Génération automatique de composants logiciels sûrs à partir de spécifications formelles B

Dorian Petit

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre : 03-34

THÈSE

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Discipline Informatique

présentée par

Dorian PETIT

Sujet de la thèse :

Génération automatique de composants logiciels sûrs à partir de spécifications formelles B

Soutenue publiquement le 18 décembre 2003 devant le jury composé de :

Jean-Paul Bodeveix	Professeur, Université de Toulouse	Rapporteur
Jeremy Dick	Telelogic	Examinateur
Arnaud Fréville	Professeur, Université de Valenciennes	Directeur
Georges Mariano	Chargé de recherche, INRETS/ESTAS	Examinateur
Jean-François Monin	Professeur, Université de Grenoble	Rapporteur & Président de jury
Vincent Poirriez	Maître de conférence, Université de Valenciennes	Co-directeur

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre : 03-34

THÈSE

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Discipline Informatique

présentée par

Dorian PETIT

Sujet de la thèse :

Génération automatique de composants logiciels sûrs à partir de spécifications formelles B

Soutenue publiquement le 18 décembre 2003 devant le jury composé de :

Jean-Paul Bodeveix	Professeur, Université de Toulouse	Rapporteur
Jeremy Dick	Telelogic	Examinateur
Arnaud Fréville	Professeur, Université de Valenciennes	Directeur
Georges Mariano	Chargé de recherche, INRETS/ESTAS	Examinateur
Jean-François Monin	Professeur, Université de Grenoble	Rapporteur & Président de jury
Vincent Poirriez	Maître de conférence, Université de Valenciennes	Co-directeur

Remerciements

Le travail présenté dans ce mémoire a été réalisé au sein de l'Institut National de Recherche sur les Transports et leur Sécurité (INRETS) dans l'équipe Évaluation des Systèmes de Transport et de leur Sécurité (ESTAS) et au sein du Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines de l'université de Valenciennes et du Hainaut Cambrésis dans l'équipe Recherche Opérationnelle et Informatique. Ce travail a été co-financé par l'INRETS et la région Nord-Pas de Calais.

Je suis très reconnaissant envers Messieurs Jean-Paul Bodeveix, Professeur à l'université de Toulouse III, et Jean-François Monin, Professeur à l'université Joseph Fourier de Grenoble, de me faire l'honneur de rapporter ce mémoire. Je suis également reconnaissant envers Monsieur Jeremy Dick de Telelogic pour avoir accepté d'examiner cette thèse.

Je remercie monsieur le Professeur Arnaud Fréville d'avoir dirigé cette thèse. Des remerciements particuliers vont à messieurs Vincent Poirriez, maître de conférences à l'université de Valenciennes et Georges Mariano, Chargé de Recherche à l'INRETS pour m'avoir initié à la recherche durant un stage de maîtrise d'informatique et pour m'avoir guidé dans les travaux présentés dans ce mémoire.

Je tiens à remercier messieurs Gérard Couvreur et Frédéric Semet, respectivement directeur des équipes de recherche ESTAS et ROI pour m'avoir accueilli et pour m'avoir donné les moyens de mener à bien ma recherche. J'associe bien évidemment les membres respectifs de ces deux équipes à ces remerciements pour les échanges que j'ai eu qui m'ont permis d'élargir ma culture générale, notamment dans le domaine des transports ferroviaires.

Je tiens à remercier toutes les personnes que j'ai rencontré et qui m'ont permis durant mon parcours d'avancer dans ma perception de certains problèmes : Jean-Louis, Joaquin et à la «B Team» Arnaud, David, Jérôme, Samuel, Stéphane et Yann.

Merci également aux personnes qui ont rendu ces quelques années de recherche agréables et ont rendu les pauses café plus vivantes. Merci donc à Damien, Nicolas, Phillipe, Sophie, Thierry, Xavier et de nouveau la «B Team».

Enfin, de grands remerciements sont à adresser à ma famille qui m'a permis de poursuivre mes études et tout particulièrement à Christelle qui m'a soutenu tout au long de l'élaboration de ce travail malgré sa propre recherche.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

L'industrie du logiciel a fortement évolué ces dernières décennies. Cette évolution a eu pour objectif d'améliorer le processus de production du logiciel et a conduit au développement de diverses techniques et méthodes de développement. De ces travaux visant à l'amélioration du processus de production ont émergé diverses techniques de développement de logiciels.

Certains domaines d'activité nécessitent de porter une attention particulière aux logiciels développés. Par exemple, les systèmes de transport public sont des systèmes pour lesquels un dysfonctionnement des logiciels est inacceptable. Ces logiciels dont le dysfonctionnement peut entraîner des pertes humaines ou financières considérables sont dits critiques. Comme pour tous les domaines d'activité, les domaines critiques n'ont pas échappé à l'avènement du logiciel. En effet, les développements de systèmes comme ceux se retrouvant embarqués dans les rames de métro de type VAL ou METEOR (construits par Siemens Transportation Systems) ont évolué : le fonctionnement du pilote automatique des métros de type Val est assuré par assemblage de composants matériels sûrs alors que ceux des métros de type METEOR sont assurés par un logiciel. Ce changement est motivé par la diminution des coûts de développement car les composants matériels sûrs sont coûteux.

Cette évolution des systèmes matériels vers des systèmes logiciels ne s'est pas faite au détriment de la sûreté de fonctionnement du système. Pour répondre à ces nouvelles exigences émanant des industries développant des systèmes logiciels critiques, les techniques et les méthodes de développement ont du évoluées. La validation traditionnelle consistant à vérifier *a posteriori* que, sur des ensembles de données passées en entrée du système, l'exécution mène bien au résultat attendu ne répondait plus au niveau d'exigences de sûreté de fonctionnement.

L'une des techniques qui a émergé de cette évolution est le développement formel. Sans aller trop loin dans la définition de ce qu'est un développement formel (qui sera abordé dans le premier chapitre), nous pouvons le définir comme un développement permettant de s'assurer par un raisonnement mathématique de la conformité du code vis-à-vis des exigences.

La méthode B est une des techniques de développement logiciel qui a été développé durant les années 80. Cette méthode a été utilisée essentiellement dans le développement de système de transport guidé et notamment pour le développement du pilote automatique embarqué du métro METEOR que nous avons déjà cité.

Le développement de systèmes logiciels critiques nécessite de maîtriser le processus de développement de ce système. L'objectif de ce mémoire est d'étudier l'un des aspects du processus de développement de la méthode B : la génération de code. C'est un aspect important de la méthode, car il permet de

différencier B des autres méthodes par cette intégration de la phase de génération de code. Nous verrons que la maîtrise du processus de génération de code nécessite une compréhension et une modélisation des moyens mis à disposition en B pour composer les spécifications. De cette modélisation, nous retirerons, d'une part, une vision différente de la composition de spécifications B nous permettant de simplifier la présentation de ce système de composition, et, d'autre part, un support pour la génération de code et donc la traduction de cette composition. La génération de code sera orientée vers la génération de composant logiciel. Le processus de génération est adaptable et permet de viser différents langages cibles pour le code. Nous examinerons également l'impact de la génération de composant sur les processus de développement de logiciels.

Structure du document

Le premier chapitre de ce mémoire sera consacré à la présentation de trois techniques de développement logiciel. La première sera le développement formel, la deuxième le développement à base de composants et la troisième le développement par contrats. Nous montrerons que ces trois techniques sont complémentaires et que dans une approche «hybride», il est possible de tirer des avantages de chacune des trois méthodes.

Le deuxième chapitre sera consacré à l'étude de la méthode B. Cette étude se focalisera sur les aspects langages supports de la méthode ainsi que sur l'aspect développement des spécifications. Nous présenterons également la génération de code qui est actuellement possible à partir de spécifications B. En fin de chapitre, nous étudierons un algorithme de manipulation de spécifications que nous avons implanté. Cet algorithme nous servira dans la suite des travaux.

Le troisième chapitre effectuera un zoom sur un aspect du langage B important pour la génération de code : la modularité. Nous étudierons cette modularité et montrerons qu'il est en fait possible d'aborder la modularité de B d'une manière plus simple que ce qui est actuellement fait. Nous utiliserons cette vision pour modéliser le système de modules de B à la manière des systèmes de modules des langages ML. Cette modélisation nous permettra de manipuler la modularité B et les modules B dans la suite des travaux sur la génération de code.

Le quatrième chapitre concernera spécifiquement la génération de code. Il reprendra les aspects qui ont été développés dans les chapitres précédents afin d'obtenir du code à partir des spécifications B. Trois points importants seront discutés dans la présentation de la génération de code : l'aspect génération de composants logiciels, l'aspect génération de code annoté et enfin l'impact de cette génération de code sur le processus de développement B.

Le cinquième chapitre sera consacré à la présentation d'une plateforme logicielle dans laquelle nos travaux s'intègrent. Nous reviendrons sur les développements liés aux travaux présentés dans ce mémoire notamment par un aspect quantitatif. Nous développerons également quelques perspectives pour la poursuite des travaux que nous avons exposés dans ce mémoire.

Chapitre 1

Problématique

Sommaire

1.1 Introduction

Le logiciel prend une place de plus en plus importante dans les «systèmes». Ainsi, là où, il y a encore quelques années, se trouvait un opérateur humain pour surveiller la température d'une centrale thermique, se trouve désormais un logiciel. Un autre exemple est celui du pilotage des rames de métro. Celui-ci, effectué par un opérateur humain, a évolué et l'opérateur humain a dans un premier temps été remplacé par un pilote automatique «câblé» (les fonctions de sécurité sont réalisées par assemblage de composants électroniques sécuritaires) pour ensuite être remplacé par du logiciel dans les métros automatiques les plus récents. L'émergence des «technologies de l'information» a également été un moteur pour l'essor de l'industrie du logiciel.

Face à cette émergence du logiciel dans la vie quotidienne, l'industrie du logiciel a du évoluer au cours du temps afin de répondre aux exigences métiers en perpétuelle évolution. Cette évolution ne s'est pas pas faite sans heur et l'industrie a traversé plusieurs «crises» dans son existence. Ces crises lui ont fait prendre conscience de la nécessité d'augmenter la qualité de ces produits tout en conservant, voire en augmentant, sa productivité.

Une des préoccupations des industriels du logiciel est donc d'augmenter la qualité de leur produits. Cette amélioration passe par une maîtrise du processus de développement. Ce processus de développement est découpé en plusieurs phases. La première phase, l'analyse des besoins, consiste à cerner le problème et les contraintes du système à développer. La seconde phase, la spécification globale, consiste à décrire le système du point de vue des fonctionnalités attendues. La troisième phase, la conception architecturale et détaillée, enrichit la description précédente en détaillant la décomposition du problème, les choix d'implantation. La phase suivante, l'implantation, consiste, quant à elle, à réaliser le système décrit dans la phase précédente. La dernière phase, l'intégration et la validation, correspond à mettre en service le système dans l'environnement adéquat (l'environnement cible) et à vérifier que le système effectue bien ce qui est attendu. Cette structuration du processus de développement est un premier pas vers l'amélioration de la qualité du logiciel, mais une question reste essentielle : est-on sûr de ce que fait le logiciel développé ? Et plus généralement, le logiciel qui vient d'être développé répond-il aux exigences de sécurité ?

Pour répondre à ces questions, des recommandations, des standards, des normes ont été développés. En ce qui concerne les systèmes ou produits des technologies de l'information, des critères d'évaluation ont été développés afin de garantir des niveaux de sécurité. Le premier recueil de ces critères date de 1985 ; il s'agit de [Tcsec85] publié par le département de la défense américaine. D'autres travaux ont ensuite été menés pour harmoniser ces critères au niveau mondial. Les *Information Technology Security Evaluation Criteria (ITSEC)* se sont appuyés sur diverses initiatives nationales. L'effort d'harmonisation a été poursuivi et a donné les *Common Criteria (CC)* [Cc96]. Ces critères d'évaluation mettent en avant certaines techniques à utiliser suivant le niveau de sécurité voulant être atteint. Dans les niveaux les plus élevés, l'utilisation des méthodes formelles (une définition de ces termes est donnée en 1.2.1) est recommandé, voire exigée (à partir du niveau 4 des ITSEC par exemple). Dans les domaines des transports ferroviaires, une norme européenne ([En5012801]) détermine les exigences en matière de développement

logiciel destiné à être utilisé dans ce domaine d'activité.

Une autre préoccupation de l'industrie du logiciel est d'accroître sa productivité. Les travaux qui ont été menés dans ce domaine avaient pour objectif de résoudre la «crise du logiciel» qui existe depuis la fin des années 60, début des années 70. Cette crise a donné naissance au génie logiciel qui a depuis évolué de part les techniques utilisées pour répondre à l'objectif de productivité.

Dans ce chapitre, nous aborderons trois techniques de développement logiciel relativement indépendantes dans leur concept. La première est le développement formel, la deuxième est le développement à base de composants et la troisième est la programmation par contrats. Nous montrerons dans ce chapitre que ces trois techniques qui *a priori* paraissent indépendantes, peuvent être utilisées de manière complémentaire afin de répondre aux deux préoccupations de l'industrie du logiciel : la qualité et la productivité.

1.2 Le développement formel

Le développement formel de logiciel est une technique de production de logiciel destinée à répondre à des préoccupations de fiabilités du logiciel.

Les domaines d'application pour lesquels l'utilisation des méthodes formelles semble une évidence sont les domaines où les défaillances s'expriment en termes de vies humaines ou de pertes financières conséquentes. Les logiciels de cette catégorie sont dits critiques.

1.2.1 Définitions

Avant d'avancer dans l'exploration de ces techniques formelles, rappelons quelques définitions concernant les termes utilisés :

D'après le petit Larousse ([Larousse]),

> *Méthode n. f. Ensemble ordonné de manière logique de principes, de règles, d'étapes permettant de parvenir à un résultat.*
> *formel, elle adj. Qui est formulé avec précision, qui n'est pas équivoque.*

D'après Monin [Monin00], *une méthode est une démarche permettant d'atteindre progressivement un objectif donné.* Il parle plutôt de technique formelle que de méthode formelle. En effet, l'auteur considère que l'aspect méthode (au sens étapes menant à un résultat) est nettement moins développé par rapport à l'aspect technique (au sens moyens mis à disposition pour atteindre un résultat) formelle.

D'après Hinchey et Bowen [Hinchey et al.95], *une méthode formelle est un ensemble d'outils et de notations (avec une sémantique formelle), utilisés pour spécifier de manière non ambiguë les spécificités du système et supportant les preuves de propriétés sur ces spécifications et les preuves de corrections d'une implémentation éventuelle par rapport à la spécification.*

En recoupant ces définitions, une méthode formelle peut être définie comme une technique permettant de spécifier un système de manière non ambiguë et permettant de raisonner par le biais de preuves ou de manipulations mathématiques.

Les méthodes formelles sont généralement divisées en deux grandes familles (comme dans [Ofta97, chapitres 1 et 2]). La première famille comporte les méthodes dites constructives et la seconde les méthodes par vérification de modèles. Les méthodes de la première famille vont «construire» au fur et à mesure le produit final attendu alors que celles de la seconde famille vont vérifier la correction de l'expression de ce produit sous forme d'automates (par exploration des états atteignables). Dans ce mémoire, nous ne considérerons que les méthodes constructives (bien que la limite ne soit pas stricte d'après [Ofta97, chapitre 2]) et plus particulièrement la méthode B.

1.2.2 De l'informel au formel

Les étapes d'un développement formel sont schématisées dans la figure 1.1. Cette figure est une vue très simplifiée mais permet néanmoins de représenter les trois grandes phases du développement. La première phase concerne l'élaboration du cahier des charges. Ce cahier des charges peut prendre diverses formes qui peuvent aller du simple document en langage naturel au document dans lequel un début de formalisation a été mené par l'utilisation par exemple de méthodes graphiques comme UML ([Booch et al.99]).

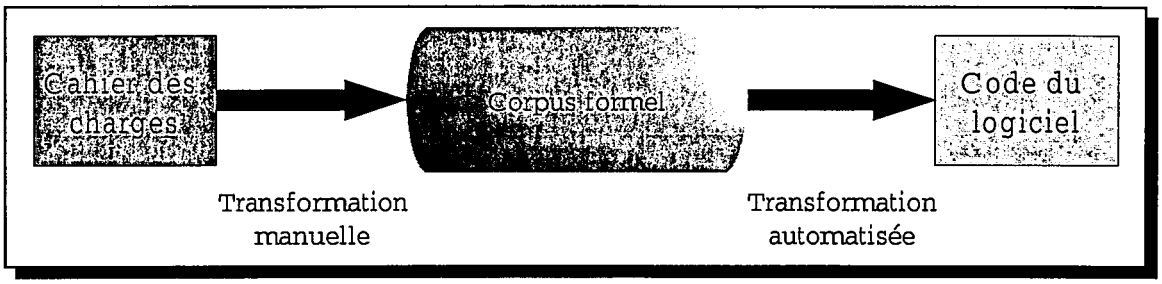


FIG. 1.1 – Du cahier des charges au code

Le passage d'une spécification informelle à une spécification formelle est le point le moins automatisable d'un développement. Parmi les travaux visant à faciliter cette tâche, nous pouvons dégager deux démarches dont l'objectif est de faciliter cette transformation.

La première consiste à utiliser un langage spécifique pour la rédaction du cahier des charges de manière à ce que le langage naturel utilisé puisse être traduisible en spécifications formelles. Bien entendu ces travaux visent à réduire les constructions du langage naturel utilisable, car les langages naturels sont par nature des langages ambigus. Dans cette famille, on trouve les travaux de Norbert E. Fuchs et al : [Fuchs et al.95, Fuchs02]. Ces travaux permettent à partir d'un texte rédigé dans un langage appelé *Attempto Controlled English (ACE)* de générer des clauses Prolog.

La seconde démarche, plus explorée que la précédente, consiste à utiliser une notation intermédiaire entre le cahier des charges et la spécification formelle. Cette notation intermédiaire est souvent graphique pour faciliter le dialogue entre le spécifieur et le client pour qui le logiciel doit être développé. Cette démarche a pour but de traduire le cahier des charges en levant les ambiguïtés liées à l'utilisation d'un langage naturel mais également à éclaircir certains points précis parfois mal exprimés ou mal compris. Dans l'éventail des techniques et notations utilisées comme représentation intermédiaire, nous citerons les statecharts, utilisés dans [Seshia et al.] pour générer des spécifications ESTEREL ou encore dans [Sekerinski98] pour générer des spécifications B. Dans [Bon00], les réseaux de Petri sont utilisés comme représentation intermédiaire. Une autre notation utilisée est celle des diagrammes OMT utilisés dans [Facon et al.96, Meyer et al.99, Facon et al.00] pour générer des spécifications B. L'évolution naturelle de ces travaux est l'utilisation de la notation UML qui se retrouve dans différents travaux : [Meyer01, Ledang01, Levy et al.02, Laleau et al.01a, Saez et al.01].

1.2.3 Avantages et inconvénients

Nous allons dans cette section discuter du pourquoi, comment, quand, etc, utiliser les méthodes formelles. Pour cela, nous nous appuierons sur trois articles majeurs du domaine. [Hall90] de A. Hall qui discute sous forme de sept mythes les avantages et les inconvénients des méthodes formelles. Cette représentation sous forme de mythes est reprise par J.P. Bowen et M.G. Hinchey dans [Bowen et al.94] qui ont ensuite commis [Bowen et al.95] dans lequel les auteurs préconisent le respect de dix commandements pour l'utilisation de méthodes formelles. Même si pour les mythes, les articles donnent l'argumentaire permettant de classer les différentes affirmations comme des mythes, ces affirmations restent des indicateurs des points sensibles sur lesquels une attention particulière doit être portée dans le cadre de l'évaluation d'une méthode formelle et de la pertinence de son utilisation.

Les mythes de Hall dans [Hall90] sont [1] :

- **(Hall, mythe 1)** ***Formal methods can guarentee that software is perfect.***
  L'évidence de la classification de cette affirmation comme mythe repose sur deux arguments : d'une part, toutes les preuves ne peuvent être menées à terme (car le monde réel ne peut être modélisé dans son intégralité), et d'autre part, il peut exister des erreurs dans celles qui le sont (par ajout d'un théorème non vérifié par exemple). À ceci, il est important d'ajouter, comme nous l'avons vu dans la section précédente, que l'activité de développement formel ne débute qu'après une phase d'interprétation d'un cahier des charges. Cette interprétation peut être source d'erreurs : prouver une spécification se fera vis-à-vis d'un référentiel issus de l'analyse du cahier des charges, si cette analyse est fausse, c'est-à-dire non conforme à ce que le client attend, aucun système de preuve ne pourra le détecter.
- **(Hall, mythe 2)** ***Formal methods are all about proving.***
  L'activité des méthodes formelles ne se réduit pas à vérifier qu'un programme est correct. Leurs usages peuvent être bénéfiques même si elles ne sont utilisées que dans des phases en amont de la

[1] Afin de conserver le sens originel exact, la version anglaise a été conservée. Elle sera également conservée dans les mythes et commandements qui vont suivre.

production du logiciel (plus les erreurs sont découvertes tard dans le cycle de développement du logiciel, plus elles seront coûteuses à réparer). La méthode Z a été utilisée dans cette optique dans le projet CICS conduit par IBM. Seule la spécification a été faite en Z, le code a été développé de manière non formelle sur la base de ces spécifications.

- **(Hall, mythe 3)** ***Formal methods are only usefull for safety-critical systems.***
  Ce point rejoint le précédent, même sur des applications non critiques, la formalisation peut permettre de trouver des erreurs qui ne seraient parfois découvertes que lors de l'exploitation du logiciel, augmentant considérablement les coûts de maintenance.
- **(Hall, mythe 4)** ***Formal methods require highly trained mathematicians.***
  La méthode B est un exemple de méthode basée sur une théorie simple, la théorie des ensembles qui ne nécessite pas une formation particulière. Ces formalismes basés sur la théorie des ensembles sont d'ailleurs enseignés dans les premiers cycles universitaires.
- **(Hall, mythe 5)** ***Formal methods increase the cost of development.***
  Le manque de recul face à l'usage des méthodes formelles peut être une explication de ce mythe. L'investissement de départ nécessaire pour l'application de techniques formelles n'est pas négligeable et l'amortissement de cet investissement à long terme n'est pas forcément pris en compte. De plus, un contre exemple est donné par le projet CICS (voir les données dans [Houston et al.91]) : les coûts de développement ont été diminués de 9%.
- **(Hall, mythe 6)** ***Formal methods are unacceptable to users.***
  L'utilisation d'une méthode formelle facilite la compréhension d'un problème et est donc un avantage pour l'utilisateur. De plus une certaine «démocratisation» de l'usage des méthodes formelles a eu lieu, elles ne sont désormais plus considérées comme inutilisables.
- **(Hall, mythe 7)** ***Formal methods are not used on real, large-scale software.***
  Les différents projets industriels utilisant les méthodes formelles (CICS, METEOR, ...) ont montré que «le passage à l'échelle» était possible.

À ces sept mythes viennent s'ajouter ceux de Bowen et Hinchey énoncés dans [Bowen et al.94] :

- **(B & H, mythe 1)** ***Formal Methods delay the development process***
  Ce mythe est surtout dû à une mauvaise estimation des temps de développement logiciel, et ceci pas uniquement dans le cadre de l'usage de techniques formelles. De plus, des projets, comme le développement de l'unité de calcul flottant T800 de Inmos ou le projet CICS de IBM, ont montré l'inexactitude de cette affirmation.
- **(B & H, mythe 2)** ***Formal Methods are not supported by tools***
  Cette critique est de moins en moins fondée avec le développement et l'amélioration des outils support des méthodes formelles ([Owre et al.99], [Clearsy], [Coq]...).
- **(B & H, mythe 3)** ***Formal Methods mean forsaking traditionnal engineering design methods***
  Comme nous l'avons vu précédemment, les méthodes formelles peuvent tirer profit d'autres méthodes non (totalement) formelles (comme UML par exemple) pour aider à définir les besoins.

L'intégration de différentes méthodes peut d'ailleurs avoir lieu à tous les niveaux du développement.

- **(B & H, mythe 4)** ***Formal Methods only apply to software***
  Différents travaux ont montré que l'utilisation de techniques formelles ne s'applique pas uniquement au logiciel mais peut s'appliquer au développement matériel ([Warnock et al.02]) ou même au développement système (comme dans l'extension de B, le B système).
- **(B & H, mythe 5)** ***Formal Methods are not required***
  Même dans des domaines d'application où la sûreté de fonctionnement n'est pas la préoccupation majeure, l'utilisation de méthodes formelles peut être bénéfique en abaissant les coûts de production (voir le mythe 5 de Hall).
- **(B & H, mythe 6)** ***Formal Methods are not supported***
  Ce mythe rejoint le deuxième des mêmes auteurs, excepté que la critique ne reste pas ciblée sur les outils mais plus globalement sur tout ce qui entoure une méthode formelle. Avec le nombre croissant de personnes utilisant les méthodes formelles, une communauté d'utilisateurs s'est formée, souvent regroupée autour d'une méthode (comme le Bforum pour la méthode B [1]). Ceci facilite les échanges entre utilisateurs et concepteurs des méthodes.
- **(B & H, mythe 7)** ***Formal Methods people always use Formal Methods***
  Les techniques formelles ne peuvent être utilisées sur l'ensemble d'un système car cela serait trop coûteux et inutile. Elles doivent donc cohabiter avec d'autres techniques de développement.

Tous ces mythes et les argumentations menées autour d'eux sont repris et synthétisés par Bowen et Hinchey dans [Bowen et al.95] sous la forme de commandements à appliquer dans le cadre d'un développement formel. Ces commandements sont repris dans la figure 1.2. Un résumé succinct de l'argumentaire mené pourrait être : les méthodes formelles sont une des techniques intéressantes pour mener à la production de systèmes plus sûrs, mais ce n'est en aucun cas **la** technique absolue.

Parmi les commandements, quatre sont particulièrement intéressants, il s'agit du deuxième, du cinquième, du neuvième et du dixième. Le deuxième stipule qu'«appliquer des méthodes formelles à tous les aspects du logiciel est non nécessaire et coûteux». Il apparaît donc clairement que l'application de techniques formelles ne peut se faire que sur des portions d'un système (dans les cas de développements d'applications réelles). Le terme composant apparaît d'ailleurs explicitement dans ce commandement : (*Having [...] identified those* ***components*** *of the system that will benefit from a formal treatment, [...]*). Cette notion de composant se retrouve implicitement dans le dixième commandement parlant de la réutilisation, notion pour laquelle les composants sont destinés. Le cinquième et le neuvième commandements se rejoignent dans l'idée qui y est défendue : les techniques formelles peuvent être utilisées en complément d'autres techniques. Chaque technique a ses particularités et ses avantages, il paraît ainsi judicieux d'intégrer différentes approches pour tirer partie des avantages de chacune et contrecarrer leurs inconvénients.

[1] http ://www3.inrets.fr/wws/info/bforum

(1)	*Thou shalt choose an appropriate notation*	(6)	*Thou shalt document sufficiently*
(2)	*Thou shalt formalize but not overformalize*	(7)	*Thou shalt not compromise thy quality standards*
(3)	*Thou shalt estimate costs*	(8)	*Thou shalt not be dogmatic*
(4)	*Thou shalt have a formal methods guru on call*	(9)	*Thou shalt test, test, and test again*
(5)	*Thou shalt not abandon thy traditional development methods*	(10)	*Thou shalt reuse*

FIG. 1.2 – Les dix commandements de Bowen et Hinchey

À la vue des avantages et inconvénients des méthodes formelles, et en regard à notre préoccupation, l'obtention du code d'un logiciel par des techniques formelles, il nous apparaît important qu'une méthode formelle permette à la fois d'intégrer les développements effectués formellement dans un développement plus large par le biais de l'utilisation de la notion de composant, mais qu'elle permette également l'utilisation de techniques de validation issues d'autres techniques non formelles.

Dans la suite de ce chapitre, nous allons donc aborder deux autres techniques de développement logiciel, techniques qui paraissent en accord avec ces idées vues précédemment. Ces techniques sont le développement à base de composants et la programmation contractuelle.

1.3 Le développement à base de composants

1.3.1 Définitions

De nombreuses définitions du terme composant existent. Il est impossible de les donner toutes, mais nous nous efforcerons dans cette partie de donner les définitions les plus consensuelles.

La première définition naïve et très générale qu'il est possible de donner au terme composant logiciel est : un «morceau de logiciel». Cette première définition peut être complétée par celle donnée dans [Microsoft95] dans laquelle un composant est défini comme un bout de logiciel réutilisable, sous une forme binaire qui peut être connecté avec d'autres composants provenant d'autres fournisseurs «sans trop d'effort».

Clemens Szyperski définit dans [Szyperski98] un composant comme étant une unité de composition avec des interfaces contractualisées et un contexte de dépendance exprimé explicitement. Il ajoute ensuite qu'un composant peut être déployé indépendamment et qu'il est sujet à composition par une tierce personne[2].

[2]la formulation anglaise est : A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by

Dans [Crnkovic et al.02], les auteurs recommandent de fournir pour la description d'un composant :
- un ensemble d'interfaces permettant de décrire les interactions possibles avec d'autres composants ;
- du code exécutable pouvant être associé au code d'autres composants.

Il est également recommandé, afin d'améliorer la qualité des composants, de fournir :
- une spécification des caractéristiques non-fonctionnelles requises et fournies ;
- un code permettant de valider la connection d'un composant avec un autre ;
- toutes informations complémentaires comme les cas d'utilisations, comment les besoins (exprimés dans les spécifications) ont été pris en charges, ...

Dans ces définitions se trouvent les notions principales servant à caractériser ce qu'est un composant. La première, qui correspond au principal objectif de l'utilisation de composants, est la notion de *réutilisation*. La deuxième notion importante est la notion d'*interaction* entre composants, ce qui est également liée à la notion de tierce personne qui va utiliser les composants. Le troisième point à noter est plus une caractéristique de l'entité composant elle-même plutôt qu'une notion utilisée lors de développement à base de composants, *un composant est-il une entité binaire* ? Nous commencerons par traiter le problème du format des composants pour ensuite aborder et présenter les autres notions : la réutilisabilité qui comme nous le verrons sera liée à une notion d'adaptabilité puis les interactions entre composants et plus particulièrement la notion d'interface de composant.

Des composants binaires ou non ?

Dans [Meyer99], B. Meyer distingue deux visions des composants : une restrictive et une plus large. La vision plus restrictive est celle donnée par C. Szyperski dans son livre *Component Software - Beyond Object-Oriented Programming* ([Szyperski98]). Cette vision considère un composant comme une entité logicielle *binaire*. Cette notion de *binaire* est précisée dans [Szyperski00] comme signifiant exécutable par une machine. La seconde vision plus large est celle défendue par Meyer dans laquelle un composant peut être vu comme n'importe quelle entité logicielle (binaire, paquetage, classe, ...).

Dans ce mémoire, nous considérerons la vision la plus large. En effet, du point de vue conception de composants, le passage de la vision large à la vision plus restrictive ne posera pas de problèmes, cela ne correspond qu'à une étape supplémentaire de développement qui sortira du cadre formel (utilisation de compilateur).

La réutilisabilité

Le développement à base de composants est basé sur un concept fondamental : la réutilisabilité. Ce concept repose sur des observations de domaines extérieurs au logiciel. Pourquoi la construction de nouveaux circuits numériques repose sur des composants existants, pourquoi la construction automobile repose sur l'assemblage de diverses parties provenant d'origines aussi diverses alors que pour le logiciel, tout repart de zéro à chaque fois ? Cette notion de réutilisation de produit existant n'est pas nouvelle, c'est M. Douglas McIlroy dans [Mcilroy68] qui pour la première fois (ou du moins l'une des premières fois)

third party

lors de la conférence sur le génie logiciel en 1968 a émis l'idée de la construction de logiciel reposant sur des plus petites briques : des composants logiciels.

Meyer énonce six points, dans [Meyer00, chapitre 4], améliorés par la réutilisation :

- **la ponctualité** en ayant la capacité de produire des logiciels rapidement ;
- **un effort de maintenance diminué** en déportant la maintenance des composants aux fournisseurs de ces composants ;
- **la fiabilité** en utilisant des composants provenant de sources réputées et particulièrement compétentes pour le développement de tel composant spécifique. Le second argument en faveur de la fiabilité est qu'un composant aura été testé par d'autres utilisateurs ;
- **l'efficacité** en laissant des spécialistes du domaine développer des composants de ce domaine (utilisation du meilleur algorithme pour effectuer une tâche) ;
- **la cohérence** car développer à base de composants permet d'aborder le développement d'une manière uniforme, et
- **l'investissement** en «amortissant» le développement d'un composant sur plusieurs projets.

Sans reprendre tout l'argumentaire développé dans [Meyer00], l'idée générale est que la réutilisation repose sur la décomposition du travail. Ainsi, en ce qui concerne la ponctualité, c'est-à-dire la capacité à fournir une application dans des délais réduits, si une partie du travail est déjà effectuée dans certains composants, le volume logiciel à développer s'en trouve réduit et donc les temps de développements et les potentiels dépassements de délai le seront également. En ce qui concerne la maintenance, l'apport de la réutilisation est de reporter la maintenance d'un composant au fournisseur de ce composant. Cette notion de déportation des tâches de développement est également un atout du point de vue fiabilité et efficacité. En déportant le travail vers des «experts» du domaine, vous tirez partie de leurs connaissances métiers. De plus, développer du logiciel avec comme objectif sa réutilisation permet d'amortir son coup de développement. Ainsi, les techniques de mise au point ou d'optimisation du composant pourront être plus poussées et plus coûteuses sans que cela n'entame la rentabilité du développement.

Adaptabilité et extensibilité

La réutilisabilité comme nous l'avons présentée précédemment est à elle seule une caractéristique importante mais pas suffisante pour le développement à base de composants. Les notions complémentaires sont l'adaptabilité et l'extensibilité. Celles-ci feront qu'un composant développé dans un certain environnement, en utilisant certaines particularités puisse être adapté à d'autres besoins. Cette nécessité est également exprimée dans [Crnkovic et al.02] comme une caractéristique nécessaire à la réutilisation.

Pour répondre à ces aspects complémentaires, Meyer donne cinq exigences sur les structures de modules. Dans la vision de Meyer, la notion de module est à prendre dans son sens le plus abstrait, comme une *«unité de décomposition du logiciel»* (voir glossaire de [Meyer00]). Ce concept de décomposition sera instancié dans chaque langage par telle ou telle technique particulière (classes pour le langage Java, paquetage pour Ada, module pour les langages ML,...). Ces exigences servent à caractériser les structures de modules qui pourront être compatibles avec l'objectif de réutilisation et d'adaptation. Un module qui suivra les exigences suivantes sera un bon candidat à la formation d'un composant. Les exigences sont :

- **Variation de type.** Un composant, comme une pile par exemple, devra être capable de s'adapter

à différents types de données (comme des entiers, des réels et des structures plus complexes qui pourront être stockées dans la pile).

- **Regroupement de routines.** Intrinsèquement, la notion d'ajout d'un élément dans une pile est intimement liée à la notion d'accès ainsi qu'à la notion de retrait. Une structure de module devra donc permettre le regroupement de ces diverses opérations au sein d'une même entité.
- **Variation d'implémentation.** Pour résoudre un problème, il est possible de choisir différentes structures de données ainsi que différents algorithmes. Un seul composant ne pourra prendre en compte toutes les possibilités. Meyer introduit la notion de famille de modules pour cette exigence.
- **Indépendance de représentation.** Cette exigence vise à permettre à un client d'un composant de s'abstraire de la représentation concrète qui a été faite lors de l'implantation du composant. Cette exigence permet d'introduire la notion de rétention d'informations consistant à cacher certains aspects comme les structures de données utilisées. Ceci permet ainsi de changer cette structure sans que cela ait le moindre effet du côté du client du composant.
- **Factorisation des comportements communs.** Toutes les similarités dans une famille de modules doivent être mises en commun dans un souci de réutilisabilité.

Interactions entre composants

La construction de logiciels basés sur des composants suppose que ces composants interagissent entre eux. Ces interactions peuvent prendre différents aspects, comme la simple utilisation des fonctionnalités ou la composition avec d'autres composants ou encore l'extension d'autres composants. Toutes ces interactions dépendent du modèle de composant utilisé. Plusieurs modèles de composant existent actuellement, citons les plus utilisés comme JavaBeans ([Microsystemsb], Enterprise JavaBeans ([Microsystemsa]), CORBA ([Omga]), CORBA Component Model (CCM) ([Omgb]), COM ([Microsofta]) ou encore .NET ([Microsoftb]).

Nous n'allons pas étudier en détails ces différentes architectures, mais un des aspects essentiels est la notion d'interface de composants. L'interface d'un composant est la donnée permettant de caractériser un composant en spécifiant comment l'utiliser. En effet, un composant est censé agir comme une boîte noire[3], c'est-à-dire comme une entité rendant certains services sans pour autant divulguer la manière dont ce service est rendu (rétention d'informations vue précédemment), cette information n'étant pas nécessaire aux utilisateurs du composant. De plus, cacher les détails de l'implantation permet de mettre à jour un composant sans que cela n'affecte les applications en place utilisant déjà ce composant.

Les informations primordiales données dans l'interface d'un composant sont :

- le nom des opérations (des services) fournies,
- leurs paramètres et
- les types acceptés pour ces paramètres.

À ces informations viennent s'en ajouter d'autres qui se retrouvent dans différents modèles de composants. Dans CCM, par exemple, existe la notion de facettes, des interfaces différentes fournies par le composant que le client utilisera pour interagir, ou de réceptacles, points de connections avec un autre

[3]Il est également possible d'utiliser des composants «boîtes blanches», mais ceux-ci posent des problèmes car l'utilisation de tels composants crée une dépendance entre l'implantation du composant et l'application qui l'utilise.

composant. Les autres informations qui peuvent également être pertinentes dans un composant sont les contrats, nous reviendrons sur cet aspect dans la partie 1.4.

1.3.2 Avantages et inconvénients

Les avantages des développements à base de composants ont été exposés dans les définitions que nous avons présentées dans la section précédente. Ces avantages sont la réutilisabilité, l'adaptabilité et l'extensibilité qui permettent une diminution des coûts de développement et une augmentation de la qualité de ces développements.

Dans le chapitre introductif de [Crnkovic et al.02], Ivica Crnkovic et Magnus Larsson énoncent cinq inconvénients/risques concernant les développements à base de composants :

- **Les temps initiaux de développement et les efforts nécessaires sont accrus.** Comme pour les méthodes formelles, cet accroissement des coûts à court terme mènera néanmoins à leur diminution à plus long terme lorsqu'une politique de réutilisation aura réellement été mise en place au sein de l'entreprise de développement.
- **Les exigences sont souvent peu claires et ambiguës.** La gestion des exigences est déjà un défi en soit, mais ajouter à cela le fait que par nature un composant est destiné à être réutilisé dans diverses applications dont certaines sont encore inconnues, ajoute à la complexité.
- **L'utilisation et la réutilisation sont des notions antagonistes.** En effet, pour qu'un composant soit réutilisable, il faut qu'il soit assez général et adaptable, ce qui le rendra plus compliqué à utiliser.
- **Les coûts de maintenance des composants sont plus élevés.** Ceci est relié à la notion de gestion des exigences vue précédemment : comme un composant intervient dans divers projets/produits, après modification d'un composant, il faut vérifier sa conformité avec toutes les exigences.
- **La fiabilité des applications est sensible aux changements.** Ce point rejoint le deuxième vu précédemment, mais cette fois-ci, du côté de l'application. Tout changement au niveau de l'application, comme le changement d'autres composants, peut avoir un impact sur la fiabilité d'un composant par une méconnaissance de toutes les caractéristiques du composant.

