# Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B

Marie-Laure Potet

**Document d'Habilitation à Diriger des Recherches**
**Institut National Polytechnique de Grenoble**

# Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B

Marie-Laure Potet

Le 5 décembre 2002

| | | |
|---|---|---|
| Composition du jury : | Roger Mohr | *Président* |
| | Egon Börger | *Rapporteur* |
| | Nicolas Halbwachs | *Rapporteur* |
| | Véronique Viguié Donzeau-Gouge | *Rapporteur* |
| | Paul Jacquet | *Examinateur* |
| | Jean-Louis Lanet | *Examinateur* |

# Remerciements

Merci à Roger Mohr, Professeur à l'Institut National Polytechnique de Grenoble, d'avoir accepté de présider ce jury. Je le remercie pour l'intérêt qu'il a porté à mon travail.

Merci à Egon Börger, Professeur à l'Université de Pise et spécialiste des Abstract State Machines, d'avoir pris le temps de rapporter sur les résultats présentés dans ce manuscrit.

Merci à Nicolas Halbwachs, Directeur de Recherche CNRS au laboratoire Vérimag, d'avoir accepté la tâche de rapporteur local. Les discussions que nous avons eues, ainsi que ses remarques, m'ont permis d'affiner ma vision du domaine.

Merci à Véronique Viguié Donzeau-gouge, Professeur au Conservatoire National des Arts et Métiers de Paris, pour l'intérêt qu'elle porte à mes travaux et pour les discussions qui en ont suivi.

Merci à Jean-Louis Lanet, Chercheur au centre Recherche et Développement de Gemplus, pour les différents échanges que nous avons eus et qui m'ont permis de mieux appréhender les enjeux des méthodes formelles en milieu industriel.

Merci à Paul Jacquet, Professeur à l'Institut National Polytechnique de Grenoble, d'avoir accepté de participer à ce jury, malgré ses responsabilités actuelles. Je le remercie de m'avoir initié à la recherche et pour le soutien qu'il m'a toujours apporté.

Bien entendu mes remerciements vont à tous mes collègues, académiques

ou industriels, qui par des discussions, commentaires et collaborations ont, d'une manière ou d'une autre, alimenté mes travaux. Un merci particulier à Jean-Raymond Abrial, d'avoir inventé la méthode B, mais aussi pour tous les échanges que nous avons eus. Bien sûr ce travail est aussi le fruit de mes activités d'enseignement, et je remercie en particulier toute l'équipe du projet génie logiciel de l'Ensimag pour les longues discussions que nous avons pu avoir. Enfin tous les étudiants avec lesquels j'ai été amené à travailler ont, par leurs questions et idées, contribué à l'avancement de mes travaux, je les en remercie.

Ce travail est aussi le résultat du soutien de toute ma famille, qui m'a toujours épaulée et encouragée dans cette démarche. Un merci particulier à Colin et Christophe pour leur aide et pour avoir, malgré tout, bien profité de cet été 2002 !

# Table des matières

# Chapitre 1

# Introduction

En 1988 j'ai soutenu ma thèse intitulée *Preuves et stratégies pour la synthèse déductive de programmes*. En 1989 j'ai été recruté sur un poste de maître de conférences à l'Ensimag. Voici maintenant ce document, *Spécifications et développements formels : étude des aspects compositionnels dans la méthode B*, rédigé dans le cadre de l'Habilitation à Diriger des Recherches. Bien entendu, la rédaction d'un tel document est toujours l'occasion de faire un point, ne serait-ce que pour en déterminer le contenu et le style. Quels ont été les fils conducteurs des différents travaux que j'ai menés ou auxquels j'ai participé? Quelles ont été les complémentarités entre mes activités d'enseignement et de recherche? Enfin, quelle teneur ai-je envie de donner à ce manuscrit, et pour quelles raisons? Je profiterai donc de cette introduction pour situer mes activités et mes travaux.

## 1.1 Les fils conducteurs

Mes activités d'enseignement et de recherche se situent dans le cadre de ce que je qualifierai de *Génie Logiciel Formel*, c'est-à-dire du processus de construction de programmes, utilisant fortement le fait que les langages de programmation sont sémantiquement définis. Si, de plus, nous disposons d'un cadre logique permettant de raisonner sur les programmes, il devient alors possible de raisonner aussi sur le processus qui permet de les construire. En effet un programme est un objet qu'on peut exécuter, mais aussi évaluer et optimiser. Il est par exemple possible de vérifier des propriétés valables pour

toute exécution. De la même manière si les spécifications sont elles-mêmes énoncées dans une notation ayant une sémantique bien définie, on peut aussi valider des propriétés plus en amont du cycle de développement, produire des tests de conformité pour lesquels on dispose d'un oracle, mais aussi valider la correction des implémentations vis-à-vis des spécifications. Ceci constitue les fondements du développement formel de logiciels. Bien entendu, il faut aussi s'intéresser à l'applicabilité, à la mise en œuvre et aux limitations de ces techniques formelles. C'est l'aspect génie logiciel ou méthodologique qui permet de savoir quels outils conceptuels appliquer, comment et dans quels objectifs, ceci afin de pouvoir évaluer et quantifier l'apport des techniques utilisées. Le dernier aspect est l'outillage, qui permet une prise en charge automatisée d'une telle approche. En plus de décharger l'utilisateur d'une partie du travail, les outils assurent une certaine garantie sur l'utilisation d'une technique[1]. Les lignes directrices de mes activités d'enseignement et de recherche ont donc comme objectif les études suivantes :

| les programmes comme objets sur lesquels on peut raisonner<br>et<br>le développement comme un processus formalisé, ou du moins formalisable. |
|---|

Et, pour être complètes, ces études doivent être déclinées suivant plusieurs axes : les besoins, les fondements théoriques, la méthodologie et l'outillage. Ces aspects complémentaires se retrouvent dans mes différentes activités. Côté enseignement, cette complémentarité prend la forme de cours fondamentaux, dans lesquels les principes sont exposés, et dans le montage et le suivi de projets, qui permettent aux étudiants la mise en œuvre de la pratique. Côté recherche, les contacts et les collaborations industrielles permettent de localiser les problèmes réels et de chercher des réponses satisfaisantes en coût et en complexité.

Bien qu'une Habilitation à Diriger des Recherches porte principalement sur les activités de recherche, j'ai décidé de m'attarder, au moins dans cette introduction, aussi sur mes activités d'enseignement car ces deux volets de mon métier sont intrinsèquement mélés. L'enseignement du génie logiciel, les interrogations des étudiants, le suivi et le montage de projets poussent naturellement à affiner son approche personnelle du domaine.

1. Les vérifications statiques éliminent par exemple les entrées non interprétables.

## 1.2 Enseignement

J'illustrerai la mise en œuvre de ces objectifs dans la formation des ingénieurs Ensimag, en décrivant certains enseignements auxquels j'ai fortement participés, et les évolutions apportées. Ces enseignements concernent principalement les aspects langage et compilation ainsi que le génie logiciel.

L'étude de la compilation permet aux étudiants de comprendre et maîtriser finement l'outil principal qu'ils utilisent, les langages de programmation. Cette thématique est donc fortement enseignée à l'Ensimag et maintenant aussi dans le département Télécommunications. Dès le début de mon activité à l'Ensimag j'ai participé activement à cet enseignement, puis assuré les cours. En plus d'initier les étudiants aux rudiments des techniques d'analyse et de génération de code, l'objectif visé est aussi de sensibiliser les étudiants aux concepts sous-jacents aux langages de programmation. Cet aspect est important pour des ingénieurs qui seront amenés à utiliser, concevoir et voir évoluer les environnements liés au développement logiciel. Il permet aux étudiants de comprendre plus finement les choix méthodologiques des langages, et les implications à la fois sur leur utilisation et sur les outils qui leur sont associés. On peut par exemple illustrer ceci, dans le cadre des langages objets, par différentes approches possibles de l'héritage et de la redéfinition implicite. Veut-on, ou non, admettre l'héritage multiple ? Quels sont alors les problèmes sémantiques ? De la même manière la redéfinition implicite d'une méthode peut prendre différentes formes [2] suivant le langage. En Java, dans le cas d'une redéfinition, seul le type de l'objet sur lequel s'applique la méthode peut varier. Dans le langage Ada, une redéfinition impose de remplacer toutes les occurrences d'un même type par son sous-type, même pour les paramètres [3]. Les paradigmes méthodologiques sont différents ... et les conséquences pratiques aussi ! La co-variance peut introduire des " trous de typage ", c'est-à-dire des erreurs à l'exécution liées à l'absence de code exécutable associé aux types dynamiques des objets. On voit à travers cet exemple le lien entre les choix méthodologiques et la puissance des outils. Cet exemple met aussi en évidence la sensibilisation nécessaire des étudiants aux aspects sémantiques, et au point délicat de la séparation entre vérification statique et vérification dynamique (ou entre message du compilateur et

2. Quel est l'instanciation d'un profil de méthode sur une sous-classe?
3. Ce principe est connu sous le nom de co-variance.

erreur à l'exécution !). La sensibilisation à ces aspects devient importante, puisque l'évolution du développement logiciel nécessite de plus en plus de pouvoir apporter des garanties sur la qualité du produit[4]. On voit donc se développer des outils permettant certaines analyses sur le produit, ou sur le processus de développement, outils dont il faut pouvoir évaluer les résultats, afin de construire une approche globale du processus de validation et vérification.

Sensibiliser et initier les étudiants aux techniques et méthodes employées en génie logiciel n'est pas nécessairement facile dans le cadre d'un cours. Nous avons choisi de le faire à l'Ensimag à travers le projet de compilation, qui est devenu un projet compilation et génie logiciel, puis un projet génie logiciel. Les objectifs visés, du point de vue génie logiciel, ont été volontairement limités aux aspects conception générale et conception détaillée, au codage et à l'activité de vérification. L'objectif premier de ce projet était d'écrire un compilateur, afin de renforcer les aspects évoqués précédemment, c'est-à-dire la maîtrise fine, par les étudiants, d'un de leurs outils principaux de travail. La particularité d'un tel projet est de pouvoir disposer dès la phase amont, d'une spécification précise du cahier des charges, c'est-à-dire du langage à compiler. Cette spécification peut, de plus, être formelle (syntaxe, sémantique statique et sémantique à l exécution), ce qui est directement exploitable dans un processus de développement. Donnons un exemple. La sémantique statique du langage à compiler est décrite par une grammaire attribuée qui définit, sans ambiguïté, les conditions à vérifier. Cette définition peut être directement utilisée à différentes visées : produire le manuel utilisateur des erreurs statiques, construire l'implémentation par un processus systématique de codage qui assure une certaine correction et, finalement, produire un ensemble de tests jugé satisfaisant. L'aspect formel, et les particularités du problème, permettent aux étudiants de se poser des questions d'ordre génie logiciel et d'avoir un retour direct sur les stratégies adoptées. Par exemple il leur semble souvent incongru d'élaborer les messages d'erreurs avant la programmation, voir même impossible, bien que la spécification fournie contienne directement l'information nécessaire. Un autre aspect est lié à l'écriture des spécifications des modules. En effet, un compilateur met en jeu un grand nombre de structures de données, qui peuvent être complexes. Nous utilisons dans ce projet un style de spécification basé sur les types abstraits, qui permet de spécifier

4. ou du moins sur son processus de production.

et d'utiliser de manière homogène ces spécifications. Bien que ces spécifications ne sont pas formelles, elles pourraient l'être. Un effort est donc fait sur la manière d'écrire les spécifications (cas d'erreurs, préconditions, ...) et de les implémenter (programmation défensive, traitement des erreurs par exception, ...).

Bien que le cas d'étude ait en lui-même des particularités, et que les aspects amont du déroulement d'un projet ne soient pas traités, ce projet s'avère dans la pratique une bonne introduction aux aspects génie logiciel, l'aspect formel permettant de disposer de critères, et ainsi d'évaluer les résultats obtenus. Par exemple le lien entre les tests fonctionnels liés à l'étape de vérification statique et les taux de couverture obtenus par l'outil Logiscope, lors d'une expérimentation, ont permis aux étudiants d'évaluer leur approche du tests. L'enseignement par projet permet ainsi de se confronter aux aspects pratiques et est généralement bien perçu par les étudiants car ils produisent une application. Du côté des enseignants, la mise en place d'un projet nécessite de fixer les objectifs du projet, par exemple vis-à-vis des connaissances que doivent acquérir les étudiants à travers ce projet, et la manière de vérifier si ces objectifs sont atteints. L'objectif du projet évoqué ci-dessus, par exemple, est de passer du fait que les étudiants produisent des applications qui " marchent à peu près " au fait qu'ils puissent évaluer la qualité de ce qu'ils ont produit, et ceci en connaissance de cause.

## 1.3 Recherche

Les travaux que j'ai menés au LSR, dans l'équipe VaSCo[5], sont plus précisément relatifs à la spécification, au développement formel, jusqu'au niveau exécutable, et au test de conformité. Ces travaux portent à la fois sur les aspects fondamentaux des méthodes formelles, sur les aspects méthodologiques de leur mise en œuvre, et sur le développement de solutions effectives, en lien avec les difficultés rencontrées par les applications industrielles. Après mes travaux sur la synthèse déductive de programmes (thèse), j'ai étudié différentes approches : la théorie des types [PMP94], les spécifications algébriques [BEJ+95] et les approches orientées modèle. Ces travaux m'ont permis d'approfondir le lien entre preuves et programmes et, dans le cas des langages algébriques, de m'intéresser plus particulièrement à la notion

5. Validation, Spécification et Construction de logiciels

de construction modulaire. En parallèle j'ai participé à des travaux, développés dans l'équipe, sur la dérivation de programmes à partir de spécifications [BP90, EJPS91, Gue96] et je me suis aussi intéressée aux aspects méthodologiques à travers différentes études de cas [Pot93, LP96, LOP98].

En 1995 j'ai eu l'occasion d'étudier plus précisément la méthode B, initialement dans le cadre d'un enseignement. Cette méthode permet d'assister les différentes étapes du développement formel : l'écriture de spécifications de haut niveau, la preuve de propriétés invariantes et le raffinement des spécifications jusqu'à du code. Cette approche est encodée dans des outils, dont le principal est l'atelier B développé par la société ClearSy. La réalisation d'une telle approche, et de l'outillage associé, pose un certain nombre de difficultés non triviales. Nous en énumérons quelques unes en vrac : la définition précise du langage accepté, l'efficacité et la correction du processus de production des lemmes à vérifier, la réalisation d'un démonstrateur de théorèmes et la traduction vers un langage de programmation. Par exemple la gestion des noms doit porter à la fois sur les noms au niveau des spécifications, comme dans tout langage de programmation, mais aussi sur les noms intervenant dans les lemmes. De la même manière, la production de lemmes doit viser différents objectifs difficilement compatibles : ne pas produire des lemmes redondants, assurer la traçabilité avec les spécifications et être efficace (pour certains projets la phase de production des lemmes peut atteindre une vingtaine de minutes). De plus, bien que cette méthode soit basée théoriquement sur des résultats somme toute assez classiques, elle est la première à intégrer, dans la théorie du raffinement, une notion de composants qui peuvent être raffinés, jusqu'à du code, et composés. C'est cette particularité qui la rend opérationnelle sur des projets industriels (le développement de Météor contient par exemple environ un millier de composants B). Un des aspects innovants de cette approche est donc la mise ensemble de ces différents concepts, dans une approche qui s'avère adaptée dans la pratique, et qui est supportée par un outil d'envergure industrielle.

Le hasard, mes recherches antérieures, et mes goûts m'ont amené à développer des travaux sur cette approche. Mes premières études ont porté sur la composition des preuves d'invariant, associée à la construction incrémentale de spécifications [BPR96]. Suite à cela, nous avons été en contact avec les industriels concernés par la mise en œuvre de développements B et de l'outillage, afin de résoudre certains problèmes. Nous avons ainsi poursuivi

nos études, pour nous intéresser aussi à la composition des preuves de raffinement. Ces travaux ont permis de définir des conditions décidables de validité, qui ont été incorporées dans l'Atelier B. Plus généralement ces travaux nous ont permis d'aborder le problème de la mise en œuvre d'une approche, et d'un langage, basés sur la composition et le raffinement. Parallèlement à cela, nous avons aussi développé des travaux méthodologiques sur l'application de la méthode B, en particulier sur le B événementiel, qui permet de spécifier et de raisonner aussi sur les aspects comportementaux. Le processus de raffinement devient alors plus complexe car il inclut aussi le raffinement de comportement. Nous avons proposé une approche permettant d'assister ce processus, en utilisant une expression explicite du contrôle [Leb00, Nah01] et en synthétisant automatiquement certains lemmes intermédiaires nécessaires. Ces travaux sont en cours d'expérimentation et ne seront pas détaillés ici. Finalement, en lien avec la société ClearSy, nous menons des études sur le langage et la méthode B, pour améliorer la qualité et l'intérêt des outils. Ces études se déroulent principalement dans le cadre d'un projet RNTL (projet BOM, 2000-2003) et portent sur différents aspects. Elles s'instancient, au LSR, par le développement d'une boite à outils qui permet en particulier d'expérimenter les différents calculs formels liés à la théorie du raffinement.

Parallèlement à cela, et dans une moindre mesure, je participe à des collaborations industrielles relatives au développement d'outils pour la compilation de systèmes embarqués. En effet, les systèmes à fortes contraintes matérielles (cartes à puce, DSP ...) nécessitent des techniques de compilation sophistiquées, qui doivent être adaptées à chaque nouvelle architecture. L'objectif est donc de construire des outils reciblables, c'est-à-dire qui peuvent s'adapter aux différentes architectures et qui doivent aussi pouvoir être validés. Ces travaux sont menés dans le cadre de collaborations industrielles, d'abord avec STMicroelectronics, puis maintenant avec Motorola. C'est aussi l'objectif du projet RNTL BOM dont le but est de produire à partir de développements B du code C adapté aux cartes à puce et aux différentes plates-formes visées.

## 1.4 Contenu de ce manuscrit

Ce manuscrit sera centré sur la méthode B et en particulier sur tous les aspects qui ont trait à la compositionnalité. Les raisons de ce choix sont

multiples :

- Ces travaux ont maintenant une forme relativement aboutie, qui fait que le contenu de ce document sera en partie pérenne.

- Le B-Book, ouvrage de référence sur la méthode B, ne traite pas vraiment de la compositionnalité des preuves. L'aspect langage est en partie décrit, mais ni les principes ni les théorèmes sous-jacents ne sont détaillés. Le travail présenté ici est donc, d'une certaine manière, un complément au B-Book. Les études que nous avons menées sont nécessaires pour valider les principes adoptés et, comme le besoin semble se faire sentir, pour évaluer les impacts de certaines extensions.

- Enfin, même si la méthode B n'est pas amenée à perdurer sous sa forme actuelle, ces études restent valides. En effet comme ceci a déjà été dit, toute méthode est le résultat d'un certain nombre de choix dont il est important d'étudier les conséquences. Dans tout autre cadre, des questions similaires se posent. Quel autonomie est laissée à l'utilisateur ? En fonction de ce choix qu'est ce qui doit être pris en charge par les outils ? L'expérience de la méthode B est donc nécessairement profitable à la communauté.

Finalement, par les différents travaux que j'ai menés, j'ai eu la chance d'être en contact avec les développeurs de l'Atelier B, avec les industriels utilisant la méthode B pour le développement d'applications de grosse envergure et, aussi, avec Jean-Raymond Abrial. Ceci m'a permis de profiter de l'expérience de chacun, à la fois sur les problèmes de mise en œuvre de la méthode B et sur les solutions possibles, mais aussi sur les difficultés techniques et théoriques liées à la méthode. Il m'a donc semblé intéressant, pour la communauté, de faire partager l'expérience et les connaissances que j'ai pu en retirer. En effet, le contexte particulier des applications, de la méthode et des outils, principalement développés dans un cadre industriel, fait que la culture et le savoir-faire sont forts, mais en partie sous forme orale ... et donc volatile !

Mes autres activités de recherche ne seront donc pas directement développées dans ce manuscrit, bien qu'elles aient influencé les travaux présentés ici (compilation de code embarqué, principe de décomposition dans les approches basées comportements ...). Le plan de ce rapport est le suivant. Le

chapitre 1 décline les avancées qui ont rendu possible l'applicabilité industrielle des méthodes formelles, et les besoins. Un accent particulier est mis sur la vérification de code, puisque c'est ici qu'il y a souvent rupture de la chaîne formelle. La méthode B est ensuite introduite, avec son contexte industriel particulier. Le chapitre 2 reprend rapidement les fondements de la méthode B, c'est-à-dire les substitutions généralisées et le calcul de plus faible précondition. Il présente ensuite de manière approfondie les opérateurs qui seront à la base du développement modulaire en B : la substitution simultanée, qui permet de composer des traitements développés indépendamment, et l'appel d'opération, qui permet d'instancier des développements. Les propriétés de ces opérateurs, vis-à-vis de la préservation des preuves, sont étudiées en détail. Ces aspects ne sont pas vraiment traités dans le B-Book ou ne correspondent pas à ce qui est actuellement accepté par les outils. Dans la première partie du chapitre 3, les principes sous-jacents à la maîtrise de la complexité par composition (décomposition) et par abstraction (raffinement) sont étudiés dans le cadre général. Nous nous intéressons par exemple à l'influence sur le processus de validation par la preuve, à l'intégration de ces aspects dans une démarche de développement incrémental et à la mise en œuvre de ces principes dans des langages. Enfin, dans la seconde partie de ce chapitre, nous décrivons l'approche adoptée dans la méthode B et la mise en œuvre de la compositionnalité au niveau des spécifications, des développements et des preuves. Cette partie reprend les travaux que nous avons développés, en essayant, dans la mesure du possible, de ne pas rentrer trop dans les détails de la méthode B.

Ce manuscrit est le résultat des différents travaux que j'ai pu mener et des projets auxquels j'ai participés. Il est aussi le résultat de nombreux échanges que j'ai pu avoir avec les principaux intéressés par la méthode B, les utilisateurs et les développeurs d'outils, et avec d'autres personnes de la communauté des méthodes formelles. Je remercie en particulier nos partenaires du projet BOM, Gemplus, ClearSy et le LIFC, les différents interlocuteurs que j'ai pu avoir à Matra Transport International, Alstom et à la RATP, ainsi que Jean-Raymond Abrial.

# Chapitre 2

# Méthodes Formelles

Les méthodes formelles ont déjà été utilisées avec succès dans des applications industrielles. En plus des besoins motivant d'autres approches que les techniques classiques du développement logiciel, ceci a été rendu possible par les progrès réalisés, qui ont su se concrétiser en des outils. Ce chapitre a pour objectif de lister certaines lignes directrices sur lesquelles les efforts doivent encore porter, afin de faire des méthodes formelles des techniques utilisables à part entière.

Je présenterai ensuite rapidement la méthode B, et son contexte très particulier de développement, et j'instancierai les efforts à faire sur cette méthode, ceci constituant le cœur de mon activité de recherche.

## 2.1 Quelques faits

La présentation ci-dessous est issue de différentes analyses menées sur les méthodes formelles et plus particulièrement des synthèses suivantes : une étude de E.M. Clarke et J. M. Wing dans le cadre du *Formal Method Working Group* [CW96], le rapport de D. Craigen, S. Gerhart et T. Ralstom fait pour le compte du *National Institute of Standards and Technology* des États-Unis [CGR93], le rapport de l'*Observatoire Français des Techniques Avancées* sur l'application des techniques formelles au logiciel [OFT97], et le rapport plus récent de R.E. Bloomfield and D. Craigen *Formal Methods Diffusion : Past Lessons anf Future Prospects*, produit pour l'agence fédérale allemande pour la sécurité de l'information [BC99]. Une bibliographie peut aussi être trou-

vée dans [JLNP97]. Dans une moindre mesure, cette présentation est aussi le résultat de mon expérience personnelle. En effet, dans le cadre de mes travaux j'ai eu l'occasion de collaborer, informellement ou non, avec les industriels utilisant et outillant la méthode B (en particulier Matra Transport International maintenant Siemens Transportation Systems, Alstom Transport, Gemplus et ClearSy). C'est pour cela que j'emprunterai mes exemples essentiellement aux cas des applications B.

