éventuellement en conservant aussi les mêmes noms). Cette première relation ne pose donc pas de problème.

Arguments Les profils des opérations d'une machine abstraite doivent être conservés au cours du raffinement de cette machine [1, Chap. 11, page 37-38] [9, page 46] et ne posent donc pas de problème.

Nous devons toutefois établir la relation entre les objets manipulés par les opérations (plus précisément, leurs arguments) dans la spécification, et les types implémentés dans le programme. Cette relation peut être vue comme un morphisme d'algèbre.

Telles qu'elles sont définies dans le *B-Book*, les opérations de B peuvent *a priori* utiliser n'importe quel type de valeur définissable en B (y compris des types ensemblistes, fonctionnels,...). La question se poserait alors de pouvoir traduire ces types complexes dans le langage du programme.

Toutefois, de façon pratique, dans le cadre d'un atelier de développement B, les types autorisés sont drastiquement restreints, comme nous l'avons vu en 2.4.3.

Par conséquent, s'il existe un générateur de code, il inclut nécessairement une traduction des types de la machine dans des types du langage de programmation. Cette traduction est documentée et c'est cette correspondance que l'on utilisera pour la soumission et l'interprétation des tests. Si le passage de la machine d'implantation au programme se fait "à la main", cette traduction devra être suffisamment documentée pour permettre de reconstruire cette correspondance.

Réutilisation des résultats Notons en passant que, par construction du programme, les résultats des diverses opérations auront nécessairement le type attendu; autrement dit, pour reprendre les notations des traces, chacun des $a_{i,j}$ argument d'une opération, ou chacun des $r_{i,j}$ résultat d'une opération, aura à l'exécution un type donné dans le langage de programmation.

Exemple Réutilisation des résultat dans une trace exprimée en B, et exemple de réalisation dans un programme en C.

Trace (exprimée en B)

... **maximum** $\updownarrow r_{3,1}$ **enter**$(r_{3,1}) \updownarrow$ ...

Trace (exprimée en C)

```
...  m = maximum(); r3_1= m;  enter(r3_1);  ...
```

□

La réutilisation des résultats d'opérations antérieures dans une trace ne pose donc pas de problèmes.

En conclusion, l'activation du programme développé avec l'Atelier B est réalisable, à condition que les éléments que nous venons d'inspecter soient vérifiés.

Comment faire *init* ? Toute machine abstraite comporte une partie *initialisation*, que nous avions notée **init** dans le cadre des traces.

Cette opération est, dans l'environnement de l'Atelier B, effectivement toujours disponible (et son nom dans le programme peut être connu) [10]. Ce point n'est pas évoqué dans le *B-Book* mais est néanmoins important, puisque nous devons savoir nous re-placer dans l'état initial au début de chaque soumission d'une trace.

Echec de la soumission Il nous faudra aussi considérer (de façon classique) les cas où la soumission elle-même ne réussit pas:

- le programme "crashe" ce qui met à jour une discordance entre la spécification et le programme, autrement dit une faute: une trace qui devrait être exécutable (puisqu'elle est licite) ne s'exécute pas.
- le programme "boucle" ce qui peut être détecté, comme dans toutes les méthodes de test, par une estimation au pire du temps d'exécution [7, page 687]; il s'agit là encore d'un cas d'échec de la soumission, toutes les opérations d'une trace licite devant se terminer.

3.3.3 Vérification du résultat du test

Après la soumission, nous disposons d'une suite

$$\begin{aligned}
&\updownarrow R_{1,1} \ldots R_{1,k'_1} \\
&\updownarrow R_{2,1} \ldots R_{2,k'_2} \\
&\ldots \\
&\updownarrow R_{n,1} \ldots R_{n,k'_n}
\end{aligned}$$

de valeurs retournées par le programme.

Succès du test Les flèches $\updownarrow$ font partie de la suite des valeurs de retour et indiquent que l'exécution de l'opération considérée s'est achevée.

Nous devons maintenant vérifier que les résultats obtenus lors de la soumission sont bien ceux attendus.

Nous dirons que la soumission de la trace $\mathfrak{t}$, décrite par la suite de substitutions V_0, V_1, ..., V_n est en succès si et seulement si, pour toutes les variables $r_{i,j}$ décrivant les résultats des invocations d'opérations de $\mathfrak{t}$, l'on a[13]

$$[r_{i,j} := \mathit{Abstr}(R_{i,j})]\mathbf{prd}_{r_{i,j}}([V_0; \ldots; V_i])$$

La fonction *Abstr* est la *fonction d'abstraction* liée à cette soumission.