Les avantages énoncés précédemment confirment l'intérêt d'une approche «hybride» formelle et composant que nous avions énoncée dans la section précédente. En effet, le développement de composants permet d'amortir les coûts de développement par réutilisation du produit obtenu. Cet amortissement est un avantage nouveau dans le cadre de développement formel car la réutilisation, même si elle existe dans les méthodes formelles, n'est pas «poussée» au niveau de ce qu'il est possible de faire avec des composants. Les cinq inconvénients apportent une réciprocité à l'intérêt de l'hybridation. En effet, une préoccupation forte ressort de ces inconvénients : la mise en place de techniques à base de composants nécessite une attention particulière sur la qualité des composants. L'usage de techniques formelles est un moyen d'améliorer la qualité.

1.4 Le développement par contrats

1.4.1 Définitions

Le développement par contrats encore appelé développement contractuel est une technique de développement logiciel qui s'est développée courant des années 90. C'est surtout Bertrand Meyer qui a été un moteur pour son développement. Le principe général de cette technique de développement est basé sur la notion de contrat entre celui qui utilise le code développé et celui qui le développe : le client doit satisfaire certaines conditions pour utiliser les fonctionnalités du logiciel et sous ces conditions, le fournisseur du logiciel assure que certaines autres conditions seront remplies après l'exécution. Le moyen d'exprimer ces conditions ou contrat est l'utilisation d'assertions.

Une assertion est une formule logique, un prédicat, permettant comme le mentionne Bertrand Meyer dans son livre [Meyer00], d'exprimer ce que le logiciel doit faire alors que le code exprime comment il doit le faire. Les assertions correspondent donc à la spécification de l'implantation. Les informations ajoutées ont pour objectif d'augmenter la qualité du logiciel en :

- aidant à la production d'un code plus «intègre» et plus «robuste» ;
- servant de support à la documentation du code ;
- fournissant un support pour le test du logiciel et
- servant à la gestion des exceptions.

Un exemple d'utilisation des contrats

Le meilleur moyen de présenter les principes et les usages des assertions est de prendre un exemple illustratif. Pour cet exemple, nous allons considérer l'écriture d'une pile bornée, c'est-à-dire une structure «dernier entré premier sorti» dont le nombre d'éléments est limité. Nous utiliserons le langage Eiffel ([Meyer92]) comme langage support pour l'écriture du code.

Les assertions que nous allons rencontrées dans l'exemple sont :

- la pré-condition,
- la post-condition et
- l'invariant

En plus de ces assertions, il est également possible de vérifier à un endroit quelconque d'un programme qu'une propriété est vérifiée, nous appellerons cette construction la vérification ponctuelle.

En Eiffel, les pré-conditions sont repérées par le mot clef `require`, les post-conditions par `ensure` et l'invariant par le mot clef `invariant`.

Les principaux services qu'un objet pile (les termes que nous emploierons, bien qu'issus des langages orientés objets, ne dénotent en rien une quelconque obligation de l'utilisation de tels langages pour la programmation par contrat) doit fournir sont : les méthodes d'interrogation sur l'état de la pile (est-elle pleine ou vide, combien d'éléments la constituent...) et les méthodes de manipulation de la pile (ajout et retrait d'éléments).

Une pile peut être représentée par un tableau, un indicateur de sa taille et l'indice désignant le haut de la pile :

la pile

```
stack_size : INTEGER                    -- taille de la pile
stack_top : INTEGER                     -- indice de haut de pile
the_stack : ARRAY[INTEGER]              -- représentation de la pile
```

Les deux principales fonctionnalités de manipulation d'une pile sont l'ajout d'une valeur (la méthode `push`) et le retrait d'une valeur (la méthode `pop`). Le code correspondant à la méthode d'ajout est le suivant :

la méthode push

```
push ( addval : INTEGER ) is            -- Push 'addval' on top of the stack
        require
                not is_full
        do
                stack_top := stack_top + 1
                the_stack.put(addval,stack_top)
        ensure
                count = old count + 1
                top = addval
        end
```

Le contrat de cette méthode stipule que le client, avant d'utiliser la méthode, doit s'assurer que la pile n'est pas pleine. Une fois ce contrat rempli, la méthode assure que le nombre d'éléments de la pile (dénoté par l'entité `count`) est incrémenté d'une unité (`count = old count + 1`) et que le sommet de pile (dénoté par l'entité `top`) est bien l'élément passé en paramètre de la méthode (`top = addval`). Une notion apparaît dans cet exemple, c'est la notion de valeur précédente. En effet, il est dans certains cas nécessaire d'exprimer les propriétés en fonction de la valeur précédente (la valeur d'une entité avant l'appel d'une méthode).

En plus des notions de pré et post conditions, il existe également une notion d'invariant. Pour notre exemple, l'invariant est le suivant :

l'invariant

```
invariant
     is_empty = (count = 0)
     is_empty implies not is_full
     is_full implies not is_empty
```

La première ligne de cet invariant exprime la relation entre le résultat de la méthode `is_empty` et la valeur nulle de `count`. Les deuxième et troisième lignes expriment les implications existantes entre `is_empty` et `is_full`. L'invariant peut être considéré comme un contrat à la fois du côté client et du côté fournisseur. Ainsi, dans cette vision, si `P` désigne la pré-condition d'une méthode, `Q` la post-condition et `I` l'invariant, l'habituelle vision en triplets de Hoare $\{P\}\ C\ \{Q\}$ peut se voir comme $\{P \wedge I\}\ C\ \{Q \wedge I\}$.

Bien que l'invariant fasse partie du contrat du côté client, des mécanismes d'encapsulations (s'ils

existent dans le langage) permettront d'interdire la modification directe de la pile et obligeront l'utilisation des méthodes définies dans la classe pile pour la modifier. Ceci assurera la validité du contrat du côté client sans surcoût, par contre, du côté fournisseur, il reste à sa charge de vérifier que toutes ces méthodes ne violent pas l'invariant.

1.4.2 Avantages et inconvénients

Les avantages des approches à base de contrats sont semblables aux avantages des approches formelles. En effet, les assertions contribuent à l'amélioration du logiciel de différentes manières. La simple formulation des propriétés apporte déjà une amélioration. En exprimant par une formule logique ce que le logiciel doit faire, un premier pas est fait vers la compréhension du problème, une fois ce pas franchi, l'élaboration du logiciel est facilitée et son adéquation avec le produit attendu est accrue.

La seconde amélioration vient de l'utilisation des assertions comme un support à la validation du logiciel. Cette validation, qui est ici à considérer dans son sens le plus large, c'est-à-dire toutes techniques permettant de trouver des erreurs dans le logiciel ou de prouver sa correction, repose sur deux grandes familles de techniques. La première de ces techniques est l'emploi d'analyses statiques pour vérifier la validité des contrats, la seconde est l'emploi de techniques dynamiques consistant à vérifier les propriétés au moment de l'exécution, soit dans une phase de mise au point, soit même durant l'exploitation du logiciel.

L'emploi d'analyse statique à partir du code annoté peut prendre différents aspects suivant le niveau d'expression du langage des annotations. La figure 1.3, extraite de [Leino01], reprend la hiérarchie des techniques suivant leur rapport investissement nécessaire/quantité d'erreurs trouvées. Parmi les ancêtres de cette technique, nous pouvons citer LCLint ([Evans et al.94]) qui a ensuite donné lieu à Splint ([Evans et al.02]) qui concerne la vérification de programme en langage C. Ces outils ne considèrent pas en entrée un langage des assertions du type du langage des prédicats mathématiques. L'objectif de ces outils est de trouver les erreurs simples telles que les violations des règles de visibilité, l'accès hors des bornes d'une structure (de tableaux par exemple)...

Dans le cadre de la programmation par contrats et de l'usage des assertions, les outils de vérification des contrats sont particulièrement intéressants. Ces outils suivent un principe assez simple : du code annoté sont extraites des formules logiques qu'un prouveur de théorème tente ensuite d'établir ou de réfuter suivant la tactique utilisée. Dans cette famille d'outils, parmi les travaux les plus avancés et les plus anciens figurent ceux menés au sein du «Compaq's Systems Research Center». Ces travaux portant sur l'extension des vérifications statiques («extended static checking», ESC) ont porté dans un premier temps sur le langage Modula-3 ([Detlefs96]) pour ensuite s'intéresser au langage Java ([Flanagan et al.02]). Un prouveur de théorème, Simplify, est fourni avec les systèmes ESC/Modula-3 et ESC/Java. Une approche semblable est menée dans l'outil Why ([Filliatre03]) développé par J.C. Filliâtre. La différence avec l'approche ESC est qu'aucun prouveur n'a été spécifiquement développé car l'outil permet de se «connecter» à différents prouveurs externes comme Coq ([Coq]), PVS ([Owre et al.99]) ou HOL Light ([Harrison]).

La seconde approche permettant d'améliorer la qualité du logiciel est une approche dynamique. Cette approche consiste à «observer» l'exécution du logiciel. Tester à l'exécution est la méthode de validation originelle du développement par contrats développé pour Eiffel. Le principe consiste à vérifier que les

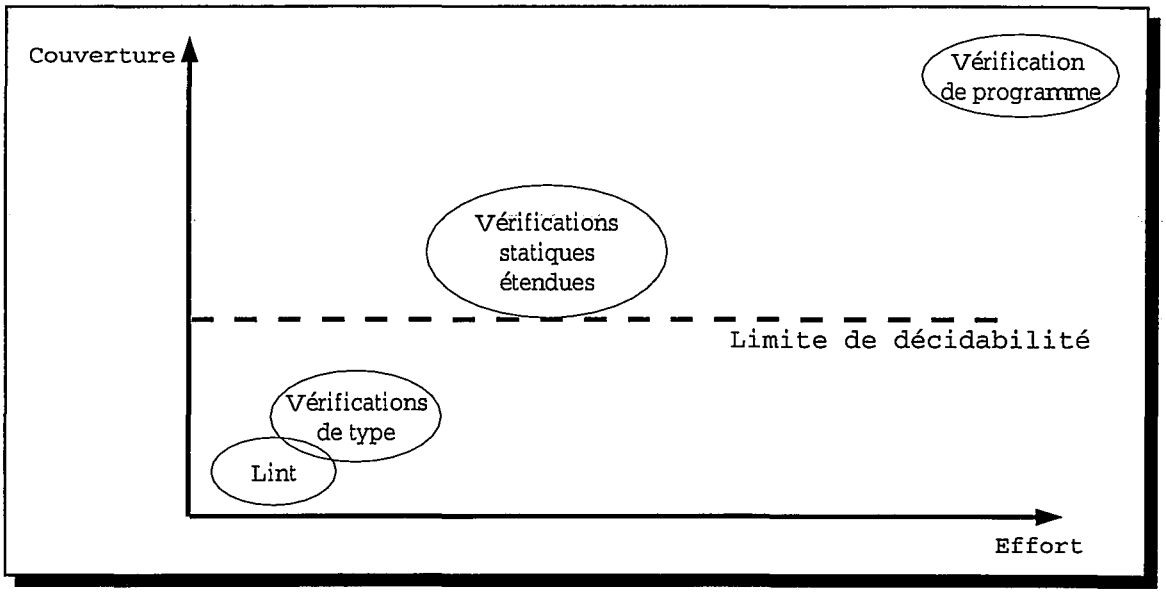


FIG. 1.3 – Rapport investissement nécessaire/quantité d'erreurs trouvées suivant les techniques utilisées

propriétés exprimées par les assertions sont bien établies au cours de l'exécution. Pour cela, si le langage, et surtout le compilateur associé intègre la notion d'assertion, aucune modification n'est nécessaire (c'est le cas de Eiffel par exemple), par contre, si le compilateur ne prend pas en compte les constructions d'assertions, il est nécessaire de passer par un pré-compilateur qui va «équiper» le code des instructions correspondant aux assertions.

Comme pour les avantages, les inconvénients des approches à base de contrats sont semblables à ceux des approches formelles. Les principales difficultés sont l'expression des contrats et leur validation vis-à-vis des exigences. Construire le bon contrat pour une fonctionnalité donnée peut être difficile.

1.4.3 Des composants contractualisés

Le principal objectif du développement à base de composants est comme nous l'avons énoncé précédemment la réutilisation. Or, réutiliser un composant logiciel nécessite quelques précautions. Cette idée est exposée dans l'article de Jean-Marc Jézéquel et Bertrand Meyer sur l'explosion du lanceur Ariane 501 ([Jezequel et al.97]). Le rapport de la commission d'enquête ([Ariane96]) identifie clairement la cause de l'accident : la réutilisation du système de référence inertielle (SRI) d'Ariane 4 pour Ariane 5. En effet, le SRI d'Ariane 4 avait été validé sous l'hypothèse que la vitesse horizontale ne dépassait pas un certain seuil. Or avec Ariane 5, du fait de trajectoires différentes, la vitesse horizontale a dépassé ce seuil et a mené le logiciel à mettre le système hors service. La critique principale de [Jezequel et al.97] est l'absence de spécification dans le module du logiciel réutilisé. La spécification de l'hypothèse de la limite de vitesse aurait pu être détectée lors des phases de tests du logiciel ou de relecture critique du code.

Le problème de la construction de composant logiciel réutilisable sûr est encore un problème ouvert. [Crnkovic et al.02] explore plusieurs techniques/approches utilisables pour y arriver, mais un des aspects importants qui ressort de cet ouvrage est la nécessité d'équiper les composants de contrats pour spéci-

fier les conditions d'utilisation et donc de réutilisation du composant ([Luders et al.02]). Ces contrats pourront être utilisés pour valider la composition de tel composant avec tel autre, en vérifiant que les composants respectent mutuellement leurs contrats. Pour un composant validé par son fournisseur, deux types d'erreurs peuvent survenir lors de son utilisation. La première vient de son environnement d'exécution qui peut différer de l'environnement dans lequel le composant aura été validé. La seconde source d'erreurs est la non adéquation du composant avec ce que le client en attend.

L'expression des propriétés dans les pré et post-conditions ainsi que dans l'invariant du composant est une manière de vérifier, d'une part, que le composant est utilisé dans un environnement adéquat, et d'autre part, que les services rendus par ce composant correspondent bien à ce qui est attendu par le client.

1.4.4 Quelques langages avec support d'assertions

Nous ne pouvons faire une liste exhaustive des langages avec support d'assertions tant le développement de l'approche contractuelle se développe et avec elle les développements visant à ajouter des assertions dans les langages existants. Nous nous limiterons à donner quelques exemples de langages supportant l'approche par contrats pour montrer que certaines différences existent soit dans le langage même des assertions soit dans les outils supports.

Les langages auxquels nous allons nous intéresser sont les langages Eiffel, Java, Ada et OCaml. Eiffel car ce langage est pionnier en termes de développement par contrats et intègre une préoccupation composant. Le langage Java est, quant à lui, actuellement très utilisé et bénéficie d'une large communauté dynamique, preuve en est le nombre d'outils qui ont été développés aussi bien pour le support des assertions que pour le développement à base de composants. Nous distinguerons le langage Java dans sa version 2 ([Java2]) mais également les extensions Jass ([Bartetzko et al.01]) et ESC ([Detlefs et al.98]) pour ce même langage. Un autre langage que nous allons considérer est le langage Ada ([Ada94]) et plus particulièrement le sous-langage Spark Ada ([Barnes03]). Ce langage est un langage intéressant du point de vue du mécanisme de composition utilisé, les paquetages. Un autre aspect qui fait que l'étude de ce langage est important est son utilisation dans le domaine d'application qui nous intéresse, le transport ferroviaire. Le dernier langage est OCaml, ce langage a la particularité d'être un langage offrant la possibilité d'utiliser à la fois des modules au sens SML ([Harper et al.86]) mais également les constructions objets.

Le tableau 1.1 reprend la présence ou non des différentes caractéristiques liées au support d'assertion des différents langages. Ces caractéristiques correspondent aux possibilités d'exprimer :

- des pré et post conditions (ligne *Pré et Post*) ;
- des propriétés ponctuelles, c'est-à-dire à n'importe quel endroit du code source (ligne *Assert*) ;
- un invariant de classe, de module, de paquetage... suivant la technique de structuration utilisée dans le langage (ligne *Invariant*) ;
- des invariants et variants de boucle (ligne *Variant/Invariant de boucle*) ;
- des relations entre valeur avant appel d'une méthode, d'une fonction... et valeur après cet appel (ligne *valeur avant*) ;
- des assertions quantifiées universellement ou existentiellement (ligne *quantificateurs*).

Remarques. La différenciation que nous faisons en séparant d'un côté les pré et post-conditions et de l'autre les assertions sert à refléter les possibilités des différents compilateurs, outils supports aux langages avec contrats. Même si les constructions de pré et post-conditions sont sémantiquement équivalentes à la construction assert qu'il est possible de rencontrer dans certains langages, leur usage peut être différent. En effet, la différenciation permet par de simples options de compilation (dans le cas de Eiffel par exemple) de ne prendre en compte que certaines catégories (soit uniquement les pré-conditions, soit les pré et post-conditions sans les assertions, etc).

Sur le même principe, l'impossibilité d'utiliser les quantificateurs pour l'expression des contrats peut être contournée par l'appel d'opérations (fonctions, procédures, méthodes,...) implantant un parcours exhaustif des valeurs possibles. Par exemple, $\forall x \in [2..10]$ s'implante facilement par une fonction récursive, une boucle «tant que», etc, solution permettant de vérifier dynamiquement la propriété quantifiée. Par contre, dans le cas où l'ensemble des valeurs est infini, il est impossible de faire un parcours des valeurs possibles. Dans ce cas, seule une solution statique est envisageable. En fait, dans un souci d'efficacité, les parcours d'ensembles finis mais de cardinalités élevées seront également exclus des traitements dynamiques et un traitement statique sera privilégié. Le traitement statique correspond à l'utilisation d'un outil de preuve comme Coq, HOL, PVS,... déjà cités précédemment.

La dernière ligne du tableau exprime le mode de fonctionnement pour l'utilisation des assertions. Parmi ces modes de fonctionnement, nous en avons recensé trois qui sont, premièrement, un fonctionnement en mode pré compilateur, c'est-à-dire avant d'être compilés, les codes sources sont modifiés pour transformer les expressions d'assertions en structures de contrôle du langage. Le second mode de fonctionnement est le mode qui est dénoté *intégré* dans le tableau. Ce mode est utilisé quand la prise en compte des assertions est directement faite par le compilateur. Le troisième mode de fonctionnement que nous avons différencié a été appelé *outils externes*. Ce mode correspond, comme son nom l'indique, à une prise en compte des assertions dans des outils externes n'ayant aucun lien avec le compilateur. L'utilisation d'outils externes correspond à l'approche statique alors que la pré compilation ou la gestion intégrée des assertions dans le compilateur correspond à l'approche dynamique.

	Eiffel	Java	Java/Jass	Java/ESC	Spark Ada	OCaml
Pré et Post	✓		✓	✓	✓	
Assert	✓	✓	✓	✓	✓	✓
Invariant	✓		✓	✓	✓	
Variant/Invariant de boucle	✓		✓	✓	✓	
valeur avant	✓		✓	✓	✓	
quantificateurs			✓	✓	✓	
Fonctionnement	intégré	intégré	pré-compilé	outils externes	outils externes	intégré

TAB. 1.1 – Caractéristiques de quatres langages avec support d'assertions

Les caractéristiques qui ont été mises en avant dans le tableau 1.1 ne sont pas les seules caractéristiques à prendre en compte pour une comparaison des différents langages. L'objectif n'est pas ici de dire que tel langage est meilleur que tel autre, mais de lister les possibilités offertes de base par le langage.

Ainsi, comme nous l'avons signalé en remarque, l'absence de pré et post conditions dans un langage n'est pas bloquant, car si la construction *Assert* existe, il suffit de mettre en début d'appel de méthode un *Assert* et pour la post condition de la mettre en fin de méthode. Mais l'absence de différenciation implique une gestion moins fine des mécanismes d'assertions.

1.5 Conclusions

Dans ce chapitre, nous avons présenté les principes généraux de trois méthodes de développement de logiciel. Chacune de ces méthodes présente des avantages et des inconvénients. Les méthodes formelles sont parfois «lourdes» à mettre en pratique et ne s'intègrent pas forcément idéalement dans un processus de développement de logiciel développé au sein d'une entreprise. Par contre, l'approche composant semble elle être très bien acceptée dans le monde industriel mais son utilisation nécessite une attention particulière sur la qualité des produits développés. Cette qualité peut être améliorée par les approches formelles mais également par l'approche contractuelle. Cette dernière, facile à mettre en œuvre, repose sur l'expression des propriétés du produit final désiré sous forme de contrats. Malgré cette apparente facilité, comment trouver les contrats les plus judicieux ? Certains travaux comme [Nimmer et al.01] ou [Arnout02] vise à trouver *a posteriori* les contrats, mais l'utilisation de méthodes formelles permettrait de «construire» ces contrats plus facilement et en plus de prouver que le logiciel les respecte bien.

Ces trois approches qui dans un premier temps peuvent paraître indépendantes sont en fait particulièrement complémentaires. Comme nous l'avons vu en section 1.2, la notion de composant est implicite lorsque l'on parle de développement formel. Et cette notion de composant ne peut raisonnablement pas s'envisager sans mécanismes de spécifications des fonctionnalités de ces composants, spécifications qui peuvent être en partie obtenues par l'expression de contrat. Outre le rapprochement, ou plutôt l'entrelacement des concepts formels, composant et contrats, chaque technique possède ses mécanismes et ses outils de validation. Intégrer dans un processus de développement chacune de ces techniques semble être un moyen d'accroître la qualité des produits logiciels mais également de faire décroître les coûts et les temps de développement.

Ces conclusions sur l'apport d'une «hybridation» des trois approches vont nous amener dans la suite à garder à l'esprit un certain nombre de préoccupations :

- la génération de code à partir de spécifications formelles ne doit pas oublier les contrats présents dans ces spécifications,
- la notion de composant est naturelle dans les approches formelles et
- toutes les techniques (notamment celles utilisées dans les approches contractuelles) facilitant la validation (du code ou des spécifications) permettent d'améliorer la productivité.

Chapitre 2

La méthode B

Sommaire

2.1 Introduction

Dans ce chapitre, nous allons présenter la méthode de développement formel B ainsi que le langage B associé. Nous présenterons la méthode en trois parties : nous aborderons tout d'abord les aspects «basiques»[1] du langage B en section 2.2 qui nous permettront de comprendre les entités manipulées lors de l'écriture de spécifications B mais également comment celles-ci sont manipulées. Ensuite, nous aborderons l'aspect composition de spécifications en partie 2.3. Cela nous permettra de comprendre comment sont décomposées les spécifications et dans quel but. Nous présenterons ensuite (section 2.4) la méthode B elle-même, c'est-à-dire le déroulement du processus de développement des spécifications et les différentes activités menant à la production d'un logiciel sûr. L'obtention de ce logiciel sûr, le code, sera abordée dans la section suivante (2.5). Dans cette partie, nous présenterons ce qui nous semble important pour la méthode B du point de vue de la génération de code. Enfin, nous nous attacherons à étudier un algorithme de dépliage de spécifications B. Nous montrerons les difficultés qui ont été rencontrées et les enseignements que nous avons tirés de l'implantation de cet algorithme.

2.2 Fondements de la méthode *B*

2.2.1 Présentation générale

La méthode B est née dans les années 80. Le livre référence de la méthode est le *B Book* ([Abrial96]) écrit par Jean-Raymond Abrial, le fondateur de la méthode. Les similitudes entre la méthode B et les méthodes/notations Z et VDM s'expliquent par le fait que Jean-Raymond Abrial ait participé à l'élaboration de ces méthodes avant de se consacrer à B. Un historique plus complet de la méthode B est mené dans [Mariano97].

La méthode B a bénéficié du soutien de l'industrie ferroviaire française qui a permis le «passage à l'échelle» de l'utilisation de la méthode. L'exemple industriel le plus abouti de l'utilisation de la méthode B concerne le pilote automatique embarqué (PAE) de METEOR de la ligne 14 du métro parisien ([Behm96] et [Behm et al.]).

Lors du développement de la méthode, deux outils ont été mis en œuvre : l'Atelier B (développé par Stéria Meditérannée maintenant CLEARSY[2]) et le B-Toolkit (développé par B-Core[3]). Ces outils permettent de développer formellement un logiciel, de sa spécification abstraite à sa forme concrète qui sera utilisée pour générer le code correspondant, le tout en assistant le développeur (que nous appellerons également le spécifieur) dans les phases de preuves qui doivent être menées pour établir la sûreté du développement. Bien qu'ayant une base commune, ces outils ont divergé légèrement au cours du temps et le B «accepté» par les deux outils n'est pas exactement le même entre la «version anglaise» et la «version française». Le B de la version anglaise sera qualifié d'originel alors que celui de la version française sera qualifié d'évolué. Ceci sans aucune connotation péjorative pour l'un ou pour l'autre. Cette différence s'explique justement par l'utilisation industrielle qui a eu lieu en même temps que le développement de l'outil français, qui a amené certaines évolutions de la méthode et des notations pour satisfaire les

[1] L'emploi du terme basique est lié à la terminologie qui sera utilisée dans le chapitre suivant sur la modularité.
[2] http ://www.atelierb.societe.com/
[3] http ://www.b-core.com/btoolkit.html

besoins industriels. Dans ce mémoire, nous considérerons principalement la version «française» du B. Pour considérer la version anglaise, il suffira d'ajouter les restrictions inhérentes à la différence de vision de la notation et de la méthode. De plus ces restrictions n'affectent que peu la génération de code.

En plus de ces outils commercialisés, d'autres outils «libres»[4] apparaissent depuis quelques temps. Ces outils sont désormais regroupés au sein de l'effort commun intitulé BRILLANT ([Brillant]). Ces outils «académiques» ont pour objectifs de fournir des plateformes d'expérimentations pour la méthode B. Les développements qui ont été menés dans les travaux présentés dans ce mémoire s'intègrent d'ailleurs dans la partie BCaml du projet.

2.2.2 La notation en machine abstraite

La notation en machine abstraite, plus connue sous l'acronyme AMN (*Abstract Machine Notation*), est la notation (uniformisée) utilisée à tous les niveaux de spécifications de la méthode B. Cette uniformisation de différents niveaux d'abstraction se rapproche des *Abstract State Machines (ASMs)* [Gurevich99]. La notion d'état est également partagée avec les ASMs et sera présentée en section 2.2.3. Ensuite, les différents niveaux de spécifications seront abordés en 2.2.4. Les substitutions généralisées qui définissent comment modifier l'état d'une spécification seront présentées en section 2.2.5.

2.2.3 L'état d'une spécification B

Les données

L'état d'une spécification B est une caractérisation de cette spécification. L'état fait référence à diverses entités : des ensembles, des variables et des constantes ainsi que des propriétés sur ces entités. Deux familles sont distinguées pour ces entités : les données dites *abstraites* et celles dites *concrètes*. Les données abstraites sont les données qui vont évoluer au cours du développement (voir le raffinement en section 2.2.4) alors que les données concrètes n'évolueront pas. Une spécification B s'exprime par l'utilisation de clauses. Les clauses servant à spécifier l'état d'une spécification sont :

- `ABSTRACT_VARIABLES` (ou `VARIABLES`), `CONCRETE_VARIABLES`, `CONCRETE_CONSTANTS` (ou `CONSTANTS`) et `ABSTRACT_CONSTANTS` qui définissent les données d'une spécification ;
- `SETS` qui définit des ensembles, soit abstraits soit énumérés qui seront utilisés pour typer les variables et les constantes ;
- `INVARIANT` qui permet de typer les variables (abstraites et concrètes) et d'exprimer les propriétés de ces variables ;
- `ASSERTIONS` qui permet d'exprimer des propriétés déduites de l'invariant, propriétés destinées à faciliter la preuve ;
- `PROPERTIES` qui permet de typer les constantes (abstraites et concrètes) et d'exprimer les propriétés de ces constantes ;
- `VALUES` qui permet de donner les valeurs aux constantes concrètes et aux ensembles abstraits.

[4] La notion de libre dans le logiciel est encore assez floue pour une majorité de personne, nous l'employons ici abusivement sous la signification : dont les sources sont librement récupérables et modifiables

Le typage des données

Le typage des données d'une spécification B repose sur l'appartenance à un ensemble. Cette appartenance peut être exprimée soit dans un prédicat de typage soit dans une substitution de typage.

Dans le premier cas (le prédicat de typage), l'information de typage est donnée explicitement. Par exemple pour spécifier qu'une variable *vv* est de type entier, dans l'invariant, l'information $vv \in NAT$ sera présente. Le deuxième cas de typage ne correspond pas à une information explicitement donnée dans la spécification, mais plutôt à une inférence de type simple. L'information de typage est déduite d'une «affectation» d'une valeur à l'entité. Par exemple si *p_out* est un paramètre de sortie d'une opération, de la substitution $p\_out := TRUE$, il est possible de déduire le type *BOOL* du paramètre *p_out*.

Une particularité du système de typage est la notion de super type. Cette notion est induite par la possibilité d'inclusion des ensembles. Ainsi, d'après les expressions $a \in E$ et $E \in T$, il est possible de déduire de la première que a est de type E, mais de la deuxième que a est de type $\mathbb{P}(T)$. Ce calcul du type le plus large correspond à un calcul de super type. Les règles de calcul se trouvent dans [Abrial96, chapitre 2 à partir de la section 2.2].

Les types B reposent sur des types de base ($\mathcal{B}$) et des constructeurs de types. Les types de base sont :
- $\mathbb{Z}$, les entiers ;
- *BOOL*, les booléens ;
- *STRING*, les chaînes de caractères ;
- Ident, les ensembles abstraits.

Les constructeurs de types sont :
- $\mathbb{P}$, l'ensemble des parties finies d'un ensemble ;
- $\times$, le produit cartésien entre deux ensembles ;
- *Struct* ; le type des enregistrements.

Ce système de type a été étendu par [Bodeveix et al.02] afin de construire des informations de type plus précises. En effet, l'objectif des travaux de J.P. Bodeveix et de M. Filali est de pouvoir traduire le langage des types de B dans des systèmes basés sur le λ-calcul typé comme PVS. Comme PVS n'effectue pas de synthèse de type pour les variables λ, il est nécessaire de construire cette information. Or l'information construite par le langage de type définie dans [Abrial96] ne fournit pas une information assez précise pour la traduction vers le langage PVS. Le langage de type est étendu par les constructions «fonctions partielles», «fonctions totales», «intervalles» et «séquences». Ainsi une fonction de E dans F habituellement typée par $\mathbb{P}(E \times F)$ sera typée par $E \longrightarrow F$ dans ce nouveau système de typage.

Ce langage de type étendue est donc le suivant :

$$\underbrace{\varepsilon\tau = \mathcal{B} \mid \mathbb{P}(\varepsilon\tau) \mid \varepsilon\tau \times \varepsilon\tau \mid \underbrace{\{l_i : \varepsilon\tau\}}_{\textit{enregistrements}}}_{\textit{le langage de type B classique}} \underbrace{\mid \underbrace{a..b}_{\textit{intervalle}} \mid \underbrace{\varepsilon\tau \longrightarrow \varepsilon\tau}_{\textit{fonction totale}} \mid \underbrace{\varepsilon\tau \nrightarrow \varepsilon\tau}_{\textit{fonction partielle}} \mid \underbrace{seq(\varepsilon\tau)}_{\textit{séquence}}}_{\textit{extension du langage de type}}$$

Cette extension du langage de type nécessite la définition d'une fonction permettant de ramener les informations plus précises du nouveau langage de type à une notation «classique» B. Cette fonction (notée $\uparrow$) est présentée dans la figure 2.1 extraite de [Bodeveix et al.02]. Elle est similaire au calcul de super type dont nous avons déjà parlé précédemment.

$(ident)\uparrow = ident$	$\mathbb{N}\uparrow = \mathbb{Z}$	$a..b\uparrow = \mathbb{Z}$
$(E_1 \rightarrow E_2)\uparrow = \mathbb{P}((E_1\uparrow) \times (E_2\uparrow))$	$E_1 \nrightarrow E_2\uparrow = (E_1)\uparrow \times (E_2)\uparrow$	$(seq(t))\uparrow = \mathbb{P}(\mathbb{Z} \times (t)\uparrow)$
$(\{l_i : t_i\})\uparrow = \{l_i : (t_i)\uparrow\}$	$(E_1 \times E_2)\uparrow = (E_1)\uparrow \times (E_2)\uparrow$	$(\mathbb{P}(t))\uparrow = \mathbb{P}((t)\uparrow)$

FIG. 2.1 – Du langage de type étendu au langage de type B classique

L'utilité de construire des types plus précis peut paraître faible, nous reviendrons plus en détails en conclusion de ce chapitre sur les choix faits dans notre démarche de génération de code. À ce stade de la réflexion, une des justifications peut simplement être que la génération de code avec pour langage cible un langage comme celui utilisé dans PVS nécessite cette précision au niveau des informations de type ; comme nous ne nous posons aucune restriction *a priori* sur les langages cibles, il nous faut considérer ce langage de type étendu.

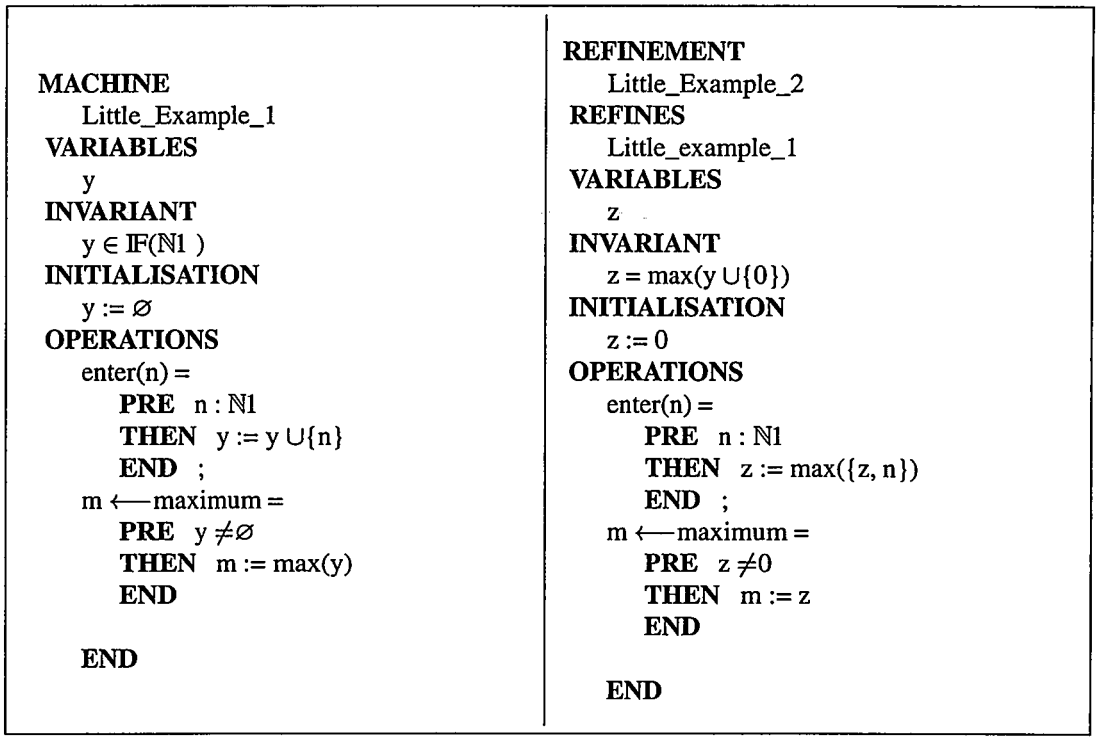
2.2.4 Le raffinement

Le raffinement est le principe de base du développement B. Il est utilisé pour transformer les spécifications abstraites du logiciel désiré en spécifications concrètes qui seront traduites dans un langage de programmation. Ce processus est illustré dans la figure 2.3. Cette transformation pas à pas a deux objectifs essentiels. Le premier est de déterminiser les traitements et les états des spécifications B au fur et à mesure des raffinements. Ces deux transformations portent respectivement le nom de raffinement algorithmique et raffinement de données. Le raffinement de données va permettre de passer de structures abstraites (des ensembles, des relations) vers des structures plus concrètes (des entiers, des tableaux). Une partie des invariants des raffinements, appelés invariants de collage, est utilisée pour lier les données des niveaux de spécification consécutifs. Le raffinement algorithmique va, quant à lui, permettre d'adapter les traitements aux modifications des données, mais également de faire des choix algorithmiques pour réaliser les traitements.

Prenons l'exemple du BBook `Little_Example` (la machine et le raffinement sont présentés dans la figure 2.2) qui décrit un module permettant de garder en mémoire la valeur la plus grande d'un ensemble. Le raffinement de données consiste dans le raffinement à transformer la variable `y` représentant l'ensemble des valeurs en une variable `z` représentant désormais uniquement la valeur maximale. Cette transformation implique de spécifier le lien entre l'invariant et le raffinement par l'invariant de collage `z=max(yU{0})`. Le raffinement algorithmique découlant de ces modifications est illustré dans les transformations des corps des opérations.

La phase de déterminisation des traitements s'accompagne également d'un affaiblissement des pré-conditions. En effet, les pré-conditions d'un raffinement doivent être établies en utilisant en hypothèse les pré-conditions de l'abstraction du composant B. Les conditions à vérifier pour pouvoir appeler une opération peuvent donc être relâchées/affaiblies.

Le deuxième objectif de la transformation incrémentale est d'«injecter» les propriétés du logiciel présentes dans le cahier des charges mais pas forcément incluses dans la version abstraite des spécifications. Cette technique correspond plus à une méthodologie plutôt qu'une spécificité du raffinement. L'usage d'une telle technique de spécification est montré dans les présentations des développements menés autour de METEOR chez Matra Transport International (maintenant Siemens) ([Dehbonei et al.95,



MACHINE
Little_Example_1
VARIABLES
y
INVARIANT
$y \in \mathbb{F}(\mathbb{N}1)$
INITIALISATION
$y := \varnothing$
OPERATIONS
enter(n) =
PRE $n : \mathbb{N}1$
THEN $y := y \cup \{n\}$
END ;
$m \longleftarrow$ maximum =
PRE $y \neq \varnothing$
THEN $m := max(y)$
END

END

REFINEMENT
Little_Example_2
REFINES
Little_example_1
VARIABLES
z
INVARIANT
$z = max(y \cup \{0\})$
INITIALISATION
$z := 0$
OPERATIONS
enter(n) =
PRE $n : \mathbb{N}1$
THEN $z := max(\{z, n\})$
END ;
$m \longleftarrow$ maximum =
PRE $z \neq 0$
THEN $m := z$
END

END

FIG. 2.2 – Le petit exemple

Behm et al.97, Behm et al.]. La phase de développement des spécifications peut être divisée en deux, une première phase de spécifications globales dans laquelle les besoins sont exprimés et une seconde phase de spécifications détaillées dans laquelle les spécifications globales sont développées pour les amener à l'état final désiré, en l'occurrence, le code lorsqu'il s'agit de développement logiciel.