Le but de cette section est de mettre en évidence certains verrous sur l'application des méthodes formelles, en présentant à la fois des avancées technologiques attendues et des besoins liés à la pratique industrielle. Le résultat des efforts à mener pourra être vu, en partie, comme la rencontre de ces différentes exigences.

### 2.1.1 Méthodes formelles

On peut caractériser une méthode formelle selon trois axes : l'aspect formel, l'aspect outillage et l'aspect méthodologique.

Une méthode formelle doit offrir à l'utilisateur un langage lui proposant des notations et des concepts permettant d'exprimer ses attentes. Ces langages peuvent permettre d'exprimer des spécifications, des propriétés ou des programmes. Une méthode est dite formelle si son langage possède à la fois une syntaxe et une sémantique mathématiquement bien définies, ainsi qu'une logique sous-jacente permettant de raisonner sur les spécifications, les programmes et les propriétés.

L'outillage proposé est un aspect crucial d'une méthode. Il doit permettre d'assister, totalement ou en partie, les différentes étapes du développement de logiciels. Ces outils peuvent être automatiques, ou demander une intervention humaine, suivant la nature du problème. Dans ce dernier cas il est important de trouver la bonne adéquation entre l'utilisateur et l'outil. En plus d'offrir un certain niveau d'automatisation, l'intérêt des outils est, en obligeant l'utilisateur à se plier à un cadre imposé, de détecter certaines erreurs. En effet on peut remarquer que les phases de vérification statique d'un langage permettent souvent de révéler des erreurs conceptuelles.

En reprenant les critères proposés dans [CW96] et [OFT97], les deux

premiers aspects d'une méthode formelle (fondements mathématiques et outillage) peuvent se caractériser par :

- la capacité d'expression et de modélisation ;
- la capacité d'abstraction ;
- les possibilités de compositionnalité ;
- le compromis entre expressivité et décidabilité.

L'abstraction et la compositionnalité, sur lesquelles nous reviendrons par la suite, sont deux caractéristiques permettant le passage à l'échelle. Comme dans toute technique, ces deux notions sont des outils indispensables pour maîtriser, dans une certaine mesure, la complexité. La notion de compromis est relative à la puissance des outils et à leur capacité d'automatisation.

L'aspect méthode doit offrir des guides permettant d'appliquer les démarches proposées de manière la plus systématique possible. C'est cette dimension qui autorise le passage à l'échelle. Les guides doivent porter sur différents axes : la mise en œuvre de la notation et de la méthode ; les familles d'application visées et l'évaluation des résultats obtenus ; l'inclusion dans le cycle de développement, en particulier vis-à-vis de l'assurance qualité ou sécurité. L'aspect méthode est bien le moins maîtrisé actuellement. Il nécessite des retours d'expérience précis et détaillés.

### 2.1.2 Méthodes formelles et cycle de vie

Les méthodes formelles peuvent être utilisées à différentes étapes du cycle de vie, et pour des objectifs différents. Nous listons ces utilisations possibles en cherchant, dans la mesure du possible, à donner des critères sur l'impact sur le cycle de vie.

#### Phase de spécification

Au niveau de la spécification, c'est-à-dire du passage des besoins au cahier des charges, l'intérêt des méthodes formelles est clair. Elles obligent à un effort de *formalisation* qui, *a priori*, fournit une aide à la modélisation. En effet la formalisation peut obliger à un certain degré de questionnement,

d'autant plus si cette formalisation est assistée par des outils.

Il semble difficile de quantifier l'impact d'une spécification formelle sur le cycle de vie d'un logiciel. Néanmoins les retours d'expérience de grands projets ont montré que la phase de spécification est souvent la plus critique. A condition de ne pas introduire de bruit auprès des clients, l'utilisation des méthodes formelles au niveau de la spécification semble donc d'un rapport coût/bénéfice avantageux.

### Conception et codage

Partant d'une spécification, les méthodes formelles peuvent être utilisées pour passer progressivement de la spécification au code. Suivant la distance entre les deux, cette étape peut essentiellement prendre deux formes :

1. la production automatique de code ;
2. l'assistance à la transformation de la spécification au code.

**Processus automatique.** Le premier type d'outils est adapté lorsqu'il existe un processus systématique de traduction qui, de plus, fournit un code ayant les qualités requises. Ceci signifie d'une part que la spécification est, d'une certaine manière, déjà exécutable et d'autre part qu'elle décrit effectivement le comportement désiré au niveau de l'exécution. Suivant les applications visées, ce dernier point est important. En effet, dans le cas de systèmes embarqués sur des ressources particulières (DSP, cartes à puce ...) le développeur peut vouloir maîtriser le code obtenu, celui-ci devant être parfaitement adapté à la cible. Dans ce cas il n'est pas toujours possible de proposer des outils entièrement automatiques, au mieux ils doivent être configurables.

Lorsque l'automatisation est possible, le gain est indéniable puisqu'il supprime un niveau de développement. Ceci revient à offrir au développeur un langage de plus haut niveau.

**Processus assisté.** Lorsque la spécification est plus abstraite, ou bien n'a pas été conçue comme décrivant le comportement attendu à l'exécution, le processus de transformation peut être assisté formellement. C'est la technique du raffinement. Le développeur doit alors écrire lui-même les spécifications

plus précises et la correction sera vérifiée formellement. Le gain est ici moins tangible car l'activité de développement devient plus coûteuse. Néanmoins, si cette activité est mise en place dans une approche globale de la qualité, ce gain peut être évaluable. Ceci a été le cas par exemple dans le projet Météor [Ba99] où les tests unitaires ont été supprimés, pour les composants développés formellement. Cette décision a été rendue possible par le fait que la détection des erreurs potentielles du compilateur, et des pannes matérielles, était garantie par la technique du Processeur Sécuritaire Codé (voir section 2.3.1).

### 2.1.3 Outils

En dehors des différentes étapes du cycle de vie, l'introduction de méthodes formelles, tôt dans le cycle de vie, permet d'utiliser des outils liés au fait de disposer de langages sémantiquement définis. Comme pour les langages de programmation il est alors possible d'analyser, d'évaluer et d'exécuter les modèles. Certains de ces outils sont eux-mêmes basés sur des fondements mathématiques.

#### Analyse statique

Les outils d'analyse statique peuvent prendre différentes formes, suivant les propriétés à valider et la complexité des algorithmes d'analyse. Ces outils peuvent aller de la vérification de propriétés particulières à chaque problème, à l'analyse de propriétés générales (absence d'erreurs à l'exécution, absence de blocage ...). En dehors de l'assurance supplémentaire induite par la vérification d'une propriété particulière, il est en général difficile d'évaluer l'impact d'une telle vérification sur le processus global de validation. Si les propriétés ne constituent pas en elles-mêmes un critère de qualité ou de validation (contrairement aux propriétés que nous avons qualifiées de générales par exemple), il est souvent impossible de déduire l'impact de ces vérifications sur l'activité de test, par exemple.

#### Animation, prototypage et simulation

En fonction des particularités du langage, il peut être possible d'exécuter ou de jouer des scénarios sur la spécification. Ceci permet de valider la

spécification, en vérifiant l'adéquation de son comportement avec les besoins attendus.

#### Production de jeux de tests

L'utilisation de méthodes formelles au niveau des spécifications peut avoir des retombées directes au niveau du test. Dans le cas du test de conformité par exemple, qui vise à établir la correction du code vis-à-vis de la spécification, elles permettent de disposer d'un oracle effectif et sont utilisées pour produire, automatiquement ou non, des suites de tests. La phase de test obérant généralement le coût global de développement de manière significative, il est évident que toute technique qui permet d'alléger cette phase est globalement bénéfique.

### 2.1.4 Besoins

Les méthodes formelles peuvent être adoptées pour des objectifs très différents.

#### Caractère imposé

Certains donneurs d'ordre, notamment dans les applications à caractère critique, imposent leur propre démarche de production de logiciel. C'est par exemple le cas du ministère britannique de la défense, dont le standard [MoD91] impose à ses contractants une approche basée sur les méthodes formelles.

#### Certification

Elle permet de donner un label qualité à un produit. Par exemple les normes ITSEC [ITS93] et les Critères Communs [ST99] sont des références pour l'évaluation des systèmes et produits informatiques. Pour les évaluations les plus élevées (E6 pour ITSEC et EAL7 pour les Critères Communs) les méthodes formelles sont requises pour valider la conception. Dès le niveau EAL5 des Critères Communs, les méthodes formelles interviennent pour la définition des politiques de sécurité.

#### Amélioration de la qualité ou de la productivité

Dans certains domaines d'application, comme par exemple les cartes à puce [Lan01], les applications ne sont pas très complexes mais nécessitent d'être développées rapidement, et avec une certaine exigence de qualité, ceci afin d'être concurrentiel. Les méthodes formelles semblent trouver ici leur raison d'être puisqu'il n'y a pas de problème de passage à l'échelle. De plus, les applications cartes à puce peuvent nécessiter différents développements en fonction des cibles visées. *A priori* les méthodes formelles devraient permettre, à terme, de réutiliser et d'adapter des développements, à condition d'offrir un certain niveau de généricité.

#### Maîtrise de la complexité

Dans certains cas, la complexité des applications peut amener à chercher un autre processus de développement, et la rigueur imposée par un cadre formel. C'est par exemple le cas du ferroviaire où le changement de technologie (utilisation du logiciel à la place du matériel sur les fonctions sécuritaires) a consisté à définir une approche globale de la sécurité, intégrant un développement formel pour le logiciel. L'augmentation croissante des systèmes réalisés, et la place de plus en plus prépondérante prise par le logiciel, peuvent amener à adopter l'utilisation de méthodes formelles.

## 2.2 Quelques pistes

Nous listons ci-dessous quelques pistes à développer, qui permettraient aux méthodes formelles soit de répondre à de nouveaux besoins, soit de les rendre plus efficaces, à la fois en les rendant plus faciles d'utilisation et en les intégrant mieux dans un cycle de développement.

### 2.2.1 Défis technologiques

Il reste des défis technologiques relatifs à des ordres de grandeur à dépasser. Donnons quelques exemples tirés de [BC99]. Ces chiffres correspondent à l'application brute des technologies citées, sans les combiner avec d'autres approches.

- vérification de l'équivalence entre circuits d'une taille d'environ un million de portes ;

- vérification par modèle de circuits de systèmes de plus de $10^{20}$ états ;
- analyse statique de programmes de taille supérieure vers 150 Kloc ;
- développement formel de la spécification à du code de l'ordre de 80 Kloc ;
- vérification formelle de compilateurs pour des langages dédiés ;
- spécification et modélisation d'un ordre supérieur à 30 000 lignes de spécification.

Tous ces défis ont pour objectif d'améliorer l'outillage, afin de permettre de traiter des applications de plus en plus conséquentes. L'avant dernier challenge est un peu particulier. Il ne consiste pas seulement à améliorer les performances d'une technique particulière. Si le langage est complexe il s'agit d'un véritable passage à l'échelle, du point de vue de la complexité. Ceci est le cas par exemple pour le langage Java où beaucoup de travaux de formalisation ont été menés [HM01]. Néanmoins, à l'heure actuelle, ces travaux ne couvrent ni tous les aspects offerts par le langage, ni le processus global de validation des propriétés du langage à son environnement d'exécution.

Un défi technologique non cité dans les travaux précédents, mais qui apparait de plus en plus d'actualité, est le besoin de pouvoir produire des justificatifs associés aux développements formels. En effet le but est de pouvoir convaincre des personnes (commanditaire, évaluateur, certificateur ...) de la rigueur du raisonnement et de la pertinence du formalisme. Pour cela il faut pouvoir construire des preuves, les présenter de manière compréhensible et, dans certains cas, les embarquer dans du code, comme dans l'approche PCC (Proof Carrying Code) [Nec97]. L'intérêt ici est de pouvoir fournir du code contenant l'argumentation des propriétés formelles attendues. Il suffit alors, lors du chargement d'un tel code, de vérifier la preuve, ce qui est nettement plus simple que de la reconstruire.

### 2.2.2 Concepts fondamentaux

Les différentes études précitées [CGR93], [CW96], [OFT97] et [BC99] soulignent toutes les mêmes concepts fondamentaux à développer. Les deux concepts clés, comme dans toute approche scientifique, sont la décomposition

et l'abstraction [dRLP98], [dR98]. Les développements attendus sont à la fois d'ordre théorique et méthodologique.

### Composition et décomposition

Les méthodes formelles doivent offrir des moyens pour construire et manipuler des modèles de taille importante. Elles doivent donc offrir des mécanismes permettant de composer les spécifications, les modèles et les théories.

La décomposition est le processus qui consiste à exhiber les différentes parties d'un système, et leur assemblage. C'est le choix pertinent de cette décomposition qui permet de couper la complexité inhérente à un problème. L'aspect méthodologique est de savoir comment décomposer, au vu des mécanismes offerts par l'approche, et des objectifs visés. Il faut par exemple étudier les classes de propriétés, et les classes de systèmes, pour lesquels il est possible de proposer des décompositions efficaces qui permettent de maîtriser la complexité, en particulier vis-à-vis des outils.

### Abstraction et raffinement

La maîtrise de la complexité passe aussi par la notion d'abstraction, qui permet de manipuler une description simplifiée d'un système. Une bonne abstraction est celle qui met l'accent sur les détails qui sont significatifs, et qui supprime les autres, pour un point de vue donné. Remarquons que le processus d'abstraction est souvent basé sur des hypothèses simplificatrices, qui doivent être validées. On peut inverser le processus et obtenir la notion de raffinement. Partant d'une spécification abstraite, sur laquelle certaines propriétés sont établies, le raffinement permet de construire des spécifications plus concrètes, tout en préservant les propriétés établies. De manière générale le processus d'abstraction/raffinement impose de pouvoir assurer une cohérence formelle entre deux niveaux de spécifications.

Les méthodes formelles doivent donc offrir des notions permettant de décrire différents niveaux de spécification, et des outils, au moins conceptuels, permettant d'assurer la cohérence entre ces niveaux. Du point de vue méthodologique, il est intéressant d'étudier différentes formes d'abstraction, en fonction des objectifs visés (par exemple des propriétés à préserver). Dans certains cas, ces études peuvent déboucher sur des outils permettant d'assis-

ter, ou de mécaniser, certaines étapes du processus d'abstraction.

### Structuration

La décomposition et l'abstraction sont deux outils qui permettent de structurer les spécifications et les développements. La décomposition introduit une structuration horizontale et l'abstraction une structuration verticale. La maîtrise de la complexité passe par ces deux notions, intrinsèquement dépendantes. En effet la puissance d'un principe de décomposition est relatif à la capacité d'abstraction offerte par le langage, qui permet de ne considérer que les aspects significatifs des différentes parties [Boe99]. L'utilisation conjointe de ces outils permet d'aborder un système complexe par parties, et ceci à tous les niveaux d'abstraction. On parle alors de structuration en couche.

La mise en œuvre de ces deux principes nécessite de bien définir leur interaction et de pouvoir raisonner sur des structurations en couche. Le principe sous-jacent à cette approche est la compositionnalité des raisonnements. Le résultat d'un développement formel, et sa validation, s'obtient donc en considérant la démarche globale mise en place pour l'obtenir. En effet, hormis pour des cas très particuliers, retrouver les bonnes abstractions nécessite souvent un effort intellectuel important. Ceci a, par exemple, été le cas pour la première application de la méthode B. Dans le cas du SACEM (Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance) une tentative de validation basée sur la preuve de programme a donné des résultats incomplets. Elle a permis de valider les programmes vis-à-vis des spécifications détaillées mais n'a pas permis de valider la conception détaillée par rapport à la spécification [BDM97].

La notion de structuration en couche permet de proposer et d'évaluer des approches globales pour le développement ou la vérification de systèmes complexes. On peut par exemple aborder une structuration en couche par une approche *top-down*, qui correspond à un principe de décomposition, ou par une approche *bottom-up* qui correspond à la composition. Dans son article intitulé *Composition: a Way to Make Proofs Harder* [Lam98], Leslie Lamport cherche à évaluer l'intérêt de la décomposition et de l'abstraction, dans le cas d'un processus de preuve. Il prend ici comme critère la longueur des

preuves. Il remarque que, dans l'absolu, la décomposition est un processus inutile. Il demande beaucoup de travail (trouver la décomposition, élaborer les spécifications des sous-systèmes ...) et revient, in fine, à faire, au mieux, les mêmes preuves que celles faisables sur le système non décomposé. Dans le cas de l'abstraction, il souligne que des gains tangibles peuvent être obtenus, en faisant " abstraction " du coût de l'invention de l'abstraction, et de sa construction. Néanmoins ces outils sont, dans la pratique, indispensables du point de vue conceptuel et peuvent devenir réellement efficaces s'ils permettent de mécaniser certaines étapes. C'est par exemple le cas lorsqu'il est possible de combiner la vérification par modèle avec la décomposition ou l'abstraction.

### Guides méthodologiques

La dernière notion qui apparait est donc le besoin établi de méthodologie, qui permet de fixer des objectifs et de mettre en place une approche globale du développement ou de la vérification de systèmes. Ceci revient à acquérir une certaine maturité, permettant d'utiliser et de combiner à bon escient les outils conceptuels ci-dessus. De tels guides ne peuvent être établis de manière pertinente que pour des familles particulières d'applications, ou des objectifs de vérification spécifiques. Par exemple la sécurité est *a priori* un domaine idéal d'application des méthodes formelles. En effet, en plus de son caractère obligatoire ou recommandé, les bonnes pratiques en matière de sécurité consistent à regrouper dans quelques composants clefs les fonctions de sécurité. La taille des composants concernés est donc généralement petite et les interactions de ces composants avec leur environnement sont aussi parfaitement définies [BlML01]. De plus les critères de certification, comme les Critères Communs, imposent une certaine démarche basée sur le raffinement et des vérifications de cohérence.

Des guides méthodologiques doivent viser les objectifs suivants :

- définir une approche globale, qui identifie les objectifs visés par l'approche formelle, les différentes étapes et l'évaluation des résultats obtenus ;

- offrir un processus de documentation et d'explication, et plus généralement s'inscrire dans un cycle de vie global, incluant l'activité de test.

- gérer le développement jusqu'au code, si besoin est.

La mise en place de tels guides permet de codifier un processus de développement ou de vérification et ainsi de franchir un degré supplémentaire dans la maîtrise de ce processus. Il devient alors possible de développer des outils formels qui prennent en charge certaines étapes et qui offrent une certaine forme de réutilisation du processus de développement lui-même. Par exemple le développement B effectué dans le cadre du projet Météor utilise un raffineur automatique, qui assiste une partie du processus d'implantation [BM99]. Ce raffineur propose des représentations informatiques des formes de données classiquement utilisées par le ferroviaire et certaines (méta-)preuves ont été réalisées une fois pour toutes. Un tel outil permet donc de réutiliser une partie d'un développement. De la même manière on peut citer le projet ABS (Atelier B-SCADE), développé par la RATP, qui consiste à faire collaborer les approches B et SCADE, pour développer formellement des familles de systèmes [Cha02]. La méthode B permet de construire un modèle abstrait, sur lequel des propriétés générales sont validées. Chaque système particulier est alors décrit à l'aide du langage SCADE, en se basant sur une librairie de composants. L'outil ABS a pour but d'établir le lien entre ces deux niveaux de spécification et d'obtenir une double chaîne de génération de code : le code certifié, obtenu à partir de la chaîne SCADE, et le code prouvé, obtenu par la chaîne B.

### 2.2.3 Vérification formelle et code exécutable

Les applications des méthodes formelles, que ce soit en vérification ou en développement, ont rarement été menées au niveau du code. En effet les langages de programmation sont, dans toute leur généralité, souvent trop complexes pour pouvoir appliquer efficacement des raisonnements formels. On peut citer par exemple des constructions comme les exceptions, les threads ou les pointeurs qui compliquent très largement la sémantique. Le modèle considéré est donc généralement une abstraction qui, même si elle semble proche de l'implémentation, ne correspond pas à l'exécution. C'est d'autant plus vrai dans le cadre des systèmes embarqués où le code exécuté peut être complexe, car devant répondre à des exigences particulières. Soit le code est écrit à la main à partir de la spécification soit on laisse le soin à un outil de produire un code ayant les bonnes propriétés. Dans un processus formel les

points à prendre en compte sont :

1. la validation des propriétés intrinsèques au code, c'est-à-dire dépendantes du modèle d'exécution (aspects non fonctionnels, préservation de la mémoire, absence d'erreurs à l'exécution, restrictions liées au compilateur ...) ;

2. la préservation de la chaîne formelle en assurant la correction du code obtenu, ce qui se ramène généralement à la correction du processus de traduction ;

3. l'adaptabilité du processus de traduction.

### Propriétés sur le code

Comme nous l'avons remarqué précédemment les développements formels contiennent, peu ou prou, un changement de langage. On passe nécessairement d'une vision déclarative à une vision opérationnelle. Il faut donc étudier finement ce qui est pris en charge par le processus formel et ce qui doit, de plus, être assuré par la traduction. En particulier les outils d'analyse statique peuvent compléter le processus de développement formel. Les aspects non fonctionnels échappent généralement à la portée des méthodes formelles.

### Correction du code

Un point délicat est d'assurer d'une manière ou d'une autre la correction du processus de traduction, c'est-à-dire que les propriétés établies en amont restent effectivement valides sur le code exécuté. Dans l'absolu, les langages se prêtent en général bien à des raisonnements formels car ils disposent naturellement d'une spécification, qui est leur sémantique. La correction d'un compilateur se découpe traditionnellement en deux étapes [BBa92] : la validation de la spécification du compilateur et la validation de l'implémentation. La première étape met en jeu à la fois la sémantique du langages source et celles du langage cible. La seconde étape se réduit à une preuve d'implémentation, vis-à-vis de la spécification déjà validée. Néanmoins cette approche pose beaucoup de difficultés, la première étant de disposer d'une sémantique formelle, mais correcte, d'un langage. Remarquons que la définition formelle de la sémantique est un domaine de recherche en soi, comme le montrent les

travaux actuels autour du langage Java et de son environnement d'exécution [HM01].

Une autre approche consiste à utiliser des outils formels non pas pour vérifier complètement un traducteur mais pour s'assurer qu'il préserve certaines propriétés générales, comme des domaines sémantiques par exemple. Ceci peut venir en complément d'autres techniques plus classiques, comme les guides méthodologiques préconisés par les processus de certification, ou la redondance de code. Par exemple le générateur de code SCADE est certifié au niveau A de la norme DO-178B de l'avionique. Ceci signifie que le traducteur a été développé en respectant les étapes et les critères de cette norme. Dans le cas de la redondance, une solution consiste à utiliser une double chaîne de traduction. On peut ainsi comparer les exécutions ou, si c'est le cas, simplement vérifier l'identité du code produit.

Même si elle n'est pas suffisante pour valider le code exécutable (problème des pannes matérielles par exemple), la validation formelle d'un traducteur constitue un verrou important, car tout processus de développement logiciel passe, peu ou prou, par un tel outil.