Le succès du test s'écrit donc finalement:

$$\begin{aligned}
&\forall i = 0, \ldots, n, \forall j = 1, \ldots, k'_i, \\
&\quad [r_{i,j} := \mathit{Abstr}(R_{i,j})]\mathbf{prd}_{r_{i,j}}([V_0; \ldots; V_i])
\end{aligned}$$

Fonction d'abstraction De manière similaire à ce que nous avons montré pour la soumission, et plus particulièrement en ce qui concernait les arguments des opérations, nous pouvons affirmer:

- d'une part, que dans le cadre le plus large (celui du *B-Book*), les valeurs retournées par les opérations peuvent être d'un type quelconque.

  Toutefois, dans le cadre de l'Atelier, là encore les types admis sont beaucoup plus restreints; en l'occurrence, ce sont les mêmes que pour les paramètres d'entrée [9, pages 32–33].

[13] On notera dans la définition que l'on ne considère que les substitutions d'indices 0 à i. Du point de vue du prédicat à vérifier, il reste le même (en effet, $r_{i,j}$ n'est pas modifié par les substitutions V_{i+1},...,V_n, et $\mathit{Abstr}(R_{i,j})$ ne peut en aucun cas être une variable). Par contre, la construction du prédicat est plus rapide, et l'étape de vérification est donc plus courte.

- d'autre part, nous devons émettre les mêmes hypothèses que pour les paramètres d'entrée en ce qui concerne les correspondances des types (entre le programme et la spécification).

  En l'occurrence, les types pour lesquels nous disposons d'une correspondance pourront être directement exploités. Pour les autres, il ne pourra s'agir que de *deffered sets*, auquel cas nous utiliserons la méthode décrite dans la section 2.4.5.

La fonction d'abstraction étant donnée, la vérification du prédicat suit les règles de la logique de B.

On voit donc que la vérification des résultats amènent deux conclusions: tout d'abord, comme dans toute approche boîte-noire du test, la présence du raffinement et son impact sur les types pose un problème, que ce soit d'ailleurs à la soumission ou lors de la vérification. Par contre, le fait que l'on travaille sur des programmes développés dans un atelier, et que l'atelier pose des conditions sur la forme de ce programme, simplifie beaucoup l'approche du test.

Nous insistons sur le fait qu'il s'agit là de conditions nouvelles par rapport à celles développées dans le *B-Book*. Mais ces conditions sont suffisantes pour la réalisation du test.

3.3.4 Jeu de test exhaustif

Nous sommes maintenant en mesure de définir le jeu de test exhaustif, ainsi que la notion de succès pour la soumission d'un jeu de test à un programme.

Définitions Le *jeu de test exhaustif* lié à une Machine Abstraite est constitué par l'ensemble de toutes les traces licites que l'on peut construire à l'aide des opérations définies dans la Machine Abstraite.

La soumission d'un jeu de test à un programme est en succès si toutes les traces (licites) constituant le jeu de test sont en succès. Elle est en échec dans le cas contraire.

Le jeu de test exhaustif pour une machine abstraite SP est donc l'ensemble $\mathfrak{TL}_{SP}$ décrit en section 3.1.2.

Le succès du jeu de test exhaustif s'écrit donc

$$\mathfrak{Success}_{SP} =$$
$$\bigwedge_{\substack{\mathfrak{t}=[V_0];..;[V_n] \\ \mathfrak{t}\in\mathfrak{TL}_{SP}}} \quad \forall i = 0,..,n, \forall j = 1,..,k'_i,$$
$$[r_{i,j} := Abstr(R_{i,j})]\mathbf{prd}_{r_{i,j}}([V_0;\ldots;V_i])$$

3.3.5 Validité et non-biais du jeu de test exhaustif

Comme nous l'avons déjà fait remarquer, la méthode B ne donne pas explicitement la sémantique attendue de la part du programme. Nous savons que le raffinement et la machine d'implantation doivent vérifier un certain nombre d'obligations de preuve. Nous pouvons donc proposer une première notion de correction pour B: un programme est correct s'il est une "implémentation correcte" de la dernière étape du raffinement d'une Machine Abstraire.