La décomposition en plusieurs étapes permet de faciliter la preuve de correction des spécifications. Cela permet de découper la preuve en plus petits fragments plus facilement réalisables.

Vocabulaire B. À chaque niveau de raffinement est associé un «type» de spécification dans le langage B. Au niveau le plus abstrait, il s'agit des *machines abstraites* (ou `MACHINE`). Pour le niveau le plus concret, il s'agit des *implantations* (ou `IMPLEMENTATION`) et pour les niveaux intermédiaires, il s'agit des *raffinements* (ou `REFINEMENT`). Chacune de ces spécifications s'appelle un *composant* B. Le terme composant est à prendre dans un sens différent de celui vu dans le chapitre précédent. Lorsque la confusion sera possible, la distinction sera clairement indiquée. La chaîne de raffinements de la machine abstraite vers l'implantation forme un *module B*. Une unique machine peut également être appelée module. Lorsqu'une différenciation sera nécessaire, le qualificatif *développé* sera utilisé pour dénoter un module possédant une implantation et le qualificatif *abstrait* pour les modules constitués uniquement d'une machine abstraite.

2.2.5 Les substitutions généralisées

Les substitutions généralisées servent à exprimer l'aspect dynamique des spécifications B : les opérations. Nous différencierons les substitutions en deux catégories, les substitutions dites indéterministes

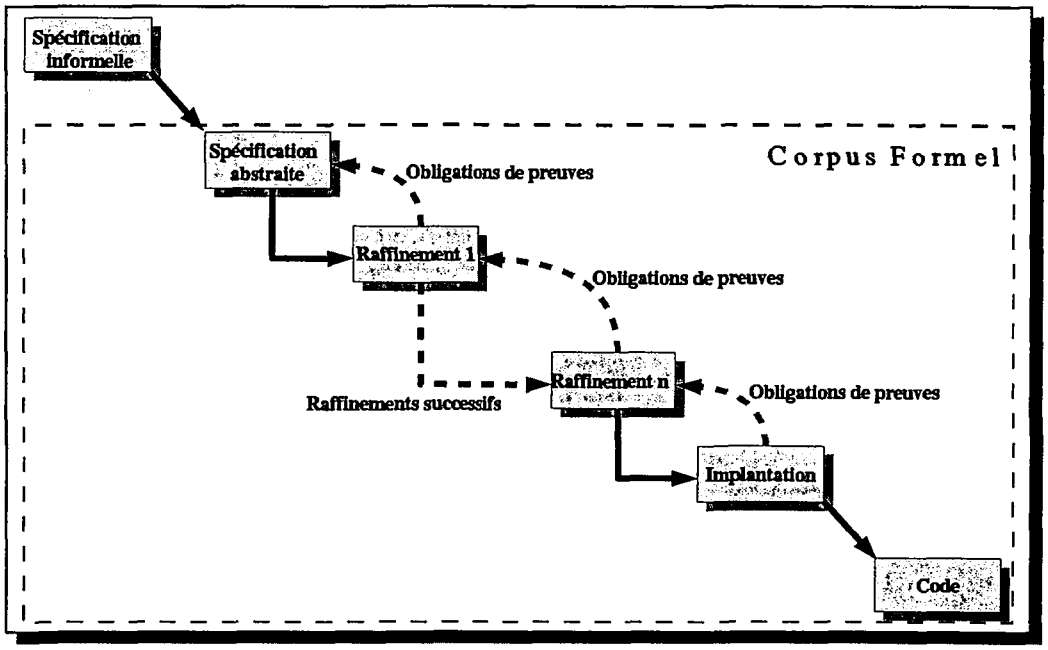


FIG. 2.3 – Principe du développement par raffinements

(présentées dans le tableau 2.1) ne pouvant pas apparaître au niveau le plus concret des spécifications (l'implantation) et celles (présentées dans le tableau 2.2) pouvant y apparaître. Parmi la deuxième catégorie de substitutions, certaines peuvent apparaître au niveau le plus abstrait de spécification (la machine) et d'autres non. Dans le tableau 2.2, cela se traduira par la présence ou l'absence d'un X dans la colonne Mch.

Notation Dans les tableaux 2.1 et 2.2 les notations suivantes seront employées :

- S, S1, ... désigneront une substitution ;
- P, P1, ... désigneront un prédicat ;
- xx désignera une liste de variables ;
- Expr désignera une expression ;
- C désignera une condition, c'est-à-dire une expression booléenne implantable composée des opérateurs $\wedge$, $\vee$, $=$, $\neq$, $\leq$, $\geq$; ... ;
- T, T1, T2, ... désigneront un terme, c'est-à-dire un sous ensemble des expressions.

Les substitutions du tableau 2.2, hormis les deux premières (Pré-condition et Assertion), sont très proches de ce qu'il est possible de retrouver dans les noyaux des langages impératifs. La couche implantation de spécification sera donc très proche du code à générer. À cette couche concrète de spécification correspond un sous langage de B appelé B0.

Nom de la substitution	Notation	Description
Choix borné	CHOICE S1 OR S2 ... END	Choix parmi un nombre fini d'alternatives
Choix non borné	ANY xx WHERE P THEN S END	Choix parmi un nombre possiblement infini d'alternatives. xx est caractérisée par P.
définition locale	LET xx BE P IN S END	Définition locale d'une variable xx caractérisée par P
Choix borné conditionné	SELECT P1 THEN S1 WHEN P2 THEN S2 ... END	Choix conditionné (pas forcément exclusif) parmi un nombre fini d'alternatives
Simultanée avec partage	S1 || S2	Exécution simultanée des deux substitutions
devient tel que	xx :(P)	xx prend une valeur caractérisée par P
devient élément de	xx :∈ Expr	xx prend une valeur de l'ensemble Expr

TAB. 2.1 – Les substitutions indéterministes

2.3 Composition de spécifications *B*

2.3.1 Principe général

Comme pour le développement logiciel, la méthode B fournit la possibilité de décomposer un système (logiciel) en plusieurs sous systèmes plus faciles à spécifier, développer et prouver. Pour des besoins d'écriture de spécifications et de partage d'information, plusieurs possibilités sont fournies pour décomposer. Les termes *clause de liaison* ou *clause d'assemblage* sont utilisés pour désigner ces possibilités. Nous aborderons dans cette partie les différentes clauses de liaisons et montrerons en quoi elles sont utiles. Plus de détails seront donnés dans le chapitre 3, notamment en ce qui concerne les différentes visibilités associées à chacune des clauses.

Nous commencerons par les principales `INCLUDES`, `IMPORTS`, `SEES` et `USES` pour aborder ensuite d'autres clauses qui, soit étendent les possibilités de composition des précédentes clauses, soit servent de «sucre syntaxique» pour la combinaison de plusieurs clauses. La figure 2.4 illustre la majorité des traits de modularité de B. Les dépendances dans un projet B peuvent être représentées comme un graphe acyclique. Ce graphe peut être ramené à un arbre en ne considérant que les liens `IMPORTS` et `REFINES`. Dans ce cas restreint, le terme arborescence des dépendances sera employé.

Indépendamment des différentes clauses d'assemblage, deux principes de base sont à rappeler pour le partage de données en B : le premier principe est qu'**une variable déclarée dans un composant B ne peut être modifiée par un autre composant**. Ceci est nécessaire pour «modulariser» les preuves d'invariants : si une variable était modifiée dans un composant externe, il faudrait vérifier que l'invariant du composant où la variable est définie continue à être vérifié. L'utilisation des opérations du composant permet de localiser les preuves au composant lui même. Le second principe à rappeler est que **la**

Nom de la substitution	Notation	Description	Mch
identité	skip	Ne modifie pas d'état	X
Précondition	PRE P THEN S END		X
Assertion	ASSERT P THEN S END	Exécution de S sous l'hypothèse P	X
Bloc	BEGIN ... END		X
Conditionnelle	IF P THEN S1 ELSE S2 END		X
Condition par cas	CASE Expr OF EITHER T1 THEN S1 ELSE T2 THEN S2 ... END	Choix conditionné (et exclusif) parmi un nombre fini d'alternatives	
Variable locale	VAR xx IN S END	Définition de variables locales	
Tant que	WHILE C DO S INVARIANT P VARIANT Expr END		
séquence	S1 ; S2		
devient égal	xx := Expr		X
Appel d'opération	xx <– op(yy)	Appel d'une opération d'un autre composant	X

TAB. 2.2 – Les substitutions déterministes

décomposition en B suit le paradigme «un écrivain, plusieurs lecteurs». Ceci a comme conséquence que seules les opérations d'un module peuvent modifier (par l'utilisation d'une clause de liaison et en utilisant les opérations définies dans le module pointé par cette clause) l'état d'un autre module (celui pointé par la clause de liaison).

2.3.2 Les liens INCLUDES et IMPORTS

Les clauses INCLUDES et IMPORTS servent à partager les fonctionnalités (les opérations) et les données définies dans un autre module B.

Ce sont des clauses de liaison dont l'usage est proche de celui fait des constructions de modularité usuelles des langages de programmation. Leur objectif est de permettre la décomposition d'un problème en se servant des fonctionnalités développées pour des sous problèmes. La différence entre ces deux clauses est que INCLUDES est une clause de composition pour les machines abstraites et les raffinements alors que la clause IMPORTS est une clause de composition du niveau implantation. Dans le livre de H. Habrias ([Habrias01]), la différence est exprimée en terme d'«utilisation encapsulée» (restriction 7, page 312) : dans une implantation, toute utilisation d'un autre module se fait de manière encapsulée alors que dans une machine ou un raffinement cette restriction ne s'applique pas. «De manière encapsulée» signifie que :

- les variables concrètes sont visibles ;
- les variables abstraites ne sont visibles que dans les invariants et
- les opérations sont visibles.

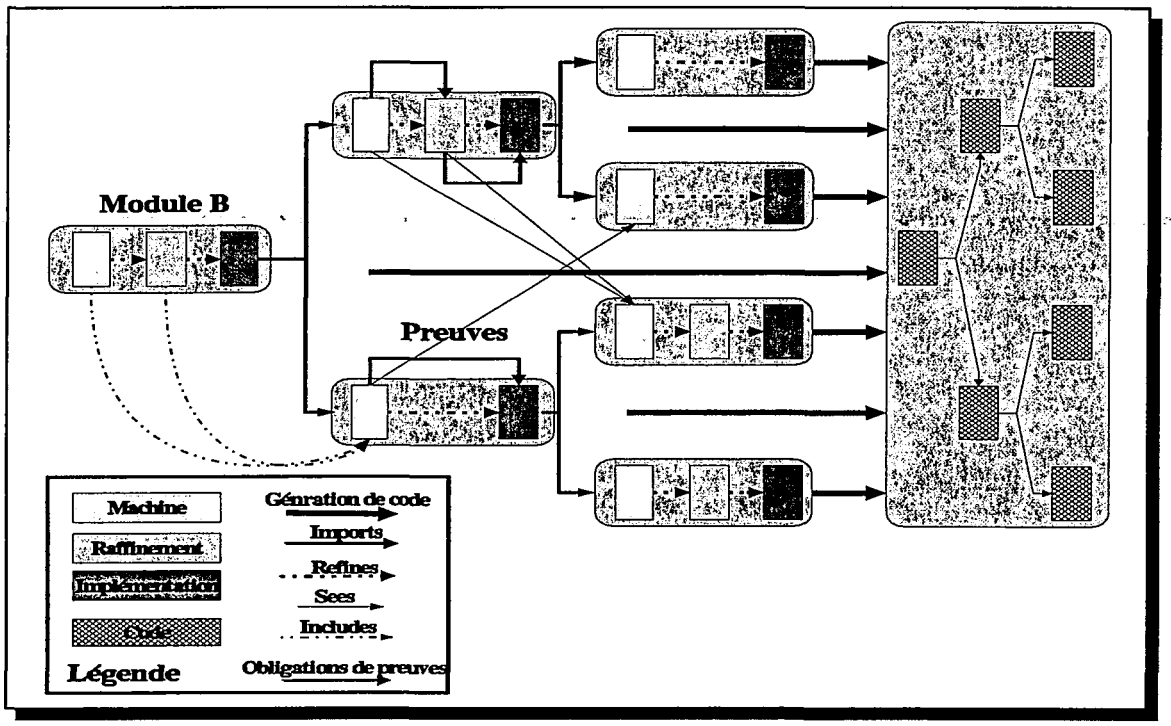


FIG. 2.4 – Décomposition d'un projet B

Ces deux liens sont les seuls à autoriser la modification d'état des composants. La notion de visibilité des entités sera reprise plus en détails dans le chapitre 3.

2.3.3 Les liens SEES et USES

Les liens SEES et USES sont des liens transversaux dans l'arborescence des dépendances d'un projet B. Ils ne sont utilisés que pour consulter les informations des autres composants. Les liens SEES et USES étant des «courts-circuits» dans l'arborescence des dépendances, il est nécessaire que les composants soient attachés à cette arborescence par l'utilisation soit du INCLUDES soit du IMPORTS.

2.3.4 Les autres liens

En plus des quatre liens de bases vus précédemment, il existe deux autres liens, le PROMOTES et le EXTENDS, qui servent à «promouvoir» les opérations d'autres composants. Promouvoir une opération d'un composant Mp dans un composant M revient à intégrer cette opération dans le composant M (les composants M et Mp seront en relation par un lien INCLUDES ou IMPORTS). Ainsi l'opération est considérée comme faisant partie de la spécification M. La clause PROMOTES permet de promouvoir spécifiquement une opération d'un composant inclus ou importé alors que la clause EXTENDS revient à inclure (respectivement importer) dans une machine ou un raffinement (respectivement une implantation) un composant et promouvoir toutes les opérations de ce composant inclus (respectivement importé).

2.4 Processus de développement *B*

Les sections précédentes ont été consacrées aux fondements sur lesquels repose la méthode B, nous allons maintenant aborder plus spécifiquement le processus de développement utilisé dans le cadre d'un développement B. Ce processus peut être décomposé en deux grandes tâches, d'une part, l'action de spécification, et d'autre part, l'action de preuve. À ces deux grandes actions, il est possible et nécessaire d'ajouter une action de test. Cette action de test ne fait à proprement parler pas partie du développement formel, tant que ce dernier est considéré comme idéal, ce qui en réalité n'est jamais le cas. Une argumentation a déjà été menée dans le chapitre 1 section 1.2.3 sur la nécessité du test même dans le cadre d'un développement formel. La méthode B n'échappe pas à la règle (voir [Waeselynck et al.95]).

L'objectif du développement formel que nous fixons est l'obtention du code correspondant aux spécifications. La méthode B a déjà été utilisée avec d'autres objectifs, comme la certification de propriétés ([Lano et al.96, Julliand et al.98, Petin et al.98, Lano et al.98, Requet00]) ou de protocole ([Lanet, Abrial et al.97, Bieber et al.94, Bieber95]. Le processus que nous présentons couvre donc l'intégralité de ce que peut faire la méthode B. Ce processus est présenté dans la figure 2.5 de manière simplifiée, une spécification plus formalisée et intégrant une nouvelle approche sera présentée dans le chapitre 4. Dans cette figure, les ellipses correspondent aux activités et les rectangles aux «produits» (spécifications, code). Une flèche correspond à une interaction entre deux activités ou la production/modification d'un produit. Un développement B s'organise autour de trois activités : la spécification, la preuve et le test. Nous allons aborder ces trois activités dans la suite.

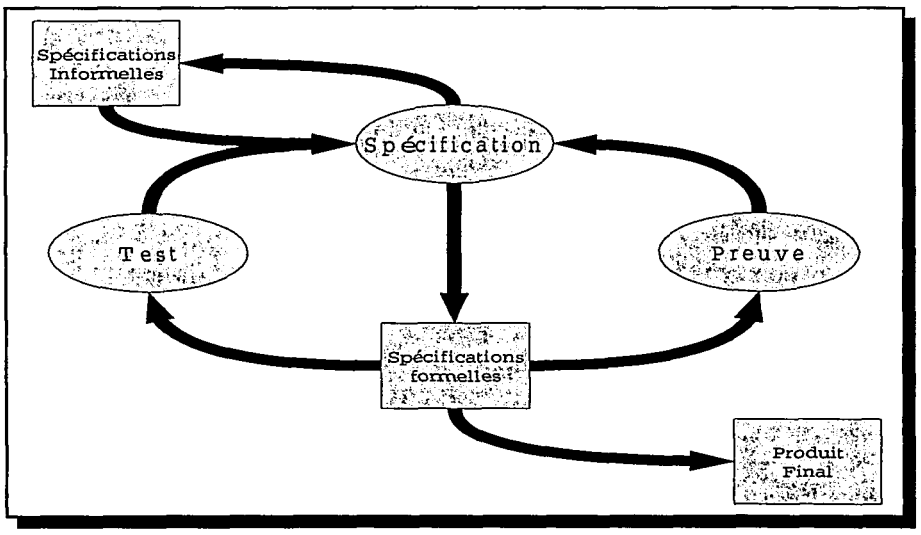


FIG. 2.5 – Processus de développement B : une vue simplifiée

2.4.1 Spécification

L'activité de spécification a plusieurs objectifs. L'un d'eux est de produire à partir du cahier des charges, de spécifications informelles ou à partir de représentations intermédiaires (voir 1.2.2), des spécifications formelles en intégrant les différents besoins et les propriétés du système désiré. Durant cette

transcription/traduction, certains manques peuvent être mis en évidence et nécessiteront une interaction avec le client afin qu'il complète sa spécification informelle, qui peut être ambiguë, incomplète voire contradictoire. Une autre activité de la phase de spécification est le développement des spécifications elles-mêmes, soit par raffinement soit par décomposition. Ces deux activités ont déjà été développées respectivement en sections 2.2 et 2.3.

2.4.2 Test de spécification

Le test peut paraître une activité superflue dans le processus de développement B. Or, même dans une vision idéaliste, dans laquelle aucune erreur ne serait commise dans la phase de développement B, le test reste une nécessité. L'argumentation menée dans [Waeselynck et al.95], prônant la continuation de l'activité de test dans le processus B, s'appuie sur quatre raisons :

- la possibilité de mettre en avant des problèmes méthodologiques de la méthode ;
- l'introduction d'une redondance dans les vérifications ;
- la vérification de la prise en compte des exigences informelles et
- la vérification des règles introduites par l'utilisateur.

Hormis le premier point qui n'a «sûrement» plus lieu d'être à l'heure actuelle, les autres points restent encore d'actualité. En effet, à la date de rédaction de l'article (1995), les outils supports de la méthode et la méthode elle-même pouvaient manquer de maturité (1995 est l'année de la première commercialisation de l'Atelier B). Cette critique portant sur la «jeunesse» de la méthode et des outils doit désormais être nuancée, mais les trois autres points restent d'actualité.

La redondance est une technique souvent utilisée dans les activités critiques, aussi bien pour la validation, que pour la génération de code (où plusieurs chaînes de production du code sont utilisées) ou même sur l'utilisation du logiciel (technique du vote parmi plusieurs chaînes de calcul d'une même donnée, deux parmi trois par exemple, technique très utilisée en avionique).

La troisième raison concerne le problème du passage de l'informel au formel. Comment garantir que la spécification abstraite B, spécification qui sera le référentiel pour les preuves ultérieures (de la chaîne de raffinement), est cohérente avec les exigences du cahier des charges. [Behnia00, Behnia et al.98, Waeselynck et al.97] sont des travaux qui portent sur ce point précis de cohérence des exigences.

La dernière raison peut se généraliser à la vérification des «interventions humaines». En effet, comme nous l'avons vu dans le chapitre précédent, l'argumentaire de [Bowen et al.95] rappelle que toute intervention humaine peut être source d'erreurs. Les travaux de L. Van Aertryck ([Aertryck et al.97, Aertryck98]) sont un exemple de travaux de vérification pouvant détecter ce type d'erreurs. Ces travaux utilisent les spécifications afin de générer les cas de test pour les tests unitaires.

Face à ces arguments, il paraît essentiel de donner les moyens de faciliter le test des spécifications afin de vérifier les points soit qui ne peuvent être couverts par les outils de preuves soit qui peuvent l'être plus facilement par des outils de test. La génération de code peut être un moyen d'y arriver ; il est possible d'envisager la génération d'un code en un langage cible qui sera pris en entrée d'un outil de test.

2.4.3 Preuve

La dernière activité d'un développement B est la preuve. Cette activité est celle qui peut nécessiter l'investissement le plus conséquent en termes de temps et de moyens humains et financiers. Cette activité a pour objectif de prouver que les développements des spécifications sont conformes aux règles B. Deux mécanismes de développement existent en B : le raffinement et la composition ; nous allons présenter les différentes obligations de preuves (qui expriment ce qui doit être vérifié) sur un développement B. Lorsque une preuve aura été effectuée, l'obligation de preuve sera qualifiée de déchargée. Ces obligations nous serviront à comprendre les mécanismes B de partages, de compositions et de raffinements.

Notation : L'expression des obligations de preuves nécessite de définir les entités que nous allons utiliser.

- Le contexte, c'est-à-dire les hypothèses, servant dans une obligation de preuves sera noté Γ.
  Ce contexte sera constitué, sauf mention contraire, d'informations récoltées dans le composant d'où l'obligation de preuves provient (informations provenant des clauses `CONSTRAINTS`, `INVARIANT`, `ASSERTIONS`, ...). En plus de ces informations, d'autres proviendront de règles B comme par exemple la règle stipulant qu'un paramètre de machine qui est un ensemble doit appartenir à l'ensemble des parties finies non vides des entiers. Dans ce cas, une obligation de preuve sera générée afin de vérifier que le paramètre est bien de ce type, mais ces obligations de preuves seront omises dans la suite. Souvent l'expression complète du contexte sera omise, pour des raisons de concision mais également parce que nous ne tenons pas à fournir une spécification des obligations de preuves à vérifier (qui peut d'ailleurs être trouvée dans [Abrial96]), mais comprendre les mécanismes B qui interviendront dans la suite de notre document.
- L'invariant d'un composant sera noté *Inv*.
- L'initialisation sera notée *Init*
- Une opération d'un composant sera représentée par :
  $p\_out \leftarrow op(p\_in) ==$ **PRE** *Pre* **THEN** *body* **END**
  où *p_out* et *p_in* désignent respectivement la liste des paramètres en sorties (respectivement en entrées) de l'opération, *Pre* désigne la pré condition de l'opération et *body* le corps de l'opération.
- Une obligation de preuve sera de la forme $\Gamma \Rightarrow [S]Inv$, ce qui signifie sous les hypothèses Γ, il faut prouver que la substitution S établit bien l'invariant *Inv*. Une substitution établit un invariant si cette substitution appliquée à l'invariant se réduit à vrai (par utilisation du contexte).
- Quand plusieurs composants entrent en jeu, une indexation (par *imp* par exemple dans le cas de l'importation) sera utilisée pour désigner les entités. Dans le cas où la relation liant deux composants est le raffinement, l'indexation *abst* (respectivement raf) sera utilisée pour désigner les entités du composant raffiné (respectivement du composant raffinant).
- Pour désigner une entité E dans laquelle les paramètres du composant où elle a été définie sont instanciés, nous utiliserons E_{inst}
- Les indexations peuvent être combinées, ainsi pour désigner l'initialisation d'une machine importée dans laquelle les paramètres du composant importé sont instanciés, nous utiliserons $Init_{imp\_inst}$

Preuve d'une machine B

Dans une machine B, il est nécessaire de vérifier que l'état initial (donc l'initialisation) du composant satisfait l'invariant :

Initialisation machine **(OP 1)**

$$\Gamma \Rightarrow [Init]Inv$$

Remarque(s) : Γ ne contient pas l'invariant du composant

et que chaque opération n'invalide pas l'invariant :

Opération **(OP 2)**

$$\Gamma \wedge Pre \Rightarrow [body]Inv$$

Remarque(s) : à vérifier pour toutes les opérations de la machine

Preuves de raffinement

Au niveau du raffinement, il faut vérifier que le «delta» introduit dans le raffinement ne remet pas en cause l'abstraction, aussi bien au niveau de l'initialisation que des opérations.

Initialisation raffinement **(OP 3)**

$$\Gamma_{abst} \Rightarrow [Init_{raf}]\neg[Init_{abst}]\neg Inv$$

Opération raffinement **(OP 4)**

$$\Gamma_{abst} \wedge \Gamma_{raf} \wedge Pre_{abst} \Rightarrow Pre_{raff} \wedge [body_{raf}]\neg[body_{abst}]\neg Inv_{raff}$$

Remarque(s) : à vérifier pour toutes les opérations

Dans cette dernière obligation de preuve, on voit clairement que la pré condition du raffinement doit être établie suivant la pré condition de l'abstraction. Ceci illustre le concept d'affaiblissement de la pré condition.

Aparté sur la nécessité de la double négation. Nous allons ici donner l'intuition de la signification de la double négation dans les obligations de preuves liées au raffinement.

Une première interprétation de la nécessité de cette double négation est : un raffinement **ne** doit **pas** être en **dé**saccord avec son abstraction. Dans cette interprétation se retrouve la double négation.

Pour aller plus loin dans l'intuition, nous allons prendre un exemple de raffinement et nous considérerons les chemins parcourus pour arriver au résultat. Prenons par exemple la substitution :

$$x := x+2 \square x := x-2$$

L'application de cette substitution signifie qu'il est possible soit d'ajouter 2 à x soit de retirer 2 à x (cette

substitution est donc indéterministe). Un raffinement de cette spécification pourrait être :

$$\boxed{y := y+1} \text{ avec l'invariant de collage } \boxed{y := 2*x}$$

En considérant cette spécification, il faut démontrer d'après l'OP 4 (en ne considérant que la partie sur les corps d'opération et en notant I l'invariant du raffinement) :

$$\boxed{[y := y+1] \neg [x := x+2 \,\square\, x := x-2] \neg I}$$

En développant la substitution de choix :

$$\boxed{[y := y+1] \neg ([x := x+2] \neg I \wedge [x := x-2] \neg I)}$$

Cette expression s'interprète par : les deux chemins possibles doivent être valides. En continuant le développement, on obtient :

$$\boxed{[y := y+1] \neg (\neg [x := x+2] I \wedge \neg [x := x-2] I)}$$

puis :

$$\boxed{[y := y+1] (\neg\neg [x := x+2] I \vee \neg\neg [x := x-2] I)}$$

qui est équivalent à :

$$\boxed{([y := y+1] [x := x+2] I) \vee ([y := y+1] [x := x-2] I)}$$

Ce qui peut s'interpréter par : la substitution de raffinement doit être cohérente avec l'une des alternatives de l'abstraction. Voila une présentation intuitive de la nécessité de la double négation.

Preuves de composition

Les liens de compositions interviennent essentiellement sur la construction du contexte Γ du composant dans lequel apparaît la clause de liaison. Les seules preuves spécifiques pour les liens de composition sont celles qui vérifient que le composant est bien utilisé dans les conditions prévues (passage de paramètres) :

Instanciation d'un module **(OP 5)**

$$\Gamma \Rightarrow C_{inst}$$

Remarque(s) :
- à vérifier pour les liens `INCLUDES` et `IMPORTS`
- Γ ne contient pas l'invariant

Initialisation et `INCLUDES` **(OP 6)**

$$\Gamma \wedge \Gamma_{incl} \Rightarrow [Init_{incl\_inst}; Init] Inv$$

Remarque(s) : Γ ne contient pas l'invariant

Initialisation et `IMPORTS` **(OP 7)**

$$\Gamma \wedge \Gamma_{imp} \Rightarrow [Init_{imp\_inst}; Init] \neg [Init_{abst}] \neg Inv$$

Remarque(s) :
- Γ ne contient pas l'invariant
- $Init_{abst}$ désigne l'initialisation de l'abstraction de l'implantation

Une autre incidence de la composition sur la preuve est l'utilisation d'opération dans les corps d'opération. Ceci n'entraîne pas de nouvelle obligation de preuve, car l'appel d'opération est remplacé par le texte abstrait de l'opération appelée.

Une fois le processus de preuve terminé, il est possible de générer un code sûr.

2.5 La génération de code à partir de la méthode *B*

2.5.1 La génération de code actuelle

Dans un projet B, chaque module B développé donne lieu à la génération d'une entité/d'un fichier de code dans le langage cible (voir figure 2.4). Les langages cibles sont au nombre de trois dans l'Atelier B supportant la méthode : le C, le C++ et l'Ada. Comme chaque étape de raffinement est considérée comme un «delta» par rapport à la spécification un cran plus abstraite, la construction du produit final code nécessitera de récolter des informations dans toute la chaîne de raffinement de la machine abstraite à l'implantation.

La possibilité de générer du code à partir des spécifications impose certaines restrictions sur ces spécifications : les paramètres des modules B doivent être traduisibles et sont donc restreints à des types concrets (le typage des données est présenté en section 2.2.3). Les paramètres d'entrées et de sorties des opérations sont également soumis à cette même restriction. Avec ces restrictions, ajoutées au fait que les paramètres ne sont pas raffinés, et ne peuvent donc pas passer d'un caractère abstrait à concret, la machine abstraite d'un module peut être considérée comme l'interface du module B. Ceci signifie que la machine abstraite contient toutes les informations servant à caractériser les services rendus par le composant.

Une particularité du code généré par la méthode[5] est que l'environnement d'utilisation du code est considéré comme sûr. En effet, dans le cas d'un développement formel complet, c'est-à-dire avec toutes les preuves déchargées, chaque module B est prouvé être utilisé dans les conditions définies dans les spécifications (invariant et pré conditions). Un seul composant échappe à cette règle, il s'agit du composant racine de l'arbre des dépendances des modules. Ce composant n'est utilisé par aucun autre composant du projet, mais lors de l'intégration du projet B dans l'application plus globale développée, ce qui sera majoritairement le cas pour les applications réelles (voir chapitre 1), ce composant sera utilisé. Ceci a pour conséquence que chaque projet B possède un module point d'entrée très simple, permettant de faire la liaison entre le «monde non formel» et le «monde formel». La figure 2.6 présente le module d'entrée (machine et implantation) de l'étude de cas bien connu BOILER développé par Jean-Raymond Abrial. Ce composant ne sert qu'à lier d'autres composants du projet afin d'utiliser leurs fonctionnalités.

[5]En ce qui concerne la génération de code, on utilisera indifféremment les termes méthodes et outils support de la méthode. Le référentiel outil que nous considérons est l'Atelier B.

```
MACHINE
    Princ_1
OPERATIONS
    main = BEGIN
        skip
    END
END
```

```
IMPLEMENTATION
    Princ_2
REFINES
    Princ_1
IMPORTS
     Constants_L_1, Arithmetic_1, MinMax_1,
Cycle_A_1, Acq_1, Emis_1
OPERATIONS
    main = VAR
        compteur_1
    IN
        compteur_1 := MAXINT ;
        WHILE
            (compteur_1 ≠ 0)
        DO
            main_Acquisition ;        main_A ;
            main_Emission ;  compteur_1  :=
            compteur_1 -1
        INVARIANT
            compteur_1 ∈ NAT
        VARIANT
            compteur_1
        END
    END
END
```

FIG. 2.6 – Module racine du projet BOILER

2.5.2 Les difficultés et les besoins

La caractéristique «boîte noire» des outils commerciaux rend la génération de code difficilement adaptable. Pourtant, ce besoin d'adaptabilité se fait ressentir notamment quand le domaine d'application est un autre domaine que le transport guidé. L'industrie de la carte à puce est un bon exemple d'un domaine d'application où l'usage des méthodes formelles est requis par les ITSEC ou les Common Criteria. Les préoccupations de cette industrie sont différentes de celle du transport guidé, exception faite du besoin de sûreté de fonctionnement, qui lui a, comme dans le ferroviaire, une importance primordiale. En effet, le code dédié à être embarqué au sein d'une rame de métro n'est pas soumis aux mêmes contraintes de ressources que le code embarqué sur une carte à puce. La société GEMPLUS[6], l'une des plus importantes sociétés du marché, s'intéresse à l'utilisation de la méthode B dans le cadre de ces développements [Lanet, Lanet et al.98, Requet et al.00, Motre et al.00, Motre00, Requet00, Lanet00]. L'utilisation de la méthode et des outils supports existant ne posent pas de problèmes tant que le travail mené correspond à la validation de propriétés d'un modèle. Par exemple, le travail mené dans [Requet00], vise à prouver la correction d'un sous ensemble de règles de la machine virtuelle Java Card. Dans ce cas le produit résultat attendu n'est pas le code (au moins dans un premier temps), donc les préoccupations sur le code ne sont pas mises en évidence. Au contraire, lorsque la méthode est utilisée avec pour objectif la production d'un logiciel qui sera embarqué dans une carte à puce, les outils montrent immédiatement leurs limites.

[6] http ://www.gemplus.com/

L'une des préoccupations du projet *B with Optimized Memory (BOM)* ([Bom]), dans lequel GEMPLUS est l'un des partenaires industriels (avec CLEARSY), est le développement d'un générateur de code optimisé mémoire. Un exemple d'optimisation de la mémoire est la traduction des entiers : habituellement traduit sur 32 bits, un entier de plus petite taille sera utilisé sur les supports carte à puce (taille qui sera déterminée suivant la plateforme cible). Cette préoccupation ne peut être prise en compte dans les outils actuels.

L'exemple de la carte à puce nous permet d'illustrer les besoins en terme d'extension des générateurs de code de la méthode B. La difficulté majeure de cette extension est l'absence de spécifications du fonctionnement des générateurs de code. Les principaux points à éclaircir sont :

- comment traduire les instructions du langage B0 ;
- comment s'effectue la collecte d'informations dans la chaîne de raffinement ;
- comment traduire les types B en type du langage cible ;
- comment traduire les paramètres des modules B ;
- comment traduire la modularité de B en «modularité»[7] du langage cible.

Les trois premiers points seront traités dans le chapitre 4. Les deux derniers points concernant la modularité seront également traités dans ce chapitre, mais ils nécessiteront une étude plus approfondie qui sera menée dans le chapitre 3.

2.6 Le dépliage de spécification *B*

Dans cette section, nous allons étudier un algorithme de transformation de spécifications B. Cette algorithme nous a paru un bon point de départ pour l'étude des spécifications/du langage B. Dans cette partie, nous allons présenter cet algorithme que nous avons implanté et nous allons montrer en quoi il peut être utile à la génération de code.

2.6.1 Présentation générale

Déplier ou aplatir (les deux termes seront utilisés indifféremment) des spécifications *B* consiste à construire un composant B équivalent à un ensemble de composants B. Cette opération vise à éliminer les clauses de liaisons entre les composants B mais également la relation de raffinement qui existe au sein d'un module B.

L'équivalence entre un ensemble de composants et un seul composant s'exprimera en terme de raffinement. Le résultat du dépliage d'un ensemble S de composants B doit être un raffinement du composant racine de cet ensemble S (en adoptant la vision arborescente des dépendances vue en 2.3).

La notion de dépliage est implicitement déjà présente dans le BBook ([Abrial96, chapitre 11 section 11.2.6]). La construction de la machine équivalente à deux machines reliées entre elles par un lien INCLUDES y est (partiellement) présentée. Dans les travaux de M.L. Potet et Y. Rouzaud ([Potet et al.98]) le terme «flattening» est employé pour désigner l'opération analogue pour les liens IMPORTS et SEES. Mais,

[7] La notion de modularité n'existant pas dans tous les langages, le terme est ici à prendre dans son sens le plus large : des constructions permettant la division d'un problème en plusieurs unités distinctes.

c'est dans les travaux de S. Behnia ([Behnia00]) que la description de l'algorithme est la plus complète. L'implantation que nous avons faite de l'algorithme est basée sur cette formulation de l'algorithme.

Cependant, nous avons apporté une extension à l'algorithme en relâchant certaines contraintes. En effet, l'algorithme présenté dans [Behnia00] est dédié au test des spécifications *B*. Pour cela, l'aspect observable des spécifications testées est fondamental, il faut donc que le composant résultat du dépliage soit une machine abstraite. Cela induit certaines restrictions sur l'architecture des développements *B* acceptés en entrée du dépliage (voir [Behnia00, section 3.3.2]).

Dans notre approche de l'utilisation du dépliage, notre vision est différente. Nous ne considérons pas le résultat du dépliage comme une machine ou un raffinement, mais comme un composant B, dans le même esprit que la notation en machine abstraite vue en 2.2.2 qui regroupe différentes notations.

Le dépliage se base sur un mécanisme d'enrichissement des spécifications. Enrichir un composant B revient à ajouter dans ce composant les informations se trouvant dans un autre composant relié au premier soit par un lien de composition soit par la relation de raffinement. Dans les sections suivantes, nous allons présenter les mécanismes d'enrichissement liés à l'importation et au raffinement. Nous verrons comment sur ces deux mécanismes il est possible de spécifier l'algorithme de dépliage. Une présentation plus complète de l'enrichissement et de l'algorithme d'aplatissement peut être trouvée dans [Behnia00, chapitre 3].

2.6.2 Enrichissement du lien IMPORTS

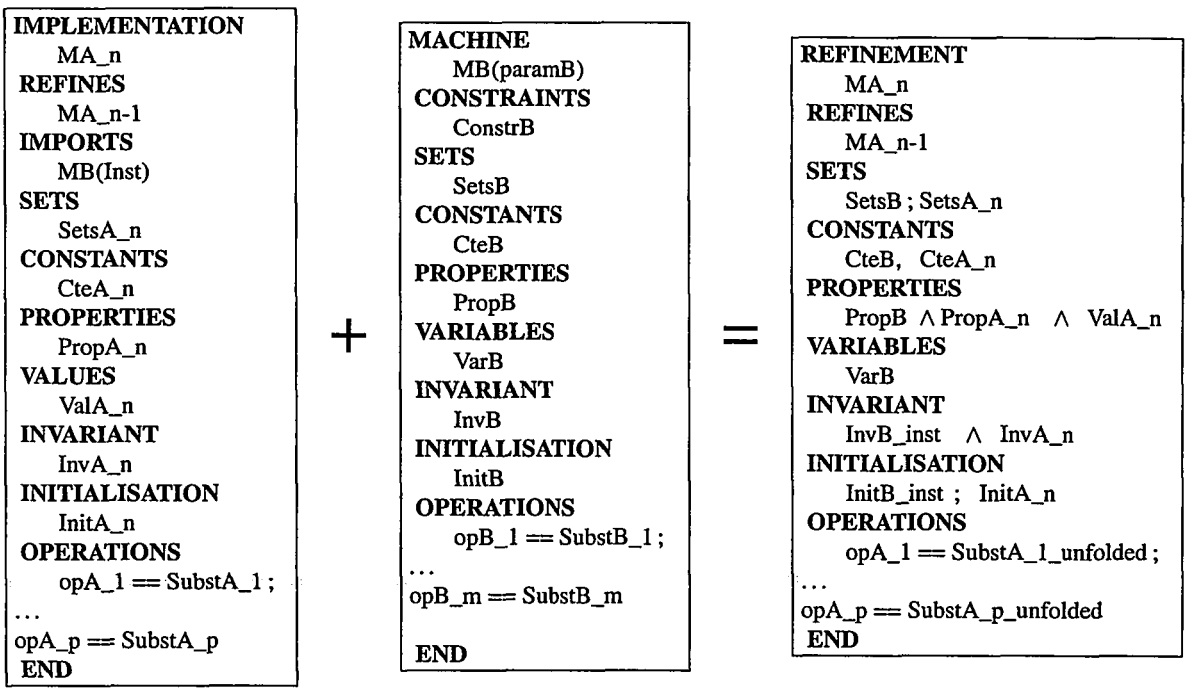


FIG. 2.7 – Enrichissement du lien IMPORTS

Deux composants (une implantation et une machine abstraite) reliés par un lien d'importation seront aplatis en un composant raffinement. La construction du composant résultant est présentée dans la figure

2.7.