### Adaptabilité

Comme ceci a déjà été dit, dans le cas des systèmes embarqués le processus de traduction doit être adaptable. En effet chaque domaine d'application ou chaque constructeur peut avoir ses propres contraintes en terme de code produit. Dans le cas du ferroviaire par exemple la technologie utilisée pour assurer la correction à l'exécution est la technique du processeur codé, qui nécessite en entrée une forme particulière du langage Ada. De la même manière, dans le domaine des applications s'exécutant sur des matériels de petite surface (carte à puces, Digital Signal Processeur) le code développé doit être adapté au matériel visé. L'adaptabilité du processus de traduction doit permettre soit de restreindre le langage cible, en raison par exemple des conditions imposées par les normes, soit d'adapter les schémas de traduction pour obtenir du code ayant les qualités requises. Il s'ensuit que le processus de validation doit être adaptable, en même temps que le processus de traduction. De plus les contraintes matérielles peuvent nécessiter des optimisations sophistiquées, qui compliquent encore le problème de la validation des traducteurs.

## 2.3 Méthode B : ses particularités

La méthode B a sa propre histoire. Elle est née d'un besoin industriel et n'est pas issue du milieu académique. Elle fait partie des réussites les plus importantes de l'application des méthodes formelles dans un cadre industriel. En effet il y a eu très peu d'expériences de développements formels basés sur la preuve, et aucun de cette envergure. De plus, les applications réalisées à l'aide de la méthode B, particulièrement le projet Météor [Ba99], reposent sur une véritable intégration de l'utilisation d'une méthode formelle dans une chaîne de développement et de validation, ceci pour un système dans son entier.

Ce contexte fait que les travaux scientifiques autour de cette méthode sont de nature un peu particulière. Je vais préciser ce contexte, ces implications et comment se situent mes travaux et le contenu de ce manuscrit.

### 2.3.1 Applications industrielles

Dans cette section nous présentons rapidement les différents cas d'applications industrielles, afin de mettre en évidence les besoins.

#### Applications ferroviaires

L'historique de la méthode B est très liée au domaine du ferroviaire [BDM97], [Hab01][1]. Très schématiquement, elle est née avec le système SACEM, *Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance*, réalisé par GEC Alsthom Transport. SACEM est l'un des premiers systèmes ferroviaires dont les fonctions de sécurité ont été réalisées par des traitements programmés sur microprocesseurs. La méthode B a trouvé sa puissance avec le projet Météor, développé par Matra Transport International, maintenant Siemens Transportation Systems. Un projet de même envergure est en cours de développement, dans la suite de Météor, pour la ligne Canarsie du métro de New-York.

Pour fixer l'ordre de grandeur de ces applications, nous donnons ci-après un tableau avec les mesures correspondant au projet SACEM, mis en service

1. Voir la préface de Fernando Méjia.

en septembre 96, et au projet Météor, mis en service en août 98. Ce tableau est extrait de [BDM97][2].

| | SACEM | Météor |
|---|---|---|
| lignes B | 3500 | 107000 |
| composants B | 28 | 1075 |
| lignes de code | (Modula) 2500 | (Ada) 87000 |
| lemmes | 550 | 29000 |

Les industriels concernés ont produit différentes publications sur leurs expériences [MD94, Beh96, Ba99]. Dans [BDM97] P. Behm (Matra Transport International), P. Desforges (RATP) et F. Méjia (GeC Alsthom) présentent l'utilisation des méthodes formelles dans l'industrie du ferroviaire et les applications développées à l'aide de la méthode B. Cet article décrit les différents facteurs ayant permis d'augmenter la taille des applications (amélioration des outils, formation des ingénieurs impliqués ...). Dans [Ba99], l'application Météor est décrite. En dehors des chiffres correspondant à ce développement, l'article met en évidence comment cette méthode s'est insérée dans une approche globale de la sécurité, quel processus de validation a été mis en place et suivi, ainsi que la complémentarité entre la validation effectuée chez le constructeur et celle effectuée chez le donneur d'ordre. La politique générale de sécurité repose aussi sur la complémentarité entre le développement formel et la technique du processeur codé [For89]. Cette technique est basée sur un codage de l'information qui sera calculé et vérifié à l'exécution. En cas de divergence entre l'information codée et l'exécution du logiciel le processeur codé met l'équipement dans un état sûr (coupure de courant, freinage d'arrêt d'urgence ...). Cette technologie repose sur un principe de sécurité intrinsèque, bien adapté au cas du ferroviaire [DEF03] : en cas de panne, le système se stabilise dans un état sécuritaire.

### Cartes à puce

Les cartes à puce constituent un autre domaine d'application des méthodes formelles. En effet, les besoins de sécurité sont importants, puisque ces cartes constituent généralement un élément clé d'une politique de sécurité. Gemplus, parmi d'autres approches formelles, a développé différentes études

2. La notion de composant correspond à une entité de spécification, de raffinement ou d'implémentation. Grosso modo, un composant est une machine à état.

de cas, à l'aide de la méthode B. Une partie de ces travaux sont relatifs à la certification [Mot00, MT00]. D'autres études portent sur les mécanismes de transaction [SL99] et les protocoles [LL98]. Enfin, un autre enjeu important est le domaine des Java cartes, qui préparent la génération des cartes ouvertes, sur lesquelles des applications seront embarquées au cours de la vie de la carte.

Un grand nombre de travaux s'intéresse donc à la validation formelle, et à la sécurité, des environnements Java et JavaCard [HM01]. Le but est d'assurer la sécurité du byte code Java exécuté. Le processus défensif d'exécution se décompose entre le vérificateur de byte code, qui assure une certaine cohérence du code, et l'interpréteur. Dans le cadre des applications cartes, la vérification de byte code est souvent décorrélée de l'exécution, car le chargement dynamique de code n'est actuellement pas envisagé. Ceci permet en particulier de sérialiser le processus : le byte code est d'abord vérifié puis exécuté par un interpréteur offensif (sans vérification particulière). Le vérificateur de byte code peut alors être externalisé hors de la carte, ce qui rend possible le fait d'embarquer une machine Java sur une carte. La communication est assurée par l'intermédiaire des fichiers CAP (Converted APplets). Une partie des travaux de Gemplus a porté sur la preuve de certaines propriétés intrinsèques du vérificateur de byte code [LR98, Req00]. L'objectif est maintenant de construire des implantations du vérificateur et de l'interpréteur, qui répondent aux contraintes carte à puce. La technique du raffinement est utilisée pour assurer la correction de l'implémentation. Une première expérience a été réalisée dans le cadre du projet MATISSE et est présentée dans [BCR03]. Pour satisfaire aux contraintes des cartes, un traducteur ad-hoc a été développé chez Gemplus. Cette expérience se prolonge dans le cadre du projet RNTL BOM [BDB+03] dans lequel un des objectifs est de développer un traducteur complet, pour les applications carte à puce. L'objectif est d'embarquer un interpréteur de byte code, pour un sous-ensemble du langage.

D'autres expérimentations industrielles sont en cours, qui utilisent en particulier l'approche B événementielle. Ces études se placent généralement plus en amont (analyse de cahier des charges et analyse système) mais n'ont pas encore fait l'objet de publication.

### 2.3.2 Un produit pour l'industrie

La méthode B a donc directement été conçue pour être utilisée industriellement. Ceci a nécessité de prendre en compte, dès la conception de cette approche, un certain nombre de contraintes relatives à la fois aux concepts et à leur mise en œuvre, et à la nécessité de disposer d'un outillage support, répondant aux contraintes des industriels.

#### Enjeux

Un des premiers enjeux est la simplicité des concepts afin de permettre à des ingénieurs d'utiliser avec succès l'approche proposée, et ceci avec un coût raisonnable de formation. Le langage B a donc été conçu avec des constructions simples, mais suffisantes pour les applications visées. Par exemple le langage offre peu de mécanismes de généricité ou de typage sophistiqués. De plus la méthodologie proposée, assistance du développement formel des spécifications abstraites au code, doit elle aussi pouvoir être mise en œuvre par les ingénieurs, sans nécessiter une maîtrise complète de la théorie sous-jacente. Il s'ensuit que la méthode B impose un certain nombre de restrictions, en particulier dans le cas des développements modulaires, pour que les mécanismes de preuve sous-jacents soient transparents.

Un autre aspect important est la quantité de travail demandée aux développeurs. La différence n'est pas toujours significative pour des projets de faible taille, mais prend une grande importance dans le cas des développements industriels. Cette contrainte a eu des retombées importantes au niveau de la conception du langage (raffinement par ajout d'information, limitation du nombre d'obligations de preuve ...).

#### Un langage support et outils

L'utilisation d'une méthode, dans un cadre de production, nécessite d'offrir rapidement des outils de qualité, c'est-à-dire un langage mais aussi un environnement supportant la démarche. La conception d'un tel environnement est en soi un problème compliqué. En particulier, définir un langage ayant les qualités requises nécessite de concilier les facilités d'emploi avec les problèmes d'analyse et d'implantation. Par exemple la gestion des noms est, dans la méthode B, un problème plus complexe que dans les langages de programmation car il faut gérer à la fois l'espace des noms qui apparaissent

dans les spécifications, mais aussi l'espace des noms associés aux preuves. De plus certains outils sont en eux-mêmes intrinsèquement difficiles à élaborer, comme le démonstrateur automatique dont les performances conditionnent en grande partie la réussite des projets. L'environnement doit aussi prendre en charge l'approche proposée, tout en offrant une certaine souplesse d'utilisation. Enfin il est à noter que l'ouvrage de référence, le B-Book, présente la méthode B et ses fondements, mais n'est pas un manuel de référence du point de vue des outils.

La dernière contrainte associée à l'outillage, et non la moindre, est d'assurer une pérénité des outils, de les faire évoluer tout en assurant une compatibilité vis-à-vis des applications déjà développées. L'Atelier B par exemple a été initialement développé chez GeC Alsthom, puis repris par la société Digilog, afin de le rendre indépendant d'un constructeur. Digilog a été repris en partie par Stéria. L'Atelier B est actuellement maintenu, développé et promu par ClearSy, une filiale de Stéria.

### Un monde clos

L'outil principal autour de la méthode B, l'Atelier B, est un outil propriétaire et qui est fermé, pour la simple raison qu'il n'a pas été conçu pour ne pas l'être, de la même manière qu'on ne peut généralement réutiliser un composant que s'il a été prévu pour être réutilisable. Il est donc construit de manière monolithique, pour l'extérieur, et ne permet pas de développer facilement des outils annexes. De plus son histoire fait qu'il a été développé avec des technologies différentes et que certaines parties sont difficilement adaptables. Néanmoins certains outils ont été développés par des universitaires et offrent des représentations intermédiaires (voir par exemple le Site B Grenoble[3]).

Il y a relativement peu de documentation autour de cette approche et pas de documentation sur l'aspect langage et outil, à part le manuel de référence de l'Atelier B. Il s'ensuit qu'il est difficile, et parfois rebutant, de bien maîtriser cette approche. Paradoxalement la méthode B est fortement enseignée, particulièrement en France (voir le site B Grenoble par exemple). En effet elle illustre assez simplement les concepts de vérification et de dé-

3. http://www-lsr.imag.fr/Les.Groupes/scop/B/B.html

veloppement formel (jusqu'à du code qui " tourne ") et l'outil permet assez facilement aux étudiants de réaliser des études de cas convaincantes. La communauté des enseignants a aussi développé de nombreuses documentations, travaux pratiques et exemples qui devraient être, à terme, promus.

### 2.3.3 Axes à développer

L'histoire particulière de B a un certain nombre de conséquences, qui contribuent au fait qu'il y a relativement peu de travaux universitaires directement autour de cette approche. Par exemple la proposition d'extensions doit, dans l'idéal, préserver l'équilibre entre la puissance de l'approche et l'outillage et, de plus, être pertinente du point de vue applications. L'outil principal n'étant pas un produit universitaire, il est difficile de s'y raccrocher.

#### Les particularités

La méthode B est une des rares approches qui intègre la prise en charge complète du développement, de la spécification au code. Basée sur la preuve, elle intègre :

- l'écriture de spécifications ;
- la preuve de propriétés invariantes ;
- un processus de raffinement ;
- un générateur de code ;
- un générateur d'obligation de preuves ;
- un démonstrateur automatique et un démonstrateur interactif ;
- un mécanisme de composition des spécifications et des développements.

Elle offre donc des particularités intéressantes à exploiter, relatives à l'intégration réussie entre un langage et une méthodologie, et ceci dans le cadre d'une approche complètement outillée. De plus les expériences industrielles ont été probantes et fournissent des résultats intéressants, en terme de retour d'expérience. Tous ces aspects dépassent largement le cadre de la méthode B,

comme ceci a été illustré en début de ce chapitre. En particulier l'expérience sur l'aspect méthodologie assistée et supportée par des outils est réutilisable dans d'autres contextes.

### Les besoins

Nous listons ci-dessous un certain nombre de pistes à développer, ou en cours de développement, qui correspondent à des besoins issus de la pratique. Les détails relatifs aux outils, projets et travaux référencés ci-après sont accessibles à partir du site B Grenoble, par exemple.

La tâche la plus complexe d'un développement formel est l'activité de preuve interactive. Elle nécessite de se concentrer à la fois sur la preuve à mettre en place, et sur sa mise en œuvre à l'aide des commandes offertes par le démonstrateur interactif. Un outil expérimental, EmacsPri[4], offre une interface plus souple et plus conviviale et semble faciliter le développement des preuves. Il propose de plus de nouvelles tactiques, qui permettent de traiter plus simplement certaines preuves[5]. Un point important à développer, pour supporter le processus de preuve interactive, est aussi d'assurer la traçabilité entre les obligations de preuve et le texte des spécifications. En effet, avant d'entreprendre une preuve, il faut la comprendre. Cette traçabilité n'est pas toujours facile à établir, en particulier lorsque les développements mettent en jeu beaucoup de composants.

La définition d'un processus de traduction paramétrable et validable du sous-ensemble implémentable de B vers un langage de programmation est aussi un besoin. En effet, comme ceci a été décrit dans la section 2.2.3 les exigences de traduction varient suivant les domaines d'application et les cibles. Cet aspect est abordé dans le projet RNTL BOM qui a pour objectif de définir un tel processus et de produire un traducteur B vers C adapté aux contraintes carte à puce. Ceci nécessite de pouvoir à la fois adapter le langage implémentable considéré (par exemple pour prendre en compte les différents formats d'entiers proposés par le langage C) et les schémas de traduction.

4. accessible sur le site de ClearSy

5. Par exemple une tactique d'analyse par cas dédiée à l'opérateur de surcharge d'une relation.

Certaines améliorations peuvent bien sûr être apportées au langage. Les plus importantes semblent être celles relatives à la notion de composant et de développement structuré. En effet, comme nous l'avons vu, ces possibilités sont à la base de la maîtrise de la complexité. Dans le cadre de la méthode B cette complexité se retrouve à la fois au niveau des spécifications et des développements, mais aussi au niveau du processus de validation par la preuve. Les besoins qui se font sentir sont relatifs à la possibilité de réutilisation, d'aide à la mise en place de développements modulaires et à l'assouplissement de certaines contraintes qui rendent actuellement complexes de tels développements. Une expérience de réutilisation d'un développement formel (celui de Météor) est en cours dans le cadre de la ligne Canarsie du métro de New-York et devrait donner des pistes intéressantes à explorer.

La méthode B se prête actuellement bien au processus de développement logiciels, de la spécification au code. Des besoins se font sentir pour des applications plus en amont dans le cycle de vie. On peut citer la formalisation de cahier des charges, qui permet en plus de la description formelle de faire des validations par la preuve, l'élaboration de spécifications prouvées et la modélisation au niveau système, qui met en jeu à la fois des composants logiciels et d'autres composants (environnement, composant matériel ...). Les extensions du B événementiel offrent une souplesse permettant de s'intéresser à ce niveau de modélisation. La notion d'évènement permet d'internaliser dans les spécifications le comportement global d'un système (contrairement aux opérations qui sont " appelées " de l'extérieur) et de décrire la causalité entre des événements. La prise en compte de la notion d'évènement demande peu d'extensions à la théorie [Abr96a, AM98]. Ceci signifie que l'outil actuel est utilisable, en particulier pour construire des systèmes par raffinement[6]. Par contre l'aspect méthodologique et le support offert par la méthode sont à développer.

Mon laboratoire, le LSR, est impliqué dans des travaux sur ces axes. Nous sommes un des partenaires du projet RNTL BOM. Dans ce projet notre contribution est principalement relative à l'aspect validation du traducteur et de certains outils visant à optimiser le code produit. Nous développons aussi des outils permettant d'assister la construction, le raffinement et la preuve

6. Un traducteur du format B-événementiel vers l'atelier B est disponible sur le site de ClearSy.

de systèmes B événementiels [BC00, Leb00, Nah01, RBB02]. Ces travaux se concrétisent par la mise en place d'une boite à outils, appelée BoB, qui offre un certain nombre de fonctionnalités permettant de manipuler le formalisme de B. Enfin, un des axes importants de mes travaux a été relatif au développement modulaire de spécifications et de développements dans la méthode B. Je me suis rapidement intéressée à ces aspects, ceux-ci étant dans la lignée de mes préoccupations. A la suite des premiers travaux sur la modularité[7], des " cas pathologiques " m'ont été soumis par les industriels utilisant et outillant la méthode B. Ceci a été à la base d'une partie des travaux que j'ai menés. J'ai donné des conditions permettant d'éliminer ces cas pathologiques[8] (conditions vérifiées par l'atelier B à partir de la version 3.5). Ces travaux m'ont amenée à développer le cadre formel permettant de valider ces conditions et plus généralement à m'intéresser à l'importance des différentes restrictions imposées par la méthode B, à la fois du point de vue théorique mais aussi du point de vue pratique.

Les difficultés rencontrées dans la méthode B sont inhérentes à l'objectif visé : mettre en place une démarche basée sur le raffinement modulaire, dans un langage à état, en acceptant une certaine forme de partage de variables. Ces difficultés sont de plusieurs ordres. Premièrement la théorie du raffinement n'a été que peu développée dans ce sens. La plupart du temps elle présuppose que l'état global du système est connu. Les premiers travaux relatifs à des opérateurs permettant de combiner des transformateurs de prédicats définis sur des espaces de variables non communs sont peu nombreux et assez récents [Mar91, Nau92, BB98]. Une seconde difficulté est que la notion d'état est généralement intrinsèquement globale à un système et se prête peu à la modularité. Enfin le partage introduit nécessairement des difficultés pour raisonner formellement. Nous retrouvons en particulier le problème bien connu des alias. Ceci signifie que la mise en place d'une telle approche ne peut être que le résultat d'un compromis entre la théorie et la pratique. Elaborer un tel compromis est une tâche difficile, qui satisfait rarement pleinement les utilisateurs. Remarquons que la méthode B a été initialement conçue sans permettre de partage, celui-ci a été introduit par la suite, à la demande des utilisateurs.

7. en collaboration avec Didier Bert et Yann Rouzaud
8. en collaboration avec Yann Rouzaud

J'ai choisi de centrer le thème de ce manuscrit sur le développement modulaire dans la méthode B, en considérant à la fois les aspects théoriques, les aspects langage et la mise en pratique des choix faits dans cette méthode. D'autre part, mes différentes activités dans le cadre de B et de la compilation des langages m'ont permis d'avoir une connaissance assez fine de cette approche. Je profiterais donc de ce manuscrit pour développer certains aspects non nécessairement détaillés dans le B-Book, aspects qui peuvent avoir un intérêt pour la communauté. Ce travail est l'aboutissement d'un certain nombre de discussions avec différentes personnes, en particulier J.R Abrial et les personnes de ClearSy en charge de l'atelier B.

# Chapitre 3

# Substitutions généralisées

Le but de ce chapitre est de faire une introduction synthétique au langage des substitutions généralisées, qui est à la base de l'écriture des spécifications dans la méthode B, et du mécanisme de production des obligations de preuve. Une partie des résultats est issue du B-Book mais ce chapitre apporte un certain nombre de compléments. Nous développons en particulier deux aspects importants du point de vue de la compositionnalité : les opérations et leurs appels, et la composition parallèle de substitutions. Les résultats sur la composition parallèle sont étendus pour traiter tous les cas de partage autorisés dans la méthode B et un algorithme effectif du calcul de la plus faible précondition est donnée pour cet opérateur. Le traitement des appels d'opérations est rapidement évoqué dans le B-Book mais ne concerne que le niveau des spécifications. La règle donnée est suffisante pour les preuves, puisque seules les spécifications sont considérées, mais ne permet pas de construire le corps des appels, lors du processus de raffinement. Cette étape peut s'avérer nécessaire, par exemple pour produire ou optimiser du code. Dans ce chapitre nous étendons la définition du B-Book et établissons les propriétés préservées par les appels. De plus, nous donnons des conditions permettant de remplacer le passage de paramètre par copie par un passage par référence. Cette optimisation est importante dans le cas des logiciels embarqués, pour lesquels la place mémoire est restreinte.

Les résultats présentés dans ce chapitre ont été développés et implantés en grande partie soit dans le cadre du projet RNTL BOM soit dans le cadre d'une boite à outils développée au LSR (l'outil BoB).

## 3.1 Langages de spécification

Bien que le but de ce chapitre ne soit pas de présenter la méthode B, nous débutons par un exemple, afin d'introduire la notion de machine qui permet de modéliser des entités qui ressemblent à des composants d'un langage de programmation classique (ou à des objets). Les différents constituants d'une machine sont :

- un état interne (constantes et variables) ;
- une initialisation ;
- des opérations ;
- des propriétés invariantes.

La méthode B est basée sur une théorie des ensembles que nous ne détaillerons pas dans ce manuscrit [Abr96b]. Les ensembles permettent de modéliser les données du modèle (constantes et variables). L'initialisation et les opérations sont décrites à l'aide du langage des *substitutions généralisées*, qui peut être vu comme un langage de programmation abstrait. Ce langage est décrit dans ce chapitre. Enfin les propriétés sont exprimées à l'aide de la logique du premier ordre, étendue à la théorie ensembliste considérée (non détaillée ici). Le tableau ci-dessous récapitule ces différentes notions :

| Modèle | Langage |
|---|---|
| données | ensembles |
| traitements | substitutions généralisées |
| propriétés | prédicats du premier ordre |

Avant de rentrer dans les aspects plus théoriques, nous donnons un exemple de machine abstraite, simple à comprendre[1]. La machine *RESERVATION*, donnée ci-après, décrit un service de réservation de places. Les places sont numérotées de 1 à $nb\_max$, cette dernière valeur étant un paramètre de la spécification. Les opérations offertes permettent de tester s'il reste des places (opération `place_libre`), de réserver une place (opération `réserver`) et de libérer une place déjà réservée (opération `libérer`).

### Les données

Le paramètre $nb\_max$ est le nombre maximum de places. Ce paramètre vérifie certaines contraintes (clause CONSTRAINTS) : il est supérieur ou égal à 1 et inférieur ou égal à *MAXINT*. Cette dernière contrainte permettra d'implanter simplement la spécification dans un langage n'offrant que des entiers bornés. D'autre part, pour des questions de lisibilité, on définit l'ensemble *SIEGES* comme l'intervalle $1..nb\_max$ (clause DEFINITIONS).