Malheureusement, cette définition ne dit rien de ce qu'est une "implémentation correcte" d'une machine abstraite: elle se contente de cacher derrière une expression ce qui est caché habituellement dans les Ateliers de développement B. Une interprétation possible est la suivante: le programme est une traduction de la machine d'implantation (qui ne comporte en effet que des structures de données et des structures algorithmiques proches d'un langage de programmation); l'exécutable de ce programme est réalisé par une édition de liens avec les librairies qui implémentent les librairies B.

Cette interprétation ne répond pas plus à la question d'origine: quelle est la sémantique du programme? En particulier, quels sont les résultats attendus du programme lorsque l'on lui soumet un test donné?[14]

Nous proposons comme sémantique une sémantique observationnelle, basée sur les traces.

Nous disons qu'un programme écrit à partir d'une spécification en B est correct s'il est en succès pour le test exhaustif.

Nota Afin de justifier plus complètement cette proposition, il serait intéressant de comparer plus précisément la notion de correction au sens de B (*i.e.* une machine d'implantation dont toutes les obligations de preuve, de raffinement comme de cohérence, sont vérifiées) avec la notion de correction d'un programme par rapport au succès des traces licites.

Nous pouvons maintenant préciser les notions de validité et de non-biais d'un contexte de test, telles qu'elles sont définies dans [2] (voir section 4).

- Un contexte de test est *valide*, si, modulo les hypothèses, il n'accepte que des programmes corrects (*i.e.* le succès du test implique la correction du programme).

[14] Notons que la réponse n'est pas non plus la réponse simpliste: "Les valeurs que rend la machine abstraite d'implantation pour ce test donné." D'une part, cette machine abstraite a beau être déterministe, on ne peut pas prévoir systématiquement les résultats d'un test donné (ces résultats dépendront aussi de l'environnement d'exécution, du système, des interventions ou réponses de l'utilisateur,...). D'autre part, une telle sémantique s'appuierait sur les obligations de preuves du raffinement, dont nous savons justement qu'elles peuvent comporter des erreurs: nous introduirions alors dans l'oracle les erreurs que nous avons évitées dans la construction du jeu de test!

- Un contexte de test est *non-biaisé*, si, modulo les hypothèses, il ne rejette pas de programme correct (*i.e.* la correction du programme implique le succès du test).

Dans le cadre de B, ces notions s'expriment donc de la façon suivante:

Non-biais Si le programme est issu d'une succession de raffinemeents prouvés corrects, et en admettant que la traduction de la machine d'implantation en programme ait été effectuée convenablement, alors ce programme est en succès.

Validité Le succès du programme implique qu'il est équivalent (modulo la traduction) à une machine d'implantation qui satisfait les obligations de preuve (du raffinement et de cohérence).

Autrement dit, il existe une succession de raffinement menant à une machine d'implantation dont ce programme est une traduction.

□

Nous noterons $\mathfrak{H}_{Min}$ les hypothèses minimales de testabilité, *i.e.* l'ensemble des hypothèses que nous avons formulées sur la forme de la spécification et du programme, à savoir:

- sur la spécification:
  - la Machine Abstraite satisfait les obligations de preuve de B.
- sur l'environnement de développement:
  - les librairies d'implantation (*C-lib*) des machines abstraites de librairies (*B-lib*) sont correctement implémentées. Notons qu'il est envisageable de tester la correction de cette implémentation en considérant chaque machine abstraite de librairie et en testant l'implantation qui lui correspond.
  - le compilateur, l'éditeur de lien,... font leur travail correctement (cette hypothèse est classique).
- sur la soumission des traces:
  - à chaque opération *op* de la machine abstraite (*i.e.* de la spécification) on sait faire correspondre une opération op_P du programme P; en particulier, on sait réaliser l'initialisation *init*;
  - les types des paramètres des opérations sont des types simples, tels que ceux décrits section 2.4.3;
  - à chaque type simple *type* de la machine abstraite on sait faire correspondre un type $type_P$ dans le programme P.

- sur le pilote (*driver*) de test et l'oracle:
  - on dispose d'une procédure qui sait déterminer l'échec de la soumission (le programme "crashe", le programme "boucle"); une telle procédure est classique dans un pilote de test;
  - on dispose d'une fonction d'abstraction *Abstr*.

Par construction, nous pouvons alors affirmer que le contexte de test formé

- des hypothèses minimales $\mathfrak{H}_{Min}$,
- du jeu de test exhaustif $\mathfrak{TL}_{SP}$,
- et de la procédure de test $\mathfrak{Success}_{SP}$

est *valide* et *non-biaisé*.