Remarques concernant l'enrichissement du lien IMPORTS :

- Les paramètres de *MB* doivent être instanciés dans les expressions reprises du composant importé.
- Seule la concaténation d'information provenant des clauses d'initialisation a un ordre spécifique. L'ordre des initialisations (d'abord celle du composant importé puis celle du composant important) se justifie d'après l'obligation de preuve numéro 6.
- Les opérations du composant important sont conservées, mais les appels aux opérations du composant importé sont remplacés par leur corps d'opération.

2.6.3 Enrichissement du lien REFINES

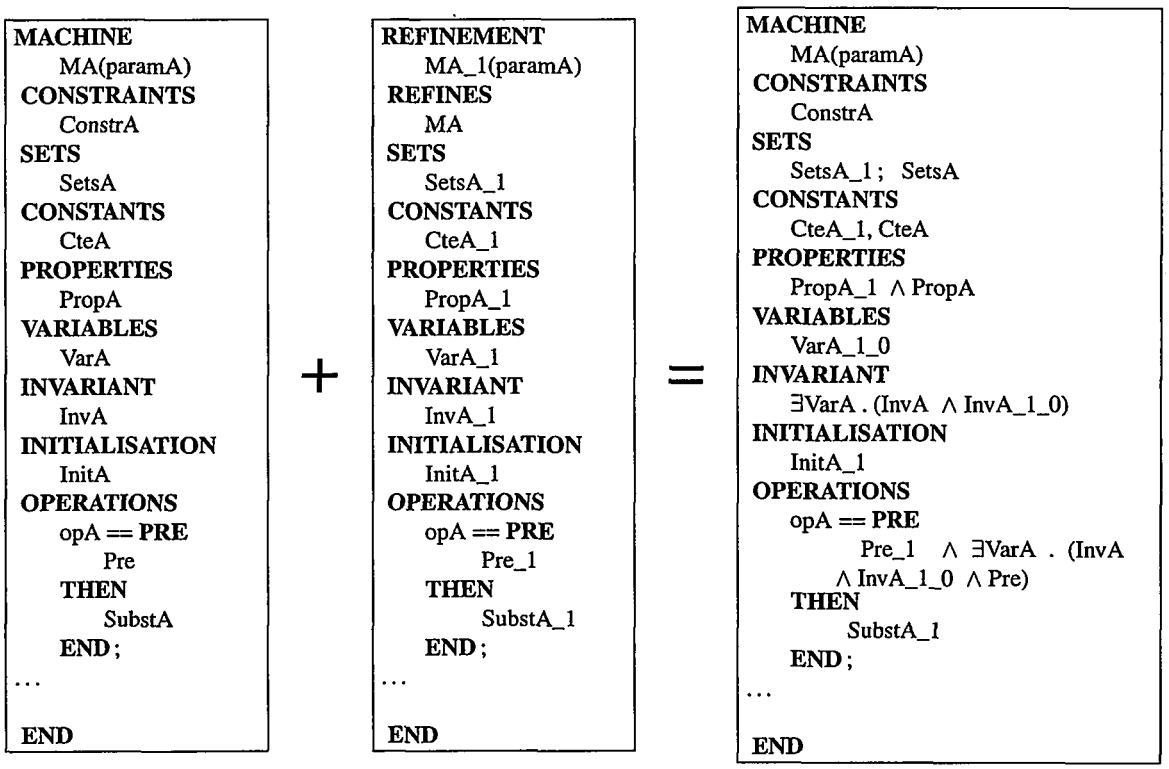


FIG. 2.8 – Enrichissement du lien REFINES

La figure 2.8 présente la construction du composant intégrant le delta d'un raffinement par rapport à son abstraction.

Remarques concernant l'enrichissement du lien REFINES :

- Certaines variables du raffinement peuvent être homonymes à une variable de l'abstraction. Dans ce cas, la variable du raffinement est renommée (en utilisant une variable fraîche) afin de conserver les deux versions de la variable (et donc les propriétés respectives définies dans l'abstraction et le raffinement). Ce renommage doit être propagé à l'ensemble du composant.

Par exemple, si un nom de variable xx est utilisé à la fois dans le raffinement et dans son abstraction, chaque occurrence de xx sera remplacée par xx_0 (xx_0 est supposée être une variable fraîche). De plus le prédicat $xx = xx\_0$ est ajouté à l'invariant du raffinement.

2.6.4 Dépliage des autres liens

Les deux liens IMPORTS et INCLUDES étant proches dans leur utilisation, il n'est pas étonnant que le mécanisme d'enrichissement lié au INCLUDES soit très proche de celui du IMPORTS. La seule différence réside dans les composants manipulés lors du dépliage. Dans le cas du lien IMPORTS, le dépliage se fait entre une IMPLEMENTATION et une MACHINE alors que dans le cas du lien INCLUDES le dépliage se fait entre soit une MACHINE soit un REFINEMENT et une MACHINE.

Il n'y a pas de mécanismes d'enrichissement liés aux clauses SEES et USES. En effet, lors de l'enrichissement des liens REFINES, IMPORTS et INCLUDES, les informations, dont l'accès est rendu possible par l'un des deux liens transversaux, sont regroupées dans un composant ; il n'y a donc plus lieu d'accéder à un autre composant. Ce composant dans lequel les informations sont regroupées est l'ancêtre commun dans l'arbre des dépendances du projet. De part les règles d'architecture des projets B, ce composant existe obligatoirement.

2.6.5 Algorithme de dépliage

L'algorithme est donc basé sur les mécanismes d'enrichissement vus précédemment. Dans l'algorithme, les notations suivantes seront utilisées :

- $N_1 \sqsubseteq N_2$ signifie que N_1 est raffinée par N_2
- $N[T_1, ..., T_n]$ signifie que N importe les machines $T_1, ..., T_N$
- $enrich_{imp}$ désigne le mécanisme d'enrichissement du lien IMPORTS,
- $enrich_{inc}$ désigne le mécanisme d'enrichissement du lien INCLUDES,
- $enrich_{ref}$ désigne le mécanisme d'enrichissement du lien REFINES.

L'algorithme de dépliage correspond à un parcours structurel de l'arbre des dépendances. L'arbre de dépendance est une vision particulière du graphe des dépendances des spécifications B. Cette vision est obtenue en ne prenant en compte que les liens de raffinement et d'importation.

L'aplatissement d'un ensemble de composants T sera désigné par $\Phi(T)$.

- *si* $T = N_1 \sqsubseteq ... \sqsubseteq N_n[T_1, ..., T_m]$,
  alors $\Phi(T) = enrich_{ref}(\Phi(N_1 \sqsubseteq ... \sqsubseteq N_{n-1}), \Phi(N_n[T_1, ..., T_m]))$
- *si* $T = N_1 \sqsubseteq ... \sqsubseteq N_n$,
  alors $\Phi(T) = enrich_{ref}(\Phi(N_1 \sqsubseteq ... \sqsubseteq N_{n-1}), \phi(N_n))$
- *si* $T = N[T_1, ..., T_m]$,
  alors $\Phi(T) = enrich_{imp}(\Phi(N), \Phi(T_1), ..., \Phi(T_m))$
- *si* $T = N$ *(N un composant unique), alors* $\Phi(T) = enrich_{inc}(N)$

Remarque. La relaxation des contraintes sur l'ensemble des composants acceptés en entrée du dépliage impose le remplacement de la dernière règle de l'algorithme par :

- *si* $T = N$ *(N un composant unique, une machine), alors* $\Phi(T) = enrich_{inc}(N)$

– *si $T = N$ (N un composant unique, un raffinement ou une implantation), alors $\Phi(T) = N$*

Implantation de l'algorithme

L'implantation de l'algorithme de dépliage s'est faite au sein de la plateforme BCaml. La plateforme fournit les fonctionnalités de base d'analyse des spécifications B. L'implantation de l'algorithme était un bon «test» pour juger de l'utilité de la plateforme. Cela nous a permis également de mettre en avant certains besoins comme la gestion d'un graphe des dépendances d'un projet B ou d'une librairie de calcul des substitutions pour l'instanciation des paramètres ou le remplacement des appels d'opérations par leur corps.

Dans sa formulation de l'algorithme de dépliage, S. Behnia met en avant les difficultés liées à l'implantation de l'algorithme. Nous allons rappeler ces difficultés et proposer les solutions que nous avons retenues dans l'implantation.

Conflit de nom

Le premier problème est un problème de conflit sur les noms des entités, plus connu sous les termes capture de variables. Dans une spécification B, des variables peuvent avoir le même nom sans pour autant désigner la même entité. La solution proposée dans [Behnia00] est de mettre en place un mécanisme de renommage en amont de l'application de l'aplatisseur. Cette solution a l'inconvénient de perdre la traçabilité entre le code B initial et le code déplié. Nous avons jugé intéressant de garder le plus possible cette traçabilité afin de «remonter» plus facilement les problèmes rencontrés dans le code déplié au niveau des spécifications. Un autre problème pourrait provenir de l'utilisation de convention de nommage dans les spécifications, conventions qui pourraient être utilisées en aval du développement B (pour désigner certaines entités spéciales, par exemple).

La solution que nous avons choisie est une solution qui adopte une vision moins syntaxique des identificateurs. Au lieu de voir un identificateur (un nom de variable) comme une chaîne de caractères, nous le voyons comme une chaîne de caractères et une «empreinte» unique. Cette vision est extraite de [Leroy00a]. Ainsi deux identificateurs représentés par le même nom de variable n'auront pas la même empreinte. L'empreinte peut être construite durant l'analyse des spécifications ou lors d'une phase aval à cette analyse.

Particularité de la clause SEES

Comme nous l'avons vu précédemment, aucun mécanisme d'enrichissement n'est lié à la clause SEES. Outre la nécessité de conserver une trace des composants du projet initial (voir [Behnia00, Résolution du lien SEES section 3.3.4.2]), la prise en compte de ce lien est nécessaire pour l'élaboration de la clause INITIALISATION. L'ordre des initialisations est déterminé par un parcours de l'arbre des dépendances afin qu'un composant importé ou vu soit initialisé avant d'être «utilisé» (au sens : avant d'accéder à ses entités).

Problème lié à la fusion des états

Le troisième problème mis en avant est lié à la fusion des substitutions lors de l'utilisation du ||. En effet l'opérateur || oblige que ses parties gauches et droites portent sur des ensembles de variables différents. Or, par aplatissement, il est possible de devoir expanser dans la substitution `xx := 1 || op` l'appel d'opération *op* provenant d'une machine incluse par exemple. Si cette opération est la substitution *skip*, la substitution aplatie sera `xx := 1 || skip`. Dans un composant B, l'utilisation de la substitution *skip* signifie : l'état (ou les variables) du composant n'est pas modifié. Or, lors du processus d'enrichissement, l'état d'un composant est enrichi par l'état d'un autre composant, et donc la substitution *skip* ne porte plus uniquement sur les variables du composant dans lequel la substitution était utilisée à l'origine.

[Behnia00] propose deux solutions pour résoudre ce problème. La première consiste à définir une nouvelle substitution permettant de spécifier la portée de la substitution *skip*. Cette nouvelle substitution pourrait être de la forme : $skip_{xx,yy,...}$, où $xx, yy, ..$ représente l'espace de porté de la substitution. Cette solution est dans le même esprit que ce qui est proposé dans [Bodeveix et al.99] et [Bodeveix et al.00]. Les auteurs proposent une nouvelle sémantique de l'opérateur || proche de la notion du parallèle rencontrée sur des machines à architecture parallèle et à mémoire partagée. La seconde solution, plus simple consiste à remplacer le *skip* par une substitution «qui ne fait rien» également comme `VAR yy IN yy' :=1 END`. C'est cette solution que nous avons retenue du fait de sa simplicité de mise en œuvre.

2.6.6 Utilisation du dépliage pour la génération de code

Ce que l'algorithme de dépliage est capable de faire peut se résumer à construire un composant B équivalent à un ensemble de composants B. Ce composant peut ensuite être utilisé comme une base pour la génération de code. Cette idée d'aplatir les spécifications afin de générer du code est présente dans [Petit et al.01]. Elle a d'ailleurs été utilisée dans [Petit et al.02b, Petit et al.03a, Petit et al.03c, Petit et al.03b]. L'intuition de l'utilisation du dépliage pour la génération de code est assez facile. Un composant B dans lequel les informations concrètes utiles à la génération de code sont présentes pourra être utilisé pour produire le code correspondant, le modèle concret B étant proche des langages cibles actuels.

Cette technique du dépliage a également été utilisée dans le projet *BOM* ([Bom]) pour la génération du code ([Badeau et al.03]). La différence entre les deux approches, outre le fait que l'objectif principal de la génération de code est différent, est que le dépliage des deux approches n'est pas le même. Dans l'approche BOM, l'aplatisseur est très proche de ce qui est actuellement fait dans l'Atelier B. La génération de code n'est abordée que du point de vue module B, c'est-à-dire qu'à chaque module B correspondra une entité/un module/un paquetage dans le code cible. De plus, les informations collectées se réduisent aux informations utiles à l'exécution du code : les données concrètes et les corps d'opérations concrètes. Le dépliage que nous avons présenté est moins restrictif, d'une part il autorise de manipuler plusieurs modules B et d'autre part il construit un composant contenant le plus d'informations possibles. Ainsi, les variables et constantes abstraites, les invariants et les pré conditions sont conservées dans la vision finale dépliée. Nous détaillerons les avantages de la conservation du maximum d'information sur les spécifications dans le chapitre 4.

2.7 Conclusions

Dans ce chapitre, nous avons présenté les fondements de la méthode B et du langage associé. Nous avons également rappelé les diverses possibilités de partage entre les composants B. Tout ceci est utile au développement d'un projet B. Nous avons montré le cheminement d'un développement B de l'expression des exigences jusqu'au produit final : le code. Cette présentation du processus de développement nous a permis d'introduire l'aspect essentiel de la méthode B : la preuve des spécifications. Cette phase de preuve permet de s'assurer que le code produit est correct vis-à-vis des spécifications abstraites.

En fin de processus de développement, il est possible de générer le code du logiciel spécifié. Les limitations sur la génération de ce code ont été rappelées, avec comme principal défaut, l'impossibilité d'adapter les outils de génération. Les besoins concernant la génération de code peuvent être résumés par deux termes : **maîtrise et adaptabilité**. En effet, la diffusion et l'usage de la méthode B seront favorisés si les outils l'accompagnant ne se cantonnent pas à un usage spécifique pour un domaine d'activité particulier. Pour le moment, les seules expériences industrielles de grande échelle de l'utilisation de la méthode B sont des applications ferroviaires. L'exemple de la carte à puce montre qu'une bonne gestion de la génération de code est déjà un plus pour la méthode.

Le choix de l'algorithme de typage s'est porté sur un algorithme utilisant un langage de type étendu. Ce choix est représentatif de notre démarche. En effet, actuellement, Le code obtenu par le processus de développement B est considéré s'exécuter dans un environnement que nous qualifierons d'idéal, c'est-à-dire qu'aucun «phénomène» extérieur ne peut remettre en cause le bon déroulement du code. Or, nous avons signalé dans le chapitre 1 qu'un développement formel devra le plus souvent être intégré dans un développement non formel, ce qui rend l'hypothèse d'«exécution dans un environnement idéal» infondée. Notre démarche va donc prendre en compte cet aspect, ce qui se traduira dans la suite par un besoin constant de construire une information la plus précise possible.

Nous avons également présenté dans ce chapitre un algorithme de manipulation des spécifications B. L'implantation de l'algorithme de dépliage est un travail conséquent tant en terme de temps de développement (environ 7 Hommes*mois[8]) qu'en terme de complexité (le travail nécessaire à la compréhension de l'algorithme et sa complétion sont une part importante). Du point de vue quantitatif, l'implantation de l'algorithme de dépliage a nécessité environ 4000 lignes de code[9]. Même si cet algorithme peut être utilisé pour la génération de code, son implantation a soulevé un problème : la manipulation des concepts de modularité de B. Cette constatation nous a donc mené à étudier la modularité du langage B. Nous allons présenter ce travail dans le chapitre suivant.

[8]La quantification est ici donnée à titre indicatif, mais tente de refléter au mieux le temps de développement (les temps d'adaptation nécessaire aux personnes ayant participé au développement ont été soustrait du temps total).

[9]Plus de précision seront apportées en ce qui concerne la taille du code des différents développements menés dans le cadre du travail présenté dans ce mémoire dans le chapitre 5. Le nombre avancé ici correspond à l'implantation de l'algorithme de dépliage ainsi qu'aux développements annexes qui ont du être menés.

Chapitre 3

Une nouvelle vision de la modularité en B

Sommaire

3.1 Introduction

Dans le chapitre précédent, nous avons présenté la méthode B dans son ensemble. Dans ce chapitre, nous allons nous intéresser à un aspect particulièrement important : la modularité des spécifications. La modularité est un concept essentiel à maîtriser pour la génération de code.

La modularité de B a déjà été étudiée dans différents travaux. [Bert et al.96] développent une algèbre des composants pour y exprimer les clauses INCLUDES et USES. Cet article propose également l'idée d'une possible corrélation entre une machine *B* et un module, ainsi que le besoin d'indépendance entre le langage *B* et ses constructions de modularité. [Buchi et al.] proposent un nouveau lien qui étend les possibilités de partage des données entre les composants *B*, en transformant le paradigme «un écrivain et plusieurs lecteurs», actuellement utilisé en *B*, en «plusieurs écrivains et plusieurs lecteurs». [Dimitrakos et al.00] étudient également les différentes clauses de liaisons et mettent l'accent sur les conditions à remplir pour conserver les propriétés des composants. [Potet et al.98] étudient les clauses de liaisons IMPORTS et SEES et les problèmes qui peuvent être engendrés par leurs usages. En particulier, cet article démontre qu'il est possible de construire un projet *B* dont la structuration induit que le développement est considéré à tort comme correct. Les auteurs proposent néanmoins des critères, basés sur une analyse du graphe des dépendances, pour une élimination des configurations incorrectes.

Par ailleurs, d'autres travaux se sont consacrés à l'expression des caractéristiques de *B* au moyen d'autres formalismes ; citons les exemples d'un début de formalisation de *B* par Isabelle/HOL [Chartier98], par PVS et Coq [Bodeveix et al.99], [Bodeveix et al.00], [Pratten95]. Sans entrer dans une présentation détaillée de ces travaux, remarquons un point essentiel : certes, l'objectif d'une (méta) formalisation externe de *B* constituerait un apport conséquent au fondement théorique de *B*, cependant les travaux cités n'envisagent pas la méthode *B* dans sa globalité et ne traitent pas de la modularité. La formalisation reste cantonnée au langage basique. *Sans que cela soit une critique*, ce constat montre bien que l'expression formalisée de la modularité de *B* constitue à la fois un objectif et une difficulté centrale.

Dans ce contexte, notre approche est orthogonale et pragmatique, car ayant pour finalité la réalisation d'un outil, nous ne pouvons éviter d'aborder le langage *B* dans sa globalité et nous traitons la modularité de *B* **sans** faire intervenir (directement) les fondements théoriques de *B*.

Dans ce chapitre, nous commencerons par présenter la modularité de B telle qu'elle est faite dans le BBook ([Abrial96]). Nous verrons ensuite comment en adoptant une vision différente de cette modularité, il est possible de simplifier sa présentation. Cette vision nouvelle sera alors utilisée pour instancier un calcul de modules présenté dans [Leroy00a] dont nous rappellerons les principes essentiels.

3.2 La modularité *B*

3.2.1 Les paramètres des modules B

Les modules B peuvent être paramétrés par deux sortes de paramètres : les paramètres dits scalaires ([Steria98]) et les paramètres ensembles. Les paramètres scalaires correspondent à des données de type

simple comme des entiers, des booléens. Ils peuvent également être du type d'un ensemble passé en paramètres. Les paramètres ensembles sont quant à eux restreints à des ensembles équipotents à des intervalles d'entiers non vides. Ces ensembles sont considérés comme des types de base dans le composant. Un exemple d'en-tête de machine est présenté dans la figure 3.1. La clause **Constraints** sert à typer et contraindre les paramètres.

MACHINE
Exemple (parametre_scalaire1, parametre_scalaire2, parametre_ensemble)
CONSTRAINTS
parametre_scalaire1 $\in \mathbb{N}$ $\wedge$ parametre_scalaire2 $\in$ parametre_ensemble

FIG. 3.1 – Exemple de paramétrisation d'un module B

3.2.2 Présentation «classique»

La modularité est présentée dans le BBook ([Abrial96]) comme un ensemble de tables de visibilité. Cette présentation est assez difficile à assimiler et fait du système de modules de B un système *a priori* complexe. G. N. Watson dans [Watson97] va même jusqu'à qualifier «la prolifération des constructions du système de modules de B et des règles associées» de «rebutante». Son argumentation tient, entre autres, sur la dizaine de pages nécessaires en annexe D de [Abrial96] pour décrire les différentes tables de visibilité. De ces tables, nous avons repris (et adapté afin d'obtenir une cohérence avec le manuel de référence du langage B [Steria98]) dans les tableaux 3.1, 3.2, 3.3 et 3.4 les tables concernant les liens **Imports** et **Sees**.

Dans ces tableaux, chaque ligne correspond à une catégorie d'entités du composant vers lequel une dépendance est exprimée dans la spécification (dans les deux tableaux, soit **Imports** soit **Sees**). Au terme catégorie sera souvent adjoint le qualificatif syntaxique. Ces termes seront employés pour désigner les groupes d'entités ayant une même fonction dans la spécification. Dans les tableaux, chaque ligne représentera une catégorie syntaxique pour le langage B. Chaque colonne correspond à une clause du composant dans lequel est exprimée la dépendance. Les notations qui sont utilisées dans les tableaux sont :

- ✓ dénote un accès totale à l'entité (aussi bien en lecture qu'en écriture)
- ✓ (lecture seule) dénote un accès limité à la consultation. Cet accès limité à la consultation correspondra en ce qui concerne les opérations, aux opérations ne modifiant pas l'état de la machine dans laquelle ces opérations sont définies.
- dans les invariants de boucles dénotera la particularité de l'accès à une entité uniquement dans les invariants de boucles.
- L'absence de marquage particulier dénotera l'absence d'autorisation d'accès à une catégorie d'entités.

Ce qui est à remarquer dans les différents tableaux est que l'autorisation d'accès à certaines entités dépend de l'endroit dans lequel la demande d'accès est faite. Ainsi, dans le cas d'une liaison **Imports** entre un composant `M_imp` et une machine `M` (`M_imp` importe `M`), une variable abstraite `VAR` de la machine

Relatif au **Includes**	**Constraints**	**Includes**	**Properties**	**Invariant/ Assertions**	**Initialisation/ Opérations**
Paramètres					
Ensemble			✓	✓	✓
Constantes Concrètes			✓	✓	✓
Constantes Abstraites			✓	✓	✓
Variables Concrètes				✓	✓ (lecture seule)
Variables Abstraites				✓	✓ (lecture seule)
Opérations					✓

TAB. 3.1 – Visibilité des entités d'une machine incluse par une machine

Relatif au **Uses**	**Constraints**	**Includes**	**Properties**	**Invariant/ Assertions**	**Initialisation/ Opérations**
Paramètres				✓	✓
Ensemble			✓	✓	✓
Constantes Concrètes			✓	✓	✓
Constantes Abstraites			✓	✓	✓
Variables Concrètes				✓	✓ (lecture seule)
Variables Abstraites				✓	✓ (lecture seule)
Opérations					

TAB. 3.2 – Visibilité des entités d'une machine utilisée par une machine

Relatif au **Imports**	**Imports**	**Properties**	**Values**	**Invariant/ Assertions**	**Initialisation/ Opérations**
Paramètres					
Ensemble		✓	✓	✓	✓
Constantes Concrètes		✓	✓	✓	✓
Constantes Abstraites		✓		✓	dans les invariants de boucles
Variables Concrètes				✓	✓ (lecture seule)
Variables Abstraites				✓	dans les invariants de boucles
Opérations					✓

TAB. 3.3 – Visibilité des entités d'une machine importée par une implémentation

Relatif au **Sees**	**Imports**	**Properties**	**Values**	**Invariant/ Assertions**	**Initialisation/ Opérations**
Paramètres					
Ensemble	✓	✓	✓	✓	✓
Constantes Concrètes	✓	✓	✓	✓	✓
Constantes Abstraites		✓		✓	dans les invariants de boucles
Variables Concrètes					✓ (lecture seule)
Variables Abstraites					dans les invariants de boucles
Opérations					✓ (lecture seule)

TAB. 3.4 – Visibilité des entités d'une machine vue dans une implémentation

M pourra être utilisée dans les invariants de boucle des opérations de M_imp, alors que dans le reste du corps des opérations l'accès à cette variable est interdit.

3.2.3 Une vision différente de la modularité *B*

La vision que nous avons donnée des règles de visibilité lors de la composition de modules est une vision qui ramène la présentation à un ensemble de tableaux exprimant pour chaque lien les entités exportées ou non d'un module B. Or parmi ces tableaux, certaines informations ne concernent pas la composition en elle-même, mais le langage B lui même. Ainsi, dans le tableau 3.5 présentant les règles de visibilité d'une machine B par rapport aux entités de cette machine, il est possible de tirer quelques règles sur l'accessibilité des entités sans même parler de modularité. Ainsi dans la clause **Properties**, seuls les ensembles et les constantes sont accessibles. Il n'est donc pas nécessaire que lors de la présentation des constructions de modularité cette particularité soit rappelée. Pour la vision que nous allons donner dans

	Constraints	**Includes**	**Properties**	**Invariant/ Assertions**	**Initialisation/ Opérations**
Paramètres	✓	✓		✓	✓
Ensemble		✓	✓	✓	✓
Constantes Concrètes		✓	✓	✓	✓
Constantes Abstraites		✓	✓	✓	✓
Variables Concrètes				✓	✓
Variables Abstraites				✓	✓
Opérations					

TAB. 3.5 – Visibilité des entités d'une machine dans cette même machine

cette section, la première nécessité est de gommer les différences entre les différents types de composants B (machine, raffinement et implantation). Les deux niveaux machine et raffinement sont identiques du point de vue de la modularité ; les règles de visibilité s'appliquant aux deux types de composants sont identiques. La différence entre les niveaux machine et raffinement et le niveau implantation réside dans les constructions autorisées dans une implantation. En effet, il est logique que dans un niveau abstrait des spécifications certaines constructions qui sont autorisées ne le soient plus dans un niveau concret (les substitutions non déterministes par exemple). Notre vision de l'accessibilité des entités dans un composant est regroupée dans les tableaux 3.6 et 3.7.

	Constraints	**Clauses de liaison**	**Properties**	**Values**
Paramètres	✓	✓		
Ensemble Abstrait		✓	✓	✓
Set Énuméré		✓	✓	✓
Constantes Concrètes		✓	✓	✓
Constantes Abstraites		✓	✓	
Variables Concrètes				
Variables Abstraites				
Opérations				

TAB. 3.6 – Règles de visibilité dans un composant B (1)

	Invariant	Opérations	Opérations B0	Boucle
Paramètres	✓	✓	✓	
Ensemble Abstrait	✓	✓	✓	✓
Set Énuméré	✓	✓	✓	✓
Constantes Concrètes	✓	✓	✓	✓
Constantes Abstraites	✓	✓		✓
Variables Concrètes	✓	✓	✓	✓
Variables Abstraites	✓	✓		✓
Opérations		✓	✓	

TAB. 3.7 – Règles de visibilité dans un composant B (2)

Le but de cette vision est de ramener l'expression des différentes règles de visibilité à une simple notion d'exportation ou non des entités du composant. En effet, il n'est pas nécessaire de traiter les restrictions d'accès vues précédemment dans les règles d'accès liées à la modularité du langage. Cette vision est classique dans les langages de programmation qui permettent souvent de définir des parties publiques et des parties privées. Cette nouvelle vision nous amène à définir pour chaque lien de composition du langage B les entités exportées ou non. Ceci est présenté dans le tableau 3.8.

	Includes	Imports	Sees	Uses
Paramètres				✓
Ensemble Abstrait	✓	✓	✓	✓
Set Énuméré	✓	✓	✓	✓
Constantes Concrètes	✓	✓	✓	✓
Constantes Abstraites	✓	✓	✓	✓
Variables Concrètes	✓	✓	✓	✓
Variables Abstraites	✓	✓	✓	✓
Opérations	✓	✓	✓ (lecture seule)	

TAB. 3.8 – Règles de visibilité entre deux composants B

En plus de ces règles de visibilité, il faut rappeler deux principes importants du langage B :

Encapsulation des données :

Le principe d'encapsulation en B stipule que toute modification de l'état d'un module B doit se faire par l'utilisation des opérations définies dans ce module. Ceci permet de ne faire la preuve du respect de l'invariant d'un module que lors de la validation de ce module.

Appel d'opérations :

Pour les mêmes raisons que pour le principe d'encapsulation, les opérations d'un composant B ne peuvent être utilisées dans le même composant B. L'invariant d'un composant n'étant pas vérifié à l'intérieur du corps d'une opération, mais seulement avant et après l'appel de cette opération, appeler une opération dans une autre reviendrait à transgresser l'hypothèse selon laquelle avant un appel d'opérations, l'invariant est vérifié.

Résumé : Dans cette nouvelle vision de la modularité, les concepts à mettre en œuvre se ramènent simplement à l'exportation ou non des entités suivant le lien de composition utilisé et deux principes de base, l'encapsulation et la modularisation de la preuve d'invariant.

3.3 Les «systèmes de composition»

Dans cette partie, nous allons explorer les différentes possibilités qui nous étaient offertes pour l'expression de la modularité de B par/dans un système de composition usuel. Par système de composition usuel, nous entendons un système permettant de mettre en œuvre un principe de base du développement logiciel : la décomposition d'un problème en plusieurs sous-problèmes. Ce système de composition, s'il est intuitif, permettra une compréhension plus facile de la modularité de B. Il nous a semblé judicieux de s'intéresser aux systèmes utilisés dans les langages de programmation pour deux raisons. La première, parce qu'un tel choix répond *a priori* à l'objectif de simplicité et d'intuition. En effet, les langages de programmation sont en général connus des développeurs, ce qui facilitera leur compréhension du système de modules de B si un parallèle est possible avec des connaissances déjà acquises de leur part. La seconde raison qui nous a poussé vers des systèmes de composition proches des langages de programmation est l'objectif final du travail présenté dans ce mémoire : la génération de code. En choisissant une représentation proche des langages qui devront être finalement atteints par la phase de génération de code, nous faisons un premier pas vers le code sensé être généré en fin de chaîne.

Notons qu'il aurait été possible de s'intéresser également à des systèmes moins connus qui auraient tout aussi bien satisfait *a priori* notre objectif de simulation de la modularité de B. Mais ces systèmes, comme par exemple une évolution des systèmes à base de modules SML, les Mixin modules ([Ancona et al.98]), ne satisfont pas deux critères importants : d'une part, ils ne sont pas bien connus dans le monde du développement logiciel, et d'autre part, leur jeunesse fait de ces systèmes des systèmes en évolution ; il faut donc attendre qu'ils atteignent un niveau de stabilité suffisant.

Dans les langages actuellement utilisés, nous trouvons trois grandes familles de systèmes de composition : les systèmes reposant sur l'utilisation des techniques objets (les langages de classes), ceux reposant sur l'utilisation des paquetages (à la Ada) et ceux reposant sur l'utilisation de modules/structures (à la ML). Chacun de ces systèmes a ses particularités, mais également des points communs. Nous allons dans la suite de cette section donner les arguments qui motiveront notre choix d'un de ces systèmes. Avant nous rappellerons brièvement (la littérature présentant les divers concepts étant nombreuse) la définition de ce que sont une classe, un paquetage et un module[1]. Cette présentation des concepts s'appuiera dans le cas des paquetages et des modules sur l'«instanciation» qui est faite de ces concepts pour les langages Ada et OCaml.

Classe *«Une classe est un élément logiciel qui décrit un type abstrait de données et son implémentation totale ou partielle» ([Meyer00]).* Un type abstrait de données est une collection d'entités et

[1]Le terme module sera employé sous deux sens différents, d'une part dans sons sens le plus général, comme une unité de composition et d'autre part comme module à la ML, c'est-à-dire les modules que l'on retrouve dans les langages ML. Lorsque qu'un confusion sera possible, le sens désiré sera précisé.

un ensemble d'opérations manipulant ces entités. Une classe est à la fois un module (dans son sens le plus large : unité de décomposition) mais également un type. La création d'un objet du type d'une classe passe par une instanciation de cette classe.

Paquetage Dans le manuel de référence Ada ([Ada94]), *un paquetage est défini comme une unité de programme permettant la spécification de groupe d'entités logiquement reliées entre-elles.* La définition d'un paquetage requiert la définition, d'une part, de sa spécification qui regroupe la définition des entités définies dans le paquetage (tout ou partie des entités) ainsi que de leur type, et d'autre part, du corps du paquetage qui définit concrètement les entités (variables, types, méthodes...).

Module Un module est défini dans [Macqueen86] comme une *collection nommée de déclarations dont le but est de définir un environnement.* La notion de module dans les langages ML est très proche de la notion de paquetage. Comme pour les paquetages, la définition d'un module est divisée en deux, d'une part, l'interface du module qui spécifie ce qui est connu de l'extérieur, les entités et leur type, et d'autre part, le corps des entités constituant le module. La différence entre un module et les paquetages précédents réside dans la possibilité d'avoir plusieurs interfaces pour un même module, ce qui n'est pas possible pour les paquetages.

3.3.1 Les concepts de base

De la présentation que nous avons faite de la modularité de B, les concepts principaux qu'il semble intéressant de retrouver dans le système de composition sont :

- **l'encapsulation** des données permettant de «cacher» certaines entités au monde extérieur,
- **les modes d'utilisation multiples** d'une même entité de composition,
- **la paramétrisation** par des valeurs et des types et
- **l'instanciation** d'un même module. Ceci correspond en B à la possibilité d'utiliser plusieurs fois un même module en le renommant. Dans ce cas, le module est recopié pour donner un nouveau module.

L'encapsulation

L'encapsulation est un concept quasi universel dans les systèmes de composition. En effet, la possibilité de ne dévoiler qu'une partie d'une implantation, et donc d'en cacher une autre partie est une nécessité pour arriver à une certaine indépendance entre les différents modules. Dans les langages objets, trois catégories de visibilité se retrouvent en général : les entités peuvent être publiques, protégées, ou privées. Pour les langages utilisant les paquetages, les qualificatifs publique et privé se retrouvent également. Dans les langages de modules, l'utilisation des interfaces permet de dire si oui ou non telle entité est exportée. Cela correspond à la vision publique et privée des classes et des paquetages.

Les modes d'utilisation multiples

Un mode d'utilisation d'une même unité de composition consiste à spécifier quelles entités de cette unité de composition doivent être exportées. À chaque besoin sera associé un mode d'utilisation de l'unité de composition. Bien que relié à la notion précédente d'encapsulation, ce concept est différent. Ici, le but est de changer de vision sans pour autant changer de module. Cela correspond au besoin par exemple d'exporter toutes les opérations d'un module B lors d'une importation et de n'exporter que les opérations ne modifiant pas l'état du module dans le cas d'un accès par le lien **Sees**. Cette «vision multiple» d'un même module peut être obtenue par les systèmes à base de modules, par l'utilisation de plusieurs signatures pour le même module mais également pour certains langages de classes par le biais des interfaces (le langage Java par exemple supporte ces constructions).

La paramétrisation

La paramétrisation en B concerne comme nous l'avons déjà vu deux types de paramètres, des valeurs et des ensembles qui correspondent à des types. Autant le passage de valeurs ne pose pas de problèmes, autant le passage de types est plus problématique. En effet les types ne sont en général pas des citoyens de première classe dans les langages de programmation, ce qui fait qu'ils ne pourront pas être passés en paramètres comme les valeurs. Les solutions pour arriver à exprimer la paramétrisation existent pour les systèmes à base de modules et pour les systèmes à base de paquetages, pour les langages de classes, les solutions diffèrent suivant les langages (classes génériques en Eiffel, templates en C++). Pour les modules, il faut utiliser les constructions de foncteurs[2] permettant de passer en paramètres des modules. Il est possible de définir dans ces modules passés en paramètres des types. Au niveau des paquetages, c'est la notion de paquetage générique qui sera utilisée. Les paquetages génériques correspondent exactement à ce que l'on cherche à faire ici : paramétriser les paquetages.

L'instanciation

L'instanciation correspond à la possibilité en B de reprendre le code d'un module. Le terme instanciation est un peu fort pour désigner ce processus de «recopie». En effet, l'instanciation recouvre en général également le point précédent concernant la paramétrisation. Si les modules à instancier sont paramétrés, les mécanismes à utiliser sont les mêmes que précédemment. Par contre, si les modules B ne sont pas paramétrés, seuls les systèmes à base de classes fournissent la possibilité d'instancier plusieurs objets à partir d'une seule classe. Notons, qu'il est toujours possible de ramener un module non paramétré à un module paramétré dont les paramètres sont «vides». L'usage de ce module ne sera possible qu'après instanciation de module par un module ne contenant aucun élément, cela permettra de simuler l'instanciation utilisée en B.

[2]Cette notion de foncteur sera détaillée dans les sections suivantes, rappelons simplement ici qu'un foncteur est aux modules ce que les fonctions sont aux valeurs.