L'état du système est modélisé par un ensemble, $occupés$, qui contient les places déjà allouées. On ajoute au modèle une variable, $nb\_libre$, qui permet d'accéder directement au nombre de places libres (cette variable n'apporte pas d'information supplémentaire au modèle, elle est introduite pour illustrer la notion d'invariant). L'invariant est :

$$\begin{array}{l} occupés \subseteq SIEGES \\ \wedge\ nb\_libre \in 0\ ..\ nb\_max \\ \wedge\ nb\_libre = nb\_max - card(occupés) \end{array}$$

La propriété $nb\_libre \in 0\ ..\ nb\_max$ est une conséquence des autres propriétés puisque $nb\_max$ est positif (hypothèse sur le paramètre de la machine `RESERVATION`). Elle permet de typer la variable $nb\_libre$.

### La dynamique

Initialement, l'ensemble des sièges occupés est vide et le nombre de sièges libres est $nb\_max$. L'opération `place_libre` renvoie la valeur de la variable $nb\_libre$. L'opération `réserver` a une précondition (notation `PRE`) : elle ne peut être appelée que s'il reste des sièges libres. Dans ce cas, elle attribue

1. Un développement complet de cet exemple est accessible à l'adresse http://www-lsr.imag.fr/Les.Personnes/Marie-Laure.Potet/

une place parmi celles disponibles. Cette opération est non-déterministe : la politique d'allocation des sièges n'est pas précisée. Le non-déterminisme est décrit par la construction ANY $z$ WHERE $P$ THEN $S$ END, qui permet de " paramétrer " la substitution $S$ pour n'importe quelle valeur $z$ vérifiant le prédicat $P$. L'existence d'une place libre est garantie par la précondition. L'opération `libérer` a une précondition qui vérifie que la place à libérer est effectivement occupée.

MACHINE
$RESERVATION\ (nb\_max)$

CONSTRAINTS
$nb\_max \in 1\ ..\ MAXINT$

DEFINITIONS
$SIEGES == (1\ ..\ nb\_max)$

VARIABLES
$occupés,\ nb\_libre$

INVARIANT
$occupés \subseteq SIEGES \wedge nb\_libre \in 0\ ..\ nb\_max$
$\wedge\ nb\_libre = nb\_max\text{-}card(occupés)$

INITIALISATION
$occupés := \emptyset\ ||\ nb\_libre :=\ nb\_max$

OPERATIONS
$nb \leftarrow place\_libre\ \hat{=}$
BEGIN
$nb := nb\_libre$
END ;

$pp \leftarrow réserver\ \hat{=}$
PRE $card(occupés) \neq nb\_max$ THEN
ANY $place$ WHERE $place \in SIEGES$ - $occupés$ THEN
$pp{:=}place\ ||\ occupés{:=}occupés \cup \{place\}\ ||\ nb\_libre{:=}nb\_libre\text{-}1$
END
END ;

$liberer\ (place)\ \hat{=}$

PRE $place \in occupés$
THEN $occupés := occupés - \{place\} \parallel nb\_libre := nb\_libre + 1$
END

END

L'opérateur || permet de décrire des affectations en parallèle, portant sur des variables disjointes.

Les preuves liées à une telle spécification vont consister à s'assurer que les propriétés invariantes sont vérifiées par la dynamique. Ceci revient à vérifier que l'initialisation établit l'invariant et que toutes les opérations préservent l'invariant (si l'invariant est vrai avant l'opération, et que celle-ci est invoquée dans sa précondition, alors l'invariant est vrai après l'opération). Ces vérifications se modélisent sous la forme *d'obligations de preuve* qui sont produites, à partir des substitutions généralisées, par un calcul de plus faible précondition.

Dans la section 3.2 nous décrivons les substitutions généralisées primitives et leurs axiomatisation. La section 3.3 introduit la définition relationnelle des substitutions généralisées, qui permet de relier ce formalisme aux formes de spécification basées sur la logique (VDM, Z, TLA ...). Dans la section 3.4 nous détaillons la substitution parallèle. Enfin dans la section 3.5 nous détaillons l'appel d'opérations et ses propriétés.

## 3.2 Substitutions généralisées primitives

Le langage des substitutions généralisées est issu de la théorie des transformateurs de prédicats, développée par E.W. Dijkstra [Dij76]. Le calcul du raffinement, initialement introduit par R. Back [Bac80] et redéveloppé par C. Morgan [Mor88, MG90] et J. Morris [Mor87], permet de traiter de manière uniforme les spécifications et les programmes, en les considérant comme des transformateurs de prédicat. Le calcul du raffinement offre ainsi un cadre permettant de dériver les programmes à partir de spécifications. La notion de correction est modélisée par une relation de raffinement entre les spécifications.

Les spécifications sont classiquement décrites à l'aide de la logique, en exprimant des préconditions et des postconditions. Dans la méthode B, le formalisme utilisé est le langage des substitutions généralisées, qui ressemble à un pseudo langage de programmation. Bien que ces deux approches soient théoriquement équivalentes (voir section 3.3), le langage des substitutions généralisées est complètement adapté à la preuve, car il intègre explicitement le mécanisme de substitution dans les formules logiques. De plus il offre un style de spécification proche de celui des langages de programmation.

Les substitutions sont définies à partir d'un ensemble de substitutions généralisées que l'on peut qualifier de primitives. Pour alléger la présentation, les substitutions primitives seront ici introduites en deux classes : les substitutions que nous qualifions de mathématiques, puis les substitutions plus algorithmiques (séquencement et itération). Bien que cette classification corresponde actuellement aux substitutions admises aux différents niveaux de spécification (machine, raffinement et implémentation), elle n'a pas plus de justification.

### 3.2.1 Substitutions mathématiques

Nous donnons ici les substitutions mathématiques et leur axiomatisation en terme de plus faible précondition.

| Notation | Nom |
|---|---|
| $x := e$ | substitution simple |
| $x, y := e, f$ | substitution multiple simple |
| skip | substitution sans effet |
| $P \mid S$ | substitution préconditionnée |
| $P \Longrightarrow S$ | substitution gardée |
| $S \,[\!]\, T$ | substitution choix borné |
| $@z \cdot S$ | substitution choix non borné |

L'affectation a son sens habituel. La substitution skip est la substitution sans effet. L'affectation multiple est simultanée. Par exemple $x, y := y, x$ décrit la permutation des valeurs de $x$ et $y$. La substitution préconditionnée

permet de donner des contraintes sur les entrées (variables d'état et paramètres). La validité de la précondition, pour un état donné, sera prise en compte dans les obligations de preuve. La substitution gardée permet de donner des hypothèses sous lesquelles la substitution $S$ est utilisée. Contrairement à la substitution préconditionnée, la garde n'a pas à être prouvée. Cette substitution est utilisée en particulier pour définir des substitutions conditionnelles. La substitution choix borné permet de décrire des spécifications non déterministes. Par exemple la substitution $x := 1 \,[\!]\, x := 2$ décrit l'affectation d'une des valeurs 1 ou 2 à la variable $x$. La substitution choix non borné est une extension du non déterminisme. Par exemple la substitution $@z \cdot (z > x \Longrightarrow x := z)$ a pour effet d'affecter à la variable $x$ une valeur strictement supérieure à la précédente.

Comme dit précédemment, les substitutions généralisées peuvent être vues comme des transformateurs de prédicats. Nous donnons les axiomes pour les substitutions mathématiques. Ils permettront, par la suite, de construire les obligations de preuve. La notation est $[S]R$ où $S$ est une substitution et $R$ un prédicat.

**Définition 1** *Axiomes de base*

| *Axiome* | | | *Condition* |
|---|---|---|---|
| $[x, y := e, f]\, R$ | $\Leftrightarrow$ | $[z := f][x := e][y := z]\, R$ | $z \backslash e, f, R$ |
| $[\mathsf{skip}]\ R$ | $\Leftrightarrow$ | $R$ | |
| $[P \mid S]\, R$ | $\Leftrightarrow$ | $P \ \wedge\ [S]\, R$ | |
| $[S \,[\!]\, T]\, R$ | $\Leftrightarrow$ | $[S]R \ \wedge\ [T]\, R$ | |
| $[P \Longrightarrow S]\, R$ | $\Leftrightarrow$ | $P \Rightarrow [S]\, R$ | |
| $[@z \cdot S]\, R$ | $\Leftrightarrow$ | $\forall z \cdot [S]\, R$ | $z \backslash R$ |

La notation $x \backslash t_1, \ldots t_n$ signifie que la variable $x$ n'est pas libre dans les termes $t_i$. Si tel n'est pas le cas, il est toujours possible de renommer. Le cas de base $[x := e]R$ est la substitution uniforme de $x$ par $e$ dans $R$, non détaillée dans ce manuscrit. Le langage B n'autorise aucune forme d'alias, cette substitution est donc bien conforme à la sémantique de l'affectation.

### 3.2.2 Substitutions de programmation

La substitution séquencement a sa notation habituelle $S$ ; $T$. Elle est définie par l'axiome suivant :

**Définition 2** *Axiome du séquencement*

$$[S\ ;\ T]\,R \quad \Leftrightarrow \quad [S]([T]\,R)$$

L'itération contient les éléments de sa preuve, c'est-à-dire une expression formelle de son invariant et son variant (l'expression qui décroît à chaque passage). Les itérations, après preuve, seront donc toujours finies. La notation est :

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

où $I$ est l'invariant et $V$ le variant.

**Définition 3** *Axiome de l'itération*

$$\begin{array}{ll} & [\text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END}]\,R \\ \Leftrightarrow & \\ & (I \\ & \wedge\ \forall x\,.\,(I\ \wedge\ P \Rightarrow [S]\,I) \\ & \wedge\ \forall x\,.\,(I \Rightarrow V \in \mathbb{N}) \\ & \wedge\ \forall x\,.\,(I\ \wedge\ P \Rightarrow [n := V][S](V < n)) \\ & \wedge\ \forall x\,.\,(I\ \wedge\ \neg\,P \Rightarrow R)) \end{array}$$

L'invariant $I$ doit être établi avant l'itération et préservé à chaque passage. Le variant $V$ doit être un entier naturel qui décroit lors du passage dans la boucle. Finalement, la sortie de boucle doit impliquer la formule $R$.

### 3.2.3 Notations syntaxiques

Des extensions syntaxiques permettent de définir de nouvelles constructions et d'introduire des abréviations. Par exemple la conditionnelle se définit de la manière suivante :

| Notation | Définition |
|---|---|
| IF $P$ THEN $S$ ELSE $T$ END | $(P \Longrightarrow S) \,[\!]\, (\neg\ P \Longrightarrow T)$ |
| IF $P$ THEN $S$ END | $(P \Longrightarrow S) \,[\!]\, (\neg\ P \Longrightarrow \mathsf{skip})$ |

Il est alors possible de calculer la plus faible précondition de ces extensions syntaxiques en utilisant leur définition. Par exemple pour la conditionnelle on obtient :

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

La substitution non déterministe *any z where P then S end*, utilisée dans l'exemple, se définit par :

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

## 3.3 Terminaison et prédicat avant/après

### 3.3.1 Définition

Généralement, dans les langages de spécifications basés sur la notion d'états (Z, VDM, TLA ...), la dynamique est décrite à l'aide d'un prédicat avant/après, et, dans certains cas, par une précondition (langage VDM par exemple). Un lien peut être établi avec les substitutions généralisées par les définitions suivantes :

$$\mathsf{prd}_x(S) \quad \Leftrightarrow \quad \neg\ [S](x' \neq x)$$

$$\mathsf{trm}(S) \quad \Leftrightarrow \quad [S]true$$

Le prédicat prd est indicé par l'ensemble des variables sur lequel la substitution est définie (généralement les variables de la machine). La notation $x'$ désigne les valeurs des variables après la substitution (comme dans la notation Z). Le prédicat avant/après dépend donc de l'espace de variables sur lequel est définie la substitution. Par exemple la substitution $x := 3$ s'interprète différemment sur les espaces de variables $\{x\}$ et $\{x, y\}$. Nous avons $\mathsf{prd}_{\{x\}}(x := 3) \Leftrightarrow (x' = 3)$ et $\mathsf{prd}_{\{x,y\}}(x := 3) \Leftrightarrow (x' = 3 \wedge y' = y)$.

Les substitutions du langage B peuvent contenir des préconditions. Celles-ci seront principalement utilisées lors de la définition des opérations et devront être vérifiées lors des appels. Une telle solution est aussi adoptée dans VDM. Dans ce cas on peut calculer le prédicat de terminaison, noté trm, qui décrit la plus faible précondition pour laquelle la substitution termine. Le prédicat trm impose l'arrêt des itérations et la vérification que les opérations sont appelées dans leur précondition. La définition des prédicats trm et prd, pour chaque substitution primitive, est donnée dans l'annexe A (p. 139).

La notion de substitution est donc plus proche de la vision des langages de programmation : les modifications sont décrites de manière explicite. La notion de prédicat avant/après permet, quant à elle, un style de spécification plus déclaratif. Si aucune contrainte n'est posée explicitement alors il n'y en a pas. Par exemple le prédicat $x' = 3$ ne fait que contraindre l'évolution de la variable $x$. Si on s'intéresse à l'espace de variables $\{x, y\}$ toute valeur de $y'$ convient [2]. Ce style de spécification a de bonnes propriétés, il permet en particulier de composer les spécifications par des opérateurs logiques. A contrario, cette approche est moins adaptée pour la preuve et le raffinement. Les connecteurs logiques n'ont, par exemple, que peu de propriétés algébriques vis-à-vis du raffinement.

### 3.3.2 Forme normalisée

Les deux prédicats, trm et prd, caractérisent complètement les substitutions. En particulier toute substitution $S$ peut être mise sous la forme nor-

2. Remarquons que ceci est une source fréquente d'erreurs, on oublie de préciser que les variables ne changent pas.

malisée suivante :

**Théorème 1** *Forme normalisée*

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

Cette égalité peut être démontrée par récurrence sur les substitutions primitives, par exemple en montrant l'équivalence $[S]R \Leftrightarrow [T]R$, pour tout prédicat $R$, où $S$ est une substitution et $T$ sa forme normalisée. On obtient donc :

**Propriété 1** *Calcul de plus faible précondition*

$$[S]R \Leftrightarrow \mathsf{trm}(S) \wedge \forall x' \;.\; (\mathsf{prd}_x(S) \Rightarrow [x := x']R)$$

La forme normalisée permet d'établir le lien entre le formalisme des substitutions généralisées et les spécifications exprimées plus classiquement en logique. D'autre part, la forme normalisée est utile pour prouver des résultats généraux sur les substitutions. Par contre, dans la mesure du possible, elle n'est pas utilisée pour construire les obligations de preuve. En effet, comme nous l'avons noté en début de ce chapitre, le formalisme des substitutions généralisées est nettement plus adapté à la preuve. Illustrons cela à travers quelques exemples.

Le calcul de plus faible précondition $[x := 3]x > 0$ a pour résultat $3 > 0$. Si on utilise la forme normalisée, la formule devient $\forall x' \, (x' = 3 \Rightarrow [x := x']x > 0)$, ce qui se réduit à $x' = 3 \Rightarrow x' > 0$. Cette formulation demande une prise en compte explicite de l'égalité. De la même manière le calcul de la formule $[x := 3 \;;\; y := x + 1]y > 0$ se réduit en $[x := 3][y := x + 1]y > 0$, c'est-à-dire en $3 + 1 > 0$. Si on utilise la forme normalisée, on obtient la formule suivante (voir annexe A pour les définitions) :

$$\forall x', y' \, (\exists x'' \cdot ([x' := x'']x' = 3 \;\wedge\; [x := x'']\, y' = x + 1) \Rightarrow [x, y := x', y']y > 0)$$

c'est-à-dire :

$$\forall y' \, (\exists x'' \cdot (x'' = 3 \;\wedge\; y' = x'' + 1) \Rightarrow y' > 0)$$

La preuve nécessite la prise en compte de l'égalité et de certaines propriétés du quantificateur existentiel.

### 3.3.3 Itération et forme normalisée

Le prédicat de terminaison contient l'information que l'invariant de boucle est préservé à chaque pas et que l'arrêt du calcul est garanti :

$$
\begin{array}{l}
\mathsf{trm}\ (\text{WHILE}\ P\ \text{DO}\ S\ \text{INVARIANT}\ I\ \text{VARIANT}\ V\ \text{END}) \\
\Leftrightarrow \\
\quad I\ \wedge \\
\quad \forall x \cdot (I\ \wedge\ P\ \Rightarrow\ [S]\, I)\ \wedge \\
\quad \forall x \cdot (I\ \Rightarrow\ V \in \mathbb{N})\ \wedge \\
\quad \forall x \cdot (I\ \wedge\ P\ \Rightarrow\ [n := V][S](V < n))
\end{array}
$$

Le prédicat $\mathsf{prd}_x$, donné ci-après, ne correspond pas exactement à celui énoncé dans le B-Book. Dans ce dernier, le prédicat $[x := x']\,(I\ \wedge\ \neg\, P)$ n'est pas conditionnée par la terminaison. En toute rigueur cette condition est nécessaire, afin de d'établir les propriétés générales de la section 3.3.4. La formule donnée ici correspond à l'instanciation de la définition générale du prédicat $\mathsf{prd}$ :

$$
\begin{array}{l}
\mathsf{prd}_x\ (\text{WHILE}\ P\ \text{DO}\ S\ \text{INVARIANT}\ I\ \text{VARIANT}\ V\ \text{END}) \\
\Leftrightarrow \\
\quad (\mathsf{trm}\ (\text{WHILE}\ P\ \text{DO}\ S\ \text{INVARIANT}\ I\ \text{VARIANT}\ V\ \text{END}) \\
\quad \Rightarrow [x := x']\,(I\ \wedge\ \neg\, P))
\end{array}
$$

Dans le cas d'une itération, on peut remarquer que le calcul direct de la plus faible précondition donne une forme syntaxique (presque) identique à celle obtenue à partir de la forme normalisée. Soit $T$ la substitution WHILE $P$ DO $S$ INVARIANT $I$ VARIANT $V$ END. En utilisant la forme normalisée, $[T]R$ devient :

$$\mathsf{trm}(T) \wedge \forall x'.((\mathsf{trm}(T) \Rightarrow [x := x'](I \wedge \neg P)) \Rightarrow [x := x']R)$$

qui se simplifie en :

$$\mathsf{trm}(T) \wedge \forall x'.([x := x'](I \wedge \neg P) \Rightarrow [x := x']R)$$

puisque $x'$ est non libre dans $\mathsf{trm}(T)$. On obtient finalement, par renommage :

$$\mathsf{trm}(T) \wedge \forall x.(I \wedge \neg P \Rightarrow R)$$

Cette formule correspond exactement à la définition de la plus faible précondition pour l'itération (voir section 3.2.2, p. 50). Ceci signifie que, après simplification, les formules à prouver sont syntaxiquement identiques. Dans le cas de l'itération il est donc équivalent d'exécuter les calculs directement à partir de la substitution ou bien à partir de la forme normalisée.

### 3.3.4 Propriétés générales

Nous énonçons ci-après un certain nombre de propriétés générales qui seront utiles dans la suite. Ces propriétés peuvent, par exemple, être établies en prenant la forme normalisée des substitutions. La première propriété établit le fait que le calcul de plus faible précondition se distribue sur la conjonction (B-Book p. 287).

**Propriété 2** *$\wedge$-distributivité*

$$[S](A \wedge B) \Leftrightarrow ([S]A \wedge [S]B)$$

Ceci n'est pas vrai en général pour les autres connecteurs logiques. Cette propriété est intéressante car elle permet de couper les preuves portant sur des conjonctions. La seconde propriété décrit la monotonie du calcul de plus faible précondition, par rapport à l'implication. Elle permet d'instancier une implication générale (B-Book, p. 287).

**Propriété 3** *Monotonie de la substitution pour l'opérateur $\Rightarrow$*

$$(\forall x.(A \Rightarrow B)) \Rightarrow ([S]A \Rightarrow [S]B)$$

*$S$ modifiant les variables $x$.*

La troisième propriété garantit que, dès qu'une substitution établit un prédicat, la terminaison est aussi établie.

**Propriété 4**

$$[S]R \Rightarrow \mathsf{trm}(S)$$

Cette propriété découle directement de la précédente. A partir de l'implication $R \Rightarrow true$ on déduit par monotonie $[S]R \Rightarrow [S]true$, c'est-à-dire $[S]R \Rightarrow \mathsf{trm}(S)$. Le calcul de plus faible précondition prend donc en compte la terminaison des substitutions.

**Propriété 5**

$$\mathsf{prd}_x(S) \vee \mathsf{trm}(S)$$

Cette propriété est une conséquence de la propriété 4, en instanciant $R$ par le prédicat $x' \neq x$. On a $[S](x' \neq x) \Rightarrow \mathsf{trm}(S)$, c'est-à-dire $\mathsf{prd}_x(S) \vee \mathsf{trm}(S)$.

### 3.3.5 Condition de raffinement

Le raffinement dans la méthode B repose sur la notion d'invariant. Soit $S_1$ une substitution définie sur l'espace de variables $x_1$ et soit $S_2$ une substitution définie sur l'espace de variables $x_2$. Soit $J$ un prédicat portant sur les variables $x_1$ et $x_2$. Le raffinement de la substitution $S_1$ par la substitution $S_2$, pour l'invariant $J$, est noté $S_1 \sqsubseteq_J S_2$ et est défini par :

**Définition 4** *Raffinement par invariant*

$$S_1 \sqsubseteq_J S_2 \Leftrightarrow (J \ \wedge \ \mathsf{trm}(S_1) \Rightarrow [S_2]\neg[S_1]\neg J)$$

Cette définition a été proposée par D. Gries et J. Prins dans [GP85] et correspond à l'approche choisie dans la méthode B pour décrire et établir les raffinements. L'équivalence de cette définition avec le raffinement de données de la théorie du raffinement, dans le cas du *forward data refinement* [GM93],

a été établi par [CU89] et étendue pour la méthode B dans [Rou99]. Cette équivalence garantit les propriétés usuelles du raffinement (transitivité et monotonie), qui ne sont donc pas détaillées ici. Nous donnons ci-après quelques calculs utiles par la suite.

**Propriété 6** *Preuves de raffinement*

*Soit $P_1 \mid S_1$ et $P_2 \mid S_2$ deux substitutions. Le raffinement $P_1 \mid S_1 \sqsubseteq_J P_2 \mid S_2$ s'exprime par les conditions suivantes :*

$$(REF_1) \quad J \ \wedge \ P_1 \ \wedge \ \mathsf{trm}(S_1) \Rightarrow P_2$$
$$(REF_2) \quad J \ \wedge \ P_1 \ \wedge \ \mathsf{trm}(S_1) \Rightarrow [S_2]\neg[S_1]\neg J$$

*De plus, en utilisant les formes normalisées, la formule $(REF_2)$ devient :*

$$J \ \wedge \ P_1 \ \wedge \ \mathsf{trm}(S_1) \ \wedge \ Q_2 \Rightarrow \exists x_1' \, (Q_1 \wedge [x_1, x_2 := x_1', x_2']J)$$

*où $Q_1$ et $Q_2$ désignent les prédicats avant-après respectifs des substitutions $P_1 \mid S_1$ et $P_2 \mid S_2$ et portent respectivement sur les couples de variables $(x_1, x_1')$ et $(x_2, x_2')$.*

## 3.4 Substitution ||

Le constructeur de substitution || permet de combiner des substitutions définies sur des espaces de variables différents. Cet opérateur n'est pas un opérateur standard du calcul du raffinement mais il est important dans le cas des développements modulaires [BB98]. Il va permettre de combiner des opérations, en réutilisant leur spécification et les preuves associées. Cet opérateur est particulier : il n'est pas primitif dans le sens où il n'est pas possible de le définir directement en terme d'un transformateur de prédicats et il ne s'exprime pas non plus directement en terme de substitutions primitives. Par contre, il existe des règles qui permettent de construire une substitution généralisée équivalente.