3.3.6 Quelques idées pour le raffinement du contexte de test

La sélection est définie dans [2] comme le raffinement du contexte de test. Ce raffinement est réalisé par l'ajout de nouvelles hypothèses dans le contexte de test, soit afin de réduire le jeu de test (à oracle constant, comme l'a montré Pascale Le Gall dans sa thèse), soit afin d'augmenter le domaine de définition de l'oracle (à jeu de test constant). Nous nous intéresserons ici à la réduction du jeu de test.

En effet, le jeu de test exhaustif que nous avons défini n'échappe pas à la règle et sera (pour la plupart des spécifications) infini. Il sera donc nécessaire, afin de pouvoir réaliser des jeux de tests soumettables et donc finis, d'ajouter des hypothèses de sélection aux hypothèses initiales afin de pouvoir réduire ces jeux de tests.

Première approche Une première idée est de chercher à tester toutes les opérations décrites par la spécification. Ceci nous permet de réaliser une première répartition fdes traces constituants le jeu de test exhaustif: nous considérerons pour chaque opération *op* les traces dans lesquelles elle apparaît[15].

$$\mathfrak{TL}_{op} = \{\mathrm{t} = [V_0];...;[V_n] \in \mathfrak{TL}_{SP} \mid \exists i = 1,..,n, V_i = [w_i := a_i];[op_i];[r_i := u_i] \wedge op_i = op\}$$

Notons que nous n'avons pas réalisé un *partitionnement* des traces qui constituent le jeu de test (en général, une trace contient plusieurs opérations

[15] Dans les définitions qui suivent, $[w_i := a_i]$ est un raccourci pour $[w_{i,1} \ldots w_{i,k_i} := a_{i,1} \ldots a_{i,k_i}]$, et $[r_i := u_i]$ est un raccourci pour $[r_{i,1} \ldots r_{i,k'_i} := u_{i,1} \ldots u_{i,k'_i}]$.

différentes), mais nous pouvons maintenant considérer un sous-ensemble du jeu de test exhaustif constitué des traces licites où apparaît une opération donnée.

Direction de la sélection Pour poursuivre le raffinement du jeu de test, nous pouvons noter que, dans une trace donnée, l'opération qui nous intéresse est précédée d'autres opérations, et est suivie d'autres opérations.

$$\mathrm{t} = [V_0];...[V_{i-1}];[V_i];[V_{i+1}]...;[V_n]$$
$$V_i = [w_i := a_i];[op_i];[r_i := u_i]$$

Seules les opérations qui la précèdent (V_0,...,V_{i-1}) peuvent influer sur son comportement. Par contre, seules les opérations qui la suivent (V_{i+1},...,V_n), ainsi que le ou les résultats éventuels $r_{i,j}$ que cette opération retourne, nous permettent d'affirmer qu'elle s'est bien déroulée[16].

Nous pouvons donc poursuivre dans deux directions: soit nous sélectionnons un sous-domaine du jeu de test en cherchant à décrire ce qui précède cette opération, auquel cas nous nous intéresserons aux valeurs et aux conditions de terminaison du préfixe $[V_0];...;[V_{i-1}]$; soit nous cherchons à décrire l'influence de cette opération sur celles qui suivent, auquel cas nous nous intéresserons aux valeurs et aux conditions de terminaison de la sous-trace $[V_{i+1}];...;[V_n]$.

Rappelons que la terminaison d'une trace t est donnée par le prédicat **trm**(t), tandis que les valeurs modifiées par cette trace sont données, pour chaque variable X, par le prédicat $\mathbf{prd}_X(\mathrm{t})$.

Une étude attentive de la définition de ces deux prédicats dans le cas de la mise en séquence (définitions données section 2.5.8) montre qu'ils sont définis récursivement sur les opérations qui précèdent la dernière opération de la trace considérée, autrement dit, en partant de la fin de la trace. Il paraît donc plus intéressant, pour le test d'une opération donnée, de réaliser la sélection sur les opérations qui précèdent celle-ci.

Nous allons donc réaliser la sélection sur la sous-trace $[V_0];...;[V_{i-1}]$. Nous serons ainsi amenés entre autres à nous intéresser aux domaines de validité des gardes[17] et des préconditions[18] contenues dans la substitution qui définit une opération.