3.3.2 Choix du système

Critères de sélection

Dans la section précédente, nous avons abordé les aspects importants pour choisir parmi trois types de composition celui qui conviendrait le mieux à la simulation de la modularité de B. À ces quatre critères (dénotés par les lignes **Encapsulation**, **Vision multiple**, **Paramétrisation** et **Instanciation multiple** dans le tableau 3.9) correspondant directement au plongement des modules B dans un autre système et à l'aspect sociologique (dénoté par la ligne **Compréhensible**) que nous avons également abordé précédemment (le système retenu doit être intuitif), nous allons ajouter un critère supplémentaire, critère pragmatique visant à qualifier la difficulté à implanter un système simulant le système de modules de B (dénoté par la ligne **Implantation fournie**). Ce critère vise à évaluer la simplicité à implanter chacun des trois systèmes de composition.

Le tableau 3.9 reprend l'intégralité des critères de manière synthétique. Trois niveaux de qualification ont été retenu : +, $\sim$ et - qui dans l'ordre évaluent l'adéquation du système suivant un critère d'une adéquation forte à une adéquation faible ou nulle. Le symbole $\sim$ sera utilisé dans les cas où il est nécessaire d'utiliser un concept de manière détournée pour obtenir le résultat voulu (par exemple l'instanciation multiple pour les modules) ou lorsque ce concept n'est pas présent dans tous les langages majeurs de la catégorie (par exemple pour la paramétrisation des classes, qui existe en OCaml mais pas encore en Java).

	Critères	Classe	Module	Package
Simulation du système de modules de B				
	Encapsulation	+	+	+
	Vision multiple	$\sim$	+	-
	Paramétrisation	$\sim$	+	+
	Instanciation multiple	+	$\sim$	$\sim$
Aspects Pragmatiques				
	Implantation fournie	-	+	-
	Système éprouvé	+	+	+
Aspect sociologique				
	Compréhensible	+	+	+

TAB. 3.9 – Critères de sélection du système de «composition»

Système retenu

Il est impossible d'attribuer une valeur pour chaque niveau de qualification et pour chaque critère, tous les critères ayant une importance plus ou moins forte selon le système. Malgré cela, sur deux critères les systèmes à base de modules se montrent plus en adéquation avec B : **la vision multiple** et **la facilité d'implantation** du système. Le premier critère est une caractéristique primordiale du système de modules de B. Sans cette possibilité, bon nombre de spécifications actuellement existantes seraient rejetées. Le deuxième critère est quant à lui gage de sécurité : partir dans une voie balisée semble dans le cas d'un développement d'un système de modules prudent et sage. Pour ces raisons, nous avons choisi

un système à base de modules ML et plus exactement le système à la Harper-Lillibridge-Leroy que nous allons présenter dans la section suivante.

3.4 Le système d'introduction de modules à la Harper-Lillibridge-Leroy

Notre choix d'un système de modules s'est orienté vers le système dit à la Harper-Lillibridge-Leroy, système implantant des modules SML. Nous allons dans cette section présenter ce système, nous commencerons par une présentation générale du système, pour ensuite aborder une présentation plus pragmatique : comment utiliser ce système et qu'obtient-on après son utilisation.

3.4.1 Présentation générale

Le système à la Harper-Lillibridge-Leroy (que nous appellerons par la suite HLL) est présenté dans [Leroy96] et [Leroy00a]. Ces travaux ont pour objectif de fournir une implantation d'un système de modules de type SML paramétrée par un langage de base et par son vérificateur de types. Cette paramétrisation rend facile la réutilisation de ce système pour différents langages de base. Les termes «langage de base» désignent un langage quelconque (nous verrons en fait les restrictions sur ce langage en 3.4.2) dépourvu de toutes constructions de modularité. L'utilisation de paramètres spécifiant le langage de base et le vérificateur de type associé se justifie par la relative indépendance qui existe entre les constructions modulaires d'un langage et ses autres constructions. Cette particularité du système est particulièrement intéressante pour notre démarche. En effet, HLL nous permet d'ajouter un système de modules sur le langage B de base, reste ensuite à vérifier l'adéquation du système obtenu avec l'actuel système de modules du langage B.

Attention ! *La présentation du système de modules HLL est fortement liée à la terminologie utilisée dans les langages fonctionnels de type ML, plus précisément, de même que X. Leroy, nous utilisons le langage OCaml, «dialecte» de cette même famille. Or cette présentation est (volontairement) de nature «constructive» car elle repose sur une implantation fonctionnelle du système de modules. Le terme «module» étant une construction primitive du langage fonctionnel utilisé, il est essentiel de ne pas confondre les «modules», concept que nous souhaitons ajouter à un langage basique, et les «modules» utilisés pour décrire la construction du système HLL.*

Au contraire de [Leroy00a], qui se veut une présentation didactique sur le système de modules HLL et sur son implantation mais également une preuve de la possibilité de réutiliser ce système pour de nombreux langages, notre présentation sera très ciblée sur l'utilisation de ce système. Les détails techniques que nous n'aborderons pas pourront être trouvés dans [Leroy00a].

Vision globale de l'utilisation du système

Comme nous l'avons déjà signalé, la mise en œuvre d'un système de modules pour un langage quelconque est relativement simple. Dans cette section, nous allons donner une vision globale de l'utilisation du système HLL.

Le système HLL utilise les foncteurs pour exprimer la paramétrisation dont nous parlons dans cette section. Nous rappelons qu'un foncteur est aux modules ce qu'une fonction est aux valeurs : un foncteur permet de construire un module à partir d'un autre module qui lui est passé en paramètre.

L'utilisation du système HLL nécessite de fournir des modules définissant le langage basique. C'est par l'utilisation des foncteurs définis dans [Leroy00a] qu'il est possible de construire l'intégralité du vérificateur de la sémantique statique. La figure 3.2 reprend le fonctionnement du système HLL ainsi que le flux d'informations nécessaire à la construction de la grammaire abstraite du nouveau langage et à la construction du vérificateur de types. L'utilisateur du système n'a à fournir que deux modules

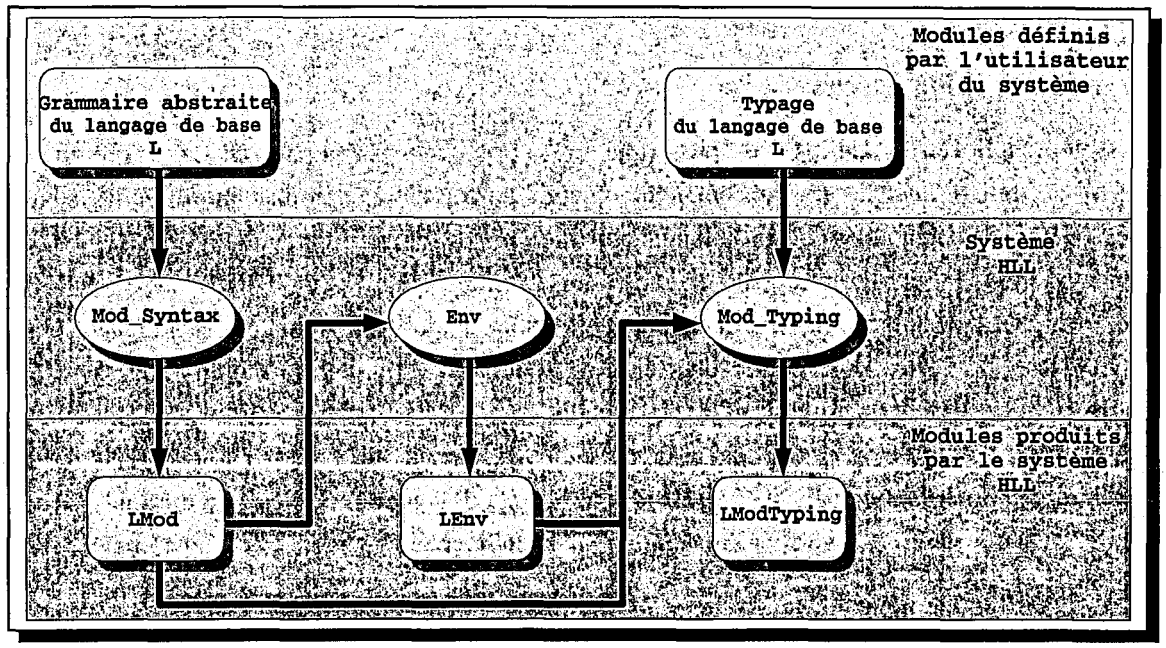


FIG. 3.2 – Utilisation du système de modules HLL

spécifiant d'une part la syntaxe abstraite du langage qu'il veut pourvoir de modules et, d'autre part, le module implantant le vérificateur statique pour ce langage. Le système HLL fournit, quant à lui, la syntaxe abstraite du langage muni de modules et le vérificateur statique de ce langage (plus un module destiné à stocker les informations nécessaires à cette vérification).

Exemple d'utilisation de HLL

Soient deux modules `L` et `LTyping` correspondant respectivement au langage basique L (L est ici un langage quelconque) et aux vérifications de types l'accompagnant.

Pour augmenter le langage basique des constructions de modularités du système HLL, il suffit de faire appel au foncteur `Mod_syntax` :

```
module LMod = Mod_syntax(L)
```

Pour effectuer les vérifications de type associées au langage, il faut construire un environnement dans lequel se feront ces vérifications ; pour cela, il faut faire appel à `Env` :

```
module LEnv = Env(LMod)
```

Il est maintenant possible de construire le module qui effectuera les vérifications par appel au foncteur

```
module LModTyping = Mod_Typing(LMod)(LEnv)(LTyping)
```

Pour l'utilisation du système HLL, le travail se situe donc uniquement au niveau de la spécification du langage basique.

Langage obtenu après instanciation

Le langage obtenu après instanciation de HLL est un langage possédant les caractéristiques suivantes :

- les modules peuvent être constitués soit de (sous-)modules, soit de valeurs, soit de types ;
- il est possible de définir des foncteurs paramétrés par des modules ;
- il est possible d'encapsuler les données soit par masquage des constituants, soit par définition de type ou de module abstrait (seule leur représentation est alors masquée) et
- il est possible de définir des types et des modules manifestes pour le partage des données (l'exportation concerne également la représentation du type ou du module dans ce cas).

3.4.2 Spécification du langage de base

Syntaxe abstraite

Pour utiliser le système HLL, il faut que la syntaxe abstraite du langage basique respecte la signature suivante[3] :

HLL : CORE_SYNTAX

```
module type CORE_SYNTAX =
  sig
    type term
    type val_type
    type def_type
    type kind
    val subst_valtype: val_type -> Subst.t -> val_type
    val subst_deftype: def_type -> Subst.t -> def_type
    val subst_kind: kind -> Subst.t -> kind
  end
```

Le type `term` correspond à la syntaxe abstraite des entités qu'il est possible de définir. Ces entités seront les variables, les constantes, les opérations, ...Ces entités auront pour type `val_type`. Enfin, `def_type` désigne les expressions de type qui peuvent être définies.

Les trois fonctions `subst_valtype`, `subst_deftype` et `subst_kind` sont utilisées pour effectuer des substitutions sur les identificateurs. En effet, le système HLL repose sur une vision des identificateurs (des `term` ou des types) basée sur le type `path` suivant :

[3] Le code du système HLL présenté est extrait de l'annexe [Leroy00b]

HLL : type path

```
type path =
    Pident of Ident.t          (* identifier *)
  | Pdot of path * string      (* access to a module component *)
```

L'utilisation de ce type rend nécessaire la substitution des identificateurs par leur chemin complet d'accès. Par exemple dans le code suivant :

Exemple module

```
module Exemple =
  struct
    type t = Tete of int | Queue of t | Vide
    let a = Vide
  end
```

le type de a ne doit pas être vu de l'extérieur comme t, mais comme Exemple.t. Les fonctions de substitutions vont permettre cette transformation. Plus de détails peuvent être trouvés dans [Leroy00a, parties 2.3 et 2.4].

Vérifications statiques

Après avoir fourni la syntaxe basique du langage, il faut fournir le module de vérification de la sémantique statique ; ce module doit suivre la signature suivante :

HLL : CORE_TYPING

```
module type CORE_TYPING =
  sig
    module Core: CORE_SYNTAX
    module Env: ENV with module Mod.Core = Core
(* Typing functions *)
    val type_term: Env.t -> Core.term -> Core.val_type
    val kind_deftype: Env.t -> Core.def_type -> Core.kind
    val check_valtype: Env.t -> Core.val_type -> unit
    val check_kind: Env.t -> Core.kind -> unit
(* Type matching functions *)
    val valtype_match: Env.t -> Core.val_type -> Core.val_type -> bool
    val deftype_equiv:
          Env.t -> Core.kind -> Core.def_type -> Core.def_type -> bool
    val kind_match: Env.t -> Core.kind -> Core.kind -> bool
    val deftype_of_path: path -> Core.kind -> Core.def_type
  end
```

Le module Env garde en mémoire l'environnement nécessaire pour effectuer les vérifications. La fonction type_term type un terme dans un environnement donné. check_valtype vérifie la bonne formation d'un type. valtype_match, deftype_equiv et kind_match testent la compatibilité de deux types ou deux *kinds*.

L'environnement du typage

Pour la compréhension de la suite de ce chapitre, il faut également rappeler la spécification de l'environnement qui sera utilisé pour exprimer les vérifications statiques :

Le foncteur Env

```
module type ENV =
  sig
    module Mod: MOD_SYNTAX
    type t
    val empty: t
    val add_value: Ident.t -> Mod.Core.val_type -> t -> t
    val add_type: Ident.t -> Mod.type_decl -> t -> t
    val add_module: Ident.t -> Mod.mod_type -> t -> t
    val add_spec: Mod.specification -> t -> t
    val add_signature: Mod.signature -> t -> t
    val find_value: path -> t -> Mod.Core.val_type
    val find_type: path -> t -> Mod.type_decl
    val find_module: path -> t -> Mod.mod_type
  end
```

Pré-requis sur le «langage de base»

Nous avons qualifié le langage de base qui peut être passé en paramètre du système HLL de quelconque. Nous avons déjà signalé qu'il doit en fait être exprimable d'une certaine manière, car sa syntaxe abstraite et sa vérification statique doivent être de la forme de celles présentées précédemment.

En plus de ces «contraintes», le langage de base impose que les catégories syntaxiques soient de deux sortes uniquement : des valeurs ou des types. L'instanciation construit une troisième catégorie, les modules. Une autre restriction concerne la définition de ces entités : il n'est pas possible de faire à la fois une définition dans l'une des catégories et dans l'autre catégorie. Par exemple, dans la construction

```
type t = Tete of int | Queue of t | Vide
```

de l'exemple de module précédent, le type `t`, qui est de catégorie syntaxique type, est défini, mais les constructeurs `Tete`, `Queue` et `Vide`, qui sont de la catégorie syntaxique valeur, sont également définis en même temps. Cette construction n'est pas permise dans un langage résultant de l'instanciation de HLL.

La dernière restriction est que les vérifications statiques nécessaires puissent s'exprimer comme des vérifications de types.

3.5 Un système HLL pour B ?

Après avoir présenté le système HLL, il est possible de se poser une question : ce système est-il adapté pour modéliser le système de modules de B ?

De prime abord, la réponse serait plutôt négative. En effet, HLL est tout d'abord destiné à ajouter des modules à un langage qui en est dépourvu. Or dans notre proposition, nous cherchons à modéliser un système de modules existant en conservant la même sémantique. De plus, le langage B est différent des langages de programmation classiques, il intègre des constructions indéterministes, des propriétés..., ce qui peut *a priori* rendre plus difficile de décliner HLL pour B.

Ensuite, les mécanismes de décomposition de B paraissent différents de ce qui existe en HLL. Le nombre de possibilité de composition est un exemple illustratif : pour B nous avons présenté quatre liens de composition ayant chacun une utilisation particulière alors qu'avec HLL, une seule manière d'accéder à un autre module existe.

Enfin, B utilise le raffinement qui est un concept n'existant pas dans les langages de programmation courants. Cet aspect du langage B peut paraître également un frein à l'expression de la modularité de B par un système à la HLL.

La séparation nécessaire du langage de base par rapport au langage de modules dans le système HLL permet de lever le doute concernant la particularité du langage B (pour les expressions indéterministes et l'expression des propriétés).

Le travail concernant la présentation de B sous un nouvel aspect nous permet d'affirmer que bien que les liens de composition B aient chacun leur utilité, ces liens ne sont pas différents du lien de composition existant dans le système HLL. Comme nous l'avons déjà signalé en 3.3.1, cela s'exprimera par des signatures de modules différentes suivant le lien de composition présent dans la spécification B originelle.

La dernière difficulté est la prise en compte de la notion de raffinement entre les composants B. Cette notion n'existant pas en HLL, nous considérerons chaque composant comme un module B-HLL (l'instanciation de HLL par le langage B) dans lequel nous ajouterons une dépendance vers son abstraction. Nous verrons dans le chapitre 4 que le raffinement n'est pas préoccupant dans le cadre de la génération de code, car il peut être éliminé.

Un exemple du résultat attendu pour la traduction d'une pile bornée est présenté dans la figure 3.3. Dans cet exemple, le prédicat exprimant la contrainte sur les paramètres n'a pas été placé dans la signature du module passé en paramètre. En effet, la vérification que la valeur `stac_size` vérifie bien la contrainte ne peut s'exprimer par une règle de vérification de type. L'intégration de la contrainte dans la signature pourrait se faire moyennant la modification du foncteur `Mod_Typing` pour que lors de la vérification de la compatibilité des types (attendus par le foncteur et fournis par le module passé en paramètre) une obligation de preuve soit générée pour la vérification de la contrainte. Cela fera partie d'une future amélioration du système B-HLL, système que nous allons présenter dans la section suivante.

3.6 Le système B-HLL

Nous venons de voir comment doter un langage quelconque de modules HLL. Il nous reste à transcrire le système de modules de *B* dans ce système à la HLL. Une première version de l'instanciation de HLL est présentée dans [Petit et al.02a]. L'instanciation que nous allons présenter ici est une évolution de ce travail. Cette évolution réside essentiellement dans la prise en compte des catégories syntaxiques

```
MACHINE
    stack(stack_size)
CONSTRAINTS
    stack_size : ℕ ∧ stack_size ≥1
 ∧ stack_size ≤MAXINT
VISIBLE_VARIABLES
    the_stack, stack_top
INVARIANT
    the_stack : (1..stack_size) →ℕ ∧ stack_top : ℕ
 ∧ stack_top ≥0  ∧ stack_top ≤stack_size
INITIALISATION
    the_stack : : (1..stack_size) →ℕ || stack_top := 0
OPERATIONS
    push(addval) =
        PRE
            stack_top <stack_size ∧ addval ∈ ℕ
        THEN
            stack_top := stack_top + 1
          || the_stack(stack_top + 1) := addval
        END
    ...
    END
```

```
module Stack =
  functor (sig val stack_size : NAT
           end) ->
  struct
    let constraints =
      stack_size : NAT &
      stack_size >= 1 &
      stack_size <= MAXINT
    let the_stack =
      (::) ((1..stack_size) --> NAT)
    let stack_top = 0
    let invariant =
      the_stack : (1..stack_size)-->NAT &
      stack_top : NAT & stack_top >= 0 &
      stack_top <= stack_size

    let push addval =
      PRE
        stack_top < stack_size &
        addval : NAT
      THEN
        stack_top := stack_top + 1 ||
        the_stack(stack_top + 1) := addval
      END;
    ...
  end
```

FIG. 3.3 – Pile bornée : spécification B et code B-HLL

que nous avons présentées en 3.2.3.

Le langage B que nous allons considérer est le langage B reconnu par les analyseurs du sous-projet BCaml ([BCaml]) du projet BRILLANT ([Brillant]). Ce langage est très proche du langage reconnu par l'Atelier B, mais diffère sur certains points comme par exemple la gestion des définitions pour lesquelles nos outils sont plus restrictifs.

3.6.1 Définitions du langage de base

Dans la syntaxe abstraite du langage, nous devons tenir compte des remarques sur les catégories syntaxiques de B que nous avons exprimés dans la section 3.2.3. Ces différentes catégories syntaxiques vont être utiles lors de l'expression du vérificateur de type qui sera présenté en 3.6.2. Deux axes sont importants pour cette prise en compte, chacun de ces axes correspondant à une dimension des tableaux 3.2.2 et 3.2.3. C'est cette nouvelle vision que nous allons utiliser dans la suite. Les types qui vont suivre permettront de prendre en compte l'axe des colonnes et le type `val_type` prendra en compte l'axe correspondant aux lignes.

term

```
type term =
    Invariant of predicate
  | ConstantConcrete of btype * expr
  | ConstantAbstract of btype
```

```
  | Properties of predicate
  | VariablesConcrete of btype * substitution
  | VariablesAbstract of btype * substitution
  | Assertions of expr
  | Operations of id list * (id*btype) list * substitution
```

Les informations de type qui se retrouvent dans les constructions correspondent aux informations présentes dans les spécifications. Celles-ci sont en B détachées de la définition des entités, mais dans notre vision du langage, nous ramenons ces informations auprès de l'entité.

Comme pour les lignes des tableaux, chaque colonne nécessite une différenciation au niveau de la syntaxe abstraite du langage. Par exemple, pour les prédicats, la syntaxe abstraite est :

Predicate

```
type predicate =
    Pred_Constraints of pred
  | Pred_Properties of pred
  | Pred_Loop of pred
  | Pred_Assertions of pred
  | Pred_Invariant of pred
```

Cette décomposition du type `predicate` permettra de distinguer dans les règles de typage les prédicats définis dans la clause `CONSTRAINTS` de ceux définis dans la clause `PROPERTIES`. Cela permettra d'autoriser l'accès aux paramètres formels d'un module B dans la première clause mais pas dans la seconde.

Au niveau des substitutions, la seule différence nécessaire concerne les niveaux d'abstractions : soit concret, soit abstrait (les raffinements étant inclus dans la partie abstraite).

Substitutions

```
type substitution =
    Subst of subst
  | SubstConcrete of subst
```

Les types utilisés en B et l'algorithme de typage sont repris des travaux de l'IRIT ([Bodeveix et al.02]) que nous avons présentés dans le chapitre précédent.

B types

```
type btype =
    PredType of expr
  | PFunType of btype * btype
  | Fin of btype
  | Untyped
  | Z
  | Bool
  | String
  | ProdType of btype * btype
  | TypeNatSetRange of expr * expr
```

```
  | TypeIdentifier of path * id list
  | RecordsType of field list
  | FunType of btype * btype
  | Sequence of btype
  | SubType of btype * expr
  | Operation of btype list * btype list
  | Unit
```

À partir de ces types, nous pouvons définir les types des entités présentes dans une spécification B de la manière suivante :

val_type

```
type val_type =
  | Var_Concrete of btype
  | Var_Abstract of btype
  | Cons_Concrete of btype
  | Cons_Abstract of btype
  | Predicate
  | Operation of btype list * btype list
```

La nécessité de décomposer en sous catégories syntaxiques s'exprime dans cette construction. Cette décomposition, suggérée par Xavier Leroy dans sa présentation du système HLL permet de prendre en compte les règles de visibilité exprimées dans les tableaux 3.6 et 3.7. À chacune des lignes de ces tableaux correspond une sous-catégorie syntaxique.

La dernière construction nécessaire à la définition de la syntaxe abstraite du langage B est l'expression des types définissables en B : les ensembles abstraits et les ensembles énumérés :

def_type

```
type def_type =
    SetAbstract of expr option
  | SetEnumerate of id list
```

3.6.2 Vérifications statiques

La seconde étape, après avoir spécifié la syntaxe abstraite du langage, est la spécification de sa sémantique statique. Nous ne présenterons pas ici l'intégralité des règles de typage, nous nous limiterons à celles concernant l'expression des règles de visibilité du langage B.

Notation

Pour l'expression des règles de typage, nous allons utiliser les notations suivantes :

- Var_visible(_) désignera n'importe quel type construit à partir du constructeur Var_visible (le _ étant utilisé comme joker). Dans l'algorithme de typage, cela se traduira par l'utilisation de reconnaissance de motifs. Bien évidemment, ceci se généralise à tous les constructeurs des différents types définis précédemment.

- Γ désigne l'environnement de typage où sont stockées les informations.
- Une règle de typage est habituellement exprimée à l'aide de triplets (Γ,e,t) qui signifient : dans l'environnement Γ l'expression e est de type t. Ces triplets seront notés $\Gamma \vdash e : t$. Dans les règles que nous allons exprimer, à la place de e, on trouvera les constructions du type `term`, et à la place de t celles du type `val_type`. Cette représentation en triplet ne suffit pas à exprimer nos règles de typage, il nous faut également une notion de contexte nous permettant d'exprimer nos règles de visibilité. Ce contexte s'exprimera par une liste de conditions devant être vérifiées par tous les identificateurs. $\Gamma, \lceil contexte \rfloor \vdash c : t$. Pour l'implantation, le contexte correspond à une fonction des types des identificateurs (soit `val_type`, soit `def_type`) vers les booléens. Par exemple, lors d'un accès à une variable concrète (de type `Var_Concrete(_)`), la fonction de contexte devra renvoyer faux si cet accés se fait dans un prédicat exprimant une `PROPERTIES` (voir la règle de typage ci-dessous).
- **TC(X)** signifie que X (une expression, une variable, ...) est correctement typé.
- **TYPE(X)** dénote le type de X. X peut être une liste d'identificateurs, dans ce cas, une liste de type sera renvoyée. Ceci sera utile pour les constructions B comme l'affectation multiple : $(XX,YY := 2,3)$ pour vérifier la compatibilités des types des deux cotés du :=.
- $\mathbf{local}_\Gamma$**(X)** (respectivement $\mathbf{extern}_\Gamma$**(X)**) signifie que dans l'environnement Γ, l'entité X a été définie localement au module courant (respectivement dans un autre module que le module courant).

Les règles de visibilité des deux tableaux 3.6 et 3.7 se traduisent simplement par des règles qui auront la forme suivant :

$$\text{(Properties)} \quad \frac{\Gamma, \lceil properties_{ctx} \rfloor \vdash TC(p) \quad \Gamma, \lceil properties_{ctx} \rfloor \vdash TYPE(p) = Predicate}{\Gamma, \lceil c \rfloor \vdash Properties(p) : Predicate}$$

La règle précédente concerne la colonne Properties des tableaux. Elle sert à vérifier le type du prédicat qui forme le contenant de la clause B `PROPERTIES`. Les autres colonnes se traitent suivant la même technique. Le contexte sera formé des conditions d'accessibilité définie dans le tableau 3.6, c'est-à-dire que seuls les ensembles et les constantes sont accessibles.

En plus des tableaux, deux règles ont été définies en 3.2.3 pour le principe d'encapsulation. Ceci a deux conséquences majeures : la première conséquence est qu'une variable d'un composant (et donc l'état d'un composant) ne peut être modifiée en dehors de ce composant. Ce principe d'encapsulation est appliqué partiellement en B car l'accès direct en lecture est quant à lui autorisé. La seconde conséquence est qu'une opération ne peut être utilisée dans le composant dans lequel elle a été définie. Pour répondre à ces deux contraintes, il est nécessaire d'adapter deux règles du typage.

La première règle est celle de l'affectation. Celle-ci se divise en deux règles, la première dans le cas où la partie gauche du := n'est pas encore typée et la seconde dans le cas contraire. Pour exprimer ces règles, il est nécessaire d'introduire une notion de propagation de contexte. En effet, si la variable (partie droite) d'une affectation n'est pas typée, la règle de typage de l'affectation doit inférer que le type de la

variable est celui de l'expression (partie droite) de l'affectation. Le nouvel environnement de typage sera noté $\{\Gamma + (X : t)\}$ dans la règle. Ceci nous donne pour la première des deux règles du typage du := :

$$(:= 1) \quad \frac{local_\Gamma(XX) \qquad \Gamma, \lceil c \rfloor \vdash TYPE(XX) =? \qquad \Gamma, \lceil c \rfloor \vdash TYPE(EE) = t}{\Gamma, \lceil c \rfloor \vdash TC(XX := EE) \qquad \{\Gamma + (XX : t)\}}$$

Dans cette règle $TYPE(XX) =?$ signifie que le type de XX n'est pas connu.

$$(:= 2) \quad \frac{local_\Gamma(XX) \qquad \Gamma, \lceil c \rfloor \vdash TYPE(XX) = TYPE(EE)}{\Gamma, \lceil c \rfloor \vdash TC(XX := EE)}$$

Dans la seconde règle, correspondant au cas où le type de la variable de la partie gauche du := est connu, il suffit de vérifier que des deux côtés du := les types sont compatibles.

Dans les deux règles, pour vérifier le typage de $XX := EE$, il faut vérifier que XX est local au composant ($local_\Gamma(XX)$).

Le même principe que celui appliqué pour le := peut être appliqué pour les appels d'opérations :

$$\text{(OperCall)} \quad \frac{extern_\Gamma(opername) \qquad \Gamma, \lceil c \rfloor \vdash (TYPE(p\_in) = TYPE(A\_in)) \wedge (TYPE(p\_out) = TYPE(A\_out))}{\Gamma, \lceil c \rfloor \vdash TC(opername(p\_in, p\_out))}$$

Un appel d'opération est correct vis-à-vis du typage si le nom de l'opération est défini en dehors du composant courant et si les expressions passées en paramètres (*A_in* et *A_out*) sont bien du même type que les paramètres utilisés lors de la définition de l'opération (*p_in* et *p_out*).

3.6.3 Les ensembles énumérés

Les ensembles énumérés B transgressent la restriction concernant l'impossibilité de définir à la fois une entité de la catégorie des valeurs et une entité de la catégorie des types. Pour résoudre ce problème, le module d'environnement du typage doit être retravaillé afin d'autoriser de telles constructions. La fonction `add_type` fait désormais à la fois un ajout dans l'environnement des types et un ou plusieurs ajouts dans l'environnement des valeurs.

3.6.4 Interfaces de modules

Pour chaque module B-HLL, il va falloir fournir quatre signatures, une pour chaque lien de liaison existant en B. Ces signatures suivront le tableau 3.8.

Nous allons présenter comment les composants B vont être transformés en module B-HLL. Nous adopterons une représentation proche de celle de OCaml pour représenter les modules B-HLL. Ceux-ci n'ont en fait pas d'existence réelle car ils sont obtenus par transformation de spécifications B en spécifications B-HLL, le tout par transformation de syntaxe abstraite. Il est néanmoins possible d'envisager écrire directement des modules B-HLL, il suffirait pour cela de coupler l'analyseur syntaxique de OCaml, partie concernant la modularité et l'analyseur de syntaxe fourni par la plateforme BCaml.

Nous allons prendre pour exemple deux composants du projet passage à niveaux que nous présenterons dans le chapitre suivant. Ces deux composants vont être le composant modélisant les capteurs sur la voie détectant l'entrée du train dans la zone du passage à niveaux et sa sortie de cette zone et le composant de supervision des actions à faire. La spécification abstraite est la suivante :

Spécification abstraite des Capteurs

MACHINE
capteurs
CONSTANTS
nb_max
PROPERTIES
nb_max ∈ NAT1
VARIABLES
origine_annonce, nb_train, ...
INVARIANT
origine_annonce ∈ BOOL ∧ nb_train ∈ NAT ∧ ...
INITIALISATION
origine_annonce := FALSE || nb_train := 0 || ...
OPERATIONS
detection_origine_annonce =
PRE origine_annonce_ok = TRUE ∧ nb_train < nb_max - 1
THEN origine_annonce := TRUE || nb_train := nb_train + 1
END ;

bb ←— Etat_Origine_Annonce =
bb := origine_annonce ;

/* Les autres opérations sont masquées*/
END

Cette spécification abstraite va être transformée pour obtenir le module B-HLL suivant :

Signature SEES pour le module capteurs

```
module type Capteurs_SEES =
  sig
    val nb_max : nat1
    val origine_annonce : bool
    .../* le reste des variables abstraites */
    val etat_Origine_Annonce : unit -> bool /* la seule opération de consultation */
  end
```

Cette signature de module correspond à la vision qu'auront d'autres modules accédant aux capteurs par le biais d'un lien **Sees**. Le module Controle voit justement le module Capteurs. Cela va se traduire dans sa spécification B par :

En-tête du composant control

MACHINE
control
SEES
Train, communication, capteurs ...

Cette en-tête de spécification se traduira dans le système B-HLL par :

Control B-HLL module

```
module Control =
  module Train = (Train : Train_SEES)
  module Communication = (Communication : Communication_SEES)
  module Capteurs = (Capteurs : Capteurs_SEES)
  open Train
  open Communication
  open Capteurs
  ... /* le reste de la spécification ne change quasiment pas */
end
```

Chaque liaison avec un autre module se ramène à une restriction sur ce module. Dans le module `Controle`, une nouvelle construction apparaît, le `open`. Cette construction n'est pas une construction reconnue par une instanciation du système HLL. Comme pour le traitement des ensembles énumérés, le foncteur `Env` doit être retravaillé pour ajouter une fonction de recopie de l'environnement du module «ouvert» dans l'environnement du module l'ouvrant. Avec cette fonction, l'utilisation de la directive `open` permettra l'utilisation de données d'un autre module sans la notation pointée.

Cette construction `open` peut également être utile à la simulation du raffinement (nécessaire pour la vérification de type, certaines entités référencées dans un composant provenant de l'abstraction). Pour ce faire, il suffit dans chaque raffinement d'ouvrir l'abstraction en utilisant une signature spécifique (exportant toutes les constructions sauf les opérations)

3.6.5 Paramètres de modules

Le dernier point important pour la transcription des composants B en module B-HLL est la prise en compte des paramètres des composants. Comme nous l'avons déjà signalé, cette prise en compte passe par l'utilisation de foncteur B-HLL. Ceci permet de passer en paramètre un module à un autre module. Il s'agit donc ici de construire un module constitué des paramètres du composant B originel.

Pour illustrer le résultat de la transformation dans ce cas, nous allons prendre la machine `Cycle_R_1` du projet spécifiant la gestion d'une chaudière (projet intitulé BOILER, les spécifications de ce projet sont présentées dans le BBook [Abrial96]). La spécification (son en-tête) de cette machine est :

En-tête du composant Cycle_R_1

```
MACHINE       Cycle_R_1(pump_number)
CONSTRAINTS   pump_number : NAT
...
END
```

Cet en-tête sera traduit en B-HLL par :

En-tête du module B-HLL Cycle_R_1

```
module Cycle_R_1 =
  functor (PARAM : sig val pump_number : NAT end) ->
    struct
        open PARAM
        ...
end
```

3.7 Conclusions

Exprimer la modularité de B en utilisant un système HLL peut paraître, comme nous l'avons signalé, ardu. Ce qui a rendu possible l'instanciation du système d'ajout de modules HLL est le travail préliminaire qui a été effectué sur la compréhension du système de modules de B. Cette présentation classique au départ a évolué vers une présentation reposant sur les catégories syntaxiques des éléments du langage B qui nous a permis d'obtenir le système B-HLL. Cette meilleure compréhension du système de modules de B est déjà en soi un apport pour la méthode, car jusqu'à présent, la modularité de B été un aspect rebutant car souvent mal présentée. Il est possible désormais de séparer les aspects modularités des aspects langage de base de B, ce qui permet de mieux cibler les difficultés.

De cette séparation des concepts de modularité et de ceux du langage de base B issue de la modélisation de B-HLL, nous allons pouvoir envisager plus sereinement la génération de code. En effet, la séparation des concepts va se répercuter au niveau de la génération de code et il va être possible de l'envisager en deux phases, d'une part la traduction des modules B en modules (modules est ici utilisé dans son sens large : unité de composition) du langage cible, et d'autre part, la traduction du langage de base B. Nous allons dans le chapitre suivant détailler ces points.

Un avantage de l'utilisation du système HLL est l'obtention «quasi gratuite» d'un vérificateur de sémantique statique pour le langage B doté du nouveau calcul des modules. Cet aspect est intéressant car notre approche se veut pragmatique, et donc l'outillage prend une part importante dans les objectifs visés.

Chapitre 4

Génération de composants

Sommaire

4.1 Introduction

Dans ce chapitre, nous allons aborder la génération de composants logiciels à partir de spécifications B. Un pré-requis indispensable pour ce travail est la maîtrise de la modularité que nous avons abordée dans le chapitre précédent. Notre système B-HLL découple le langage de base B et son langage de modularité, ce découplage va donc se répercuter au niveau de la génération de code ; il faudra traiter la génération de code sous ces deux aspects : modules et langage de base. Nous verrons également comment il est possible de générer des composants logiciels et l'impact que cela peut avoir au niveau du traitement des obligations de preuves.

Notre présentation de la génération de code va se décomposer en plusieurs parties. La première partie sera consacrée à la génération de code correspondant au langage de base et aux constructions de modularité. Dans la seconde phase, nous aborderons plus précisément l'aspect génération de composants logiciels et les extensions que notre vision de la génération de code permet d'aborder. Nous discuterons ensuite des avantages de notre processus de génération de code vis-à-vis du processus habituel. Nous aborderons dans la partie suivante plus particulièrement un des avantages de notre approche en examinant son impact au niveau du processus de développement B. Dans la dernière partie, nous reprendrons quelques exemples déjà utilisés dans les chapitres précédents afin d'illustrer la génération de code.

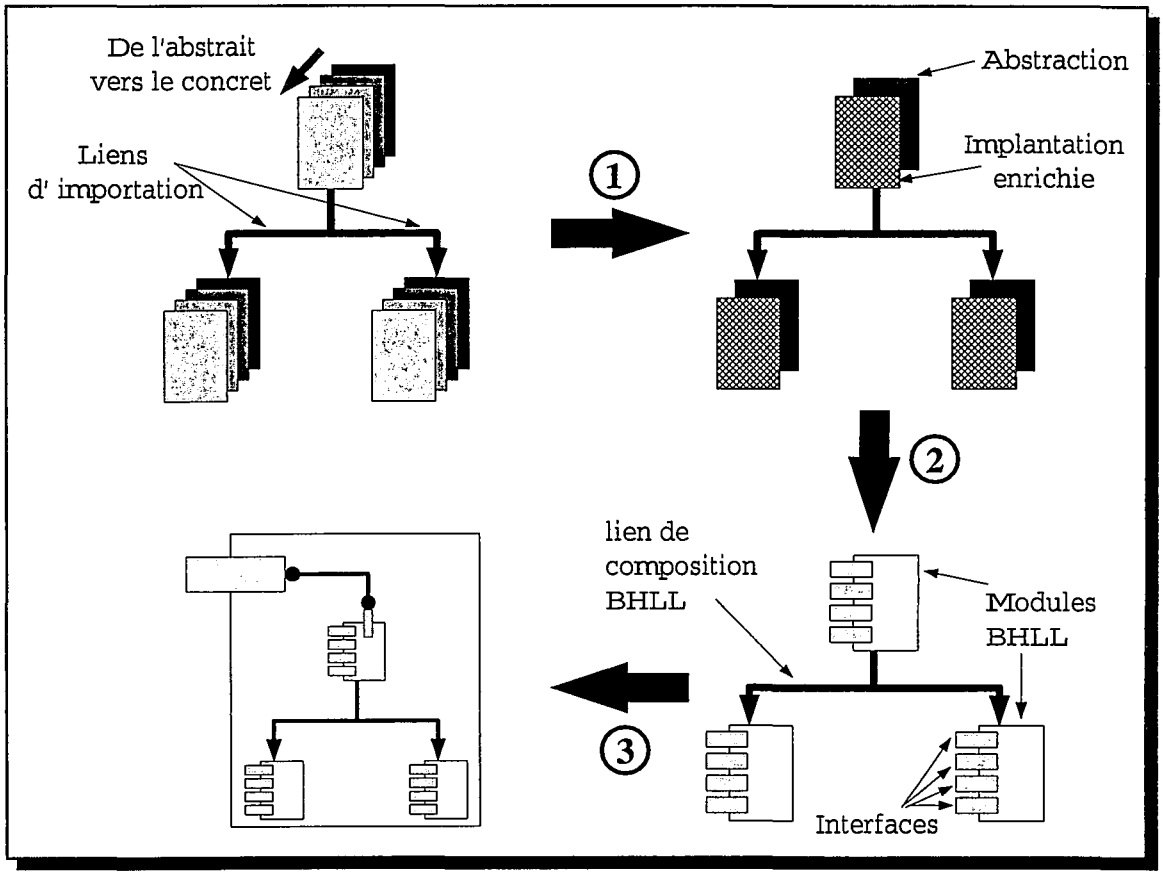


FIG. 4.1 – Processus de génération de code : des spécifications B aux composants

La figure 4.1 représente une vision globale du cheminement et des transformations effectuées dans le processus de génération de code. Comme nous le verrons, les étapes (1) et (2) peuvent être interverties.