Dans la méthode B l'opérateur || peut être utilisé dans différentes configurations. Dans une machine, il est possible de mettre en parallèle deux

substitutions qui modifient des variables disjointes et qui lisent des variables communes, ces variables étant déclarées dans la machine. Par exemple, si les variables d'une machine sont $x$ et $y$, la substitution $x := y \parallel y := x$ a pour effet de permuter les valeurs de $x$ et $y$. Par contre la substitution $x := 3 \parallel x := 4$ n'est pas autorisée. A l'extérieur des machines[3], il est possible de mettre en parallèle des opérations qui partagent des variables, mais uniquement en lecture[4]. Par exemple on peut construire la substitution $op_1 \parallel op_2$ avec $op_1 \mathrel{\hat{=}} x := z$ et $op_2 \mathrel{\hat{=}} y := z + 1$, qui a pour effet d'affecter simultanément les variables $x$ et $y$. Le premier cas d'utilisation, la mise en parallèle de deux substitutions portant sur les variables de la machine, est une facilité syntaxique. Dans la seconde forme d'utilisation, l'opérateur $\parallel$ permet de construire de nouvelles spécifications en évitant toute forme de séquentialité, qui peut induire des pas intermédiaires non nécessairement significatifs.

Nous adoptons, dans un premier temps, une vue syntaxique de cet opérateur qui permet le calcul de plus faible précondition pour les différentes utilisations de cet opérateur. Puis nous dérivons des propriétés permettant de composer les preuves. Ces propriétés étendent celles du B-Book, afin de traiter les formes de partage autorisées par les outils. Initialement la méthode B n'autorisait pas de partage, celui-ci a été introduit à la demande des utilisateurs. Ceci explique pourquoi les outils traitent, en général, plus de possibilités que celles considérées dans le B-Book.

### 3.4.1 Axiomes des substitutions multiples

Le sens de l'opérateur $\parallel$ peut être défini par des égalités (B-Book, p. 309). Elles permettent d'éliminer cet opérateur, lorsqu'il se combine avec des substitutions mathématiques, en remplaçant le terme gauche par le terme droit. Une condition implicite est que les substitutions composées ne modifient pas des variables communes (égalité 1).

3. Cette notion sera précisée dans le chapitre 4.

4. Ceci est un choix de notre part. Les configurations autorisées dans le B-Book sont très restrictives et celles admises par l'AtelierB incluent ce cas (Manuel de référence version 1.8.5).

| Equivalence | | | | Condition |
|---|---|---|---|---|
| 1) | $x := e \parallel y := f$ | $=$ | $x, y := e, f$ | |
| 2) | $\mathsf{skip} \parallel S$ | $=$ | $S$ | |
| 3) | $(P \mid T) \parallel S$ | $=$ | $P \mid (T \parallel S)$ | |
| 4) | $(T \,[\!]\, U) \parallel S$ | $=$ | $(T \parallel S) \,[\!]\, (U \parallel S)$ | |
| 5) | $(P \Longrightarrow T) \parallel S$ | $=$ | $P \Longrightarrow (T \parallel S)$ | $\mathsf{trm}(S)$ |
| 6) | $(@z \cdot T) \parallel S$ | $=$ | $@z \cdot (T \parallel S)$ | $z \backslash S$ |
| 7) | $T \parallel S$ | $=$ | $S \parallel T$ | |

Dans la règle 5, la condition sur la terminaison de la substitution $S$ est nécessaire. En effet, si cette condition n'est pas vérifiée, les substitutions obtenues ne sont pas égales. Par exemple la substitution $(x > 4 \Longrightarrow x := 2) \parallel (x > 3 \mid y := 1)$ peut se réécrire en deux formes :

$$\begin{array}{ll} x > 3 \mid ((x > 4 \Longrightarrow x := 2) \parallel y := 1) & \textit{égalités } 7, 3 \\ x > 3 \mid (x > 4 \Longrightarrow (x := 2 \parallel y := 1)) & \textit{égalité } 5 \\ x > 3 \mid (x > 4 \Longrightarrow x, y := 2, 1) & \textit{égalité } 1 \end{array}$$

ou bien :

$$\begin{array}{ll} x > 4 \Longrightarrow (x := 2 \parallel (x > 3 \mid y := 1)) & \textit{égalité } 5 \\ x > 4 \Longrightarrow (x > 3 \mid (x := 2 \parallel y := 1)) & \textit{égalités } 7, 3 \\ x > 4 \Longrightarrow (x > 3 \mid x, y := 2, 1) & \textit{égalité } 1 \end{array}$$

La première forme termine si $x > 3$ alors que la seconde termine dans tous les cas $(x > 4 \Rightarrow x > 3)$.

Le B-Book ne décrit pas d'algorithme effectif pour effectuer le calcul de plus faible précondition. Nous en avons développé un dans la boite à outil BoB[5]. Cet algorithme construit une substitution équivalente, dans laquelle les opérateurs || ont été éliminés. Il est basé sur les égalités précédentes, utilisées de gauche à droite. La seule égalité non effective est la règle 5,

5. Ce travail a essentiellement était fait par Nicolas Stouls, étudiant en Magistère.

car elle introduit une condition non vérifiable statiquement ($\mathsf{trm}(S)$). Nous l'avons remplacé par l'égalité :

$$5') \quad (P \Longrightarrow T) \parallel S \quad = \quad \mathsf{trm}(S) \mid P \Longrightarrow (T \parallel S)$$

qui permet de construire une substitution équivalente, qui contient la condition de sa validation. La correction de cette égalité est démontrée dans l'annexe A (section A.3.2). Le jeu de règles ainsi obtenu permet d'éliminer le constructeur $\parallel$ lorsqu'il se combine avec des substitutions mathématiques. Comme nous l'avons développé dans la section 3.3.3, l'itération peut être remplacée par sa forme normalisée, sans compliquer les formules obtenues. Le seul cas qui reste à traiter est le séquencement. En effet il n'y a pas de règle permettant d'éliminer le constructeur $\parallel$ en présence du séquencement, et le passage par la forme normalisée produit des formules complexes (voir l'exemple de la section 3.3.2). Néanmoins il existe un jeu d'égalités qui permet d'éliminer le séquencement, lorsqu'il se combine avec les substitutions mathématiques. Ces égalités sont rappelées dans l'annexe A (section A.2). Ces règles, associées aux précédentes, permettent d'éliminer toute occurrence de de l'opérateur séquencement, en présence de l'opérateur $\parallel$.

### 3.4.2 Propriétés

L'objectif ici est de dériver des propriétés d'une substitution de la forme $S_1 \parallel S_2$ en terme des propriétés de $S_1$ et de $S_2$. Ces propriétés seront intéressantes lorsqu'on composera des opérations car elles permettront aussi de composer les preuves (voir chapitre 4). Dans un premier temps nous cherchons à caractériser les prédicats $\mathsf{trm}$ et $\mathsf{prd}$ d'une substitution de la forme $S_1 \parallel S_2$ à partir de $S_1$ et $S_2$. Ceci n'est possible que pour des formes restreintes. Par exemple la substitution $op_1 \parallel op_2$ avec $op_1 \mathrel{\hat{=}} x := y$ et $op_2 \mathrel{\hat{=}} y := x$ ne peut pas être interprétée à partir des prédicats $\mathsf{prd}$ de ces opérations. En effet $op_1$ est une opération qui ne modifie pas $y$, son prédicat avant-après est $x' = y \wedge y' = y$ ($x' = x \wedge y' = x$ pour $op_2$). Par les égalités de la section précédente, le prédicat avant/après de la substitution $op_1 \parallel op_2$ est $x', y' = y, x$, qui ne se calcule pas directement à partir des formules associées à $op_1$ et $op_2$.

### Cas sans partage

Les restrictions introduites dans le B-Book imposent que les substitutions mises en parallèle soient définies pour des espaces de variables disjoints. On a alors :

**Théorème 2** trm *du constructeur* ||

$$\mathsf{trm}(S \parallel T) \quad \Leftrightarrow \quad \mathsf{trm}(S) \quad \wedge \quad \mathsf{trm}(T)$$

**Théorème 3** prd *du constructeur* ||

*Soit $S_1$ une substitution définie sur l'espace de variables $x_1$ et soit $S_2$ une substitution définie sur l'espace de variables $x_2$. Si $x_1 \cap x_2 = \emptyset$ alors :*

$$\mathsf{prd}_{x_1 \cup x_2}(S_1 \parallel S_2) \quad \Leftrightarrow \quad (\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{x_1}(S_1)) \quad \wedge \quad (\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{x_2}(S_2))$$

La définition ci-dessus a été proposée par Steve Dunne [Dun99]. Cette définition étend celle proposée dans le B-Book, pour respecter la propriété 5 qui établit la disjonction $\mathsf{prd}(S) \vee \mathsf{trm}(S)$ pour toute substitution $S$ (section 3.3.4, p. 55). Les preuves se font par induction sur les égalités de la section précédente. Ces preuves sont développées dans l'annexe A (section A.3.1), afin de justifier les extensions proposées. Le théorème 2 ne dépend pas du fait que les espaces de variables soient disjoints. Cette hypothèse est utile pour le théorème 3, et uniquement dans le cas des égalités 1 et 2 qui portent sur les termes de la forme $x := e \parallel y := f$ et $\mathsf{skip} \parallel S$.

### Cas avec partage en lecture

Le théorème 3 peut être étendu au cas où les substitutions $S_1$ et $S_2$ partagent des variables qui ne sont modifiées ni par $S_1$ ni par $S_2$. Dans ce cas on

dit que les variables sont partagées *en lecture*. Cette hypothèse correspond aux compositions d'opération admises (voir l'introduction de la section 3.4). On voit bien que, sous cette hypothèse, la réduction des termes de la forme $x := e \parallel y := f$ et $\mathsf{skip} \parallel S$ ne pose pas de problème particulier.

**Théorème 4** *Extension du théorème 3*

*Soit $S_1$ une substitution définie sur l'espace de variables $x_1$ et soit $S_2$ une substitution définie sur l'espace de variables $x_2$. Si $x_1 \cap x_2 = u$, on a :*

$$\begin{array}{c} ((\mathsf{trm}(S_1) \wedge \mathsf{prd}_{x_1}(S_1) \Rightarrow u' = u) \wedge (\mathsf{trm}(S_2) \wedge \mathsf{prd}_{x_2}(S_2) \Rightarrow u' = u)) \\ \Rightarrow \\ (\mathsf{prd}_{x_1 \cup x_2}(S_1 \parallel S_2) \quad \Leftrightarrow \quad (\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{x_1}(S_1)) \quad \wedge \quad (\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{x_2}(S_2))) \end{array}$$

La preuve nécessite de reprendre le cas des égalités 1 et 2 (réduction des termes $x := e \parallel y := f$ et $\mathsf{skip} \parallel S$). Cette preuve est traitée dans l'annexe A (section A.3.2).

### Composition des preuves

Il est maintenant possible d'énoncer des résultats sur la composition des preuves.

**Propriété 7** *Composition des preuves*

*Soit $S_1$ une substitution définie sur l'espace de variables $x_1$ et soit $S_2$ une substitution définie sur l'espace de variables $x_2$, les variables du n-uplet $x_1 \cap x_2$ étant partagées en lecture par $S_1$ et $S_2$. On a :*

$$[S_1]P_1 \wedge [S_2]P_2 \Rightarrow [S_1 \parallel S_2](P_1 \wedge P_2)$$

*à condition que les variables des n-uplets $x_2 - x_1$ et $x_1 - x_2$ soient respectivement non libres dans $P_1$ et $P_2$.*

La preuve est donnée dans l'annexe A, section A.3.3. Si $x_1 \cap x_2 = \emptyset$ on retrouve la propriété donnée dans le B-Book (p. 309), qui s'applique au cas où $S_1$ et $S_2$ n'ont pas de variables en commun.

La propriété 7 permet, sous les hypothèses énoncées, d'inférer des propriétés d'une substitution de la forme $S_1 \parallel S_2$, à partir de propriétés de $S_1$ et $S_2$. Elle va être à la base du principe de composition des preuves d'invariant.

### Comparaisons

Comme nous l'avons dit, les opérateurs permettant de représenter l'exécution simultanée n'ont été que peu étudiés dans la théorie du raffinement. Des formes moins contraignantes de cet opérateur ont été proposées dans le cadre de B, par nous-mêmes [BPR96] et d'autres auteurs [Cha98, Dun99]. Ces extensions peuvent introduire des miracles[6] et nécessitent de modifier les égalités de la section 3.4.1. En particulier l'égalité $\mathsf{skip} \parallel S = S$ n'est plus valide, ce qui pose problème dans le cadre de B. Dans [BB98], R. Back et M. Butler proposent trois opérateurs et développent les propriétés de ces opérateurs. L'opérateur *produit* permet de composer des transformateurs de prédicats modifiant des espaces de variables disjoints, mais qui peuvent lire des variables communes. L'opérateur *fusion* permet de composer des transformateurs de prédicats lisant et modifiant des espaces de variables partagés et le *produit dérivé* permet de composer des transformateurs de prédicats travaillant sur des espaces de variables disjoints. Les propriétés de ces opérateurs sont plus générales que celles de l'opérateur || de B, mais il n'y a pas de lois algébriques permettant d'éliminer systématiquement ces opérateurs. Rappelons que cet aspect est important du point de vue opérationnel : il permet de produire des termes de preuves " simples ". Le produit dérivé correspond à la substitution multiple telle qu'elle est définie dans le B-Book. L'opérateur produit peut correspondre à l'utilisation du constructeur || à l'intérieur d'une machine. Enfin, l'utilisation de ce même constructeur pour composer des opérations peut être vue comme un cas, très resteint, de fusion.

6. c'est-à-dire des substitutions $S$ telles que $\exists x' \mathsf{prd}_x(S) \Leftrightarrow \mathit{false}$

## 3.5 Opération

Comme introduit en début de ce chapitre, une machine abstraite B est constituée d'un état interne, d'un jeu d'opérations et de propriétés invariantes sur l'état. Ces propriétés doivent être préservées par les opérations. De même le raffinement porte sur les opérations. L'entité de base d'une spécification est donc l'opération. Dans cette section, nous développons le traitement des opérations et des appels, afin de mettre en évidence les restrictions nécessaires et les propriétés associées à ces restrictions[7]. En effet la notion de paramètres va de paire avec la notion d'alias et il est nécessaire d'introduire des restrictions afin de préserver les preuves, lors des appels. Cette section est donc un complément au B-Book, qui ne détaille ni les restrictions ni les propriétés préservées par appel.

### 3.5.1 Définition d'opération

Une opération se déclare sous la forme :

$$r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$$

où $r$ et $p$ sont des n-uplets de variables. $P$ est la précondition de l'opération, qui peut porter à la fois sur les paramètres $p$ et sur les variables globales. $S$ est une substitution qui définit le corps de l'opération. Les variables $r$ désignent la liste des paramètres par résultat et $p$ les paramètres par valeur. Les contraintes sur les paramètres formels sont les suivantes :

**Restrictions 1** *Restrictions sur les définitions*

1. *les noms des paramètres sont distincts et différents de ceux des variables ;*
2. *les paramètres par valeur ne peuvent être affectés dans le corps de l'opération.*
3. *les paramètres par résultat doivent être affectés dans le corps de l'opération et leur valeur ne peut être lue avant leur affectation.*

7. Une introduction claire au problème posé par les paramètres peut être trouvée dans [Mor92].

Ces restrictions sont celles rencontrées dans les langages de programmation, si ce n'est qu'il n'y a pas de règle de masquage de noms pour les paramètres. Par exemple, les paramètres d'entrée correspondent aux paramètres `in` du langage Ada et les paramètres par résultat aux paramètres `out`. Les deux premières contraintes sont directement vérifiées par la méthode B. La dernière donnera lieu à des obligations de preuves non prouvables, si elle n'est pas respectée[8]. La condition à vérifier revient à montrer que le prédicat $\mathsf{prd}_{x,r}(S)$ ne contient pas d'occurrence libre de $r$, les variables $x$ désignant ici les variables de l'opération.

### Preuve d'invariant

Lors de la définition d'une opération dans une machine, l'obligation de preuve consiste à montrer que cette opération préserve l'invariant de cette machine.

**Condition 1** *Preuve d'invariant*

*Soit* $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$ *la définition d'une opération op. L'obligation de preuve relative à la préservation d'un invariant* $I$ *est :*

$$(INV) \quad I \wedge P \Rightarrow [S]I$$

### Preuve de raffinement

Le raffinement de substitution a été introduit dans la section 3.3.5 et est défini par la définition 4 (p. 56). Les obligations de preuve pour le raffinement d'opération sont simplifiées, car elles exploitent les preuves d'invariant. Les paramètres ne sont pas raffinés. Cette restriction est classique : elle permet d'implanter facilement le principe de substitutivité d'une spécification par son raffinement, puisque les interfaces ne sont pas modifiées. Ce choix sera repris et discuté dans le chapitre 4. Les paramètres par valeur n'étant pas affectables, il n'y a rien à vérifier lors du raffinement. Par contre le raffinement

8. soit lors de la preuve de préservation de l'invariant, soit lors de la preuve de raffinement

devra établir que les valeurs des paramètres par résultat, fournies par le raffinement, sont compatibles avec les valeurs possibles du niveau abstrait. Ceci sera établi en vérifiant l'invariant de liaison $r_1 = r_2$, où $r_1$ désigne les valeurs abstraites du paramètre résultat et $r_2$ les valeurs concrètes.

**Condition 2** *Preuve de raffinement*

*Soit $r \leftarrow op(p) \mathrel{\hat{=}} P_1 \mid S_1$ la définition abstraite d'une opération op et soit $r \leftarrow op(p) \mathrel{\hat{=}} P_2 \mid S_2$ la définition concrète de cette même opération. Si $I$ est l'invariant préservé par la définition abstraite, et si $L$ est l'invariant de liaison entre la définition abstraite et son raffinement, alors la correction du raffinement est établie par les formules suivantes :*

$$(OP_{R1}) \quad I \wedge L \wedge P_1 \Rightarrow P_2$$
$$(OP_{R2}) \quad I \wedge L \wedge P_1 \Rightarrow [[r := r_2]S_2]\neg[[r := r_1]S_1]\neg(L \wedge r_1 = r_2)$$

L'application d'une substitution sur une substitution est définie dans la section suivante (Fig. 3.1). On peut établir que ces deux conditions sont équivalentes à celles de la définition 4 (section 3.3.5), pour $J \equiv I \wedge L \wedge r_1 = r_2$ et sachant $I \wedge P_1 \Rightarrow [S_1]I$. On a donc :

$$P_1 \mid [r := r_1]S_1 \sqsubseteq_{I \wedge L \wedge r_1 = r_2} P_2 \mid [r := r_2]S_2$$

### 3.5.2 Appel d'opération

Un appel d'opération se présente sous la forme $v \leftarrow op(e)$, où $v$ est un n-uplet de variables et $e$ un n-uplet d'expressions. Les restrictions sont les suivantes :

**Restrictions 2** *Restrictions sur les appels*

*Soit $v$ le n-uplet des paramètres effectifs par résultat et $x$ les variables de l'opération.*

1. *$v$ est un n-uplet de noms de variables ;*
2. *$v$ ne contient pas de doublon ;*
3. *les variables $v$ sont disjointes des variables $x$.*

La restriction 1 interdit des appels de la forme $t(i) \leftarrow op$, qui peuvent introduire des ambiguïtés liées à l'ordre d'évaluation. La restriction 2 élimine des appels de la forme $v, v \leftarrow op$, qui peuvent aussi être difficiles à interpréter (quelle est la valeur finale de $v$?). La dernière restriction impose que les variables sur lesquelles sont définies l'opération soient disjointes des paramètres effectifs résultats. Cette restriction est, de fait, imposée dans la méthode B. Elle correspond au principe d'encapsulation des variables d'une machine: celles-ci ne peuvent être modifiées que par l'intermédiaire des opérations.

### Sémantique par substitution

La définition ci-dessous est celle donnée dans le B-Book. Elle ne s'applique que si $S$ est définie en terme de substitutions mathématiques.

**Définition 5** *Appel par substitution*

*Soit $r \leftarrow op(p) \mathrel{\hat{=}} S$ la définition d'une opération. Le résultat de l'appel $v \leftarrow op(e)$ est défini par:*

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

Nous verrons par la suite que cette définition correspond à l'appel par copie. L'application d'une substitution sur une substitution est définie dans la figure 3.1 ci-après. Il n'existe pas de règle générale définissant l'application d'une substitution sur les substitutions séquencement et itération. Ce point sera développé dans la section 3.5.4. La définition ci-dessus est applicable à toute substitution, en prenant la forme normalisée qui ne contient que des substitutions mathématiques.

**Propriété 8** *Formes normalisées*

*Soit $r \leftarrow op(p)$ une opération définie par la forme normalis ée suivante :*

$$P \mid @x', r' \cdot Q \Longrightarrow x, r := x', r'$$

*La forme normalisée de l'appel $v \leftarrow op(e)$ est :*

$$[p := e]P \mid @x', r' \cdot [p := e]Q \Longrightarrow x, v := x', r'$$

*si les restrictions sur la définition de l'opération et sur l'appel sont vérifiées.*

Les contraintes 2 et 3 des restrictions sur les appels garantissent en particulier que la substitution $x, v := x', r'$ est bien formée. Rappelons aussi qu'il est supposé que le paramètre $r$ est initialisé avant d'être réferencé. Il est donc non libre dans $Q$.

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

FIG. 3.1 – *Substitution dans les substitutions*

### 3.5.3 Préservation des preuves par appel

Du point de vue formel, la sémantique de l'appel d'opération doit permettre de réutiliser les preuves faites sur la définition des opérations. Dans le cadre de la méthode B les preuves à préserver sont celles relatives aux invariants et au raffinement (conditions 1 et 2 de la section 3.5.1).

### Préservation des preuves d'invariant

Le théorème ci-après garantit que, si la définition d'une opération préserve l'invariant $I$, alors il en est de même des appels de cette opération.

**Théorème 5** *Préservation des preuves d'invariant*

*Soit $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$, la définition d'une opération, et soit $v \leftarrow op(e)$ un appel vérifiant les restrictions sur les appels (restrictions 2, p. 66). On a :*

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

En appliquant la propriété de monotonie sur l'antécédent (propriété 3, p. 55) pour la substitution $p, r := e, v$, et comme $p$ et $r$ sont non libres dans $I$ et que $r$ est non libre dans $P$, on obtient :

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

Or la formule $[p, r := e, v][S]\, I$ est équivalente à la formule $[[p, r := e, v]S]\, I$, puisque les variables $p$ et $r$ ne sont pas libres dans $I$. Ceci permet de déduire l'obligation de preuve relative à la préservation de l'invariant pour l'appel $v \leftarrow op(e)$. D'autre part, on peut montrer que la preuve de terminaison de l'appel se réduit à établir la formule $[p := e]P$. En effet la terminaison de l'appel est définie par la formule $[p := e]P \wedge \mathsf{trm}([p, r := e, v]S)$. Or, de la preuve de préservation de l'invariant, on peut déduire $I \wedge [p := e]P \Rightarrow \mathsf{trm}([p, r := e, v]S)$, puisque $[S]R \Rightarrow \mathsf{trm}(S)$ (propriété 4, p. 56). Il suffit donc d'établir la formule $[p := e]P$.