La sous-trace $[V_{i+1}];...;[V_n]$ quant à elle permettra de vérifier la correction de op_i. Notons que cette approche est similaire à celle introduite dans [2] avec la notion de *contexte observable* (voir section 4).

[16] Par contre, s'il y a une erreur, nous ne savons pas quelle(s) opération(s) doivent être mises en cause. Mais ce problème ne relève pas des techniques et outils du test mais de ceux de la mise au point.

[17] Comme par exemple dans une substitution IF .. THEN .. ELSE .. END .

[18] Définies par PRE .. THEN .. END .

Exemple La machine suivante (extraite de [3]) comporte une seule opération, destinée à ordonner trois entiers naturels. Cet exemple a volontairement été choisi simple (la machine abstraite ne comporte pas d'état et ne comprend qu'une seule opération).

Nous allons mettre en évidence différents cas de test dans cette spécification. Pour ce faire, nous réalisons une décomposition du domaine de test en sous-domaines.

MACHINE *Outils*

OPERATIONS $uu, vv, ww \longleftarrow \mathbf{Ordonne\_3\_NAT}(xx, yy, zz) \hat{=}$

PRE

$xx \in \mathbf{nat} \wedge$

$yy \in \mathbf{nat} \wedge$

$zz \in \mathbf{nat}$

THEN

ANY ua, va, wa WHERE

$ua \in \mathbf{nat} \wedge$

$va \in \mathbf{nat} \wedge$

$wa \in \mathbf{nat} \wedge$

$ua \in \{xx, yy, zz\} \wedge$

$va \in \{xx, yy, zz\} \wedge$

$wa \in \{xx, yy, zz\} \wedge$

$ua \leq va \wedge$

$va \leq wa \wedge$

THEN

$uu, vv, ww := ua, va, wa$

END

END

END

Une première idée est la suivante: à quelle condition la terminaison de l'opération est-elle réalisée? Elle l'est dans le cas où **trm**(**Ordonne_3_NAT**)

est vrai. **trm**(**Ordonne_3_NAT**) s'écrit (après simplification)

$$xx \in \mathbf{nat} \wedge yy \in \mathbf{nat} \wedge zz \in \mathbf{nat}$$

ce qui nous permet d'ores et déjà de limiter le domaine des entrées (et, par conséquent, le domaine de sélection) à l'ensemble **nat**.

Une deuxième idée est la suivante: quels résultats allons-nous pouvoir observer? Les variables qui contiennent les résultats de l'opération sont uu', vv' et ww', les valeurs des variables uu,vv et ww après la réalisation de l'invocation de l'opération. Si nous observons $\mathbf{prd}_X$(**Ordonne_3_NAT**), qui s'écrit après simplification

$$\begin{array}{lll} & xx \in \mathbf{nat} & \\ \wedge & yy \in \mathbf{nat} & \Longrightarrow \quad \exists ua, va, wa. \\ \wedge & zz \in \mathbf{nat} & \end{array} \begin{array}{ll} & ua \in \mathbf{nat} \\ \wedge & va \in \mathbf{nat} \\ \wedge & wa \in \mathbf{nat} \\ \wedge & ua \in \{xx, yy, zz\} \\ \wedge & va \in \{xx, yy, zz\} \\ \wedge & wa \in \{xx, yy, zz\} \\ \wedge & ua \leq va \\ \wedge & va \leq wa \\ \wedge & uu' = ua \\ \wedge & vv' = va \\ \wedge & ww' = wa \end{array}$$

nous avons $uu' = ua$, $vv' = va$, $ww' = wa$, donc

$$\begin{array}{llllll} & uu' \in \mathbf{nat} & \wedge & vv' \in \mathbf{nat} & \wedge & ww' \in \mathbf{nat} \\ \wedge & uu' \in \{xx, yy, zz\} & \wedge & vv' \in \{xx, yy, zz\} & \wedge & ww' \in \{xx, yy, zz\} \\ \wedge & uu' \leq vv' & \wedge & vv' \leq ww' & & \end{array}$$

De plus, $uu' \in \{xx, yy, zz\}$ peut se réécrire en

$$uu' = xx \vee uu' = yy \vee uu' = zz$$

de même que $vv' \in \{xx, yy, zz\}$ et $ww' \in \{xx, yy, zz\}$.