4.2 Génération de code

La génération de code a deux facettes, d'une part, l'aspect classique de construction des données et des structures de contrôle, et d'autre part, un aspect nouveau et original dans la génération de code pour la méthode B, le transport des propriétés dans le code généré.

4.2.1 Collecte des informations

La génération de code nécessite l'identification des constituants du code : la traduction de l'état de la spécification B (les variables et les constantes), la traduction des types manipulés et la traduction des structures de contrôle.

Notation : Φ_{XXX} dénote la fonction de transformation des constructions B en constructions du langage cible XXX.

Traduction des structures de contrôle

Parmi ces constituants, la partie la plus facile est la traduction des structures de contrôle des spécifications B. Ces structures de contrôle correspondent à celles disponibles dans le langage B0 de B. Ce langage B0 correspond à un noyau qui se retrouve habituellement dans les langages impératifs. La transformation des structures de contrôle dans des langages comme C, Ada, OCaml, ... langages possédant ce même noyau impératif ne pose donc que peu de problèmes. Le tableau 4.1 illustre la transformation pour le langage cible OCaml.

Une opération B se traduira en OCaml par une fonction. Afin de conserver un maximum d'expressivité du code généré au niveau du profil de la fonction générée, les paramètres de sorties d'une opération B seront traduits en un n-uplet.

Traduction de l'état d'une spécification

Autant pour la traduction des structures de contrôle les informations sont présentes dans l'implantation B, autant pour déterminer les variables et les constantes, il est nécessaire de prendre les informations dans toute la chaîne de raffinement du module B à traduire. Par exemple, les variables ou les constantes concrètes définies dans une abstraction doivent apparaître dans le code généré sans pour autant apparaître explicitement dans l'implantation. Cet enrichissement correspond à ce qui est fait dans l'algorithme de dépliage du lien `REFINES`. Il est donc possible d'extraire de l'algorithme de dépliage la partie portant sur l'enrichissement du lien `REFINES` pour éliminer les différents niveaux de raffinement. Ceci correspond à l'étape (1) dans la figure 4.1. Cette étape peut être permutée avec l'étape suivante, car la construction de l'état complet de la spécification B peut se faire aussi bien sur les spécifications B directement que sur les spécifications B-HLL.

Nom de la substitution	Substitution	Traduction
Pré-condition	PRE P THEN S END	$\Phi_{OCaml}(S)$
Assertion	ASSERT P THEN S END	$\Phi_{OCaml}(S)$
Bloc	BEGIN S END	begin $\Phi_{OCaml}(S)$ end
Conditionnelle	IF P THEN S1 ELSE S2 END	if $\Phi_{OCaml}(P)$ then $\Phi_{OCaml}(S1)$ else $\Phi_{OCaml}(S2)$
Condition par cas	CASE Expr OF EITHER T1 THEN S1 (OR Ti THEN Si)* [ELSE S2] END	match $\Phi_{OCaml}(Expr)$ with $\Phi_{OCaml}(T1)$ -> $\Phi_{OCaml}(S1)$ (| $\Phi_{OCaml}(Ti)$ -> $\Phi_{OCaml}(Si)$)* | _ -> $\Phi_{OCaml}(S2)$
Variable locale	VAR XX IN I ;S END[1]	let $\Phi_{OCaml}(XX)$=$\Phi_{OCaml}(I)$ in $\Phi_{OCaml}(S)$
Tant que	WHILE C DO S INVARIANT P VARIANT Expr END	while $\Phi_{OCaml}(C)$ do $\Phi_{OCaml}(S)$
séquence	S1 ; S2	$\Phi_{OCaml}(S1)$; $\Phi_{OCaml}(S2)$
devient égal	XX := Expr	$\Phi_{OCaml}(XX)$:= $\Phi_{OCaml}(Expr)$
Appel d'opération	XX <– op(YY)	$proj(\Phi_{OCaml}(XX),\Phi_{OCaml}(op(YY)))$

TAB. 4.1 – Transformation des substitutions déterministes

Traduction des types B

Les types autorisés pour les données concrètes sont limités par la méthode B à un sous-ensemble des types B existants. Ces types sont appelés les types concrets. Ces limitations sont issues des langages cibles initiaux de la méthode B. Ceci est exprimé dans le manuel de référence B ([Steria98, section 3.4]) : «*Comme les données concrètes doivent pouvoir être implémentées par un programme, certaines contraintes ont été fixées pour différencier les données concrètes de celles qui ne le sont pas. Ces contraintes sont forcément arbitraires, mais elles ont été établies en considérant ce que savent implémenter facilement les langages de programmation comme Ada ou C++...*»

Les alternatives. Trois alternatives peuvent être mises en œuvre pour la traduction des types dans le code généré.

La première consiste à suivre ce qui est actuellement fait dans les générateurs des outils support de la méthode, c'est-à-dire ne prendre en compte que les types concrets.

La deuxième solution consiste à utiliser des machines de base supplémentaires pour étendre les types concrets. Les machines de base sont des machines non développées en B, seuls l'abstraction et le code sont fournis. Cette solution est la solution qui a été adoptée dans le projet BOM ([Badeau et al.03]) Cette solution a l'avantage de s'intégrer facilement dans le processus de génération de code, il ne s'agit que d'un lien vers un module dont le code est fourni. Un autre avantage est la possibilité de diriger la traduction de code. Dans [Badeau et al.03], ces machines de base sont utilisées pour choisir le type entier qui devra se retrouver au niveau du code (entier sur 16 ou 32 bits, signé ou non).

Une troisième approche s'apparente à la solution précédente, et consiste à remplacer certaines expressions de types par des types du langage cible. Les types concrets ont à l'origine été choisis en fonc-

tion des possibilités des langages cibles Ada et C++, si le langage cible est maintenant un langage plus développé au niveau des types de base, il peut être intéressant d'utiliser ces types de base dans le code généré (pour générer directement des séquences par exemple, sans être obligé de les raffiner en tableaux). Cette solution a l'avantage de ne nécessiter aucune intervention au niveau des spécifications B, mais son inconvénient vient justement de la non modification des spécifications B et donc de l'injection de transformations sans aucune validation au niveau des spécifications, cette validation se reportant au niveau de la génération de code.

Recommandation de choix. Ces trois alternatives peuvent facilement être mise en œuvre et cohabiter dans notre génération de code. Il n'y a pas véritablement de choix à faire en ce qui concerne l'alternative, chacune ayant un objectif précis. En effet, la première alternative est indispensable à la traduction des types de base comme les entiers, les chaînes de caractères, ... La seconde alternative sera utilisée dans les cas spécifiques où la génération de code doit être dirigée, indépendamment du langage cible. Au contraire, la troisième alternative est directement liée au langage cible, elle permet de prendre en compte les particularités de celui ci.

Utilisation du langage de type étendue. Nous avons présenté dans le chapitre 2 un langage de type étendue pour le langage B. Ce langage de type et l'algorithme de vérification de type associé ont été utilisés dans la modélisation HLL du langage B. Cette vision étendue va être utile au moment de la génération de code afin de générer des entités avec un type le plus proche de celui exprimé dans les spécifications. Ainsi si une variable v est de type intervalle `1..10`, si le langage cible possède la possibilité d'exprimer des données de type intervalle, la variable v sera bien du type `1..10` dans le code généré. Mais si le langage cible ne possède pas cette construction, il sera possible de définir la variable v comme étant de type entier (ceci correspond à ce qui est actuellement fait dans les générateurs de code) en ajoutant la contrainte $\boxed{v \geq 1 \wedge v \leq 10}$ aux assertions (dans l'invariant et/ou après chaque instruction modifiant la valeur de v) générées dans le code.

4.2.2 Les assertions

Motivations

La conservation des propriétés des spécifications jusque dans le code est une nouveauté pour la méthode B. En effet, jusqu'à présent, le code généré est censé être prouvé correct, et donc son exécution être sans erreur. Or, comme nous l'avons signalé dans le chapitre 1 section 1.2, cette vision idéale du monde est trop «optimiste». Le code généré s'exécute dans un environnement dont les caractéristiques n'ont pas été totalement modélisées, cette modélisation étant impossible car trop complexe ; comment modéliser le processeur sur lequel le code va être exécuté, la mémoire, ... L'approche par contrats présentée en 1.4 est une technique intéressante du point de vue développement logiciel et déploiement de ce logiciel. Cette technique repose sur des contrats. Or, ces contrats sont exprimés dans les spécifications B, il est donc judicieux de les conserver dans le code.

Mise en œuvre

L'ajout des assertions dans le code généré repose sur les mêmes principes que la génération des entités (variables et constantes) et des structures de contrôle vues précédemment (section 4.2.1) : la collecte des propriétés. Les propriétés intéressantes à conserver pour la génération de code sont les invariants (de composants et de boucles) ainsi que les pré-conditions. La notion de post-condition n'existant pas en B, elle n'apparaîtra pas au niveau du code généré (pour le moment, nous verrons dans la section 4.3 l'intérêt potentiel de les utiliser).

Les pré-conditions. Les pré-conditions qui vont être transcrites dans le code seront les pré-conditions «dépliées», c'est-à-dire celles obtenues après application de l'algorithme de dépliage sur la chaîne de raffinement formant le module sur lequel la génération de code porte. D'après le principe d'affaiblissement de la pré-condition utilisé en B, il peut sembler inutile de conserver d'autres contrats que ceux exprimés dans l'abstraction. Or, ces contrats portent sur un état abstrait du composant et les variables abstraites constituant cet état sont amenées à évoluer, voire disparaître dans les raffinements. Les contrats doivent, dans le code, porter sur l'état concret du module, sans pour autant refléter l'affaiblissement des pré-conditions.

Les invariants. Les invariants sont également obtenus par application de l'algorithme de dépliage.

Limitations. Les limitations de cette approche visant à produire des contrats dans le code généré viennent des langages cibles eux même. En effet, même si l'approche par contrats s'est fortement développée ces dernières années, tous les langages ne possèdent pas un langage des prédicats assez riche pour exprimer toutes les propriétés B. Cela provient de l'absence de la prise en compte des quantificateurs existentiel et universel. Leur usage est fréquent dans les spécifications B et notre approche pour générer les contrats en fait apparaître (voir la section 2.6 sur l'aplatissement des spécifications B). Il est donc indispensable de viser des langages ayant un support d'assertions contenant les quantificateurs, tout au moins si l'approche par contrats veut être pleinement utilisée.

4.2.3 Les modules

Outre la génération du code concernant les variables, les constantes, les opérations et les assertions d'un module B, il faut également prendre en compte l'aspect composition. Cet aspect passe par l'utilisation du système B-HLL présenté dans le chapitre précédent (étape (2) dans la figure 4.1). Par l'utilisation de ce système, la génération de code se ramène à la transformation de la syntaxe abstraite des modules (sans le langage de base qui est traité par ailleurs) en langage cible. Cela se ramène à écrire la traduction de deux concepts en langage cible : les modules simples et les modules paramétrés appelés foncteurs.

La traduction des modules simples ne pose pas de problème car une majorité des langages actuellement utilisés fournissent des constructions de «modularité», que ce soit par le biais des classes, des paquetages ou autres. Par contre la traduction des foncteurs va elle poser des problèmes car ce concept n'est présent que dans peu de langages. Il faudra donc suivant les langages cibles adopter une technique différente.

Nous allons reprendre dans le tableau 4.2 les trois catégories de langages dont nous avons étudié la modularité dans le chapitre 3, les langages à objets, les langages modulaires et les langages utilisant les paquetages, et donner des solutions pour traduire les modules B-HLL. La traduction vers les langages à base de modules ML est clairement immédiate, la syntaxe abstraite de HLL étant très proche de celle de ces langages.

Module B-HLL	Objets	Packages	Modules
Non paramètré	Classe	paquetage (1)	module (1)
paramètré par des valeurs	Classe	paquetage générique	foncteur
paramètré par des types	(2)	paquetage générique	foncteur

TAB. 4.2 – Traduction des modules B-HLL en langage cible

Commentaires pour le tableau 4.2 :

- (1). La traduction en module (ou paquetage) ne laissera pas la possibilité d'instancier ce module (ou paquetage) plusieurs fois. Pour pouvoir effectuer cette recopie de module, il faudra générer un foncteur (paramétré par un module vide). La même technique pourra être utilisée pour les paquetages en utilisant les paquetages génériques.
- (2). La traduction de foncteurs B-HLL paramétrés par des types abstraits de données est une notion abordée différemment suivant les langages support. En C++, par exemple, l'utilisation de la construction `template` sera nécessaire.

4.3 Génération de composants

4.3.1 Dans B

Un composant logiciel a été défini dans le chapitre 1 comme une entité logicielle soumise à composition et possédant une caractérisation (les interfaces) spécifiant comment l'utiliser. Il existe actuellement une limite dans l'un des outils support de la méthode, l'Atelier B, la génération de code n'est possible que si il existe un module racine au projet, module non paramétré et ne dépendant d'aucun autre module. La seule notion de réutilisabilité existant actuellement en B est la réutilisation de parties de spécifications B développées dans le cadre d'autres projets. Donc seule une réutilisation interne à la méthode est possible. Cette limitation a un impact au niveau des systèmes produits, un développement B porte sur un sous-système complet et non sur des composants de granularité inférieure.

4.3.2 La paramétrisation : un premier pas

Permettre la paramétrisation du module racine de l'arborescence constituant le projet dont le code doit être généré est un premier pas vers la génération de composants logiciels. Cette paramétrisation est actuellement limitée en B à des valeurs et des types. Lors de notre modélisation du système de modules de B par des modules HLL, ce sont les foncteurs qui ont été utilisés pour simuler la paramétrisation des modules B. Cette modélisation nous permet d'étendre les possibilités de paramétrisation de B à de réels modules.

Prenons par exemple, l'exercice de recherche d'un élément dans une table. Il est intéressant de spécifier l'algorithme de recherche sans limiter les éléments apparaissant dans cette table à des entiers. Cette spécification est d'ailleurs un bon candidat à la génération d'un composant réutilisable, ou du moins comme faisant partie d'un composant plus général sur les structures de données par exemple. Le module B-HLL attendu serait du type de celui de la figure 4.2. En fait ce module B-HLL peut être exprimé en B

```
module Recherche_table :
  functor (TA_ELEMENT :
             sig
               type t
               val compare : t -> t -> INTEGER
               ...
             end)
  ->
    sig
      val ajout : TA_ELEMENT.t -> unit
      val recherche : TA_ELEMENT.t -> BOOL
      ...
    end
```

FIG. 4.2 – Spécification B-HLL de la recherche dans une table

classique par l'utilisation des liens INCLUDES et IMPORTS. Pour les composants abstraits, les spécifications sont présentées dans la figure 4.3.

MACHINE
recherche_table
INCLUDES
ta_element
...

OPERATIONS
ajout (element) =
PRE element ∈ T ...
THEN ...
END ;
resultat ⟵recherche (element) =
PRE element ∈ T ...
...

END

MACHINE
ta_element
CONSTANTS
element_neutre
SETS
T
...

OPERATIONS
resultat ⟵compare (elun, eldeux) =
PRE elun ∈ T ∧ eldeux ∈ T ...
THEN ... **END** ;
...
END

FIG. 4.3 – Spécification B de la recherche dans une table

La version concrète du composant `recherche_table` importerait simplement le composant `ta_element`.

Mais, même si cette spécification de la recherche dans une table est possible en B, la génération du code correspondant ne le sera pas. En effet, le module `ta_element` devra soit être entièrement développé en B soit posséder un code associé à l'abstraction pour que le code soit générable. Notre approche de la génération de code permet la génération de code à partir de spécifications que nous qualifierons d'ouvertes. L'aspect génération de code est un aspect important pour le développement de composant, un

autre aspect important est la preuve de ce composant. C'est ce que nous allons aborder dans la section suivante.

4.3.3 Validation modulaire

Preuves modulaires

Revenons sur la validation inter-module B. Cette validation correspond aux obligations de preuves 5, 6 et 7 de la section 2.4.3. Nous avons remarqué dans cette section sur la validation des relations entre composants que les appels d'opérations (d'autres composants) n'engendrent pas d'obligations de preuve particulières. En effet, un appel d'opération est remplacé par le corps abstrait de cette opération, ainsi, une opération constituée d'une série de substitutions `S1` suivie d'un appel à l'opération op (que nous considérons sans paramètres par souci de simplicité) suivie elle-même d'une autre série de substitutions `S2` sera soumise à l'obligation de preuve suivante (OP 5 section 2.4.3)[2] :

$$\Gamma \wedge Pre \Rightarrow [S1; op; S2]Inv$$

les opérations B sont de la forme `PRE P THEN S END` notée également `P|S`, ainsi la formulation précédente devient :

$$\Gamma \wedge Pre \Rightarrow [S1; P|S; S2]Inv$$

Par développement, cette obligation de preuve se transforme en :

$$\Gamma \wedge Pre \Rightarrow [S1; P|S]([S2]Inv)$$

puis :

$$\Gamma \wedge Pre \Rightarrow [S1](P \wedge ([S]([S2]Inv)))$$

Ce qui est équivalent à :

$$\Gamma \wedge Pre \Rightarrow [S1]P \wedge ([S1; S; S2]Inv)))$$

Cette obligation de preuve peut s'interpréter intuitivement par :

- avant l'appel d'une opération, le composant qui appelle cette opération doit être dans un état satisfaisant la pré-condition de l'opération appelée ($[S1]P$) et
- l'opération («appelante») doit établir l'invariant du composant ($[S1; S; S2]Inv$).

Pour la validation d'un composant dont la spécification fournit uniquement les informations de pré et de post-conditions, il faut transformer ces deux contrats en substitution. La substitution équivalente pour une opération dont la pré-condition est P, la post-condition Q et dont les paramètres de sortie sont désignés par E sera la substitution :

$$\textbf{PRE } P \textbf{ THEN ANY } X \textbf{ WHERE } T \wedge Q \textbf{ THEN } E := X \textbf{ END}$$

Cette substitution s'interprète par sous la pré-condition P, prend n'importe quel ensemble de variables X de type T vérifiant la post-condition Q et affecte ces valeurs de X aux variables de sortie E de l'opération. Le prédicat T qui apparaît dans cette substitution sera naturellement obtenu grâce au profil de l'opération. Ce prédicat servira à typer les données.

[2]L'obligation de preuve que nous donnons ici n'est pas correcte au sens strict B, car la construction de séquencement de substitution n'est pas une construction des machines abstraites alors que cette obligation de preuves correspond à l'obligation de preuves d'une machine abstraite. Nous cherchons ici à donner l'intuition de comment se fait la preuve d'un appel d'opération, l'ajout des parties constituant la preuve de bon raffinement n'apporte rien de plus à notre démarche, mais compliquerait la compréhension.

Spécification des contrats des composants

La nécessité de l'utilisation des post-conditions, en plus des pré-conditions, est mis en évidence dans le paragraphe précédent. En B, même si la donnée du corps d'une opération correspond à la donnée d'une post-condition, il semble judicieux de fournir en plus de cette post-condition une post-condition caractérisant uniquement les relations entre les paramètres d'entrée et ceux de sortie d'une opération. Cela permettra de différencier les conditions nécessaires à la validation «en interne» de la validité de l'opération et celles nécessaires pour valider «en externe» l'utilisation de cette opération.

Leur prise en compte mène à modifier l'obligation de preuves concernant les opérations (OP2 de 2.4.3). En supposant qu'une opération est maintenant de la forme `PRE P THEN S POST Q` (ou $P|S|_pQ$), les obligations de preuves se réécrivent en :

Opération avec les post-conditions **(OP 8)**

$$\Gamma \wedge P \Rightarrow [body](Inv \wedge Q)$$

Remarque(s) : à vérifier pour toutes les opérations de la machine

La post-condition sera une construction des machines abstraites et ne sera pas soumise à raffinement.

Règle méthodologique

L'expression des contrats d'un composant est une donnée très importante pour l'interface de ce composant. Ce contrat doit être intuitivement facile à comprendre, or dans la formation des contrats de nos composants, il semble nécessaire de clarifier l'expression de ceux ci. Prenons comme exemple, le «petit exemple» du BBook ([Abrial96, pages 552-553]) présenté en figure 4.4.

```
MACHINE
    Little_Example_1
VARIABLES
    y
INVARIANT
    y ∈ IF(ℕ1 )
INITIALISATION
    y := ∅
OPERATIONS
    enter(n) =
        PRE  n : ℕ1
        THEN  y := y ∪{n}
        END  ;
    m ←—maximum =
        PRE  y ≠∅
        THEN  m := max(y)
        END

    END
```

```
REFINEMENT
    Little_Example_2
REFINES
    Little_example_1
VARIABLES
    z
INVARIANT
    z = max(y ∪{0})
INITIALISATION
    z := 0
OPERATIONS
    enter(n) =
        PRE  n : ℕ1
        THEN  z := max({z, n})
        END  ;
    m ←—maximum =
        PRE  z ≠0
        THEN  m := z
        END

    END
```

FIG. 4.4 – Spécification de Little_example ([Abrial96, 552-553])

La pré-condition de l'opération `maximum` stipule que l'opération ne peut être appelée que si l'ensemble `y` est non vide. Cette spécification du contrat est intuitif et facilement compréhensible. Lors de l'aplatissement des raffinements, cette pré-condition va devenir

$$z \neq 0 \wedge \exists y.(y \in \mathbb{F}(NAT1) \wedge z = max(y \cup \{0\}) \wedge y \neq \varnothing),$$

ce qui est intuitivement moins évident à comprendre.

Une possibilité serait d'abstraire les contrats, pour cacher les informations inutiles. Dans ce cas ci, le contrat peut en fait s'exprimer par : *il faut au moins avoir appelé une fois l'opération* `enter`. Il faudrait donc diviser les contrats en sous contrats «atomiques» pour écrire des spécifications qui serait de la forme : `m ←—maximum = PRE not_empty THEN ... .`

Ce contrat serait la post-condition de l'opération `enter` et la définition de `not_empty` évoluerait dans les raffinements. Au niveau de la génération de code il suffit de conserver la définition en tant que fonction évaluant un prédicat (si cette construction est permise dans le langage des contrats).

Cette méthodologie est cohérente avec le principe d'encapsulation de la méthode B. Les données constituant l'état d'une spécification sont cachées de l'extérieur de la spécification, donc toutes propriétés portant sur ces données ne peuvent être établies que dans cette spécification. Dans le cas de l'adoption des contrats sous forme de pré et post-conditions, les pré-conditions portant sur l'état du composant devront être établies au préalable par une (ou la combinaison de plusieurs) post-condition(s) d'autre(s) opération(s).

4.4 Extensions des langages cibles

Dans cette section nous allons montrer un des avantages de notre démarche : l'adaptabilité du générateur de code. Ce besoin d'adaptabilité est récurrent dans le développement logiciel, et pas uniquement dans les techniques à base de composants. Nous avons déjà évoqué un cas typique illustrant cette nécessité, c'est le développement de logiciel destiné à être embarqué dans une carte à puce. Ce logiciel ne sera pas soumis aux mêmes contraintes qu'un logiciel embarqué dans une rame de métro. Pourtant ces deux domaines d'application sont des domaines dans lesquels la méthode B peut être utilisée (et l'a déjà été d'ailleurs). Cette adaptabilité peut également être un moteur à la diffusion de la méthode B dans des domaines d'application pour lesquelles la méthode n'a pas été conçue au départ.

Cette adaptabilité peut être perçue sous deux angles différents. Le premier est l'angle pragmatique, c'est-à-dire la facilité d'adapter la génération de code d'un point de vue technique de traduction donc écriture des règles de transformations. Le second angle est un angle plus conceptuel, l'adaptabilité permet de transcrire certains concepts B par certains concepts dans le langage cible, sans aborder tous les langages cibles d'une manière uniforme.

4.4.1 Facilité d'adaptation

L'usage de règles de transformation XSL ([W3c]) pour la transformation de notre syntaxe abstraite des composants B en code cible rend la procédure de génération facilement adaptable (voir la spécification de la transformation concernant la structure de boucle *while* de la figure 4.5). En effet, les règles de transformations sont simples, ce qui permet d'utiliser des processeurs simples comme les processeurs

XSLT. L'usage de tels processeurs est connu dans le monde du développement logiciel, et leur assimilation dans le cas contraire est aisée.

```
<xsl:template match="WHILE">
  <xsl:text>
      WHILE
  </xsl:text>
  <xsl:apply-templates select="child::*[position()=1]"/>
  <xsl:text>
      DO
  </xsl:text>
  <xsl:apply-templates select="child::*[position()=2]"/>
  <xsl:text>
      END /* WHILE */
  </xsl:text>
</xsl:template>
```

FIG. 4.5 – Spécification XSL de la traduction du WHILE

4.4.2 Finesse des générateurs

La deuxième facette de l'adaptabilité concerne l'adaptation des concepts B aux langages cibles. Cet avantage découle directement du point précédent sur la facilité d'adaptation de la spécification des traductions et de la conservation des aspects du langage B (hormis le raffinement qui pour la génération de code n'est pas pertinent). Il est ainsi possible de traduire au plus près les concepts. Prenons l'exemple de la traduction du concept d'encapsulation. Dans le traducteur actuel de l'Atelier B, ce concept se traduit par la génération de structures, aussi bien pour la génération de code C que pour la génération de code Ada. Ceci permet d'avoir une uniformisation des traductions vers les langages cibles. L'inconvénient de cette approche est que la traduction des concepts s'aligne sur les langages de plus bas niveau, dans ce cas ci, le langage C.

4.5 Correction du code généré

La question de la correction des transformations effectuées est une question importante à aborder dans le cadre de développement formel. Comme nous l'avons signalé en introduction du chapitre 3, certains travaux visent à la formalisation de la méthode et du langage B dans des systèmes comme Coq, PVS... ([Chartier98], [Bodeveix et al.99], [Bodeveix et al.00]). Si une telle formalisation portait sur l'intégralité du langage, la génération de code pourrait être spécifiée dans le système support à la formalisation de B et le code du générateur de code extrait automatiquement (à la manière de l'extraction de programme du système Coq par exemple). Or à l'heure actuelle, l'état d'avancement des travaux de méta spécification de B ne permet pas une telle approche. De plus, notre démarche est plus pragmatique car l'objectif final est un générateur de code opérationnel permettant l'exploration de diverses pistes pour la génération. Il nous sera donc impossible d'obtenir une preuve totale de la correction de notre génération de code, néanmoins, nous allons fournir des indications sur la validation étape par étape.

La première étape concerne la transformation des spécifications B en spécifications B-HLL. La validation de cette étape consiste à vérifier que la vision des modules B en B-HLL est correcte. Ceci a été discuté dans le chapitre 3.

La seconde étape concerne la transformation des chaînes de raffinements en une spécification et une implantation. La validation de cette transformation est obtenue par la relation de raffinement existant entre la spécification enrichie et la spécification abstraite. Les transformations qui sont appliquées pour obtenir l'implantation enrichie sont explicitées dans le BBook ([Abrial96, section 11.2.5]) qui utilise cette version transformée des spécifications pour donner l'intuition de ce qu'est la preuve de raffinement. Par le biais des obligations de preuves, l'auteur montre que la machine transformée est équivalente aux deux composants reliés par le lien de raffinement. Cette même approche est suivie dans [Behnia00, chapitre 3] qui montre que la relation de raffinement existe entre deux étapes de développement. En prenant comme étapes de développement la machine racine du module (la plus abstraite) et comme seconde étape la chaîne de raffinement complète, nous obtenons que l'aplatissement de la chaîne de raffinement est un raffinement de la machine abstraite de cette chaîne[3].

La troisième étape de validation est une étape qui devra être menée par le «client» de notre outil de génération de code, il s'agit de la validation des règles de transformation utilisées pour passer de l'arbre de syntaxe abstraite obtenu par les étapes précédentes en syntaxe (concrète ou abstraite) du langage cible. Comme la modélisation B-HLL dissocie le langage des modules du langage B lui même, la validation sera également scindée en deux. La première partie concernera la traduction des constructions de modules de B-HLL en construction du langage cible et la seconde partie concernera la traduction des constructions du noyau impératif B0 et des prédicats formant les contrats.

L'expression de ces traductions a été faite en XSL ([W3c]) pour les exemples que nous présenterons plus tard. Ce choix se justifie par la relative simplicité d'utilisation de ces moteurs de transformation comme nous l'avons montré en 4.4.1. Du point de vue de la validation, l'utilisation de processeur XSLT peut poser quelques problèmes : comment être sûr que la traduction obtenue est bien une expression du langage cible et comment être sûr que la sémantique des constructions est bien conservée par chacune des règles. Il n'y a pas de solution permettant de prouver la validité des feuilles de style XSL pour répondre à la première préoccupation hormis une vérification par tests que le résultat obtenu est bien en adéquation avec celui attendu. L'utilisation de moteur de réécriture permettrait de raisonner d'avantage sur les transformations, cela permetrai de s'assurer par exemple que le résultat obtenu par réécriture est bien une expression du langage cible. Par contre, cette solution a l'inconvénient d'être plus «lourde» à mettre en œuvre.

La seconde partie de la validation concernera chaque règle de traductions. Il faut montrer que la sémantique d'une construction B est conservée dans sa projection vers le langage cible. La solution proposée dans [Badeau et al.03] consiste à utiliser une notion d'observation de l'état, d'une part, de la spécification B, et d'autre part, de l'état du code généré. Il faut ensuite définir une relation d'équivalence entre les observations permettant de les comparer.

[3]Même si le choix de ces deux étapes de développement ne satisfont pas les règles architecturales définies par l'auteur, cela ne remet pas en cause l'existence de la relation de raffinement.

Validation par «ouverture»

Une autre partie de ce qui contribue à la validation est l'«ouverture» du générateur de code. Cette ouverture prend deux formes, d'une part, l'ouverture par l'utilisation d'outils éprouvés, et d'autre part, l'ouverture par diffusion du générateur de code.

L'ouverture par l'utilisation de techniques et outils éprouvés est un principe du développement par composant : laisser développer certains composants par des experts du domaine. L'utilisation de processeurs XSLT, qui sont des processeurs souvent utilisés et répondant à des recommandations du consortium W3C (World Wide Web Consortium), déporte le problème de la correction du moteur de transformation. Seule la partie spécification des traductions est laissée à l'utilisateur, pas la partie application de ces règles. Comme dans l'approche composant, rien n'empêche de retirer le composant processeur XSLT pour le remplacer par un autre composant implantant un moteur de réécriture.

L'ouverture par diffusion du générateur de code et des sources le constituant est un moyen pour obtenir une validation. Cette approche de validation ne repose sur aucun concept scientifique, mais elle a pourtant fait ses preuves pour des développements conséquents comme le système d'exploitation Linux et les diverses applications utilisables sur ce système. Cette approche d'ouverture peut également prendre un autre aspect que nous allons aborder dans la section suivante, c'est la possibilité de connecter d'autres outils de validation, tous les formats utilisés étant libres et modifiables.

4.6 Impact sur le processus de développement *B*

Dans cette section, nous allons aborder plus précisément le processus de développement B et la manière dont notre approche de génération de code peut intervenir dans ce processus pour l'étendre et faciliter l'utilisation et l'intégration de la méthode B dans un processus de développement non B.

La figure 4.6 présente conjointement le processus habituel de développement B et celui que nous proposons. Cette figure est une modélisation UML sous forme de diagramme d'activité des deux processus de développement. Le diagramme d'activité est divisé en trois travées[4] appelées spécification, validation et vérification.

- La travée centrale décrit les activités de spécifications, c'est-à-dire la transformation des exigences en spécifications B et le développement de ces spécifications B par raffinements et compositions.
- La travée de droite concerne les activités classiques de validation de la méthode B. Une partie de ce qui doit être prouvé a été présentée en section 2.4.3.
- La travée de gauche représente les activités de vérification pouvant être effectuées après utilisation de notre générateur de code.

4.6.1 Processus classique B

Une fois le développement commencé, l'équipe chargée de l'écriture et du développement des spécifications commence son travail. Une fois que les spécifications ont atteint un niveau de développement

[4]Une travée est utilisée dans un diagramme d'activité pour décomposer les différentes activités de ce diagramme en plusieurs familles d'activité. Une famille d'activité pourra par exemple se traduire physiquement par une équipe de développement ou de validation.

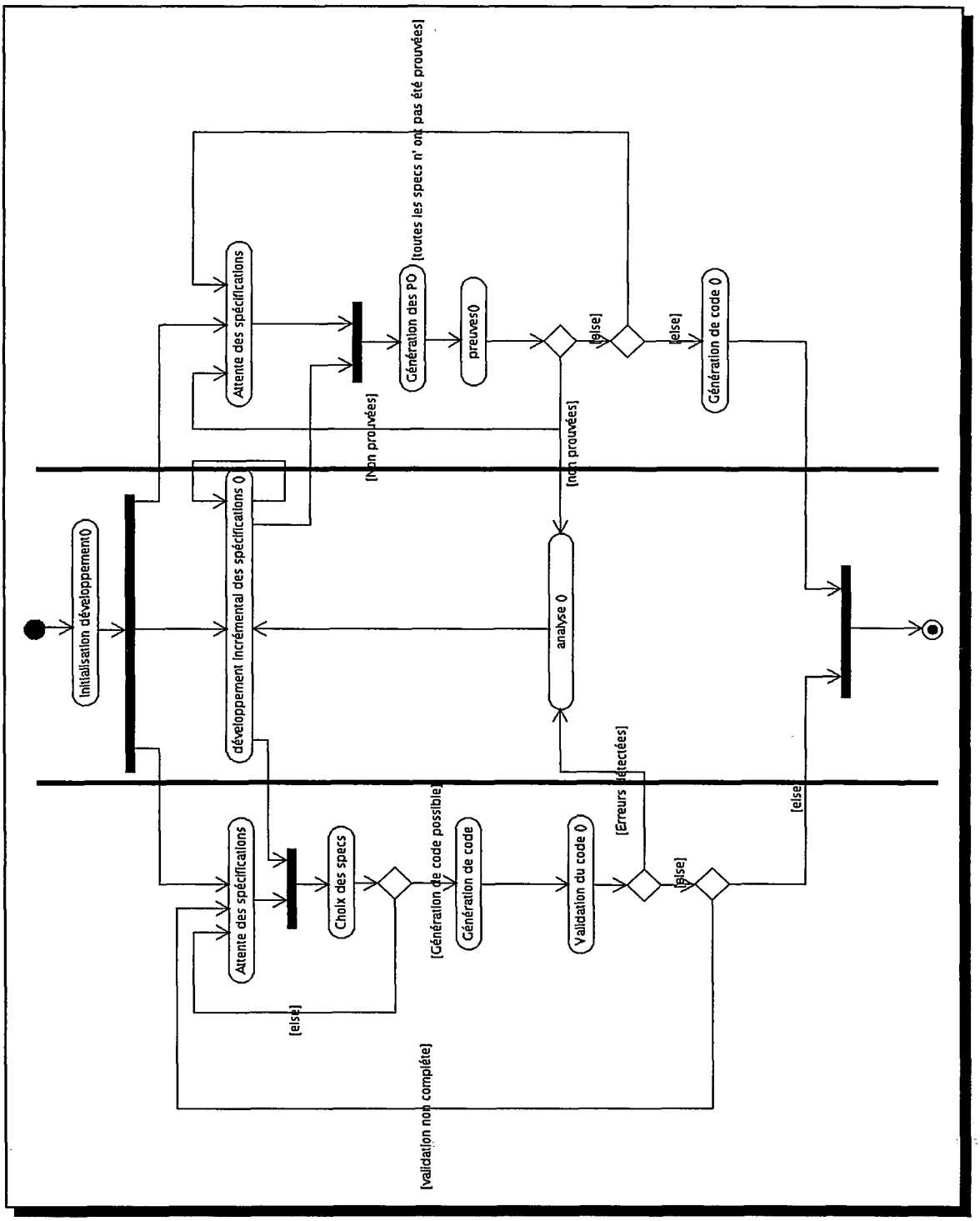


FIG. 4.6 – Deux processus de développement

suffisant, les actions de validation peuvent débuter. Ces actions consistent en la génération des obligations de preuve et leur déchargement[5]. Si toutes les obligations de preuves sont déchargées, et si toutes les spécifications ont été validées, le processus de validation est terminé et le code peut alors être généré[6]. Si des spécifications n'ont pas encore été validées, le processus de validation reprend sur un autre sous-ensemble de spécifications.

Il arrive que certaines obligations de preuves ne puissent être déchargées, ceci mettant en évidence un problème dans les spécifications B. Dans ce cas, l'équipe de spécification doit retravailler les spécifications fautives pour résoudre le problème. Ensuite, la procédure de validation recommence sur les spécifications touchées par les modifications.

4.6.2 Nouveau processus

L'approche que nous allons présenter ici a pour objectif de réduire au maximum le temps consacré à la preuve, et plus particulièrement le temps consacré à trouver par le biais de la preuve les erreurs dans les spécifications. En effet, l'étape de preuves dans la méthode B est la partie la plus coûteuse en temps, et cela, même si les outils supports à la méthode (l'Atelier B et le B ToolKit) permettent d'atteindre des taux de preuves allant jusqu'à 80% voire même 90%. Le pourcentage de preuves restant représente un nombre considérable de preuves à effectuer manuellement, ou plus exactement interactivement. Dans [Behm et al.] est exposé le retour sur expériences de l'utilisation de la méthode B pour le développement du projet METEOR. Pour ce projet, un taux de preuves de 80% représente encore 6000 lemmes à prouver interactivement.