### Préservation des preuves de raffinement

**Théorème 6** *Préservation des preuves de raffinement*

*Soit $r \leftarrow op(p) \mathrel{\hat{=}} S_1$ la définition abstraite d'une opération op et soit $r \leftarrow op(p) \mathrel{\hat{=}} S_2$ la définition concrète de cette même opération. Soit $v \leftarrow op(e)$ un appel qui vérifie les restrictions sur les appels (restrictions 2, p. 66) et tel que*

*l'expression e ne contient aucune occurrence des variables de la machine*[9] *(ni de son raffinement). On a alors:*

$$[r := r_1]S_1 \sqsubseteq_{J \wedge r_1 = r_2} [r := r_2]S_2$$
$$\Rightarrow$$
$$[v := v_1][p, r := e, v]S_1 \sqsubseteq_{J \wedge v_1 = v_2} [v := v_2][p, r := e, v_2]S_2$$

*avec J l'invariant de raffinement entre les variables de* $S_1$ *et les variables de* $S_2$*.*

Nous avons choisi d'expliciter les invariants de raffinement en renommant les variables communes entre les deux niveaux du raffinement. Puisque, lors d'un appel, le paramètres effectif résultat $v$ peut apparaître dans l'expression $e$ du paramètre effectif entrée, nous devons aussi distinguer les occurrences de $p$ et $e$. Pour la preuve nous choisissons de renommer $p$ en $p_1$ dans $S_1$ et $p$ en $p_2$ dans $S_2$ en ajoutant l'hypothèse $p_1 = p_2$.

Soit $P_1 \mid @x_1', r_1' \cdot Q_1 \Longrightarrow x_1, r_1 := x_1', r_1'$ la forme normalisée de la définition $[p, r := p_1, r_1]S_1$ et soit $P_2 \mid @x_2', r_2' \cdot Q_2 \Longrightarrow x_2, r_2 := x_2', r_2'$ la forme normalisée de la définition $[p, r := p_2, r_2]S_2$. Par la définition 4 du raffinement (p. 56) nous avons :

$(a)$ $p_1 = p_2 \wedge J \wedge P_1 \Rightarrow P_2$
$(b)$ $p_1 = p_2 \wedge J \wedge P_1 \wedge Q_2 \Rightarrow \exists x_1', r_1' \cdot (Q_1 \wedge ([x_1, x_2 := x_1', x_2']L) \wedge r_1' = r_2')$

Ces deux formules sont implicitement quantifiées universellement sur les variables $p_1$, $p_2$, $x_1$ et $x_2$. En renommant $p$ par $p_1$, la substitution $[v := v_1][p, r := e, v]S_1$ se réécrit en $[p_1, r_1 := e_1, v_1][p, r := p_1, r_1]S_1$, avec $e_1 = [v := v_1]e$, puisque $v$ est non libre dans $S_1$. La même réécriture peut être faite pour $[v := v_2][p, r := e, v]S_2$. En prenant la forme normalisée de ces appels (propriété 8, p. 67) nous devons donc établir :

$(c)$ $p_1 = p_2 \wedge J \wedge [p_1 := e_1]P_1 \Rightarrow [p_2 := e_2]P_2$
$(d)$ $p_1 = p_2 \wedge J \wedge [p_1 := e_1]P_1 \wedge [p_2 := e_2]Q_2 \Rightarrow$
$\exists x_1', r_1' \cdot ([p_1 := e_1]Q_1 \wedge ([x_1, x_2 := x_1', x_2']L) \wedge [v_1, v_2 := r_1', r_2'](v_1 = v_2))$

Les formules $c$ et $d$ découlent directement des formules $a$ et $b$ par la propriété de monotonie (propriété 3, p. 55), appliquée pour les substitutions $p_1 := [v := v_1]e$ et $p_2 := [v := v_2]e$.

9. Cette restriction correspond à la vue encapsulée, comme définie au chapitre 4.

### Préservation des postconditions

Dans le cadre de la preuve de programmes, les conditions portant sur les appels sont plus contraignantes que dans la méthode B. Elles imposent de préserver toute postcondition établie par la définition. Les postconditions peuvent porter sur les variables de l'opération, mais aussi sur les paramètres. La propriété attendue est que si on peut établir la postcondition $R$, pour les paramètres $r$ et $p$, alors on doit pouvoir établir la postcondition $[p, r := e, v]R$, où $e$ est le paramètre effectif correspondant à $p$ et $v$ le paramètre effectif correspondant à $r$. La préservation des postconditions n'est actuellement pas exigée, ni garantie, dans la méthode B. Par exemple, l'opération *op* définie par :

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

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

L'ajout de postconditions, vérifiées par les opérations, est une extension envisagée dans la méthode B. En effet, cette possibilité permet de prouver des propriétés locales à une opération, et de réutiliser ces propriétés pour les appels. Nous proposons ci-dessous des restrictions assurant la préservation des postconditions. Ces restrictions sont similaires à de celles proposées dans [BB89].

**Restrictions 3** *Restrictions pour la préservation des postconditions*

*En plus des restrictions 2 de la section 3.5.2, on a :*

1. *les paramètres effectifs par résultat n'apparaissent pas dans la postcondition ;*

2. *les paramètres effectifs par valeur ne contiennent aucune occurrence des variables de l'opération ni des paramètres effectifs par résultat.*

Pour un appel de la forme $v \leftarrow op(e)$, l'opération $op$ étant définie sur les variables $x$, et pour une postcondition $R$, ceci signifie que $v$ n'apparaît pas dans $R$ et que $e$ ne contient aucune occurrence de $x$ ou de $v$. La condition 1 est nécessairement vérifiée en B puisque l'opération ne peut pas dépendre des paramètres effectifs résultats. La condition 2 garantit que la valeur finale d'un paramètre effectif par valeur est identique à sa valeur initiale, puisqu'aucune des variables apparaissant dans l'expression ne peut être modifiée. Le théorème est alors le suivant :

**Théorème 7** *Préservation des postconditions*

*Soit une définition de la forme* $r \leftarrow op(p) \hat{=} P \mid S$ *et soit l'appel* $v \leftarrow op(e)$. *La propriété est :*

$$(\forall r, p \ \ (P \Rightarrow [S]R)) \Rightarrow ([p := e]P \Rightarrow [[p, r := e, v]S]([p, r := e, v]R))$$

*si les restrictions 3 sont respectées.*

En appliquant la propriété de monotonie sur l'antécédent, pour la substitution $p, r := e, v$, on obtient $[p, r := e, v]P \Rightarrow [p, r := e, v][S]R$. Il faut donc établir l'implication suivante :

$$[p, r := e, v][S]R \Rightarrow [[p, r := e, v]S]([p, r := e, v]R)$$

Si on prend la forme normalisée de la définition et de l'appel (voir propriété 8, section 3.5.2), il faut montrer :

$$[p, r := e, v][x, r := x', r']R \Rightarrow [x, v := x', r'][p, r := e, v]R$$

L'antécédent se simplifie en $[p := e][x, r := x', r']R$ car $r$, $x'$ et $r'$ n'ont pas de variables en commun. Le conséquent se simplifie en $[x := x'][p, r := e, r']R$ puisque $v$ est non libre dans $R$ et dans $e$ (points 2 et 3 des restrictions 3). Il faut donc établir :

$$[p := e][x, r := x', r']R \Rightarrow [x := x'][p, r := e, r']R$$

Ceci nécessite de pouvoir permuter les substitutions $p := e$ et $x := x'$, ce qui n'est possible que si l'expression $e$ ne contient pas d'occurrence de $x$.

Cette extension introduit une limitation sur la forme des appels admis (point 2 des restrictions ci-avant). Néanmoins il est toujours possible de remplacer un appel de la forme $v \leftarrow op(e)$, avec $e$ ne vérifiant pas la condition 2, par :

```
VAR l IN
    l := e;
    v ← op(l)
END
```

où $l$ est une variable locale, non libre dans $op$.

### 3.5.4 Implantation des appels

Dans la section 3.5.2, l'appel d'opération est défini en terme de substitution des paramètres formels par les paramètres effectifs (définition 5, p. 67). Cette définition n'est pas nécessairement très adaptée en pratique, car elle nécessite de ramener la définition de l'opération à des opérateurs primitifs mathématiques. Cette définition ne convient donc pas pour générer du code. En effet un appel doit pouvoir se traduire en terme du code associé à la définition de l'opération, par exemple par un appel de procédure du langage de programmation, après ajustement des paramètres. Le mécanisme proposé est donc suffisant lorsqu'on ne s'intéresse qu'aux preuves, mais est insuffisant pour l'implémentation. Dans la suite de cette section, nous nous intéressons à une expression plus implémentatoire de l'appel d'opération. Ces développements ont en particulier été faits dans le cadre du projet RNTL BOM, dans l'objectif de construire des traducteurs adaptés aux contraintes carte à puce.

Les deux modes de passage de paramètres que nous allons considérer sont *le passage par copie* et *le passage par référence*. Dans la pratique, le passage par référence est plus efficace lorsque les paramètres ont des types non atomiques. En effet, il permet d'éviter une copie, qui peut être coûteuse en place. Nous allons donner une définition opérationnelle de ces modes de passage, qui va fournir un moyen effectif de construire le texte d'un appel, en

terme de la définition de l'opération. Ces définitions permettront d'établir les conditions qui vont assurer l'équivalence d'un mode de passage de paramètre avec la définition formelle donnée dans la section 3.5.2. Elles donneront aussi un moyen effectif pour l'expansion des appels, qui est une technique classique d'optimisation de code. En effet, l'expansion consiste à remplacer un appel par les instructions correspondant à cet appel, ce qui évite la gestion mémoire liée aux appels.

### Sémantique par copie

Le passage par copie correspond par exemple au mode de passage adopté en C. L'opération travaille sur des copies locales des paramètres.

**Définition 6** *Appel par copie*

*Soit* $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$ *la définition d'une opération op. Le passage par copie d'un appel de la forme* $v \leftarrow op(e)$ *est défini par :*

$$([p := e]P) \mid \text{VAR } p, r \text{ IN } p := e \ ; \ S \ ; \ v := r \text{ END}$$

La notation VAR $z$ IN $S$ END correspond à la déclaration de variables locales. Sa définition mathématique est $@ z \cdot S$.

On peut montrer que le passage par copie est équivalent à la définition des appels par substitution (définition 5, p. 67), sous l'hypothèse que les restrictions sur la définition des opérations et les restrictions sur l'appel d'opération soient vérifiées (sections 3.5.1 et 3.5.2). Cette équivalence est prouvée dans [Pot02] et est donnée dans l'annexe B (sections B.2.2 et B.2.3).

### Passage par référence

Le passage par référence, désigné aussi par passage par variable, consiste à travailler directement sur les paramètres effectifs, sans copie. Il n'a de sens que pour des paramètres variables, qui désignent des zones mémoire adressables.

Le passage par référence peut être modélisé par le remplacement textuel[10] des paramètres formels par les paramètres effectifs, en prenant quelques précautions. Dans toute sa généralité, le remplacement textuel correspond au passage par nom, dans lequel les paramètres effectifs sont évalués à chaque utilisation. Ce mécanisme est rarement supporté par les langages de programmation, car trop complexe à implanter et à utiliser. Dans le passage par référence, l'adresse des paramètres effectifs est évaluée une fois pour toute, avant appel. La modélisation du passage par référence par remplacement textuel n'est donc correcte que si la réévaluation des paramètres est sans effet sur les adresses[11].

**Définition 7** *Appel par référence*

*Soit* $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$ *la définition d'une opération op. Le passage par référence d'un appel de la forme* $v \leftarrow op(e)$ *est obtenu par remplacement textuel des paramètres formels, par les paramètres effectifs, sous l'hypothèse que les adresses désignées par* $e$ *et* $v$ *ne soient pas modifiées par* $S$*.*

La restriction impose que, si les termes $e$ et $v$ ne se réduisent pas à une variable, les variables apparaissant dans ces termes ne sont pas affectées dans le corps de S. Dans le cas de la méthode B, le cas ne se pose pas pour le paramètre effectif $v$, qui est nécessairement une liste de noms de variable (voir les restrictions sur les appels, p. 66). Pour le terme $e$, ceci impose qu'il ne contienne pas d'occurrence d'une variable de l'opération ni d'une variable de la liste $v$.

Le passage par référence peut introduire des alias, c'est-à-dire confondre des noms initialement différents. Ceci résulte en une perte de la monotonie, c'est-à-dire que les propriétés ne sont généralement pas préservées par les

10. Nous distinguons ici sciemment substitution et remplacement textuel.

11. Une modélisation plus fine du passage par référence nécessiterait de pouvoir manipuler explicitement les adresses.

appels. Donnons un exemple[12] :

```
MACHINE Exemple
VARIABLES x
INVARIANT pair(x)
INITIALISATION x := 0
OPERATIONS
    r ← op(p)≙
        PRE  pair(p)
        THEN
            x := x + 1;
            x := x + p + 1
        END
END
```

L'opération *op* préserve l'invariant $pair(x)$, où *pair* est un prédicat vrai si $x$ est un entier pair. Considérons l'appel $v \leftarrow op(x)$, qui a pour effet de confondre la variable $x$ et le paramètre par valeur $p$. Sous l'hypothèse de l'invariant, la précondition de *op* peut bien être établie, pour le paramètre effectif $x$. La sémantique du passage par référence donne le code $x := x + 1 \ ; \ x := x + x + 1$ qui a pour effet d'affecter à $x$ la valeur $(x + 1) + (x + 1) + 1$, qui ne peut être paire. L'invariant n'est donc plus préservé.

Plus généralement, le passage par référence (ou le remplacement textuel) ne préserve pas l'égalité. Par exemple les deux substitutions $S_1$ et $S_2$ suivantes sont égales :

$$\begin{array}{lcl} S_1 & \hat{=} & x := 1 \ ; \ y := 2 \\ S_2 & \hat{=} & y := 2 \ ; \ x := 1 \end{array}$$

Par contre les substitutions $[x := y]S_1$ et $[x := y]S_2$ se réduisent respectivement à $y := 2$ et $y := 1$, et sont donc différentes. C'est pour cela que

12. Cet exemple ne respecte pas les restrictions de la méthode B, qui interdisent l'opérateur de séquencement dans les machines. Le même exemple pourrait être donné sous la forme d'un raffinement.

nous avons distinguer explicitement le remplacement et la substitution, cette dernière devant vérifier certaines propriétés formelles. Sous certaines hypothèses, il est possible de montrer que le passage par référence est équivalent au passage par copie, et donc qu'il préserve les propriétés. Ces hypothèses correspondent aux cas où l'appel n'introduit pas d'alias. Les restrictions sont les suivantes :

**Restrictions 4** *Restrictions pour le passage par référence*

1. *les paramètres effectifs résultats n'apparaissent pas dans $S$ ;*
2. *les paramètres effectifs par valeur ne contiennent aucune occurrence des paramètres effectifs résultats, ni des variables réferencées dans l'opération.*

Ces restrictions sont similaires à celles de la section 3.5.3. Dans les deux cas, la propriété à préserver est que les paramètres effectifs par valeur ne soient pas affectés par l'exécution du corps de l'opération, et que le remplacement de $r$ par $v$ ne crée pas d'alias. Sous ces restrictions, il est alors possible de définir formellement le remplacement textuel par la substitution et d'établir l'équivalence entre le passage par référence et la définition formelle de l'appel (définition 5, section 3.5.2).

Dans le cas des substitutions généralisées définies uniquement avec des substitutions mathématiques, les égalités données dans la figure 3.1 (section 3.5.2, p. 68) fournissent un moyen de propager les substitutions dans les sous-termes. Ces règles peuvent être étendues aux substitutions syntaxiques, définies à partir des substitutions primitives mathématiques. Par exemple on a :

$$[x := E]\ (\text{IF } P \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END})$$
$$=$$
$$\text{IF } [x := E]P \text{ THEN } [x := E]S_1 \text{ ELSE } [x := E]S_2 \text{ END}$$

Le cas qui pose problème est le séquencement, puisqu'il permet d'introduire des modifications intermédiaires. On n'a généralement pas $[x :=$

$E](S_1 \ ; \ S_2) = ([x := E]S_1) \ ; \ ([x := E]S_2)$, comme l'illustre le contre-exemple donné précédemment [13]. Par contre, avec les restrictions précédentes nous pouvons établir l'égalité :

$$[p, r := e, v](S_1 \ ; \ S_2) \quad = \quad [p, r := e, v]S_1 \ ; \ [p, r := e, v]S_2$$

La correction de cette règle, pour les restrictions données ci-dessus, est développée dans [Pot02] (annexe B, section B.3.2 et B.3.3). Elle repose sur le fait que l'équivalence suivante est valide :

$$[p, r := e, v][x, r := x', r']R \Leftrightarrow [x, v := x', r][p, r := e, v']R$$

avec $v$ non libre dans $R$. Cette condition est la même que celle introduite par la préservation des postconditions (section 3.5.3), elle est donc établie par les mêmes restrictions.

### 3.5.5 En résumé

Dans ce chapitre nous avons introduit le formalisme des substitutions généralisées ainsi que le calcul de plus faible précondition associé. De plus nous avons énoncé les résultats principaux sur lesquels sera basée la construction incrémentale de spécification et de développement : la notion d'opération et d'appel ainsi que l'opérateur de composition parallèle, qui permettra de combiner les spécifications. La composition des développements et des preuves est l'objet du chapitre suivant.

Concrètement nous avons complété les résultats du B-Book, dans le cas du constructeur de substitution multiple, afin de donner des règles opérationnelles d'élimination de ce constructeur (section 3.4.1) et nous avons étendu les propriétés de cet opérateur afin de couvrir les configurations admises dans les outils. Dans le cas des appels d'opérations, nous avons établi certains résultats. Dans un premier temps nous avons mis en évidence les restrictions nécessaires à la préservation des preuves d'invariant et de raffinement (section 3.5.3). Nous avons aussi établi la correspondance entre la définition formelle

13. La règle `SUB27` donnée dans le B-Book (p. 756) n'est donc pas toujours valide.

des appels par substitution et le mécanisme opérationnel du passage de paramètre par copie (section 3.5.4). Cette dernière règle décrit le passage de paramètre directement en terme de la substitution qui définit l'opération[14]. Finalement, nous avons proposé des restrictions sur les appels, permettant de préserver les postconditions (section 3.5.3) et d'implanter le passage de paramètre par référence (section 3.5.4). Cette dernière construction est utile pour optimiser les implémentations, tout en restant dans le cadre formel de la méthode B.

14. Par exemple si on veut autoriser le séquencement dans les machines, ce qui n'est actuellement pas admis, cette règle permet de traiter simplement les appels en présence de séquencement.

# Chapitre 4

# Composant et composition

Dans ce chapitre nous décrivons les principes nécessaires à la maîtrise de la complexité des développements formels, c'est-à-dire l'abstraction (ou son inverse le raffinement) et la décomposition. Nous étudions les différentes manières de les mettre en œuvre dans une approche incrémentale du développement, et leur influence sur le processus de preuve. Dans une seconde partie nous présentons l'approche adoptée dans la méthode B. Cette partie est basée sur les travaux que nous avons développés sur ce sujet [PR98, BP00, Pot03]. Nous présentons les choix méthodologiques faits, leur mise en œuvre et les théorèmes justifiant la correction de l'approche. Les résultats plus complets sont disponibles dans les articles cités.

## 4.1 Développement incrémental

Dans cette section nous nous intéressons au développement modulaire de spécification et de programme, et principalement au principe de compositionnalité et à sa mise en œuvre. Cette présentation est en partie inspirée des différents travaux autour de TLA, la logique temporelle des actions de L. Lamport, dans laquelle les processus de décomposition et de composition ont été étudiés de manière fine, et en particulier leur influence sur le processus de validation par la preuve. Dans la suite nous dénoterons par composant une spécification, un programme ou un système.

### 4.1.1 Composition horizontale et composition verticale

La maîtrise de la complexité des spécifications, des développements et des preuves, est nécessaire pour envisager le traitement d'applications de taille réelle. Il existe principalement deux processus mentaux permettant cette maîtrise : la réduction à des problèmes plus petits, et l'abstraction qui permet de ne s'intéresser qu'à une approximation d'un problème [Boo94]. Le premier outil est souvent référencé sous le terme de *composition horizontale* et permet de construire un composant en combinant d'autres composants. La composition horizontale offre un processus de simplification si le composant est séparable, c'est-à-dire s'il peut effectivement être exprimé par une combinaison de composants plus simples. De plus il devient très efficace s'il peut être combiné avec le processus d'abstraction. La mise en œuvre de ce dernier est basé sur le principe de *separation of concerns* entre l'utilisation d'un composant et son implémentation. On peut ainsi produire de nouveaux composants sur la base de leurs spécifications, sans connaître leurs implémentations, et on peut implémenter un composant qui satisfait sa spécification, sans aucune connaissance de ses utilisations [Lam83]. La *composition verticale* est le processus qui permet de considérer différents niveaux d'abstraction d'un composant, le niveau le plus précis décrivant le comportement attendu. Cette approche correspond au processus d'abstraction ou de raffinement, suivant que l'on part du composant final qu'on abstrait, ou d'une spécification qu'on implémente. Ces différents niveaux doivent pouvoir se composer, pour mettre en relation différents niveaux d'abstraction. Ceci correspond par exemple à l'approche de raffinement par étapes (stepwise refinement) initialement proposée par N. Wirth [Wir71].

La puissance d'une approche dépend du fait de pouvoir combiner ces deux types de composition, et des propriétés de compositionnalité associées. Ces propriétés dépendent bien sûr du niveau d'exigence, c'est-à-dire des propriétés sémantiques visées. Par exemple un langage de programmation, tel le langage Ada, offre des primitives de structuration horizontale et verticale. La notion de paquetage permet de séparer la spécification du corps, et les paquetages peuvent être composés à l'aide de la clause `WITH`. Il est alors possible de compiler séparément des paquetages en ne considérant que les spécifications, et il est possible de construire le code final par assemblage des codes objet de chaque corps de paquetage. Dans la suite nous nous in-

téressons à des propriétés plus sémantiques, et à la compositionnalité des raisonnements qui leur sont associés.

### 4.1.2 Composition et preuve

La composition horizontale et la composition verticale offrent donc deux outils permettant de construire des systèmes de manière incrémentale. Le raisonnement sur ces développements dépend de la manière dont ils sont abordés. On distingue classiquement deux approches : l'approche *top-down* et l'approche *bottom-up*. Ces approches correspondent à deux formes différentes de composition qui, par là-même, donnent lieu à deux formes différentes de raisonnement. Bien que ces approches ont été longuement étudiées du point de vue génie logiciel, peu de travaux ont été menés sur leur implication vis-à vis du raisonnement formel. La présentation ci-après est largement inspirée des travaux de M. Abadi et L. Lamport [AL95].