Finalement, nous avons d'une part

$$\begin{array}{lcl} xx \in \mathbf{nat} & & uu' \in \mathbf{nat} \\ yy \in \mathbf{nat} & \wedge & vv' \in \mathbf{nat} \\ zz \in \mathbf{nat} & & ww' \in \mathbf{nat} \end{array}$$

qui servent à "typer" les valeurs (ou, du point de vue de la sélection, permettent de réduire le domaine de sélection à **nat**).

Nous avons d'autre part

$$\begin{array}{l} (uu' = xx \vee uu' = yy \vee uu' = zz) \quad \wedge \\ (vv' = xx \vee vv' = yy \vee vv' = zz) \quad \wedge \\ (ww' = xx \vee ww' = yy \vee ww' = zz) \quad \wedge \\ uu' \leq vv' \wedge vv' \leq ww' \end{array}$$

La mise sous forme normale disjonctive de ce prédicat (qui est d'ailleurs la méthode proposée par Dick et Faivre dans [5]) nous fournit 27 sous-cas pour la sélection:

1	$uu' = xx$	$\wedge$	$vv' = xx$	$\wedge$	$ww' = xx$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
2	$uu' = xx$	$\wedge$	$vv' = xx$	$\wedge$	$ww' = yy$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
3	$uu' = xx$	$\wedge$	$vv' = xx$	$\wedge$	$ww' = zz$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
4	$uu' = xx$	$\wedge$	$vv' = yy$	$\wedge$	$ww' = xx$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
5	$uu' = xx$	$\wedge$	$vv' = yy$	$\wedge$	$ww' = yy$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
6	$uu' = xx$	$\wedge$	$vv' = yy$	$\wedge$	$ww' = zz$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$
7	$uu' = xx$	$\wedge$	$vv' = zz$	$\wedge$	$ww' = xx$	$\wedge$	$uu' \leq vv'$	$\wedge$	$vv' \leq ww'$

etc...

Les cas **1**, **2**, **3**, **4**, **5**, et plus généralement les cas où l'une des variables xx,yy,zz apparaît plusieurs fois sont manifestement des erreurs de spécification. La sélection du jeu de test permet donc aussi de mettre en évidence des cas pour lesquels la spécification est discutable; il s'agit là d'une observation classique.

Si nous considérons maintenant un cas "manifestement correct" (par rapport à ce que nous voulions spécifier à l'origine), par exemple le cas **6**, il vient:

$$xx \leq yy \wedge yy \leq zz$$

Nous avons bien obtenu des conditions sur les arguments de l'opération.

Nous pouvons alors considérer les deux sous-cas naturels de l'opérateur $\leq$: un pour l'égalité, un pour l'inégalité stricte. De même, nous pouvons choisir de considérer ces sous-cas soit selon $xx \leq yy$, soit selon $yy \leq zz$. Si par exemple nous choisissons $xx \leq yy$, nous avons alors deux sous-domaines: un pour $xx < yy$, un pour $xx = yy$; de même pour $yy \leq zz$, qui donne deux autres sous-domaines: $yy < zz$ et $yy = zz$. La combinaison de ces domaines donne quatre sous-domaines[19] :

$$\begin{array}{lcl} xx = yy & \wedge & yy = zz \\ xx = yy & \wedge & yy < zz \\ xx < yy & \wedge & yy = zz \\ xx < yy & \wedge & yy < zz \end{array}$$

Ce raisonnement n'est pas limité au cas **6**, mais peut être facilement étendu aux autres cas.

Toutefois, une limitation importante demeure: nous avons « intuitivement » utilisé un dépliage de l'opérateur $\leq$ en « $<$ ou $=$ ». Cette manipulation était justifiée (dans [2]) par la présence d'axiomes dont le dépliage n'était qu'une conséquence; ce n'est pas le cas dans B, qui ne fournit pas explicitement ces règles.

La même remarque peut par ailleurs s'appliquer à la transformation de $uu' \in \{xx, yy, zz\}$ en une disjonction d'égalités.

Un autre exemple est le cas de l'opération **maximum** de la machine *Little_Example_*1 que nous avons déjà étudiée.

Cette opération comporte l'expression $\mathbf{max}(y \cup \{n\})$. On perçoit que la décomposition en sous-domaine devra s'orienter vers deux sous-cas: soit n est plus grand que tous les éléments de y (auquel cas il est le maximum cherché), soit n est plus petit que le plus grand élément de y (auquel cas le maximum cherché est $\mathbf{max}(y)$).