La nouvelle approche commence, comme pour l'approche classique B, par attendre que les spécifications aient atteint un certain niveau de développement, niveau qui permettra la génération de code. Une fois ce code généré, il est possible d'utiliser des outils de validation de code se basant sur les assertions. Ces outils peuvent à la fois être des outils de vérifications statiques ou dynamiques développés par d'autres ou même développés au sein de l'entreprise. Les outils développé par Rustan et Leino[Leino01], Cheon et Leavens [Cheon et al.02] ou Evans et Larochelle [Evans et al.02] sont de bons candidats à la validation opérant sur le code.

Cette approche s'inscrit dans l'esprit de *Disappearing Formal Methods* de John Rushby ([Rushby00]). L'idée consiste à cacher au maximum les aspects les plus «rebutants» des méthodes formelles comme la preuve. Mais, même si certaines erreurs peuvent être trouvées plus facilement et surtout plus rapidement par d'autres outils, l'abandon de la phase de preuves classique B n'est pas envisagé. L'approche est également conforme au cinquième commandement de Bowen et Hinchey (*«Thou shalt not abandon thy traditional development methods»*) ; si l'entreprise a développé une expertise dans la validation de logiciel (développé par contrat ou non), il est dommage d'abandonner ces techniques de validation pour n'utiliser que celle fournie par B.

[5]Rappel : décharger une obligation de preuve consiste à prouver que le prédicat formant cette obligation de preuve est un théorème.

[6]Le critère d'arrêt peut être relâché suivant le niveau de validation désiré.

4.7 Application de la génération de code

Dans cette section, nous allons présenter les différents codes qu'il est possible d'obtenir en utilisant nos outils. Nous montrerons les diverses possibilités que nous avons présentées précédemment, notamment la souplesse d'adaptation du générateur. Nous nous appuierons également sur ces exemples pour illustrer les difficultés de traductions lors du traitement de certains concepts B et leur transformation en concepts du langage cible.

4.7.1 La pile bornée

Le premier exemple que nous allons prendre est celui de la pile bornée dont nous avons déjà présenté une spécification abstraite dans le chapitre 2.

Spécification

La spécification de la pile bornée est présentée dans la figure 4.7.

```
MACHINE
    stack(stack_size)
CONSTRAINTS
    stack_size : ℕ ∧ stack_size ≥1
 ∧ stack_size ≤MAXINT
VISIBLE_VARIABLES
    the_stack, stack_top
INVARIANT
    the_stack : (1..stack_size) →ℕ ∧ stack_top : ℕ
 ∧ stack_top ≥0  ∧ stack_top ≤stack_size
INITIALISATION
    the_stack : : (1..stack_size) →ℕ || stack_top := 0
OPERATIONS
    push(addval) =
        PRE
            stack_top <stack_size  ∧ addval ∈ ℕ
        THEN
            stack_top := stack_top + 1
          || the_stack(stack_top + 1) := addval
        END
    ...
    END
```

```
IMPLEMENTATION
    stack_1(stack_size)
REFINES
    stack
INITIALISATION
    the_stack := (1..stack_size) * {0}  ; stack_top := 0
OPERATIONS
    push(addval) =
        BEGIN
            stack_top := stack_top + 1
          ; the_stack(stack_top) := addval
        END
    ...
    END
```

FIG. 4.7 – Une pile d'entier en B

Code(s) généré(s)

Nous allons présenter plusieurs versions du code de la pile bornée. Ces versions se différencieront par des stratégies de traductions différentes. La première illustration de la traduction de la pile bornée est l'obtention d'une classe correspondant à cette pile. Le code est présenté dans la figure 4.8.

Les points importants à remarquer dans le code présenté en figure 4.7 sont :

```
class stack stack_size =
  object
    val mutable the_stack =
      (BASIC_Sets.mul (((BASIC_Sets.make_range 1 stack_size)),
                       (BASIC_Sets.make_compr [0] )))
    val mutable stack_top = 0

    method push addval  =
      assert (stack_top >= 0 & stack_top <= stack_size);
      assert (stack_top < stack_size) ;
      (
        stack_top <- stack_top + 1;
        BASIC_Sets.mul_set  the_stack stack_top addval
       ) ;
      assert (stack_top >= 0 & stack_top <= stack_size)

...
end ;;
```

FIG. 4.8 – Une classe OCaml de la pile

- L'absence de construction d'invariant de classe dans le langage OCaml qui nécessite de concaténer cet invariant aux pré-conditions et post-conditions du code généré à partir des opérations.
- La génération de code fait intervenir un module `BASIC_Sets` destiné à manipuler les constructions d'ensemble exprimées dans la spécification concrète. Ces constructions sont habituellement traduites en tableau, le choix a ici été d'abstraire cette représentation par l'utilisation de ce module `BASIC_Sets` et des services qu'il fournit.
- La paramétrisation de la pile porte uniquement sur la valeur de la taille de la pile. Dans les cas où la paramétrisation fait intervenir des ensembles abstraits (donc des types), les constructions de classes génériques seront utilisées.

La génération de code Ada pour cette même spécification implique une gestion différente de la notion de type abstrait caractérisant un module B (voir figure 4.9). Dans la génération OCaml présentée précédemment, et plus généralement dans toute génération de code reposant sur la notion de classe, l'instanciation est un mécanisme du langage cible. Ainsi, l'initialisation d'un objet se fait au moment de l'instanciation de la classe représentant cet objet. Dans un langage utilisant les paquetages (les langages à base de modules sont dans le même cas), l'instanciation n'existe pas, et donc l'initialisation de l'«objet» doit se faire de manière explicite. Le code généré devra donc tenir compte de cela et fournir les mécanismes adéquats pour s'assurer de l'initialisation avant toute utilisation. Dans le code présenté dans la figure 4.10, ceci se traduit par l'ajout dans la pré-condition d'une vérification (`initialised(this)`).

Dans le cas de la génération de code pour des modules B non paramétrés, il est possible de choisir une solution définissant un type abstrait de données pour les paquetages Ada. Cette solution est illustrée dans les figures 4.11 et 4.12. Cette solution a l'avantage de ne pas devoir passer par les packages génériques pour instancier plusieurs fois un même module (même si une instanciation doit être éffectuée pour définir la taille de la pile). Par contre, cette solution requiert un certain nombre de précautions, comme la vérification que l'initialisation a été faite. En Ada, il est possible d'initialiser les champs d'un enregistrement lors de sa création. Dans le cas contraire, il aurait fallu ajouter une condition représentant

```
generic
   stack_size :  natural  ;

package stack is
   function  is_empty return boolean ;
   procedure  push(addval : in natural );
   procedure  pop;
   function  top return natural ;
   function initialised return boolean;
end stack;
```

FIG. 4.9 – La spécification d'un paquetage Ada pour la pile bornée

```
package body stack is
--# invariant stack_top >=  0 and stack_top <=  stack_size

   the_stack : array (1..Stack_size) of natural ;
   stack_top : 0..Stack_size ;

   procedure push(addval : in  natural ) is
   begin
      --# pre stack_top <  stack_size
      stack_top:=stack_top +  1;
      the_stack(stack_top) := addval;
   end push;

   ...

begin --initialisation
  stack_top := 0
end stack;
```

FIG. 4.10 – Le corps du paquetage Ada pour la pile bornée

l'initialisation dans les pré-conditions des méthodes (ainsi qu'un champ dans l'enregistrement indiquant que l'initialisation a bien été effectuée).

```
generic
   Stack_size_parameter : natural

package stack is

   type ABSTRACT_TYPE_stack is private ;

   function  is_empty(this : in ABSTRACT_TYPE_stack ) return boolean ;
   procedure  push(this : in out ABSTRACT_TYPE_stack ; addval : natural );
   procedure  pop(this : in out ABSTRACT_TYPE_stack );
   function  top(this : in ABSTRACT_TYPE_stack ) return natural ;
   function initialised(this : in ABSTRACT_TYPE_stack) return boolean;
private
   type  the_stack_data_type is array (1..Stack_size_parameter) of natural ;
   type  ABSTRACT_TYPE_stack is
      record
         stack_size : Integer := 0 ;
         the_stack : array (1..Stack_size_parameter) of natural := (1..Stack_size_parameter => 0) ;
         stack_top : natural := 0 ;
      end record;
end stack;
```

FIG. 4.11 – La spécification d'un paquetage Ada pour la pile bornée : solution du type abstrait de données

```
package body stack is
--# invariant stack_top >=  0 and stack_top <=  stack_size

   procedure push(this : in out ABSTRACT_TYPE_stack ; addval : in  natural ) is
   begin
      --# pre stack_top <  stack_size
      this.stack_top:=this.stack_top +  1;
      this.the_stack(this.stack_top) := addval;
   end push;

...
end stack;
```

FIG. 4.12 – Le corps du paquetage Ada pour la pile bornée : solution du type abstrait de données

4.7.2 La pile bornée générique

Nous allons montrer ici une évolution de la spécification de la pile, spécification que nous allons rendre générique pour que le code généré soit également générique.

Spécification

La spécification ne diffère que peu de celle présentée précédemment. Un lien INCLUDES, respectivement un lien IMPORTS, est ajouté dans le composant abstrait, respectivement concret, vers le module implantant le type abstrait des éléments stockés dans la pile. Les transformations du code suivront les mêmes règles que celles effectuées pour la recherche dans une table (figure 4.3). Le module `ta_element` devra définir en plus une constante concrète pour l'élément neutre du type abstrait, élément qui sera utilisé dans l'initialisation de la pile.

Code généré

Le code généré pour cette pile générique utilisant la représentation des modules est présenté dans la figure 4.13 pour le code OCaml et dans 4.14 pour la spécification du paquetage Ada.

```
module Stack_gen =
  functor (Param : sig val stack_size : int end) ->
    functor (TA_element :
               sig
                 type t
                 val element_neutre : t
                 ...
               end) ->
  struct
    let the_stack = ...
  end
```

FIG. 4.13 – Une pile générique OCaml

Dans ce code la double utilisation de la construction de foncteur permet pour la première d'exprimer les paramètres du module B originel et pour le second la dépendance vers un autre module.

4.7.3 Le passage à niveaux

Le passage à niveaux est une étude de cas classique du monde ferroviaire. Cette étude de cas a été présentée dans [Einer et al.]. Elle a été utilisée pour illustrer plusieurs travaux, notamment récemment dans plusieurs articles de [Tarnai et al.03]. La figure 4.15 illustre le problème du passage à niveaux adapté à la France et la figure 4.16 le séquencement des actions que le train doit suivre quand il arrive dans le secteur d'un passage à niveaux. Cette entrée dans le secteur est détectée par des capteurs positionnés sur la voie.

Spécification

Il n'est pas envisageable de présenter l'intégralité des spécifications, nous allons donc nous focaliser sur le module de contrôle, entité centrale ayant un regard sur le système complet. C'est cette entité qui donnera au système embarqué dans le train l'information concernant le mode du passage à niveaux (Sécurisé ou non, mode utilisé dans l'automate de la figure 4.16). Les dépendances entre les différents

```
generic
   stack_size :  natural  ;
   type t is private;

package stack is

   type ABSTRACT_TYPE_stack is private ;

   procedure initialisation (this : in out ABSTRACT_TYPE_stack);
   function  is_empty(this : in ABSTRACT_TYPE_stack ) return boolean ;
   procedure  push(this : in out ABSTRACT_TYPE_stack ; addval : t );
   procedure  pop(this : in out ABSTRACT_TYPE_stack );
   function  top(this : in ABSTRACT_TYPE_stack ) return t ;
   function initialised(this : in ABSTRACT_TYPE_stack ) return boolean;
private
   type  the_stack_data_type is array (1..Stack_size) of t ;
   type  ABSTRACT_TYPE_stack is
      record
         initialised : Boolean;
         stack_size : Integer ;
         the_stack : the_stack_data_type ;
         stack_top : natural ;
      end record;
end stack;
```

FIG. 4.14 – Une pile générique Ada

modules constituants les spécifications sont présentées dans la figure 4.17. En plus des modules présents dans la figure, il faut ajouter un module `Train` spécifiant les caractéristiques d'un train (sa vitesse minimale, sa vitesse maximale...) ainsi qu'un module `communication` modélisant la communication nécessaire entre l'entité de contrôle et le système du passage à niveaux pour s'assurer que la communication est fonctionnelle.

Dans ce module de contrôle, une opération (celle dénommée `trans_BBa`, voir figure 4.7.3) est destinée à baisser la barrière.

Cette opération n'est pas raffinée dans le composant de raffinement `control_r` mais l'est dans le composant `control_i`.

Code généré

Certains modules ont déjà été présentés dans le chapitre 3 section 3.6.4 pour illustrer les interfaces des modules. La syntaxe des modules utilisée pour cette présentation est la même que celle du langage OCaml, la syntaxe des structures de contrôle sera elle traduite en suivant les mêmes règles que dans les exemples précédents (voir le code en figure 4.19).

Par contre, ce qui est important de remarquer, c'est le résultat obtenu par le processus d'élimination des liens de raffinements au niveaux des contrats du code généré. La pré-condition de l'opération `trans_BBa` sera composée des pré-conditions des différents niveaux de raffinement ainsi que des invariants. L'invariant abstrait du composant `control` est celui présenté en figure 4.20.

Vue la taille de l'invariant abstrait, il apparaît très clairement la nécessité d'abstraire les différents

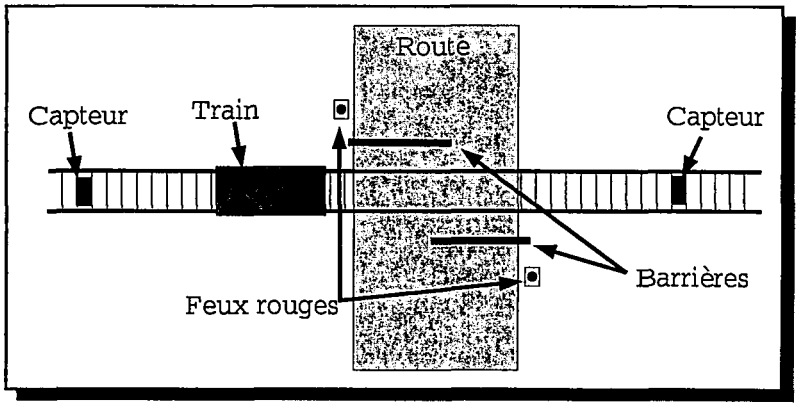


FIG. 4.15 – Le passage à niveaux

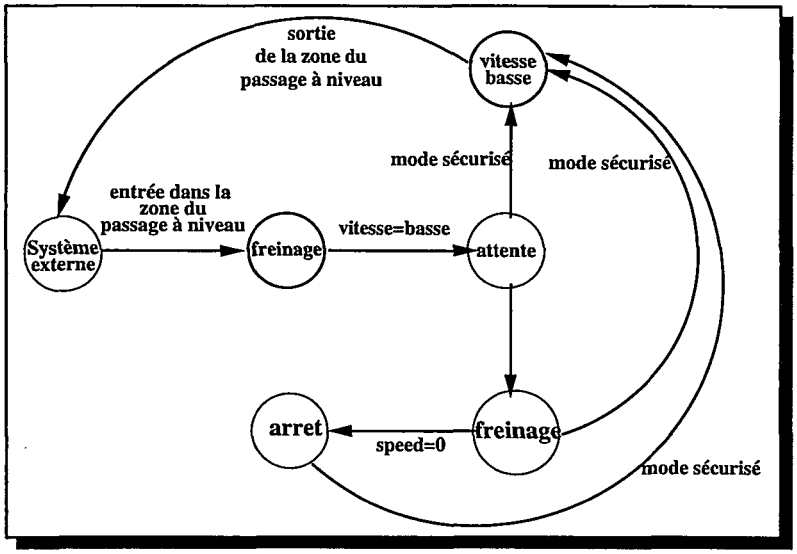


FIG. 4.16 – Séquencement des actions du train en approche d'un passage à niveaux.

contrats dans le code généré comme nous l'avons indiqué en section 4.3.3.

4.8 Conclusions

Dans ce chapitre, nous avons exposé le déroulement du processus de génération de code. Ce processus se base sur l'aplatissement des spécifications que nous avons présenté dans le chapitre 2 et sur la représentation en modules B-HLL que nous avons étudiée dans le chapitre 3.

Le premier apport de notre générateur de code est la préservation des propriétés dans le code généré. Cette prise en compte permet de valider le code généré par des outils externes à la méthode B. Cette ouverture de la méthode B à des domaines extérieurs permet d'apporter des extensions à la génération de code. Ainsi, il nous a été possible d'aborder la génération de composants logiciels. Une fois le composant logiciel généré, il peut s'insérer dans un développement à base de composants sans qu'il y ait un quel-

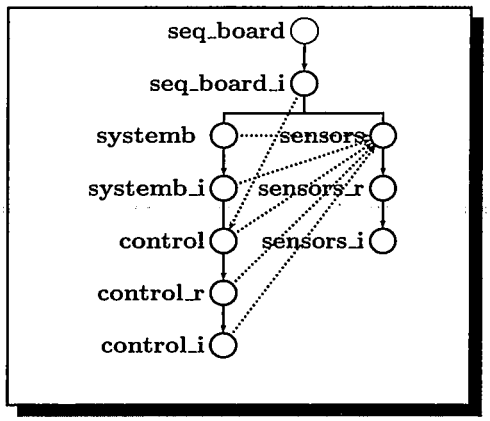


FIG. 4.17 – Graphe de dépendance des spécifications

trans_BBa =
PRE (feu = allume) ∧md = safe
THEN br := mouvement_bas
END

FIG. 4.18 – L'opération trans_BBa

```
module Controle =
  struct
    module Train = (Train : Train_SEES)
    module Communication = (Communication : Communication_SEES)
    module Capteurs = (Capteurs : Capteurs_SEES)

    open Train
    open Communication
    open Capteurs
...

    let trans_BBa =
      assert (feu_allume & mode_pn_securise);
      br := Mouvement_bas
...

  end
```

FIG. 4.19 – Le module control en module OCaml

md ∈ MODE ∧ ack ∈ ACK ∧ feu ∈ FEU ∧ zn ∈ ZONE ∧ gr ∈ GARDIEN ∧ train <: TRAINS
∧ no_train /: train ∧ card(train) <= max_train ∧ train_demande_ack ∈ TRAINS
∧ train_demande_ack∩train = ∅ ∧ br ∈ BARRIERE ∧ son ∈ SONNERIE
∧ ((feu = allume & zn = normal) => son = tinte) ∧ ((feu = allume & zn = sensible) => son = calme)
∧ ((br = haut) => feu = eteint) ∧ ((br /= haut) => feu = allume)
∧ ((feu = eteint) => son = calme)

FIG. 4.20 – L'invariant du composant abstrait control

conque impact au niveau du développement par composants. Par contre, le composant généré sera sûr car il aura été développé avec une technique formelle. Au niveau du développement B, la prise en compte de la notion de composant logiciel a entraîné des changements au niveau de la méthode. Ces changements concernent la prise en compte d'interfaces contractualisées pour la validation de spécifications B dépendantes de composant non formalisés en B.

Le second apport de notre générateur de code est la facilité d'adaptation qui découle des techniques utilisées. Cette facilité nous a permis d'illustrer la génération de code pour des langages cibles tels que Ada et OCaml. Il est clair que la génération de code dans d'autres langages cible peut s'envisager facilement. Les problèmes liés à la spécification de la génération étant identifiés, les solutions doivent être adaptées en fonction des possibilités des langages cible.

Chapitre 5

Mise en œuvre et perspectives

Sommaire

5.1 Introduction

Ce dernier chapitre sera composé de deux parties. Dans la première partie, nous présenterons la plateforme BCaml, plateforme dans laquelle nos travaux s'intègrent. La seconde partie de ce chapitre sera consacrée à la présentation d'extensions possibles de nos travaux à plus ou moins long terme. Dans cette seconde partie, nous commencerons par les perspectives nous apparaissant comme les plus immédiates à obtenir dans les sections 5.3.1, 5.3.2 et 5.3.3. Nous discuterons ensuite des poursuites de travaux à plus long terme dans les sections 5.3.4, 5.3.5 et 5.3.6.

5.2 La plateforme BCaml

Les travaux qui ont été présentés dans ce mémoire s'intègrent dans le développement collaboratif d'une plateforme d'expérimentation pour B s'intitulant BCaml. Cette plateforme fait partie de l'effort de diffusion d'outils libres pour B intitulé BRILLANT ([Brillant]). Le code de cette plateforme est entièrement libre et disponible afin de permettre une évaluation des outils par des tiers mais également pour leur permettre de se connecter facilement à la plateforme pour leurs propres expérimentations.

5.2.1 Historique de la plateforme

Le développement d'outils libres et ouverts autour de B permettant des expérimentations pour des travaux de recherche autour de B est vite apparu comme une nécessité. L'embryon de la plateforme BCaml apparaît dans les travaux de thèse de Georges Mariano ([Mariano97]) dans lesquels la nécessité d'analyser les spécifications B a rendu nécessaire le développement d'un analyseur syntaxique. Cet analyseur a ensuite été développé dans [Petit98] pour donner naissance à la plateforme BCaml. Les fondations de la plateforme sont donc la définition d'une syntaxe abstraite de B et de l'analyseur syntaxique permettant d'obtenir les arbres de syntaxe abstraite des spécifications. Ces premiers travaux qui peuvent paraître anodins ont permis de mettre en évidence certains problèmes du langage B ([Mariano et al.99]). Autour de ce noyau central d'analyse et de représentation des spécifications B, d'autres développements ont pu ensuite être menés. C'est ce que nous allons aborder dans la section suivante.

5.2.2 Présentation de la plateforme

La plateforme BCaml est présentée en partie par la figure 5.1 ; les principaux éléments sont représentés. Chaque ellipses dans la figure représente un composant implantant une fonctionnalité.

La vision que nous donnons de la plateforme est une vision reposant sur le flux de données dans les différents outils constituant la plateforme. Chaque outil produit un résultat en sortie qui peut être réutilisé par d'autres outils. Les formats d'échange avec le monde extérieur sont systématiquement des formats XML pour permettre une connection et une utilisation aisée de nos outils. La préoccupation principale étant l'ouverture au monde extérieur, les formats doivent eux aussi être ouverts.

Les différents composants constituant la plateforme, exception faite de ceux concernant la génération de code que nous aborderons dans la section suivante, sont :

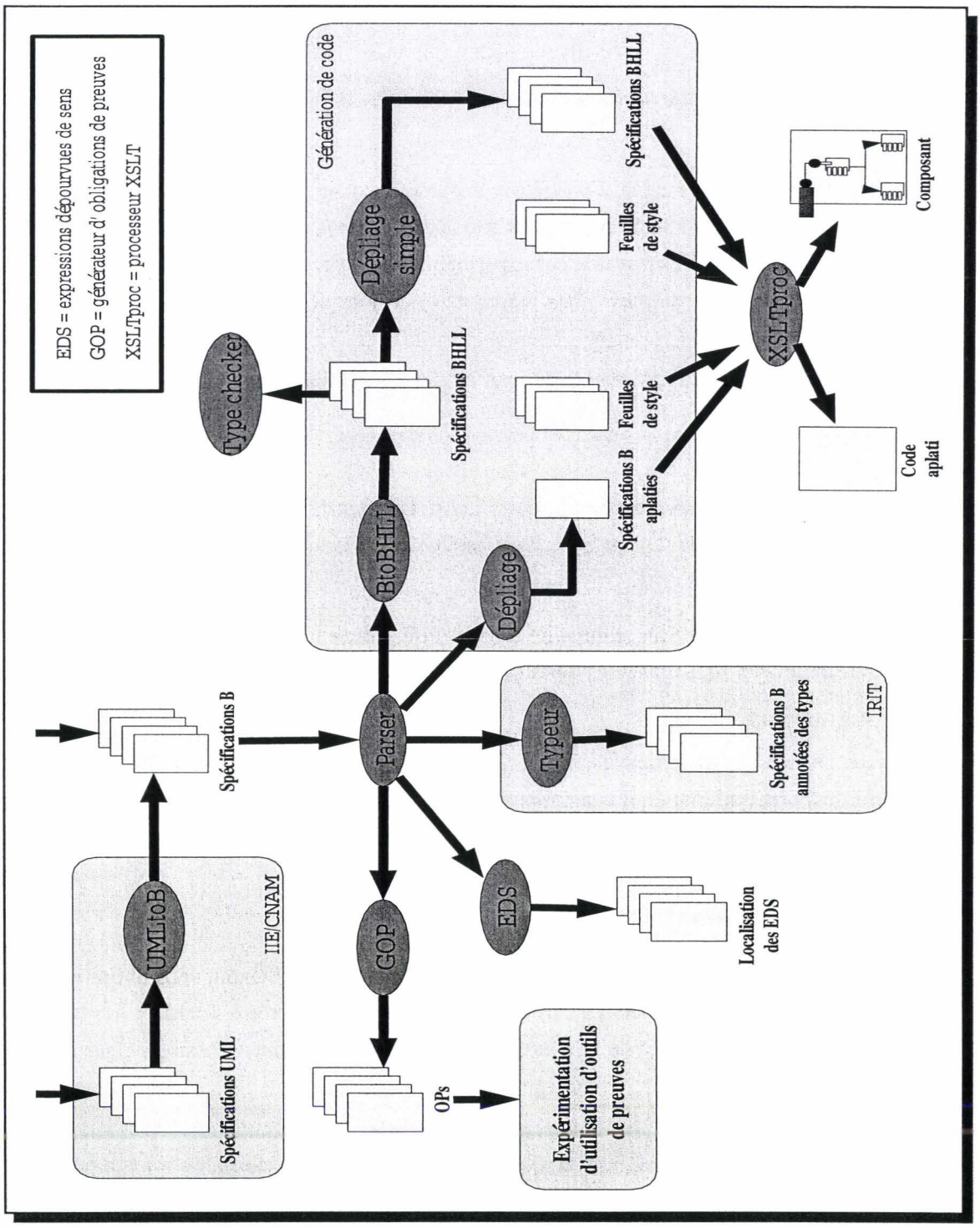


FIG. 5.1 – La plateforme BCaml

UMLtoB

Ce composant a été développé pour transformer des spécifications UML en spécifications B ([Laleau et al.01b, Laleau et al.02]). Le domaine d'application est centré sur les spécifications de base de données.

Parser

Ce composant est l'analyseur syntaxique des spécifications B.

GOP

Ce composant est le générateur d'obligation de preuves. Comme son nom l'indique, il a pour fonction de générer les prédicats qui devront être déchargés pour assurer la correction du développement B. Sur la base de ces travaux, des expérimentations sont en cours au sein de l'équipe pour connecter des prouveurs externes (Phox notemment) et pour décharger un maximum de preuves automatiquement.

C'est à ce niveau que se fait l'intervention pour modifier les obligations de preuves que nous avons présentées en 4.3.3.

EDS

Ce composant implante les travaux présentés dans [Burdy00]. Ces travaux visent à détecter les expressions dépourvues de sens dans les langages basés sur la théorie des ensembles.

Typeur

Ce composant implante un algorithme de typage différent de l'algorithme habituellement utilisé en B. L'algorithme et le langage des types implantés sont présentés dans [Bodeveix et al.02].

Les autres composants

En plus de ces composants majeurs, il faut signaler la présence de librairies destinées à implanter un calcul des substitutions, ou à construire et manipuler le graphe de dépendances d'un projet B par exemple.

5.2.3 Flot de données pour l'obtention de code

Après avoir présenté différents composants constituant la plateforme BCaml, nous allons revenir sur ceux spécifiques au générateur de code. La partie de la plateforme consacrée à la génération de code est la partie dans l'encadré sur la droite de la figure 5.1. Il existe deux chemins possibles pour générer le code.

Le premier chemin consiste à utiliser le composant de dépliage aplatissant un ensemble de spécifications B pour n'en construire qu'une seule. L'implantation du module de dépliage nécessite la connaissance du graphe de dépendance du projet B développé par ailleurs dans la plateforme. Ce composant B unique peut ensuite être utilisé pour générer le code par une simple réécriture de celui-ci par un processeur XSLT. Nous rappelons que le choix de l'utilisation d'un processeur XSLT peut être remis en cause, des outils comme ceux développés dans le projet CDuce ([Benzaken et al.03]) pourraient tout à fait convenir.

Le second chemin est celui qui a été présenté dans le chapitre 4. Les spécifications sont transformées en modules B-HLL. Sur ces modules peut être appliqué le module de vérification de la sémantique statique, module obtenu par implantation du système HLL. Ce module de vérification statique reprend l'algorithme développé dans le cadre du composant Typeur présenté précédemment. La relation de raffinement dans un module B est ensuite éliminée par l'utilisation d'une partie de l'algorithme de dépliage utilisé dans la première voie de génération. Comme nous l'avons signalé dans le chapitre précédent, la phase de dépliage simple peut être intervertie avec la transformation en module B-HLL ; dans ce cas le dépliage simple est remplacé par le composant de dépliage. De même que pour la voie précédente, la spécification de la génération de code se fait par l'écriture de feuilles de style. Bien que dans les deux approches de génération de code, la syntaxe abstraite est légèrement différente, les feuilles de style partagent une grande partie de leur spécification.

5.2.4 Les modules développés

Dans cette section, nous allons apporter des éléments quantitatifs sur les éléments logiciels qui ont été produits pour implanter le travail présenté dans ce mémoire. La figure 5.2 présente une répartition de l'effort de développement en fonction des trois grands aspects du travail liés à l'implantation des outils : l'outil d'analyse comprenant la définition de la syntaxe concrète et de la syntaxe abstraite pour B, l'outil implantant l'algorithme de dépliage et les outils liés à l'instanciation du système HLL ainsi qu'à la transformation d'arbre de syntaxe B en arbre de syntaxe B-HLL. En plus de ces trois aspects du développement, un quatrième est ajouté pour représenter les développements annexes qui ont été nécessaires mais ne sont en rien spécifiques à tel ou tel développement. Par exemple, l'implantation d'un module de manipulation d'un graphe des dépendances d'un projet B est nécessaire à l'implantation de l'algorithme de dépliage, mais sera utile pour d'autres développements, comme la gestion de projet.

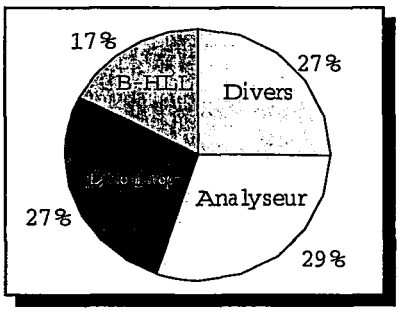


FIG. 5.2 – Taille du code : les principaux thèmes de développement

La figure 5.3 reprend les trois aspects et précise les modules constituant chacun d'eux. L'analyseur syntaxique est classiquement composé de trois modules : l'analyseur lexical (`Blexer`), l'analyseur syntaxique (`Bparser`) et la définition de la syntaxe abstraite du langage B (`Blast`).

La partie consacrée au dépliage de spécifications B est constituée du module `Depliage` qui permet de vérifier les conditions définies dans [Behnia00, Chapitre 3] autorisant l'utilisation de l'algorithme pour un sous ensemble de spécifications B. Le second module est le module qui réalise le dépliage (`Unfold`) et qui pour cela utilise les fonctionnalités définies dans le module `Unfold_lib`. `Unfold_main` est le module

principal construisant tout ce qui est nécessaire au dépliage (la construction du graphe de dépendance par exemple) et décompile le résultat en une sortie XML.

Les développements consacrés à la construction du système B-HLL comprennent la définition du module définissant la syntaxe abstraite et le typage du langage de base B (`Bhll`), la décompilation en XML (`Bhllxml`) et la transformation d'un arbre de syntaxe B «classique» (syntaxe définie dans `Blast`) en arbre de syntaxe B-HLL.

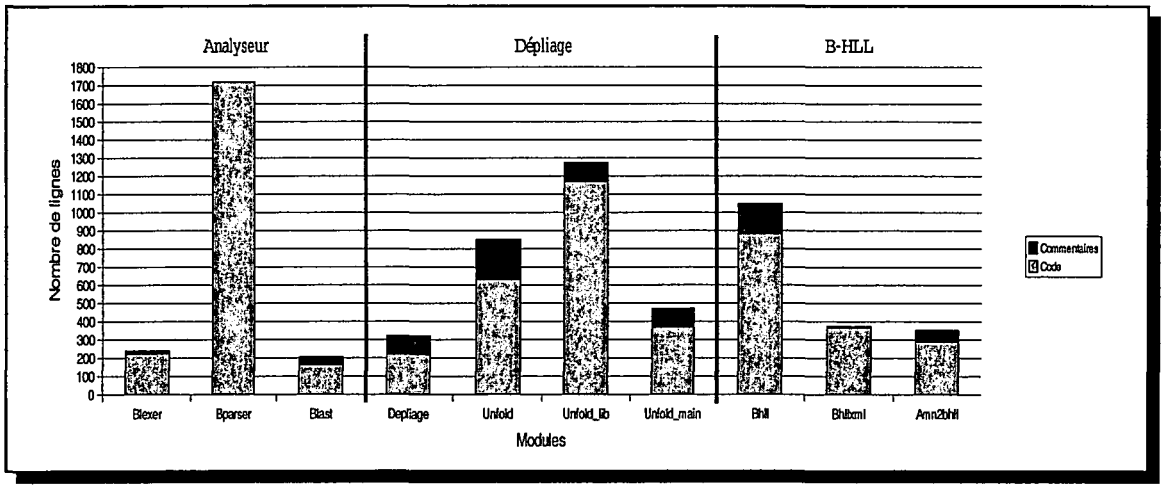


FIG. 5.3 – Les modules nécessaires à la génération de code

5.3 Perspectives

5.3.1 Langages cibles

Dans les travaux que nous avons présentés dans ce mémoire, nous avons illustré la génération de code en utilisant comme langages cibles les langages Ada et OCaml. Il est évident que d'autres langages peuvent être abordés pour la génération de code. Parmi les langages intéressants à explorer, Eiffel est un candidat naturel. En effet, ce langage fait référence en ce qui concerne le développement par contrat, et les outils développés autour de cette approche.

L'ouverture par les langages cibles à d'autres techniques et outils de validation est un aspect intéressant de la génération de code. Celle-ci pouvant s'adapter facilement, il est judicieux de tirer profit de différents outils. Parmi ces outils, nous en citerons deux particulièrement intéressants : Spark Ada [Barnes03] et Why [Filliatre03]. L'intérêt majeur de ces outils est la proximité des langages supports utilisés avec les langages traités dans le chapitre 4 (Ada et OCaml). Spark Ada utilise en effet un sous-langage Ada éliminant certaines constructions comme les tâches, constructions qui pour la majorité n'interviennent pas dans le code généré par nos outils. Le langage de l'outil Why est similaire au langage OCaml sans inférence de type. Les types des entités doivent donc être ajoutés dans le code généré. Les deux outils se basent sur les annotations pour générer des obligations de preuves pour le code. Ceci permettrait d'obtenir une preuve que le code est bien correct vis-à-vis de ses contrats.

5.3.2 Simplification des contrats générés

Dans les illustrations de génération de code que nous avons présentées, nous avons remarqué que les pré-conditions étaient «verbeuses». Il est facilement envisageable de simplifier l'expression de ces contrats avant ou après la phase de génération de code. Pour cela, il suffirait de se connecter à un système permettant de transformer les expressions des contrats en prédicats plus simples. Par exemple, en reprenant les spécifications de la section 4.3.3 (le module `Little_Example`), la pré-condition de l'opération `maximum` aplatie était :

$$\boxed{z \neq 0 \wedge \exists y.(y \in \mathbb{F}(NAT1) \wedge z = max(y \cup \{0\}) \wedge y \neq \varnothing)}$$

Or, cette pré-condition est en fait équivalente à :

$$\boxed{z \neq 0}$$

Pour réaliser cela, il est envisager de connecter l'outil Phox ([Raffalli et al.03]) pour simplifier automatiquement les prédicats des contrats générés à partir des spécifications B.

5.3.3 Intégration dans un processus de développement

Nous avons présenté dans la section 4.6 l'impact de notre approche sur le processus de développement B. Pour quantifier l'apport d'une approche mixte par rapport à une approche classique purement B, nous envisageons de développer les spécifications d'une étude de cas par les deux approches que nous qualifierons respectivement de pûre et mixte. L'approche mixte utilisera des outils de validation externes (par exemple ceux de la suite Spark-Ada) avant le processus de preuves classique B. L'approche pure B suivra le processus classique de développement B. Ces deux approches se baseront sur une première écriture des spécifications. Sur cette première version des spécifications, les deux processus de validation commenceront en parallèle. Le tableau 5.1 présente les quantités nous semblant intéressantes à obtenir par les deux approches. Les premières lignes du tableau sont consacrées à l'approche mixte et les dernières à l'approche purement B. Les résultats intéressants seront le nombre d'erreurs trouvées par les outils externes ainsi que le nombre de preuves interactives à effectuer. Ce nombre de preuves interactives devra tenir compte des incréments dans l'écriture des spécifications : une preuve interactive menant à trouver une erreur dans la spécification devra être comptabilisée autant de fois qu'il y aura eu de tentative de la décharger. Les résultats obtenus devront être comparés aux résultats de l'approche purement B pour calculer le gain, si il existe entre les deux approches (par le calcul de $\frac{NPI\_mixte}{NPI\_pure}$ par exemple).

Cette expérience permettra d'apporter une partie de la réponse à la question : quel gain est-il possible d'obtenir par une approche mixte. La réponse est partielle, car dans cette expérience, aucune technique de validation externe n'aura été mise en œuvre, la validation ne reposera que sur des outils préexistants. Une seconde étape dans cette voie serait de mener cette expérience au sein d'une entreprise de développement ayant développé ces propres techniques de validation pour un domaine d'application particulier.

5.3.4 Prise en compte des cas exceptionnels

Les mécanismes d'assertions mis en place dans la programmation par contrats sont associés dans la majorité des cas à des mécanismes d'exceptions. Ces exceptions sont utilisées pour signifier la détection

	Module_1	Module_2	Module_3	...	Total
Approche mixte					
Nombre d'erreurs trouvées par les outils externes à B					
Nombre d'OP					
Nombre d'OP triviales					
Nombre de preuves interactives (NPI_mixte)					
Nombre d'erreurs trouvées par la preuve					
Taux de preuve					
Approche B seule					
Nombre d'OP					
Nombre d'OP triviales					
Nombre de preuves interactives (NPI_pure)					
Nombre d'erreurs trouvées par la preuve					
Taux de preuve					

TAB. 5.1 – Les chiffres caractéristiques de l'utilisation mixte

et pour permettre la récupération d'événements inhabituels, telle que la rupture des contrats quand ceux ci sont activés à l'exécution. Notre approche consistant à «propager» les contrats des spécifications dans le code généré soulève naturellement une question : que se passe-t-il lorsqu'un contrat est rompu ? Cette question s'abordera différemment suivant les niveaux de sûreté ou les politiques de sûreté de fonctionnement désirés. Dans certains domaines comme les transports guidés, toute anomalie dans le système conduit à un même état, la mise en sécurité du système qui consiste à l'arrêt du train, de la rame de métro... Dans cette politique de sécurité, seul le système central doit tenir compte de la détection des anomalies, et aucune action corrective n'a besoin d'être mise en œuvre (la seule étant l'arrêt du système). Dans d'autres systèmes d'autres domaines d'activité, cette politique n'est pas applicable ; il n'est en effet pas envisageable qu'un système embarqué dans un avion applique une telle politique. Dans ce cas, il faut tant que faire se peut mettre en œuvre des mesures correctives. Ces mesures correctives peuvent être vues comme des actions dont le but est de remettre le composant dans un état correct. Cet état correct est caractérisé en B par un invariant. Une récupération d'erreur pourrait donc être assimilée à une opération dont la garde (la pré-condition) spécifierait le cas d'erreurs sensé être récupéré. Une telle opération ne pourrait être traitée de la même manière qu'une opération classique au niveau de la preuve, car les seules hypothèses connues dans ce cas seraient celles contenues dans la garde de l'opération. Une obligation de preuve d'une opération de cette catégorie (de la forme **SELECT** P **THEN** body **END**) serait de la forme :

$$\boxed{P \Rightarrow [body]Inv}$$

Cette notion d'opérations correctrices se rapproche de la notion d'événements du B événementiel. Des travaux pourraient donc être menés pour valider dans quelle mesure certaines notions du B événementiel pourraient être intégrées dans le B classique.