Dans le cadre d'une approche top-down, un composant est décomposé en parties plus petites, appelées sous-composants. Dans cette approche, le contexte d'un sous-composant est connu. On peut donc raisonner localement sur un sous-composant, en incluant les propriétés nécessaires de son environnement. Cette approche peut-être illustrée dans le cadre de la preuve de programme. Par exemple la preuve de l'assertion $true\{x := 2\ ;\ x := x-1\}x \geq 0$ se décompose en la preuve des deux assertions $x-1 \geq 0\{x := x-1\}x \geq 0$ et $true\{x := 2\}x - 1 \geq 0$. La décomposition est introduite par l'instruction de séquencement et la seconde assertion a été calculée en fonction du contexte (la postcondition attendue est la précondition de la première assertion). Le principe de la preuve par décomposition est illustré dans toute sa généralité par le théorème de décomposition de M. Abadi et L. Lamport [AL95], que nous donnons ci-dessous. Les formules $S_i$ et $R_i$ décrivent des spécifications (les détails de leur description ne sont pas nécessaires ici). Le raffinement s'exprime à l'aide de l'implication logique et la composition s'exprime par la conjonction des spécifications. La règle la plus simple est la suivante :

$$\frac{\begin{array}{c} R_1 \Rightarrow S_1 \\ R_2 \Rightarrow S_2 \end{array}}{R_1 \wedge R_2 \Rightarrow S_1 \wedge S_2}$$

Cette règle impose que les raffinements $R_1$ et $R_2$ de $S_1$ et $S_2$ puissent être établis indépendamment de tout contexte, c'est-à-dire sans information sur la manière dont $S_1$ et $S_2$ collaborent. Losqu'on s'intéresse à des aspects comportementaux, ceci est rarement le cas. Par exemple pour construire un contrôleur on utilise généralement des propriétés sur le fonctionnement de l'environnement. Le théorème de décomposition permet de prouver les raffinements en utilisant des hypothèses sur la composition, il peut s'exprimer par :

$$\frac{\begin{array}{c} S_1 \wedge S_2 \Rightarrow E_1 \\ S_1 \wedge S_2 \Rightarrow E_2 \\ E_1 \wedge R_1 \Rightarrow S_1 \\ E_2 \wedge R_2 \Rightarrow S_2 \end{array}}{R_1 \wedge R_2 \Rightarrow S_1 \wedge S_2}$$

Cette règle permet de raffiner la spécification $S_i$ en $R_i$, sous l'hypothèse $E_i$ qui décrit des propriétés de la composition $S_1 \wedge S_2$ (par exemple des propriétés sur les variables partagées). Cette règle n'est pas valide dans tous les cas. Par exemple si toutes les formules sont à $false$, sauf les $R_i$, alors toutes les prémisses sont vraies et la conclusion est fausse. On peut établir la correction de cette règle sous certaines conditions, par exemple si on ne s'intéresse qu'à des propriétés de sûreté et si $S_1$ et $S_2$ modifient des ensembles disjoints de variables, de même pour $E_1$ et $E_2$. Des versions plus complexes de ce théorème permettent de traiter d'autres cas.

Dans le cadre d'une approche bottom-up, les composants sont spécifiés indépendamment, et peuvent être réutilisables dans différents contextes. Il est donc possible de raisonner sur ces composants, hors de leur contexte d'utilisation. Un composant réutilisable peut néanmoins supposer certaines propriétés sur son environnement. Il est alors spécifié sous la forme de spécifications *rely/guarantee* [Jon81]. La condition *rely* décrit les hypothèses de comportement de l'environnement et la condition *guarantee* est le contrat respecté par le comportement du composant. On parle alors de composant ouvert. Les opérateurs de composition permettent d'assembler de tels composants, sous certaines conditions. Lorsque la composition est possible, le principe de compositionnalité doit permettre de plonger les raisonnements locaux dans le composant global. Ce principe peut demander une règle d'adaptation [Zwi89]. Pour reprendre l'exemple introduit dans le cas d'une approche par décomposition, l'approche par composition correspond à la preuve de pro-

grammes en présence de procédures ou de fonctions. Supposons que nous ayons montré de manière indépendante les deux formules $true\{x := 2\}x = 2$ et $x - 1 \geq 0\{x := x - 1\}x \geq 0$ (ces deux affectations pouvant correspondre au corps de deux procédures). Pour composer ces deux preuves nous devons utiliser la règle d'adaptation $x = 2 \Rightarrow x - 1 \geq 0$, qui permet de faire le lien entre la postcondition de la première assertion et la précondition de la seconde. Le théorème de composition de M. Abadi et L. Lamport s'exprime par la déduction :

$$\frac{E \wedge S_1 \wedge S_2 \Rightarrow E_1 \wedge E_2 \wedge S}{(E_1 \rightarrow S_1) \wedge (E_2 \rightarrow S_2) \Rightarrow (E \rightarrow S)}$$

où la notation $E \rightarrow S$ permet de spécifier un composant ouvert. Intuitivement cette spécification décrit le fait que le composant établit la propriété $S$, si l'environnement satisfait la propriété $E$. La flèche a un sens plus fort que l'implication, qui pourrait correspondre à l'interprétation attendue. En effet, l'implication $E \Rightarrow S$ est valide dès que l'environnement ne vérifie pas les propriétés attendues, la flèche va introduire un contrôle plus fin de ces situations. Comme dans le cas précédent, cette déduction n'est correcte sous cette forme que pour les propriétés de sûreté, et sous certaines hypothèses de partitionnement des variables.

La différence essentielle entre ces deux approches, en dehors de l'aspect méthodologique, est liée à l'écriture des composants. Dans l'approche top-down on construit des sous-composants, qui sont donc relatifs à un contexte englobant. Ceci nécessite de définir deux notions, les composants et les sous-composants, ainsi que l'opération de décomposition. Dans l'approche bottom-up il n'y a qu'une seule notion de composant, un composant fermé étant un cas particulier de composant ouvert (les hypothèses sur l'environnement sont réduites à $true$). Remarquons que dans les deux cas, au moins au niveau du raisonnement, on peut être amené à mettre en jeu un processus permettant d'abstraire le contexte, afin de simplifier le processus de vérification de propriétés. Dans l'approche top-down on dispose, par défaut, de toutes les informations du contexte. Le processus de preuve peut nécessiter de trouver des abstractions (les hypothèses $E_i$ du théorème de décomposition) afin d'éliminer les détails non utiles à la preuve. *L'abstraction porte donc sur le processus de preuve.* Par contre, dans l'approche bottom-up, par défaut, on n'a aucune contrainte sur l'environnement. Le processus d'abstraction consiste à trouver

les hypothèses adéquates sur l'environnement de chaque composant (les $E_i$ du théorème de composition) qui vont permettre de valider et de raffiner les composants de manière indépendante. *L'abstraction porte ici sur le processus de conception.* Par défaut, dans le premier cas on ne simplifie pas le processus de vérification car on dispose de trop d'hypothèses alors que, dans le second cas, on peut ne pas pouvoir valider les propriétés attendues, aussi par défaut d'hypothèses.

Dans le cadre du développement formel, on distingue habituellement la vérification des systèmes séquentiels de la vérifications des systèmes concurrents ou distribuées, arguant du fait que les problèmes ne sont pas les mêmes. En effet, dans le cas des systèmes concurrents, la difficulté provient du fait qu'il faut prendre en compte, et formaliser, l'interaction d'un composant avec son environnement. Pour un système séquentiel la notion de composant est assimilé à la notion de modules, et ceux-ci peuvent généralement être exprimés suivant le principe de séparation : ils peuvent être spécifiés indépendamment des autres modules. Les préconditions s'avèrent en général un mécanisme suffisant pour exprimer les hypothèses d'utilisation. Néanmoins, lorsqu'on s'intéresse à des développements séquentiels basés sur l'assemblage de modules, les problématiques se rejoignent en partie. Un module devient un composant qui va être plongé dans un environnement (l'architecture) et qui peut entretenir des interactions avec les autres modules. C'est le cas par exemple lorsque certains modules sont partagés. Les problèmes sont alors similaires à ceux rencontrés dans les systèmes concurrents avec partage de variables, et vont nécessiter des techniques analogues à celle du test de non-interférence introduit par S. Owicki et D. Gries pour la preuve de programmes concurrents [OG76]. Par exemple, dans [BB99], M. Büchi et R. Back utilisent une forme particulière de spécifications rely/guarantee pour exprimer des contraintes sur les partages possibles, dans le but de proposer des extensions à la méthode B. Dans le cadre de la théorie du raffinement, M. Butler et R. Back proposent des opérateurs de composition de transformateurs de prédicats qui trouvent leur application aussi bien pour modéliser des exécutions parallèles que pour plonger des programmes dans des systèmes plus grands [BB98]. Ceci montre la similitude des raisonnements sous-jacents. Nous avons fait le même constat au cours de nos travaux sur l'interaction entre la composition et le raffinement dans la méthode B.

### 4.1.3 Raffinement et compositionnalité

Dans cette section nous décrivons les principes sur lesquels repose la compositionnalité du raffinement. Le point de vue classiquement adopté est le suivant [KS99] : *in specification, the purpose of components is to provide modularity, where component refinements are composable into a refined specification of the total system.* Ce point de vue reprend le principe de la modularité dans les implémentations, qui peut s'énoncer par : *in implementation, the purpose of components is to provide modularity, where component implementations are composable into an implementation of the total system.* Ce principe signifie en particulier que les propriétés des raffinements restent valides[1], une fois ceux-ci composés, et que le raffinement de chaque composant fournit bien un raffinement de la composition globale. Le raffinement doit donc, dans une certaine mesure, être substituable à sa spécification. Suivant ce point de vue, la compositionnalité du raffinement garantit qu'il est possible de construire un raffinement de la spécification structurée. La règle peut s'énoncer en :

$$\frac{C_i \sqsubseteq R_i}{C_1 \oplus \ldots C_i \oplus \ldots \oplus C_n \sqsubseteq C_1 \oplus \ldots R_i \oplus \ldots \oplus C_n}$$

où $\oplus$ désigne un opérateur de composition et $\sqsubseteq$ la relation de raffinement. $C \sqsubseteq R$ signifie que le composant $R$ est un raffinement du composant $C$.

Le principe de substitutivité nécessite de préciser l'aspect observable des composants. Ceci correspond généralement à la notion *d'interface*, dans laquelle on précise les variables accessibles, le jeu d'opérations, les communications autorisées ... Un composant peut généralement être considéré selon plusieurs vues. Par exemple dans [Mey87], B. Meyer introduit deux types de vue : une *vue ouverte*, qui donne accès à toutes les informations de la spécification et une *vue fermée*, qui restreint l'information accessible aux aspects observables. La première vue est adaptée à la phase de spécification. La vue fermée encapsule le composant et permet la substitutivité par le processus de raffinement. Elle est donc adaptée à la phase de codage. Par exemple une machine à état rend visible tout son texte afin de pouvoir raisonner sur la spécification. Au niveau des implémentations elle n'offre à l'extérieur que

1. On fait référence ici aux propriétés de sûreté.

son jeu d'opérations, afin d'encapsuler son état. Son interface se réduit au jeu d'opérations et l'aspect observable aux valeurs possibles des paramètres résultat, en fonction des paramètres entrée. L'encapsulation va ainsi permettre de raffiner l'état interne d'un composant, de manière transparente pour les utilisateurs [Hoa84, Jon86]. D'autres types de spécification vont offrir d'autres notions d'observabilité (traces d'événements avec observation des failure/divergence, traces d'état avec propriétés temporelles ...).

Une relation de raffinement se définit donc en terme d'une relation entre l'observabilité de la spécification abstraite et l'observabilité de la spécification concrète. Par exemple le raffinement d'une machine à état impose que les opérations concrètes aient une précondition plus faible que la précondition abstraite et que la postcondition soit plus forte. Dans CSP par exemple le raffinement doit préserver les failure/divergence : le système concret ne doit pas s'arrêter ou boucler plus souvent que le système abstrait. Le raffinement nécessite donc de pouvoir comparer des observations. Le choix le plus simple est d'imposer que le raffinement ne modifie pas les objets dont dépend l'observabilité. Par exemple le raffinement ne permet généralement pas de modifier la représentation des paramètres d'opérations ou des variables externes, puisqu'ils sont observables. On peut donc comparer simplement les aspects observables de deux composants. Des études en cours visent à assouplir ces contraintes, en permettant le *raffinement d'interface*. Ces extensions peuvent prendre différentes formes, changement de la représentation des données observables, raffinement de la granularité d'une interaction ..., et nécessitent de définir des critères de comparaison des aspects observables. Le principe de compositionnalité doit donc aussi être étendu afin de traiter de manière homogène les différents composants concernés par une même interface.

Finalement, la validité du principe de compositionnalité des raffinements est souvent conditionnée par la manière de construire les composants et de les composer. En effet, dans le cas général, les propriétés de compositionnalité des composants informatiques sont pauvres et ne dépassent pas le cadre de la compilation. En effet, le comportement global d'un système résulte du comportement des composants qui le constituent et de l'interaction entre ces composants, ceux-ci étant généralement dépendants les uns des autres. De plus ces composants s'exécutent en utilisant de la mémoire partagée, ce qui

rend difficile le raisonnement modulaire. Ceci signifie que la compositionnalité est toujours considérée pour des langages restreints (par exemple libres de toutes possibilités d'alias) et impose une discipline stricte de développement des composants. Par exemple, une restriction courante est de pouvoir partitionner l'état d'un système. Ceci permet d'attribuer la responsabilité d'une variable à un seul composant et ainsi de localiser les modifications de cette variable. Nous avons vu que cette dernière restriction permettait par exemple de simplifier les raisonnements, comme dans les théorèmes de décomposition ou de composition proposés dans [AL95]. Le principe de compositionnalité est donc soumis à un certain nombre de contraintes. La règle donnée en début de cette section s'accompagne donc généralement de restrictions, qui portent sur la structure des spécifications, c'est-à-dire sur les assemblages autorisés.

### 4.1.4 Développement incrémental et raffinement

La compositionnalité du raffinement est une propriété particulièrement intéressante car elle permet de découper une spécification en plusieurs parties (composition horizontale) et de détailler par étape les comportements attendus de chacune de ces parties de manière autonome (composition verticale). La compositionnalité garantit que cette approche est correcte, c'est-à-dire qu'en ne se focalisant, par exemple, que sur un développement vertical, on participe à la construction du tout. Elle rend donc transparente au développeur la structure, et donc la complexité globale du raisonnement [2]. Néanmoins, et on s'en doute, ce principe est si beau ... qu'il n'est pas toujours réaliste. En plus des limitations théoriques évoquées précédemment, une autre limitation est liée à la pertinence même du principe de compositionnalité des raffinements. En effet, les outils de décomposition horizontale et verticale peuvent être utilisés à différents niveaux du cycle de développement. En phase de spécification ou de conception globale ils correspondent à des outils conceptuels, permettant de maîtriser la complexité. En phase de conception détaillée ou de codage la structuration est significative et fait intrinsèquement partie de la spécification. Ceci signifie que les exigences vis-à-vis du raffinement peuvent varier, suivant les phases du développement.

2. Ceci dans l'idéal!

Lors de la phase de codage les composants sont associés aux modules du langage de programmation et le principe de séparabilité est effectif : les aspects observables d'un composant correspondent à la notion d'interface des modules et le raffinement, basé sur la préservation des aspects observables, est bien adapté. De plus, à ce stade du développement, l'architecture est fixée et le but est bien de développer individuellement chaque composant. En phase de conception, l'objectif est de construire l'architecture ainsi que les interfaces. L'utilisation du raffinement pour cette étape, introduction pas à pas d'une structuration modulaire, correspond aussi au principe de compositionnalité du raffinement. On raffine un composant par un composant structuré, en fonction de l'architecture attendue. Par contre le raffinement peut nécessiter de préserver des propriétés propres aux architectures. Par exemple dans [MQR95], le point de vue adopté est qu'une architecture décrit non seulement les composants, les interfaces et les connections entre composants, mais aussi les relations qui n'existent pas entre les composants. Le raffinement doit alors prendre en compte le fait qu'une architecture concrète préserve les propriétés de l'architecture abstraite, mais aussi qu'elle ne permette pas d'inférer de nouvelles propriétés de l'architecture abstraite. Le raffinement doit donc être conservatif. Dans [Gar96], d'autres formes de propriétés propres aux architectures sont proposées. En phase de conception, la notion de raffinement doit donc être adaptée en fonction de l'objectif, construire une architecture ayant des propriétés requises.

En phase de spécification, la compositionnalité telle que décrite précédemment n'est généralement pas vraiment utilisable. En effet, à ce stade, la construction par assemblage répond plus à des préoccupations logiques qu'à des préoccupations liées à l'implémentabilité. Il s'ensuit que ni la structuration, ni les interfaces entre les composants ne sont nécessairement significatives pour la suite du développement. On peut par exemple être amené à restructurer les spécifications. En effet les objectifs de cette étape sont principalement relatifs à la prise en compte des besoins et à la validation de propriétés formelles. Ces dernières portent généralement, à ce niveau, sur la spécification en son entier. La composition verticale, et son principe de compositionnalité, ne sont donc pas nécessairement des outils intéressants, puisqu'ils introduisent une notion forte de localité des raisonnements. Les besoins qui se font actuellement sentir sont plutôt relatifs à la notion même de raffinement. Par exemple, certaines formes de raffinement, comme la préser-

vation des interfaces ou du non-determinisme externe, apparaissent comme trop rigides. En effet le raffinement joue, à ce stade, un rôle différent, il ne vise pas à établir la correction d'une implémentation mais à définir un processus de traçabilité des besoins. Les questions qui se posent sont alors quelles formes de raffinement sont adaptées, et comment les appliquer pour tracer à la fois la prise en compte des besoins et les propriétés formelles attendues[3]. Il y a eu, jusqu'à présent, peu de développements conséquents de spécifications formelles. Le raffinement semble néanmoins apporter un soutien méthodologique fort, dans le cas de spécifications un tant soit peu complexes. Un certain nombre de challenges ont été posés (The boiler case study [ABL96], le contrôle d'accès[4], the Cash Point Problem of FM'99, the Wildfire Verification Challenge Problem[5], parmi d'autres). La nécessité de définir et manipuler différentes formes de raffinement est préconisée dans l'approche des Abstract State Machines (ASM) et a été illustrée par différentes études de cas [Boe99].

Dans l'idéal, une approche doit donc offrir différentes formes de composition, et de raffinement, suivant les objectifs visés. Elle doit de plus permettre de raisonner sur la structure globale d'un développement. En effet, la composition horizontale et la composition verticale offrent des moyens de maîtriser la complexité, mais pas de la faire disparaître[6]. Ces moyens permettent de se focaliser sur des sous-parties d'un développement et de raisonner localement sur ces parties et s'accompagnent d'une " gymnastique " intellectuelle qui permet de réinterpréter des raisonnements locaux dans le contexte global. La difficulté de ce raisonnement dépend des propriétés considérées. Par exemple si on s'intéresse à des propriétés structurelles il est facile d'étendre un raisonnement local au système en son entier. Par contre si on considère par exemple des propriétés invariantes sur les données qui, de plus, peuvent être abstraites ou raffinées, il devient difficile de réinterpréter des raisonnements locaux. Une approche basée sur la compositionnalité doit donc offrir des outils assistant les développeurs sur ces aspects. C'est une possibilité

3. Ceci correspond aux études sur l'applicabilité des méthodes formelles au niveau système par exemple.

4. http://www-lsr.imag.fr/afadl2000/EtudeDeCas/

5. http://research.microsoft.com/users/lamport/tla/tla.html

6. Ces remarques rejoignent celles de L. Lamport dans le cas des preuves par décomposition (section 4.1.2).

offerte dans une approche comme Specware [SJ94, SJ95], où les développements sont représentés par des diagrammes, mathématiquement définis, qui permettent de raisonner sur les développements.

La conclusion de cette section est que la mise en œuvre d'une approche pour le développement formel incrémental est nécessairement complexe et limitative. La première raison est liée aux restrictions théoriques. La seconde est liée au type d'outils que l'on veut mettre entre les mains des développeurs. Ceci dépend du degré d'expertise attendu. On peut aller d'une démarche très " cablée " dans laquelle l'utilisateur n'a pas vraiment à maîtriser la sémantique des développements structurés, à des approches très souples qui nécessitent de sa part une compréhension fine des mécanismes offerts. Finalement, la dernière difficulté est liée à la mise en œuvre d'une telle approche, dans un langage et un environnement, afin d'offrir des outils apportant une aide effective aux développeurs. Les différents aspects que doivent couvrir des langages qui supportent le développement incrémental sont abordés dans la section ci-dessous.

### 4.1.5 Aspects langage

Les fondements théoriques sont généralement exprimables de manière simple, mais ceci ne signifie pas toujours qu'ils sont faciles à utiliser. Par exemple l'approche adoptée dans TLA est très élégante : toutes les manipulations sur les composants, ou presque, se réduisent à des opérateurs logiques. Les composants et les sous-composants sont exprimés par des formules de logique temporelle, les propriétés de partitionnement se codent et se vérifient logiquement, le masquage de variables est réalisé par un quantificateur existentiel et les propriétés attendues sur les composants sont exprimées par des implications logiques. Finalement la composition se réduit à la conjonction et le raffinement se réduit à l'implication. Il faut néanmoins donner les correspondances de raffinement[7] dans le cas de raffinement de données [AL91]. Une telle approche offre un cadre complet pour exprimer le développement incrémental, et les raisonnements sous-jacents. Par contre sa mise en œuvre nécessite de la part du développeur une connaissance fine de ce cadre, et

7. Refinement mapping

une maîtrise parfaite de ce qu'il fait. Il est donc nécessaire de proposer des langages permettant de manipuler ces différentes notions[8]. Il existe, à l'heure actuelle, peu de langages permettant de soutenir une démarche complète de développement incrémental. Ceci signifie qu'il n'y a pas, encore, de principes établis de conception de tels langages, comme ceci peut être le cas pour les langages de programmation ou pour les langages à base de composants. Dans la suite de cette section nous allons donner quelques caractéristiques attendues de ces langages. La présentation ci-dessous est issue de mon expérience de la méthode B et de l'étude d'autres approches, principalement algébriques. Un exemple est le langage Specware[9] [SJ94, SJ95].

Le premier besoin est d'offrir un langage de description de spécifications modulaires. De nombreux travaux ont été développés dans ce sens, particulièrement dans le cadre des approches algébriques. Un langage de description de modules offre une forme de composant basique et des opérateurs permettent de construire d'autres composants (extension, renommage, masquage d'information, mise en parallèle ...). Dans la suite nous qualifions ces composants de structurés. Certains langages permettent de ramener les composants structurés à des composants basiques. Cette approche est appelée *réduction syntaxique* dans [BS96] et permet de définir la sémantique des opérateurs de composition en terme de la sémantique de leur construction syntaxique. Il n'y a donc qu'une notion basique à appréhender. C'est généralement le cas pour les langages basés sur la logique. Par exemple le langage Z permet d'assembler des schémas (les entités de base de Z) par des opérateurs logiques et le résultat est un nouveau schéma. TLA relève de la même approche. L'approche par réduction syntaxique n'est pas toujours possible[10] et n'est pas complètement adaptée du point de vue langage car elle ne permet généralement pas de maîtriser finement le partage. En effet, un langage permettant de manipuler des composants va, par exemple, offrir des mécanismes pour faciliter la gestion des noms (règles de visibilité, de redéfinition ...). Il devient alors difficile de donner une interprétation simple par réduction syntaxique, la structure étant nécessaire. En particulier lorsqu'il y a du partage, il faut pouvoir modéliser les parties communes. Par

8. C 'est d'ailleurs l'objectif du projet TLA+.

9. qui est basé aussi sur la théorie des catégories.

10. Dans certains langages algébriques par exemple le résultat d'une spécification structurée n'est pas exprimable syntaxiquement.

exemple dans la notation Z les variables sont considérées comme communes dès qu'elles ont le même type. Dans le cas d'un langage, il faut généralement qu'elles soient issues de la même définition, ce qui est plus complexe à vérifier, mais plus sûr d'un point de vue méthodologique. La maîtrise du partage s'avère de plus importante lorsqu'on s'intéresse à la compositionnalité. En effet si on assimile " par hasard " des noms identiques il y a peu de chances de pouvoir composer les raisonnements. De fait un langage comme Z a peu de propriétés intrinsèques de compositionnalité. On est donc souvent amené à distinguer les composants basiques des composants structurés, qui portent l'information de leur construction. Finalement, les opérateurs de composition vérifient généralement certaines lois algébriques (commutativité, distributivité, ...). Un langage peut se distinguer par le fait qu'il offre, ou non, la possibilité de manipuler les structurations par l'intermédaire des propriétés de ces opérateurs. Si ces propriétés sont offertes à l'utilisateur, ceci lui donne une grande souplesse pour raisonner sur les spécifications structurées ou les transformer.