Là encore, cette approche, bien que classique, devrait néanmoins s'appuyer sur les définitions de **max** et $\cup$ telles qu'elles sont données en B pour être justifiée pleinement.

□

Pour poursuivre la sélection sur cet exemple, il conviendrait pour chaque cas exhibé de construire un « passé » $[V_0; \ldots; V_i]$ qui satisfasse le prédicat

[19] Cet exemple de décomposition de $\leq$ est inspiré de celui présenté dans [2] à l'aide de la technique de *dépliage*.

trm (*i.e.* qui soit un préfixe de trace licite) et dont les entrées appartiennent au domaine choisi. Si par exemple on choisit de ne couvrir ce cas que pour *un seul* passé cohérent[20], on effectue dans ce cas une hypothèse d'uniformité [2] sur le domaine des arguments et des résultats décrits par la substitution $[V_0;\ldots;V_i]$. De plus, si les observations fournies par cette substitution n'étaient pas suffisantes, il conviendrait alors de construire un « futur » $[V_{i+1};\ldots;V_k]$ qui les complète.

On perçoit à travers cet exemple que, bien que la technique de décomposition proposée semble permettre l'utilisation d'hypothèses de sélection classiques, un important travail d'étude des structures de B reste à mener pour guider utilement la sélection; travail qui devra s'appuyer sur les structures de B qui définissent les objets et les opérateurs des expressions et des prédicats.

4 Comparaison avec les travaux du domaine

Comme nous l'avons dit, notre démarche est une approche "black box" du test. Elle s'inspire en particulier de [2] et [6].

Dans [2], les auteurs proposent la construction de jeux de tests à partir d'une spécification en utilisant la notion de *contexte de test*. Un contexte de test comporte des hypothèses de test (hypothèses que l'on a effectuées sur le comportement du programme), un jeu de test (comportant, pour être praticable, un nombre fini de tests unitaires), et un oracle (chargé de dire, en fonction des résultats fournis par le programme en réponse à un test, si le programme est en accord avec la spécification ou pas).

Les propriétés attendues d'un contexte de test sont le non-biais (*unbias*), c'est-à-dire que, modulo les hypothèses, il ne rejette pas de programme correct, et la validité (*validity*), c'est-à-dire que (toujours modulo les hypothèses) il n'accepte que des programmes corrects (ou encore, que tout programme incorrect est rejeté). La présence ou l'absence de ces propriétés dépend des hypothèses, du jeu de test et de l'oracle.

D'autre part, cet article introduit des hypothèses de sélection (régularité, uniformité) et une technique de décomposition par cas à partir des axiomes d'une spécification algébrique (dépliage).

En ce qui concerne la vérification des résultats du test, notre approche est similaire à celle des contextes d'observations [6]. La sous-trace $[V_{i+1}]$;...;$[V_n]$ constitue en effet un contexte d'observation pour la sous-trace $[V_0]$;...$[V_i]$, c'est-à-dire qu'elle permet d'*observer* l'état de la machine abstraite (après la réalisation de V_i) en considérant l'influence de cet état sur les opérations qui suivent V_i.

La représentation en est cependant différente: dans [2] et [6], les con-

[20] En prenant en compte les prédicats **trm** et $\mathbf{prd}_X$ intermédiaires.

textes observables sont introduits afin de pouvoir observer des valeurs de sortes non-observables; dans B, les contextes observables sont introduits afin d'observer l'état de la machine, qui ne peut pas être observé directement.

L'article de Dick et Faivre présentant une procédure de sélection automatique de jeu de test pour VDM [5] suit une approche similaire à celle décrite dans [2].

VDM est une méthode de spécification basée sur la notion de fonction partielle (alors que B est basé sur les ensembles et les booléens); les opérations y sont spécifiées par des pré- et post-conditions (alors qu'en B elles sont spécifiées par des substitutions généralisées). VDM présente néanmoins de grandes similitudes avec B: on retrouve la notion d'état et d'invariant, le rôle des préconditions,...

Comme nous l'avons déjà indiqué, la première partie de l'algorithme proposé dans cet article consiste en une mise sous forme normale disjonctive des obligations de preuve exhibées par la spécification.