5.3.5 Extension de la notion de composant logiciel formel

Validation de composant B dépendant de composant extérieurs

Nous avons vu dans le chapitre 4 que la prise en compte de la notion de composant logiciel nécessite une certaine transformation du composant pour l'intégré dans le processus de preuve.

Cette approche nécessite de traduire les contrats exprimés dans un composant quelconque en prédicats reconnus dans le langage B. Nous envisageons de considérer les composants accompagnés d'une spécification dans laquelle les contrats sont exprimés en Object Constraint Language (OCL [UML, chapitre 6]). Des travaux existent pour la transformation des expressions OCL en expressions B, notamment [Marcano et al.01] et [Marcano et al.02a]. Ces travaux visent à analyser les expressions OCL en utilisant les outils B existants ([Marcano et al.02b]).

La réutilisation de ces travaux[1] permettra donc de régler la majeure partie concernant la transformation des contrats OCL en contrats B. Il restera une partie qui concerne la traduction des types du langage de spécification du composant que l'on souhaite réutiliser en langage B. En effet en B, l'absence de distinction entre les informations de typage et les informations de contrats nécessite de retranscrire dans les contrats une partie des informations de typage. Pour cela, les choix faits au niveau de la génération de code devront être réutilisés pour faire la transformation inverse du code vers B.

Prise en compte des aspects non-fonctionnels

La notion de composant vue dans le chapitre 4 peut être étendue. Les composants que nous générons satisfont les définitions qui ont été données dans le chapitre 1, mais seuls les aspects fonctionnels sont pris en compte. Comme les aspects non-fonctionnels constituent un élément important des composants, ceux-ci pourraient être pris en compte dans la poursuite de nos travaux.

Deux voies sont envisageables pour y arriver, la première consiste à remonter dans les spécifications B les notions nécessaires à l'expression des caractéristiques non fonctionnelles du composant. La seconde approche consiste à traiter ces aspects non fonctionnels en dehors du développement B soit en parallèle soit une fois les aspects fonctionnels traités.

Pour la première approche, des travaux sont en cours pour ajouter une notion de temps aux spécifications formelles B ([Colin01]). Comme pour les propriétés, ces annotations de durées ne sont *a priori* pas destinées dans un premier temps à être intégrées au code, mais comme pour les propriétés, il pourrait être intéressant pour certains domaines d'application dans lesquels les temps de réponse sont primordiaux de conserver ces indications de temps dans le code généré.

La seconde approche repose sur la séparation des aspects fonctionnels de ceux non-fonctionnels. Le principe est exposé dans la figure 5.4, il consiste a séparer les spécifications des aspects fonctionnels de ceux non fonctionnels pour ensuite mutualiser le résultat des deux voies pour obtenir un composant. Dans cette approche, les points durs seront la séparation des concepts fonctionnel et non-fonctionnel, car il ne parait pas évident qu'une distinction systématique soit envisageable. Un autre point dur sera la fusion entre les deux approches, comment dans la partie fonctionnelle utiliser les parties non fonctionnelles.

[1] Ces travaux de transformation d'expressions OCL en B sont en cours d'intégration dans la plateforme BCaml

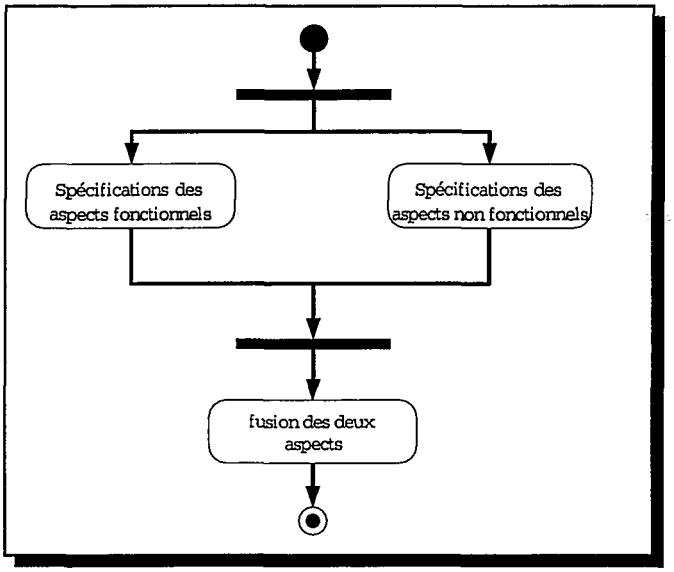


FIG. 5.4 – Développement séparé des aspects fonctionnels et non fonctionnels

Bien que la figure 5.4 paraisse simpliste, elle reflète bien une partie des recommandations que nous avons évoquées dans le chapitre 1 pour l'usage de méthodes formelles : l'utilisation d'une méthode formelle ne résoud pas tous les problèmes rencontrés lors d'un développement formel. Il est donc nécessaire de garder en permanence à l'esprit la possibilité d'utiliser d'autres techniques (soit de développement, soit de validation...) pour tirer profit de chaque technique.

5.3.6 Modèles de composant

Les composants logiciels que nous avons générés n'utilisent aucun des modèles de composants existants. La génération de composants conforme à un des modèles usuel ne nécessite pas d'effort particulier, car les données nécessaires sont déjà construites durant la phase d'analyse des spécifications. Une lacune commune aux différents modèles de composant est la non prise en compte des contrats dans les langages d'interface. Or les contrats font parties intégrantes de la spécification d'un composant et doivent donc apparaître au niveau des interfaces de ceux-ci. La génération de véritables composants sûrs nécessitera donc l'usage d'un modèle de composant dans lequel la notion de contrat sera intrinsèque au modèle.

5.4 Conclusions

Dans ce chapitre, nous avons présenté comment sont mis en œuvre les générateurs de code au sein de la plateforme BCaml. Cette plateforme destinée à l'expérimentation d'extensions ou de travaux autour de B a désormais atteint un niveau de maturité la rendant «utilisable» (au sens où les fonctionnalités de base permettant la manipulation des spécifications sont implantées). La génération de code repose d'ailleurs sur des parties qui ont été développées pour la plateforme, comme le vérificateur de sémantique statique par exemple.

Sur la base de la génération de code, quelques perspectives ont été présentées dans la seconde partie

du chapitre. Ces perspectives s'étendent du simple ajout d'un langage cible jusqu'à l'intégration d'un développement B dans un développement à base de composants.

Conclusion générale

Dans ce mémoire, nous avons abordé la génération de composants logiciels sûrs pour la méthode formelle B. Nous avons, dans un premier temps, rappelé trois approches pour le développement de logiciels : l'approche formelle, l'approche à base de composants et l'approche par contrats. Les approches à base de composants et par contrats sont intimement liées car réutiliser, le principe de base de l'approche par composants, nécessite une spécification de ce que fait le composant, ce qui est obtenu par l'approche par contrats. Les avantages et inconvénients de ces trois approches ont été rappelés. Un avantage commun aux trois approches est l'amélioration de la qualité du produit final. Les approches formelles peuvent être plus difficiles à mettre en œuvre à la différence de l'approche contractuelle par exemple, qui se trouve plus légère. De plus, le développement par composants permet une meilleure intégration du produit logiciel construit par des techniques formelles dans son environnement d'exécution final (l'utilisation de techniques formelles étant exclue pour l'intégralité d'un logiciel).

Après ces considérations d'ordre général, nous avons étudié la méthode B et le langage du même nom support à la méthode. Les principaux concepts intervenant dans la génération de code sont le langage B (les structures de contrôle et la manière de définir les entités des spécifications), la structuration des spécifications entre elles, et le raffinement pour comprendre comment les spécifications sont développées et ce qu'il est nécessaire de conserver des spécifications dans le code généré.

L'étude de la méthode et du langage B nous a mené à étudier plus précisément la modularité de B. Par une séparation des concepts du langage de base B et du langage de modularité, les concepts de modularité de B se sont ramenés à des concepts de signature de modules. Ceci a permis d'utiliser le système de modules à la Harper-Lillibridge-Leroy pour modéliser le système de modules de B. Cette modélisation permet d'exprimer les règles de visibilité de B par des règles de typage et permet également d'obtenir un vérificateur de sémantique statique pour B.

La modélisation du système de modules de B nous a permis d'aborder la génération de code plus sereinement, car les problèmes de manipulation des modules B étant déportés. La représentation HLL est une représentation utilisée dans certains langages de programmation ; elle est donc proche du niveau code (à opposer au niveau abstrait des spécifications). De plus la décomposition imposée par HLL du langage en langage des modules d'une part, et langage de base d'autre part, nous permet au niveau de la génération de code de séparer également les traitements de ces deux aspects.

Une nouveauté dans la génération de code que nous avons présentée est la conservation des propriétés des spécifications en les transportant dans le code produit. Cette conservation des propriétés nous permet d'utiliser des outils de validations du code différents des outils B actuellement fournis par les ateliers support de la méthode.

Un autre apport que nous avons présenté est la notion de composant logiciel. Même si, dans ce mémoire, l'aspect composant n'a pas été abordé en profondeur (une manière de le faire a été signalée en perspectives), il a néanmoins permis d'avancer dans cette voie par la constructions des informations primordiales pour la définition d'un composant : sa spécification (une interface contractualisée) et sa dépendance avec d'autres composants (là aussi exprimée par une ou plusieurs interfaces contractualisées). Cette approche nous a permis de soulever un problème dans la validation de spécifications dépendantes des autres spécifications : en B la seule donnée d'une interface de composant contractualisée ne suffit pas à prouver la validité d'un composant (utilisant le composant spécifié par l'interface). Une modification des obligations de preuves à mener pour la validation nous a rendu possible cette validation.

Un autre impact de notre génération de code se situe au niveau du processus de développement. Il est en effet possible d'utiliser des techniques et des outils issus des approches par contrats.

Tous les travaux que nous avons menés ici s'intègrent dans une plateforme d'expérimentation pour B intitulée BCaml. Ces travaux ouvrent des portes pour l'exploration de diverses pistes. Certaines concernent l'avancée dans les concepts composants logiciels et d'autres la validation du code obtenu par l'utilisation de techniques et outils différents de ceux utilisés actuellement en B.

Bibliographie

[Abrial et al.97] Abrial (Jean-Raymond) et Mussat (Louis). – Specification and design of a transmission protocol by successive refinements using B. *Mathematical Methods in Program Development*, éd. par Broy (Manfred) et Schieder (Birgit). pp. 129–200. – Springer, 1997.

[Abrial96] Abrial (Jean-Raymond). – *The B Book - Assigning Programs to Meanings*. – Cambridge University Press, août 1996.

[Ada94] Intermetrics, Inc. – *Ada reference Manual*, December 1994. Version 6.0.

[Aertryck et al.97] Aertryck (Lionel Van), Benveniste (Marc) et Metayer (Daniel Le). – CASTING : une m'ethode formelle de génération de cas de tests. *AFADL : Approches formelles dans l'assistance au développement de logiciel*, pp. 99–112. – Toulouse, France, mai 1997.

[Aertryck98] Aertryck (Lionel Van). – *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. – Thèse de doctorat, Université de Rennes I, janvier 1998.

[AFADL00] *Approches Formelles dans l'Assistance au Développement de Logiciels*. – LSR/IMAG – Grenoble – France, LSR/IMAG, janvier 2000.

[AFADL01] *Approches Formelles dans l'Assistance au Développement de Logiciels*. – ADER/LORIA Nancy – France, ADER/LORIA, juin 2001.

[Ancona et al.98] Ancona (D.) et Zucca (E.). – A theory of mixin modules : Basic and derived operators. *Mathematical structures in computer science*, vol. 8, n4, August 1998, pp. 401–446.

[Ariane96] Rapport de la commission d'enquête ariane 501, Juillet 1996. http ://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501.html.

[Arnout02] Arnout (Karine). – Extracting implicit contracts from .net libraries. *17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. – November 2002. Poster.

[Badeau et al.03] Badeau (Frédéric), Bert (Didier), Boulmé (Sylvain), Potet (Marie-Laure), Stouls (Nicoals) et Voisin (Laurent). – Traduction de B vers des langages de programmation : points de vue du projet BOM. IRISA, pp. 87–102. – IRISA Rennes – France, janvier 2003.

[Barnes03] Barnes (John). – *High Integrity Software. The Spark Approach to Safety and Security.* – Addison-Wesley, 2003.

[Bartetzko et al.01] Bartetzko (D.), Fischer (C.), Moller (M.) et Wehrheim (H.). – Jass - java with assertions. *Proceedings of the First Workshop on Runtime Verification.* – 2001.

[BCaml] *BCaml.* https ://savannah.nongnu.org/projects/brillant/.

[Behm et al.] Behm (Patrick), Benoit (Paul), Faivre (Alain) et Meynadier (Jean-Marc). – METEOR : A successful application of B in a large project. *Proceedings of FM'99 : World Congress on Formal Methods*, pp. 369–387.

[Behm et al.97] Behm (Patrick), Desforges (Pierre) et Mejia (Fernando). – *Application de la méthode B dans l'industrie ferroviaire*, chap. III, pp. 59–88. – In OFTA [Ofta97].

[Behm96] Behm (Patrick). – Développement formel des logiciels sécuritaires de METEOR. In Habrias [Habrias96], pp. 3–10.

[Behnia et al.98] Behnia (Salimeh) et Waeselynck (Hélène). – *External verification of a B development process.* – Rapport technique n98085, LAAS (TSF) – INRETS (ESTAS), 1998.

[Behnia00] Behnia (Salimeh). – *Test de modèles formels en B : cadre théorique et critères de couvertures.* – Thèse de doctorat, Institut National Polytechnique de Toulouse, octobre 2000.

[Benzaken et al.03] Benzaken (V.), Castagna (G.) et Frisch (A.). – Cduce : An xml-centric general-purpose language. *Proceedings of the ACM International Conference on Functional Programming.* – 2003.

[Bert et al.96] Bert (D.), Potet (M.-L.) et Rouzaud (Y.). – A study on components and assembly primitives in B. In Habrias [Habrias96], pp. 47–62.

[Bert98] Bert (Didier) (édité par). – *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier. – Montpellier, Springer Verlag, april 1998.

[Bieber et al.94] Bieber (Pierre) et Boulahia-Cuppens (N.). – Formal development of authentication protocols. *Proceedingsof BCS-FACS Sixth Refinement Workshop.* – 1994.

[Bieber95] Bieber (Pierre). – Spécification et vérification avec la méthode B d'un protocole de sécurité. *Journées Formalisation des Activités Concurrentes.* – Toulouse, avril 1995.

[Bodeveix et al.99] Bodeveix (Jean-Paul), Filali (Mamoun) et Munoz (César). – A formalization of the B method in Coq and PVS. *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques* [BUG99], pp. 32–48.

[Bodeveix et al.00] Bodeveix (J.-P.), Filali (Mamoun) et Munoz (C.A.). – Formalisation de la méthode B en COQ et PVS. In AFADL [AFADL00], pp. 96–110.

[Bodeveix et al.02] Bodeveix (Jean-Paul) et Filali (Mamoun). – Type synthesis in B and the translation of B to PVS. In ZB [ZB02], pp. 350–369.

[Bom] Bom : B with Optimized Memory. http ://lifc.univ-fcomte.fr/ tatibouet/WEBBOM/.

[Bon00] Bon (Philippe). – *Du cahier des charges aux spécifications formelles : une méthode basée sur les réseaux de Petri de haut niveau.* – Thèse de doctorat, Université des Sciences et Techniques de Lille, octobre 2000.

[Booch et al.99] Booch (G.), Rumbaugh (J.) et Jacobson (I.). – *The UML Reference Guide.* – Addison-Wesley, 1999.

[Bowen et al.94] Bowen (J. P.) et Hinchey (M. G.). – *Seven More Myths of Formal Methods.* – Rapport technique nPRG-TR-7-94, Oxford University Computing Laboratory, Programming Research Group, 1994. Shorter version in FME94 and in IEEE Software Vol. 12 Number 4.

[Bowen et al.95] Bowen (J. P.) et Hinchey (M. G.). – Ten commandments of formal methods. *IEEE Computer*, vol. 28, n4, avril 1995, pp. 56–63.

[Brillant] Brillant : B recherches et innovations logicielles à l'aide de nouvelles technologies. https ://savannah.nongnu.org/projects/brillant/.

[Buchi et al.] Büchi (Martin) et Back (Ralph). – Compositional symmetric sharing in B. *Proceedings of FM'99 : World Congress on Formal Methods*, pp. 431–451.

[BUG99] *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques.* – Springer-Verlag, 1999.

[Burdy00] Burdy (Lilian). – *Traitement des expressions dépourvues de sens de la théorie des ensembles : Application à la méthode B.* – Thèse de PhD, CEDRIC-CNAM, 2000.

[Cc96] Common Criteria for Information Technology Security Evaluation, 1996. The Common Criteria Project Sponsoring Organisations.

[Chartier98] Chartier (Pierre). – Formalisation of B in Isabelle/HOL. In Bert [Bert98], pp. 66–82.

[Cheon et al.02] Cheon (Yoonsik) et Leavens (Gary T.). – *A Runtime Assertion Checker for the Java Modeling Language (JML.* – Rapport technique n 02-05, Department of Computer Science, Iowa State University, mars 2002. To appear in SERP 2002.

[Clearsy] ClearSy. – Atelier B. http ://www.atelierb.societe.com/-index_uk.html.

[Colin01] Colin (Samuel). – Méthode B et temps réel : étude de l'intégration du calcul des durées. – 2001. Mémoire de DEA.

[Coq] Le système coq. http ://pauillac.inria.fr/coq/doc-fra.html.

[Crnkovic et al.02] Crnkovic (Ivica) et Larsson (Magnus) (édité par). – *building reliable component-based Software Systems.* – ARTECH HOUSE, 2002.

[Dehbonei et al.95] Dehbonei (Babak) et Mejia (Fernando). – Formal development of safety-critical software systems in railway signalling. *Applications of Formal Methods*, éd. par Hinchey (M. G.) et Bowen (J. P.), pp. 227–252. – Prentice Hall International, 1995.

[Detlefs et al.98] Detlefs (David L.), Leino (K. Rustan M.), Nelson (Greg) et Saxe. (James B.). – *Extended Static Checking*. – Rapport technique, Compaq Systems Research Center, December, 1998. Research Report 159.

[Detlefs96] Detlefs (David L.). – An overview of the extended static checking system. *Proceedings of The First Workshop on Formal Methods in Software Practice*, pp. 1–9. – January 1996.

[Dimitrakos et al.00] Dimitrakos (Theo), Bicarregui (Juan), Matthews (Brian), Maibaum (Tom) et Robinson (Ken). – Compositional structuring in the B Method : A Logical Viewpoint of the Static Context. *ZB'2000 – International Conference of B and Z Users* [ZB000], pp. 107–126.

[Einer et al.] Einer (S.), Schrom (H.), Slovák (R.) et Schnieder (E.). – A Railway demonstrator model for experimental investigation of integrated specification techniques.

[En5012801] EN 50128 Railway applications- Communications, signalling and processing systems- Software for railway control and protection systems, July 2001.

[Evans et al.94] Evans (David), Guttag (John), Horning (James) et Tan (Yang Meng). – LCLint : A tool for using specifications to check code. *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pp. 87–96. – 1994.

[Evans et al.02] Evans (David) et Larochelle (David). – Improving security using extensible lightweight static analysis. *IEEE Software*, 2002.

[Facon et al.96] Facon (P.), Laleau (R.) et Nguyen (H.P.). – Dérivation de spécifications formelles B à partir de spécifications semi-formelles de systèmes d'information. In Habrias [Habrias96], pp. 271–290.

[Facon et al.00] Facon (Philippe), Laleau (Régine) et Nguyen (Hong Phuong). – *From OMT Diagrams to B Specifications*, chap. Software Specification Methods : an Overview Using a Case Study. – Springer, FACIT series, 2000.

[Filliatre03] Filliatre (Jean-Christophe). – Why : a multi-language multi-prover verification tool. *Proceedings of Types Conference*. – April 2003.

[Flanagan et al.02] Flanagan (C.), Leino (K.), Lillibridge (M.), Nelson (C.), Saxe (J.) et Stata (R.). – Extended static checking for java, 2002.

[Fuchs et al.95] Fuchs (Norbert E.) et Schwitter (Rolf). – Attempto : Controlled natural language for requirements specifications. *Seventh Workshop on Logic Programming Environments*, pp. 25–32. – 95.

[Fuchs02] Fuchs (N. E.). – *Reasoning in Attempto Controlled English*. – Rapport technique, U. Schwertel, 2002.

[Gurevich99] Gurevich (Y.). – The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, vol. 67, 1999, pp. 93–124.

[Habrias96] Habrias (Henri) (édité par). – *Proceedings of the 1st Conference on the B method.* IRIN Institut de recherche en informatique de Nantes. – novembre 1996.

[Habrias01] Habrias (Henri). – *Spécification formelle avec B.* – Lavoisier-Hermès, 2001.

[Hall90] Hall (Anthony). – Seven myths of formal methods. *IEEE Software*, vol. 7, n5, 1990, pp. 11–19.

[Harper et al.86] Harper (Robert), MacQueen (David B.) et Milner (Robin). – *Standard ML.* – Rapport technique, University of Edinburgh, 1986. ECS LFCS 86-2.

[Harrison] Harrison (John). – Hol light. – http ://www.cl.cam.ac.uk/users/jrh/hol-light/.

[Havelund et al.01] Havelund (Klaus) et Rosu (Grigore) (édité par). – *Runtime Verification.* – 2001. Satellite Workshop of CAV'01.

[Hinchey et al.95] Hinchey (Michaek G.) et Bowen (Jonathan P.). – *Applications of Formal Methods*, chap. Applications of Formal Methods FAQ. – Prentice Hall, 1995.

[Houston et al.91] Houston (Ian) et King (Steve). – Cics project report, experiences and results from the use of Z in IBM. *VDM '91. Formal Software Development Methods.* – Springer, 1991.

[Java2] The source for java technology. http ://java.sun.com/.

[Jezequel et al.97] Jézéquel (Jean-Marc) et Meyer (Bertrand). – Design by contract : The lessons of Ariane. *Computer IEEE*, vol. 30, n1, January 1997, pp. 129–130.

[Julliand et al.98] Julliand (Jacques), Legeard (B.), Machicoane (T.), Parreaux (B.) et Tatibouet (B.). – Specification of an integrated circuit card. protocol application using the B method and linear temporal logic. In Bert [Bert98], pp. 273–292.

[Laleau et al.01a] Laleau (Régine) et Mammar (Amel). – An automatic generation of b specifications from well-defined uml notations for database applications. *Proceedings of International Symposium on Programming Systems.* – Mai 2001.

[Laleau et al.01b] Laleau (Régine) et Mammar (Amel). – An automatic generation of b specifications from well-defined uml notations for database applications. *Proceedings of International Symposium on Programming Systems.* – Mai 2001.

[Laleau et al.02] Laleau (Régine) et Polack (Fiona). – Coming and going from UML to B : A proposal to support traceability in rigorous is development. In ZB [ZB02], pp. 517–534.

[Lanet] Lanet (Jean-Louis). – Using the B method to model protocols. *AFADL'98*, pp. 79–90.

[Lanet et al.98] Lanet (Jean-Louis) et Requet (Antoine). – Formal proof of smart card applets correctness. *Proceedings of the Third Smart Card Research and Advanced Application Conference (CARDIS'98).* – Louvain-la-Neuve, (be), septembre 1998.

[Lanet00] Lanet (Jean-Louis). – Are smart cards the ideal domain for applying formal methods ? *ZB'2000 – International Conference of B and Z Users*, pp. 363–374. – 2000.

[Lano et al.96] Lano (K.), Bicarregui (J.) et Sanchez (A.). – Using B to design and verify controllers for chemical processing. In Habrias [Habrias96], pp. 237–270.

[Lano et al.98] Lano (Kevin), Bicarregui (J.C.) et Kan (P.). – Experiences of using formal methods for chemical process control specification. pp. 119–133. – juin 1998.

[Larousse] *Le Petit Larousse*. – Larousse.

[Ledang01] Ledang (Hung). – Des cas d'utilisation à une spécification B. In AFADL [AFADL01], pp. 131–140.

[Leino01] Leino (K. Rustan M.). – Extended static checking : A ten-year perspective. *Lecture Notes in Computer Science*, vol. 2000, 2001, pp. 157– ? ?

[Leroy96] Leroy (Xavier). – *A modular module system*. – Research report n2866, INRIA, avril 1996.

[Leroy00a] Leroy (Xavier). – A modular module system. *Journal of Functional Programming*, vol. 10, n3, 2000, pp. 269–303.

[Leroy00b] Leroy (Xavier). – Web appendix to a modular module system, 2000. http ://pauillac.inria.fr/ xleroy/publi/modular-modules-appendix/.

[Levy et al.02] Levy (N.), Marcano (R.) et Souquières (J.). – From requirements to formal specification using uml and b. *2nd International Conference in Computer Systems and Technologies CompSysTech2002*. – Sofia, Bulgaria, June 2002.

[Luders et al.02] Lüders (Frank), Lau (Kung-Kiu) et Ho (Shui-Ming). – *Specification os Software Components*, chap. 2. – In Crnkovic et Larsson [Crnkovic et al.02].

[Macqueen86] MacQueen (David B.). – *Modules for Standard ML*. – Rapport technique, University of Edinburgh, 1986. in [Harper et al.86].

[Marcano et al.01] Marcano (Rafael) et Lévy (Nicole). – Transformation d'annotations OCL en expressions B. In AFADL [AFADL01], pp. 39–49.

[Marcano et al.02a] Marcano (R.) et Levy (N.). – Transformation rules of ocl constraints into b formal expressions. *CSDUML'2002, Workshop on critical systems development with UML. 5th International Conference on the Unified Modeling Language*. – Dresden, Germany, September 2002.

[Marcano et al.02b] Marcano (R.) et Levy (N.). – Using b formal specifications for analysis and verfication of uml/ocl models. *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*. – Dresden, Germany, September 2002.

[Mariano et al.99] Mariano (Georges), Boulanger (Jean-Louis) et Tatibouet (Bruno). – *Revisiting B language syntax*. – Rapport technique, Laboratoire CEDRIC-CNAM, 1999.

[Mariano97] Mariano (Georges). – *Évaluation de logiciels critiques développés par la méthode B : une approche quantitative.* – Thèse de doctorat, Universitée de Valenciennes et du Hainaut-Cambrésis, Dec 1997, 160p.

[Mcilroy68] McIlroy (M. D.). – Mass produced software components. *Nato Software Engineering Conference*, pp. 138–155. – 1968.

[Meyer et al.99] Meyer (Eric) et Souquières (Jeannine). – Systematic approach to transform OMT diagrams to a B specification. *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques* [BUG99], pp. 875–895.

[Meyer92] Meyer (Bertrand). – *Eiffel : The Language.* – Prentice Hall, 1992.

[Meyer97] Meyer (Bertrand). – *Object Oriented Software Construction.* – Prentice-Hall, 1997. second edition.

[Meyer99] Meyer (Bertrand). – The Significance of Components. *Software Development Magazine*, November 1999. – online magazine (http ://www.sdmagazine.com/).

[Meyer00] Meyer (Bertrand). – *Conception et programmation orientées objet.* – Eyrolles, 2000. traduction de [Meyer97].

[Meyer01] Meyer (Eric). – *Développement formel par objets : utilisation conjointe de B et UML.* – Thèse de doctorat, Université de Nancy, 2001.

[Microsofta] Microsoft. – Component Object Model (COM). http ://www.microsoft.com/com/.

[Microsoftb] Microsoft. – Component Object Model (COM). http ://www.microsoft.com/net/.

[Microsoft95] Microsoft. – The component object model specification, October 1995. Version 0.9.

[Microsystemsa] Microsystems (Sun). – Enterprise Javabeans. http ://java.sun.com/products/ejb/.

[Microsystemsb] Microsystems (Sun). – Javabeans. http ://java.sun.com/products/javabeans/.

[Monin00] Monin (Jean-François). – *Introduction aux méthodes formelles.* – hermes, 2000.

[Motre et al.00] Motré (Stéphanie) et Téri (Corinne). – Using formal and semi-formal methods for a common criteria evaluation. *Eurosmart.* – Marseille (France), jun 2000.

[Motre00] Motré (Stéphanie). – A B automaton for authentication process. *WITS'2000.* – Geneva (Switzerland), jul 2000.

[Nimmer et al.01] Nimmer (Jeremy W.) et Ernst (Michael D.). – Static verification of dynamically detected program invariants : Integrating daikon and esc/java. In Havelund et Rosu [Havelund et al.01]. Satellite Workshop of CAV'01.

[Ofta97] OFTA (édité par). – *Application des techniques formelles au logiciel.* – Observatoire Français des Techniques Avancéees & Lavoisier TEC & DOC, 1997.

[Omga] OMG. – Common Object Request Broker Architecture (CORBA). http ://www.corba.org/.

[Omgb] OMG. – Corba Component Model. http ://www.omg.org/.

[Owre et al.99] Owre (S.), Shankar (N.), Rushby (J. M.) et Stringer-Calvert (D. W. J.). – *PVS System Guide.* – Computer Science Laboratory, SRI International, Menlo Park, CA, septembre 1999.

[Petin et al.98] Pétin (J.-F.), Morel (G.), Méry (D.) et Lamboley (P.). – Process control engineering : contribution to a formal structuring framework with the B method. In Bert [Bert98], pp. 198–209.

[Petit et al.01] Petit (Dorian), Mariano (Georges) et Poirriez (Vincent). – *Flattening B Components for Code Generation.* – Rapport technique n INRETS/RT-01-716-FR, INRETS, July 2001.

[Petit et al.02a] Petit (Dorian), Mariano (Georges) et Poirriez (Vincent). – Vers un système de module à la Harper-Lillibridge-Leroy pour les spécifications formelles B. *JFLA : Journées Francophones des Langages Applicatifs.* – Janvier 2002. pp 85-100.

[Petit et al.02b] Petit (Dorian), Poirriez (Vincent) et Mariano (Georges). – Development of Formal Components Using the B Method. *Proceedings of the First COLOGNET Joint Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems*, éd. par Carro (Manuel), Vaucheret (Claudio) et Lau (Kung-Kiu), pp. 35–46. – September 2002.

[Petit et al.03a] Petit (Dorian), Mariano (Georges) et Poirriez (Vincent). – Génération de composants à partir de specifications B. *Actes de AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, éd. par Jézéquel (Jean-Marc), pp. 103–119. – Janvier 2003.

[Petit et al.03b] Petit (Dorian), Mariano (Georges), Poirriez (Vincent) et Boulanger (Jean-Louis). – Automatic Annotated Code Generation from B Formal Specifications. *Symposium on Formal Methods for Railway Operation and Control Systems*, éd. par Tarnai (G.) et Schnieder (E.). pp. 37–44. – L'Harmattan, May 2003.

[Petit et al.03c] Petit (Dorian), Poirriez (Vincent) et Mariano (Georges). – The B method and the component-based approach. *Formal Reasoning on Software Components and Component Based Software Architectures.* – December 2003. Special topic session of the Seventh Conference on Integrated Design and Process Technology. To appear.

[Petit98] Petit (Dorian). – *Réalisation d'un moteur d'analyse pour les spécifications B.* – Rapport technique, INRETS, Juillet 1998. Rapport de stage de maîtrise informatique.

[Potet et al.98] Potet (Marie-Laure) et Rouzaud (Yann). – Composition and refinement in the B method. In Bert [Bert98], pp. 46–65.

[Pratten95] Pratten (C.H.). – An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. *Proc. Z Twenty Years on - What is its Future*, éd. par HABRIAS (Henri). IRIN-IUT de Nantes, pp. 149–165. – October 1995.

[Raffalli et al.03] Raffalli (Christophe) et Roziere (Paul). – *The PhoX Proof checker Documentation*, February 2003. http ://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/af2.html.

[Requet et al.00] Requet (Antoine), Casset (Ludovic) et Grimaud (Gilles). – Application of the B formal method to the proof of a type verification algorithm. *HASE 2000.* – Albuquerque (NM), novembre 2000.

[Requet00] Requet (Antoine). – A B model for ensuring soundness of the Java card virtual machine. *FMICS'2000.* – Berlin, mar 2000.

[Rushby00] Rushby (John). – Disappearing formal methods. *High-Assurance Systems Engineering Symposium.* Association for Computing Machinery, pp. 95–96. – Albuquerque, NM, novembre 2000.

[Saez et al.01] Sáez (José), Toval (Ambrosio) et Alemán (José Luis Fernández). – Tool support for transforming uml models to a formal language. *WTUML : Workshop on Transformations in UML (ETAPS 2001 Satellite Event).* – April 2001.

[Sekerinski98] Sekerinski (E.). – Graphical design of reactive systems. In Bert [Bert98], pp. 182–197.

[Seshia et al.] Seshia (S. A.), Shyamasundar (R.K.), Bhattacharjee (A.K.) et Dhodapkar (S.D.). – A translation of statecharts to esterel. *Proceedings of FM'99 : World Congress on Formal Methods*, pp. 983–1007.

[Steria98] STERIA. – *Manuel de référence du langage B version 1.7*, mars 1998.

[Szyperski98] Szyperski (Clemens). – *Component Software - Beyond Object-Oriented Programming.* – 1998, addison-wesley / acm press édition.

[Szyperski00] Szyperski (Clemens). – Point, Counterpoint. *Software Development Magazine*, February 2000. – online magazine (http ://www.sdmagazine.com/).

[Tarnai et al.03] Tarnai (Géza) et Schnieder (Eckehard) (édité par). – *Formal Methods for Railway Operation and Control Systems (FORMS).* – L'Harmattan, mai 2003.

[Tcsec85] TCSEC : Trusted Computer Systems Evaluation Criteria, 1985. DOD 5200.28-STD Department of Defence, United States of America.

[UML] OMG. – *Unified Modeling Language (UML).* formal/03-03-01, version 1.5.

[W3c] W3C (World Wide Web Consortium). – XSL transformations (XSLT). http ://www.w3.org/TR/xslt/.

[Waeselynck et al.95] Waëselynck (Hélène) et Boulanger (Jean-Louis). – The role of testing in the B formal development process. *Proceedings of 6th International Symposium on software Reliability Engineering (ISSRE'95).* Toulouse, pp. p.205–28. – IEEE Comput. Soc. Press, Los Alamitos, CA, USA, October 24-27 1995.

[Waeselynck et al.97] Waeselynck (Hélène) et Behnia (Salimeh). – *B model animation for external verification.* – Rapport technique n97392, LAAS (TSF) – INRETS (ESTAS), 1997.

[Warnock et al.02] Warnock (J. D.), Keaty (J. M.), Petrovick (J.), Clabes (J. G.), Kircher (C. J.), Krauter (B. L.), Restle (P. J.), Zoric (B. A.) et Anderson (C. J.). – The circuit and physical design of the power4 microprocessor. *IBM Journal of Research and Development*, vol. 46, n1, 2002, pp. 27–52.

[Watson97] Watson (Geoffrey Norman). – A comparison of modularity in B and Cogito. *Formal Methods Pacific*, éd. par Groves (S. Reeves L.), pp. 263–286. – 1997.

[ZB000] *ZB'2000 – International Conference of B and Z Users.* – Helsington, York, UK YO10 5DD, août 2000.

[ZB02] LSR-IMAG. – *ZB'2002 – Formal Specification and Development in Z and B.* – Grenoble, France, janvier 2002.

Titre : Génération automatique de composants logiciels sûrs à partir de spécifications formelles *B*

Résumé. Ce mémoire est consacré à l'étude de la génération de composants logiciels sûrs à partir de spécifications formelles B. B est une méthode de spécification formelle basée sur la théorie des ensembles, le calcul des prédicats du premier ordre et le raffinement. L'aspect primordial à maîtriser pour la génération de code est la modularité du langage B. Nous proposons dans ce mémoire une modélisation du système de modules du langage B utilisant un système de modules à la Harper-Lillibridge-Leroy (HLL). HLL est un système de modules (à la ML) générique qui peut être instancié pour un grand nombre de langages de programmation. Cette modélisation nous permet de clarifier certains aspects de la modularité de B et nous permet ainsi d'avoir une représentation des spécifications B qui pourra ensuite être utilisée lors de la phase de génération de code. Ce nouveau système de modules pour B nous permet également d'aborder une technique de production de logiciel, la programmation par composants. Cette technique est souvent utilisée conjointement à la programmation par contrats. La prise en compte des aspects liés à la démarche par composants découle directement de la modélisation du système B-HLL dans lequel les foncteurs permettent d'exprimer des dépendances entre composants. L'approche contractuelle est prise en compte par la conservation des propriétés exprimées dans les spécifications B. Elles sont traduites dans le code généré, ce qui est nouveau par rapport aux démarches de génération de code existantes. Notre démarche de génération de code permet donc de marier deux techniques de développement à celle de la méthode B pour tirer partie des avantages de chacunes.

Mots clefs : Méthode formelle, méthode B, modularité, génération de code, développement par composants, développement par contrats.

Title : Automatic generation of reliable components from B formal specifications

Abstract. This work is based on the study of code generation from B formal specifications. B is a formal specification method based on set theory, first order predicate calculus and refinement. The main aspect that should be qddressed in code generation is the modularity of the B language. We have expressed the B modularity with an Harper-Lillibridge-Leroy module system (an ML-like generic module system). This model clarifies some aspects of the B modularity and gives us a representation of the B modules that are used during the phase of code generation. This new module system for the B language allows us to tackle a software production technique : the components based approach. This technique is often used in conjunction with the design by contracts technique. The component aspects in the code we generate are directly translate from our B-HLL system of modules. The contracts are expressed in the B specification and keep in the code we generate (in contrast to the current approaches for code generation). Our code generation process is a blend of the design by contracts approach, the component based approach and the B method. We can take advantage of the three approaches to develop software.

Keywords : formal method, B method, modularity, code generation, component based development, design by contract.