Les auteurs font quelques remarques, concernant VDM, qui se rapprochent de celles que nous avons présentées: la nécessité d'avoir des types simples (entiers et booléens comme types de base, ainsi que les ensembles, séquences et fonctions totales sur ces types de base); ils indiquent aussi que les obligations de preuve (de VDM) devraient permettre de détecter des combinaisons de valeurs d'entrée et de pré-conditions pour lesquels la spécification n'est pas satisfiable[21]; enfin, ils indiquent que les post-conditions peuvent fournir des contraintes sur les résultats possibles des opérations: nous affirmons même que ce sont ces post-conditions qui doivent être utilisées pour la vérification des résultats du test.

L'algorithme qu'ils proposent pour la sélection est basé sur la construction d'un automate fini, qui décrit les états atteignables par la spécification (à partir de l'état initial et en composant les opérations), et les transitions (correspondant aux opérations) entre ces états. Leur but est de pouvoir placer le programme dans un état connu (en créant une suite d'opérations, semblable à une trace), nécessité par le domaine de test.

Nous avons montré dans ce rapport qu'il pouvait être impossible d'observer (dans le programme) l'état du système (tel qu'il est décrit dans la spécification). Bien qu'ils mentionnent les problèmes liés à l'observation de l'état, Dick et Faivre sont obligés (en raison du non-déterminisme de certaines opérations), de l'observer.

Nous avons ici évité ce problème en observant *indirectement* les états intermédiaires à travers les résultats retournés par chaque transition.

[21] Cette idée est poussée un peu plus loin ici, puisque nous ne considérons, avec les traces licites, que le cas où la spécification est satisfiable.

5 Conclusion

Nous avons effectué une analyse du test de programmes écrits à partir de spécification B, selon une optique « boîte noire ». Nous avons choisi et justifié comme test unitaire une notion de trace licite, ou trace exécutable, dont la terminaison est requise par la spécification. Nous avons exposé les difficultés liées à la soumission et à la vérification de ces tests, et mis en avant les spécificités liées à B. La phase de vérification met plus particulièrement l'accent sur le manque d'une véritable sémantique opérationnelle de la méthode B; nous avons proposé une sémantique observationnelle, qui coïncide avec le succès du jeu de test exhaustif. Enfin, nous proposons quelques pistes pour la sélection de jeux de tests praticables dans le cadre de B.

Cette étude ouvre donc deux problématiques. La première est celle d'une sémantique des programmes pour B. Il conviendrait, soit de confirmer la validité de notre proposition, soit de présenter une autre solution. Le deuxième thème intéressant est celui de l'étude des structures de B, afin de proposer des critères pertinents pour la sélection de jeux de tests.

6 Bibliographie

References

[1] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, parution prévue: Août 1996.

[2] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[3] Jean-Louis Boulanger. Spécification du triangle en B. Technical report, INRETS-ESTAS, May 15 1995. Le Triangle est un petit problème bien connu dans le monde du test, dont J-L. Boulanger propose une spécification en B. Le projet complet et sa documentation sont disponible sur le serveur du B User Group: `ftp://ftp.inrets.fr/ESTAS/BUG/Spec/Drafts/`.

[4] Dan Craigen, Susan Gerhart, and Ted Ralston. *An International Survey of Industrial Applications of Formal Methods*. U.S. Dept of Commerce, Technology Administration, National Institue of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, March 1993. Volume 1: Purpose, Approach, Analysis, and Conclusions (viii+117). Volume 2: Case Studies (viii+188). This document is available online at URL `http://www.ora.on.ca`.

[5] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In Springer-Verlag, editor, *FME'93*, LNCS volume 670, pages 268–284, 1993.

[6] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT'95*, LNCS volume 915, pages 82–96, 1995.

[7] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, November 1983.

[8] Gérard Guiho and Claude Hennebert. Sacem software validation. In *Proc. 12th International Conference on Software Engineering (ICSE'90)*, pages 186–191. IEEE Computer Society Press, March 26–30 1990. Nice–France. Session 9, Experience Reports 2. Chair: K. Reed, La Trobe University. Recommended by: Nancy Leveson. Gérard Guiho was with GEC ALSTHOM, Claude Hennebert was with RATP Service TT.

[9] Digilog / Stéria Méditerranée. *Le langage B, Manuel de référence, document provisoire*, June 7 1996.

[10] Clément Roques. Communication privée. Digilog, June 1996.

[11] Hélène Waeselynck and Jean-Louis Boulanger. The role of testing in the B formal development process. In *Proc. 6th International Symposium on software Reliability Engineering (ISSRE'95)*, pages 58–67. INRETS, October 24-27 1995.