Le second aspect qui doit être supporté par le langage est le raffinement. Il existe à l'heure actuelle peu de langages qui intègre le raffinement. Par exemple Z et VDM définissent une notion de raffinement, mais qui n'est pas intégré à des outils : le raffinement, comme d'autres formes de raisonnement, fait partie de la méthodologie externe à la disposition des développeurs. La description d'un raffinement contient plusieurs informations[11]. La mise en place d'une relation de raffinement, dans le cas d'une spécification basique, nécessite de donner, sous une forme ou une autre :

- la spécification raffinée ;
- le lien entre la spécification abstraite et la spécification concrète ;
- l'argumentaire du raffinement (relation entre données abstraites et données concrètes, correspondance de noms ...)

Le raffinement est donc une relation entre spécifications, qui vérifie des propriétés particulières, par exemple la transitivité qui permet de raffiner une

11. Le terme raffinement est souvent surchargé : il peut désigner la spécification raffinée ou bien la relation entre deux spécifications. Nous utiliserons ce terme indifféremment dans ses deux sens.

spécification en plusieurs étapes. Dans le cas d'une spécification construite à partir d'autres spécifications, le principe de compositionnalité permet de ramener le raffinement aux raffinements des spécifications basiques qui la composent. Le raffinement peut alors être vu comme un opérateur de construction de spécification structurée : on obtient une spécification globale dans laquelle un composant a été substitué par son raffinement (voir la règle de la section 4.1.3). Un langage peut aussi offrir des mécanismes plus souples pour raffiner des spécifications structurées, afin de se libérer du principe de compositionnalité, comme suggéré dans la section 4.1. La réduction syntaxique est un mécanisme possible. Elle permet de remplacer une spécification structurée par une spécification basique et d'oublier sa structuration. Cette possibilité est souvent assortie de restrictions qui sont nécessaires lorsque des spécifications peuvent être partagées, afin de garantir par exemple le raffinement homogène des parties communes. Le langage peut aussi permettre de raffiner directement des formes particulières de spécifications structurées, en exploitant les propriétés des structurations. Les possibilités offertes par un langage, en dehors des limitations théoriques, sont donc le résultat d'un compromis entre la puissance des concepts proposés aux utilisateurs et la difficulté de leur mise en œuvre. La compositionnalité des raffinements est par exemple un mécanisme simple puisque l'utilisateur ne manipule jamais directement les spécifications structurées.

Un langage permettant de supporter le développement formel incrémental doit donc, à la base, offrir une notion de raffinement de spécification basique et son extension aux spécifications structurées. Dans le cas de raffinement par composition, il doit néanmoins permettre de visualiser la spécification résultante, et si possible ses propriétés, comme pointée en fin de la section 4.1.4. Les particularités du langage dépendent de la manière d'exprimer le raffinement et du statut associé à la relation de raffinement. Par exemple, elle peut être intégrée dans le langage, en tant qu'opérateur particulier entre spécifications. De plus si le langage intègre des lois algébriques sur les opérateurs de composition, y compris l'opérateur de raffinement, alors les développements deviennent eux-mêmes des objets manipulables. Finalement il n'existe pas, à notre connaissance, de langage supportant un paramétrage par des relations de raffinements. En effet, ceci nécessiterait qu'un tel langage prenne en charge la validation des propriétés attendues d'une relation de raffinement (raffinement pas à pas, compositionnalité ...), ce qui change le niveau de

complexité, puisque le langage devient, dans une certaine mesure, paramétré par une méthode.

Nous terminerons cette section par un aperçu, rapide, du langage Specware. La particularité de cette approche est d'offrir une représentation mathématique de la structure des spécifications et des raffinements, qui permet donc de manipuler formellement les structurations. Un des intérêts est alors de pouvoir inférer de nouvelles propriétés à partir de la structure (un exemple est donné dans [LOP98]). Pour cela, le langage Specware offre, en plus de la notion de spécification basique :

- des constructeurs de spécifications. Une spécification structurée est appelée un diagramme de spécification. Un diagramme est un objet formel dont les nœuds sont des spécifications et les flêches des morphismes de spécification. Un diagramme peut être interprété comme une spécification basique par l'opérateur de colimite, sous l'hypothèse que certaines conditions sémantiques soient vérifiées.
- Le raffinement est défini en s'appuyant aussi sur la notion de morphismes et la composition verticale se déduit des propriétés de la structure d'un raffinement ;
- Le raffinement est étendu aux diagrammes de spécification par les diagramme de raffinement. Ces derniers ont aussi leur propre propriété de compositionnalité.

La puissance de l'approche proposée par Specware est la représentation des structurations à base d'objets formels (ici les morphismes de spécification). Ceci donne une grande souplesse. On peut composer des morphismes pour déduire de nouveaux faits sur une structuration. De plus, les diagrammes ayant leurs propres propriétés, il est possible de raisonner sur ces diagrammes. Bien que le cadre théorique soit complexe, et que les spécifications manipulées soient axiomatiques, l'approche est intéressante dans la pratique car elle permet de représenter les développements structurés par des objets formels. Le point de vue des auteurs est que ce niveau de formalisation est nécessaire pour permettre le passage à l'échelle des techniques de développement de programmes au développement de systèmes plus complexes.

L'approche adoptée dans la méthode B est une approche plus pragmatique : le développeur ne manipule que les spécifications basiques et le mécanisme de compositionnalité lui est, d'une certaine manière, transparent. Il n'a donc pas besoin de maîtriser des principes complexes. C'est ce qui rend cette approche opérationnelle. Néanmoins cette compositionnalité est, bien sûr, théoriquement possible et peut même être exprimée dans le cadre de B. C'est l'approche que nous avons adoptée pour valider la compositionnalité du raffinement. Elle donne de plus une méthode effective pour étendre " gratuitement " les calculs sur les composants. Elle permet, par exemple, de synthétiser la vue globale d'un développement à différents niveaux de détails, et ainsi d'exprimer et de vérifier des propriétés sur ces spécifications globales.

La méthode B a été conçue pour développer des programmes de manière incrémentale, tout en assurant la compositionnalité des preuves. Deux types de preuve vont être considérés : les preuves d'invariant et les preuves de raffinement. Les possibilités offertes par la méthode sont donc, comme nous le verrons, le résultat d'un compromis entre les outils pratiques offerts pour le développement incrémental de programmes et la compositionnalité des preuves. Dans la section 4.2 nous introduisons la notion de composant et de preuves associées à un composant. La compositionnalité des spécifications, et des preuves d'invariant, est développée dans la section 4.3. La compositionnalité des développements, et des preuves de raffinement, est développée dans la section 4.4.

## 4.2 Composants dans la méthode B

Dans la méthode B, les différentes types de composants sont les suivants :

- les *machines abstraites*, qui décrivent des machines à état. Les preuves sont relatives à la préservation des invariants.

- *les raffinements*, qui décrivent une vue plus détaillée d'une machine abstraite, par raffinement des données et des traitements. Les preuves sont relatives à la correction du raffinement vis-à-vis de sa spécification.

- *les implémentations*, qui sont des raffinements qui suivent des règles particulières, permettant leurs traduction directe vers un langage de programmation.

Dans cette section nous présentons les preuves associées à chaque type de composant.

### 4.2.1 Machine abstraite et obligations de preuve

La notion de machine à état de B est classique, elle est basée par exemple sur celle introduite par C. Morgan, dans le cadre du raffinement [MG90]. L'originalité est la notion d'invariant, sur laquelle nous reviendrons (voir section 4.2.3). Voici un schéma simplifié d'une machine abstraite[12] :

MACHINE $M$
VARIABLES $v$
INVARIANT $I$
INITIALISATION $U$
OPERATIONS
$u \longleftarrow nom\_op(w) \hat{=}$ PRE $P$ THEN $S$ END;
...
END

L'invariant $I$ est une propriété du composant, pour les variables $v$. On montre que cet invariant est établi par l'initialisation et que, pour chaque opération, lorsqu'elle est appelée dans un état vérifiant sa précondition, le corps de l'opération préserve l'invariant. Ces obligations de preuve s'expriment par les formules suivantes :

| | |
|---|---|
| $[U]I$ | initialisation |
| $I \wedge P \Rightarrow [S]I$ | opérations |

12. D'autres notions sont autorisées dans les machines, comme les constantes et leurs propriétés ... Nous ne les considérons pas ici, car elles ne posent pas de problème particulier pour la composition des preuves.

La notion d'invariant ne correspond pas à une notion de typage. On s'intéresse à sa validité seulement pour la granularité des opérations. Les variables peuvent, à l'intérieur du corps d'une opération, prendre des valeurs intermédiaires qui ne respectent pas l'invariant[13].

### 4.2.2 Raffinement et obligations de preuve

Un raffinement se présente sous la forme d'un *delta* par rapport à une spécification. Il ne correspond donc pas à un composant à part entière. Il permet le raffinement des données et des traitements. Nous donnons ci-dessus la forme d'un raffinement par rapport à une machine.

| | |
|---|---|
| MACHINE $M_1$ | REFINEMENT $R_2$ |
| | REFINES $M_1$ |
| VARIABLES $v_1$ | VARIABLES $v_2$ |
| INVARIANT $I_1$ | INVARIANT $I_2$ |
| INITIALISATION $U_1$ | INITIALISATION $U_2$ |
| OPERATIONS | OPERATIONS |
| $r \leftarrow op(p) \mathrel{\hat{=}}$ | $r \leftarrow op(p) \mathrel{\hat{=}}$ |
| PRE $P_1$ THEN $S_1$ END | PRE $P_2$ THEN $S_2$ END |
| END | END |

Certaines variables peuvent être communes à la spécification et au raffinement. Cette possibilité est une facilité syntaxique qui permet d'alléger l'écriture d'un raffinement. De la même manière si une opération garde la même définition, il n'est pas nécessaire de la répéter. L'invariant $I_2$ porte à la fois sur les variables de $M_1$ et de $R_2$ et établit la relation de raffinement entre l'état abstrait $v_1$ et l'état concret $v_2$. Cette relation est la suivante :

$$\{(v_2, v_1) \mid I_1 \wedge I_2\}$$

Les obligations de preuve consistent à montrer que l'initialisation abstraite est raffinée par l'initialisation concrète et que, pour chaque opération, la

13. Le séquencement et l'itération n'étant pas autorisés dans les machines, cette remarque s'applique aux raffinements et aux implémentations.

définition abstraite est raffinée par la définition concrète. La notion de raffinement adoptée ici est classique : les préconditions peuvent être élargies et les postconditions peuvent être renforcées. Pour les preuves de raffinement, les variables communes sont renommées dans le raffinement et l'invariant $I_2$ est complété par le prédicat $v_i = v_i'$, où $v_i'$ désigne le renommage de la variable $v_i$. Les obligations de preuve se calculent par les formules suivantes :

| | |
|---|---|
| $[U_2]\ \neg[U_1]\ \neg I_2$ | initialisation |
| $I_1 \wedge P_1 \wedge I_2 \Rightarrow P_2$ | précondition |
| $I_1 \wedge P_1 \wedge I_2 \Rightarrow [S_2]\ \neg[S_1]\ \neg I_2$ | opération sans résultat |
| $I_1 \wedge P_1 \wedge I_2 \Rightarrow [[r := r']S_2]\ \neg[S_1]\ \neg(I_2 \wedge r = r')$ | opération avec résultat |

De la même manière que précédemment on ne s'intéresse aux raffinements des opérations que pour les variables vérifiant l'invariant abstrait $I_1$. Une interprétation de ces obligations de preuve en terme logique, c'est-à-dire exprimée sur les prédicats trm et prd, est donnée dans le chapitre 3 (propriété 6, page 57).

### Raffinement par pas

Une propriété intéressante du raffinement est la transitivité, qui permet de construire un raffinement par étapes, et de couper ainsi la complexité d'un développement. La relation de raffinement s'obtient en composant les invariants de raffinement de la manière suivante :

$$\{(v_3, v_1) \mid \exists v'\ (I_1 \wedge I_2 \wedge I_3)\}$$

avec $v' = v_2 - (v_1 \cup v_3)$[14]. $I_1$ est l'invariant sur les variables $v_1$ de la spécification la plus abstraite, $I_2$ est l'invariant de liaison entre les variables $v_1$ et $v_2$ de la spécification intermédiaire et $I_3$ est l'invariant de liaison entre les variables $v_2$ et $v_3$.

14. Si un nom de variable disparait dans une chaîne de raffinement, il ne peut être réutilisé plus bas. C'est ce qui fait que le calcul syntaxique donné ici est correct.

### Raffinement vu comme un composant

Il est possible de déduire des obligations de preuve données ci-dessus, les propriétés suivantes (B-Book p. 525 et suivantes) :

$$\begin{array}{|l|}\hline I_1 \wedge P_1 \wedge I_2 \Rightarrow [S_2]\,\exists v'\,(I_1 \wedge I_2) \\ \hline I_1 \wedge P_1 \wedge I_2 \Rightarrow [S_2]\neg[S_1]\neg(I_1 \wedge I_2) \\ \hline \end{array}$$

avec $v' = v_1 - v_2$, c'est-à-dire les variables non communes à la spécification et au raffinement. La première formule donne l'invariant qui caractérise les variables du raffinement. La seconde propriété établit la relation de raffinement donnée précédemment ($\{(v_2, v_1) \mid I_1 \wedge I_2\}$). Ces propriétés permettent de construire un composant " complet ", et validé par construction, qui correspond au raffinement (cette possibilité n'est que théorique dans la méthode B). Voici la forme de ce composant, pour le raffinement donné précédemment[15] :

MACHINE $M_2$
VARIABLES $v_2$
INVARIANT $\exists v'\ (I_1 \wedge I_2)$
INITIALISATION $U_2$
OPERATIONS
$r \leftarrow op(p) \mathrel{\hat{=}}$ PRE $P_2 \wedge \exists v'(I_2 \wedge P_1)$ THEN $S_2$ END
END

De la même manière on peut exprimer la relation de raffinement entre les deux machines abstraites $M_1$ et $M_2$ par le composant B suivant :

REFINEMENT $RR_2$
REFINES $M_1$
EXTENDS $M_2$
INVARIANT $I_2$
END

15. Nous faisons ici abstraction des restrictions syntaxiques associées aux différents types de composant.

La clause EXTENDS permet de construire une spécification qui " recopie " la spécification $M_2$ (voir section 4.3.3). Ces constructions sont étendues aux clauses d'assemblage dans [BP00] et les preuves de correction de ces composants sont développées dans [Bon99].

### 4.2.3 Utilisation des machines

Nous décrivons ici les utilisations externes qui peuvent être faites d'une machine, et les propriétés de ces utilisations. Ces propriétés vont définir une notion d'observabilité des machines, et donner un sens aux obligations de preuve relatives à un composant.

#### Notion d'invariant

La notion d'invariant est attaché à un espace de variables, **pour un jeu d'opérations donné**. Cette notion d'invariant est bien adaptée à la forme des modules considérés : un module offre des opérations sur des données, qui permettent d'observer des valeurs de ces données, et ces valeurs vérifient certaines propriétés. Il n'existe pas vraiment d'autres langages qui offre une telle notion. Les invariants sont généralement soit des invariants de typage (comme dans [MV92]), qui doivent donc être préservés à chaque pas intermédiaire des calculs, soit des invariants locaux, qui correspondent à des assertions ponctuelles sur les variables[16]. Une forme un peu similaire d'invariant est proposée dans [LW94], pour décrire l'aspect observable des objets d'une classe et de ses sous-classes.

La notion d'invariant va jouer un rôle important dans le processus de preuve. Comme nous l'avons vu, le raffinement des opérations n'est, par exemple, considéré que pour des données vérifiant l'invariant. La spécification complète d'une opération définie par $P \mid S$ est donc $P \wedge I \mid S$. La précondition implicite $I$ sera garantie par construction lors des appels d'opérations.

16. Par exemple dans les systèmes à base de transitions on manipule des invariants de la forme $c = i \Rightarrow A$, où $c = i$ désigne un point de contrôle.

### Vue d'un composant

Un composant rend généralement visible à l'extérieur, potentiellement, sa définition complète. Les différentes utilisations d'une machine vont restreindre cette visibilité. On distinguera les vues suivantes :

- *vue ouverte* : les variables sont visibles de l'extérieur sans contrainte, ainsi que les opérations.

- *vue semi-ouverte* : les variables ne sont visibles de l'extérieur qu'en position de lecture (en tant qu'expression) et les opérations sont visibles.

- *vue fermée* : les variables ne sont pas visibles[17].

La vue ouverte est la plus souple. Elle n'est pas offerte aux développeurs. Elle correspond à l'utilisation d'une machine par le processus de production des obligations de preuve, puisque les appels d'opérations sont expansés, comme ceci a été décrit dans le chapitre 3. La modification explicite d'une variable à l'extérieur d'une machine nécessiterait de prouver la préservation de l'invariant, pour cette nouvelle modification. C'est pour cela qu'elle n'a pas été retenue[18]. Le fait de n'utiliser que des appels d'opération pour modifier les variables (vue semi-ouverte) permet d'assurer la préservation de l'invariant, sous la condition que la précondition est vérifiée, comme ceci a été établi dans le chapitre 3 (section 3.5.3). La vue fermée correspond au principe d'encapsulation et assure l'indépendance vis-à-vis des choix de représentation d'une spécification. Dans la méthode B le choix des différentes vues dépend syntaxiquement du type des composants. Dans les machines et les raffinements les vues permises sont semi-ouvertes et dans les implémentations la seule vue autorisée est la vue fermée. Ces différentes caractéristiques sont récapitulées

17. Nous ne considérons pas ici les variables dites concrètes, c'est-à-dire qui ne peuvent être raffinées. La vue fermée les laisse bien entendu visibles.

18. Pour lever cette contrainte, il suffirait d'ajouter localement une nouvelle obligation de preuve.

dans le tableau ci-après.

| | |
|---|---|
| vue ouverte : | adaptée pour la réutilisation de spécifications<br>utilisée seulement pour les preuves |
| vue semi-ouverte : | adaptée pour la réutilisation des preuves<br>proposée dans les machines et les raffinements<br>réutilisation des preuves d'invariant |
| vue fermée : | adaptée pour la réutilisation de développement<br>proposée uniquement dans les implémentations<br>réutilisation des preuves d'invariant et de raffinement |

Le choix fait dans la méthode B est plus souple que celui proposé par C. Morgan dans [MG90], qui impose uniquement la vue fermée. Il est néanmoins limitatif car toute nouvelle opération doit pouvoir s'exprimer en terme des opérations basiques offertes[19]. Dans [Hay96], I. Hayes montre que cette vue peut être trop contraignante dans un processus de réutilisation.

### Utilisation

Une machine abstraite correspond donc à une machine à état offrant à l'extérieur un jeu d'opérations. Les utilisations qui peuvent en être faites sont donc des traitements qui utilisent ces opérations pour modifier l'état interne de cette machine, dans le cas d'une vue semi-ouverte, ou pour accéder à cet état, pour la vue fermée.

Une utilisation $U_M$ d'une machine $M$ est donc une substitution construite à partir d'appels d'opérations de la machine $M$, ou d'affectations ne modifiant par les variables de $M$[20]. De plus on ne considère que les substitutions pour lesquelles la terminaison est établie.

19. La puissance dépend des combinaisons possibles des opérations.

20. La notion d'utilisation correspond à la notion de substitution externe introduite dans le B-Book.

**Définition 8** *Utilisation*

1. *tout appel d'une opération de $M$ est une utilisation ;*
2. *une affectation qui ne modifie pas les variables de $M$ est une utilisation ;*
3. *Toute combinaison d'utilisations par les constructeurs de substitution généralisée, autres que l'opérateur $||$, est une utilisation ;*
4. *$U_1 \,||\, U_2$ est une utilisation si les substitutions $U_1$ et $U_2$ ne mettent en parallèle que des opérations de lecture*[21] *(voir chapitre 3, section 3.4) ;*
5. *la terminaison doit être établie.*

Une utilisation $U_M$ sera dite *fermée* si elle ne contient aucune référence aux variables de $M$. Finalement, la méthode B introduit un certain nombre de restrictions, suivant le type de composant :

- dans les machines les constructeurs de séquentialité et d'itération ne sont pas autorisés ;
- dans les raffinements l'itération n'est pas autorisé ;
- dans les implémentations seules les utilisations fermées sont autorisées. De plus le constructeur $||$ n'est pas autorisé ainsi que certains autres constructeurs.

Les restrictions sur les machines et les raffinements sont purement d'ordre méthodologique[22]. Au niveau des implémentations, seules les substitutions correspondant à des instructions sont admises. En particulier ces instructions vérifient la loi du *miracle exclus*[23]. Par construction, les utilisations vérifient certaines propriétés.

21. On considère ici les utilisations relatives à une seule machine. Dans le cas de la composition de machines, la règle devient : si deux opérations d'une même machine sont en parallèle ce ne peut être que des opérations de lecture.

22. Le constructeur de séquentialité n'est néanmoins pas compatible avec la clause d'assemblage USES (voir section 4.3.3).

23. Un programme est un témoin de l'existence des valeurs décrites par la spécification.

### Préservation des invariants

**Propriété 9** *Préservation des invariants*

$$\boxed{I \wedge \mathsf{trm}(U) \Rightarrow [U]I}$$

L'hypothèse de terminaison est nécessaire. C'est elle qui garantit que les opérations sont appelées dans un état vérifiant leur précondition, et donc que ces appels préservent l'invariant. La preuve se fait par cas sur la définition des utilisations. Dans le cas d'un appel, cette propriété est établie par les obligations de preuve de la machine $M$, et par la préservation de ces preuves par appel (chapitre 3, section 3.5.3, p. 68). Dans le cas 2 de la définition 8, $I$ est trivialement préservé puisque les variables de $I$ ne sont pas modifiées. On peut de plus montrer que tous les constructeurs de substitution préservent la propriété 9. Par exemple on a :

$$\frac{I \ \wedge \ \mathsf{trm}(U_1) \Rightarrow [U_1]I \qquad I \ \wedge \ \mathsf{trm}(U_2) \Rightarrow [U_2]I}{I \ \wedge \ \mathsf{trm}(U_1 \ ; U_2) \Rightarrow [U_1 \ ; U_2]I}$$

Pour le constructeur || la propriété 7 (chapitre 3, section 3.4.2, p. 62) peut être appliquée.

### Préservation des raffinements

Soit $M$ une machine raffinée en $R$, par l'invariant de raffinement $J$. Soit $U_M$ une utilisation fermée de la machine $M$. L'utilisation $U_R$, obtenue en interprétant les appels des opérations de $M$ par leur définition dans $R$, vérifie la propriété suivante :

**Propriété 10** *Préservation des raffinements*

$$\boxed{U_M \sqsubseteq_J U_R}$$

Rappelons que la notation $S \sqsubseteq_J T$ est équivalente à la formule $J \wedge \mathsf{trm}(S) \Rightarrow [T]\neg[S]\neg J$ (chapitre 3, définition 4, p. 56). La propriété 10 garantit qu'on obtient un raffinement d'une utilisation à partir du raffinement de chacune des opérations qui la constituent. La preuve se fait par récurrence sur $U$. Dans le cas d'un appel, cette propriété a déjà été établie dans la section 3.5.3, chapitre 3. Dans le cas d'une affectation sur les variables non raffinées cette propriété est trivialement vraie. Le cas 3 de la définition 8 découle de la monotonie des constructeurs de substitution vis-à-vis du raffinement (p. 518 du B-Book). Par exemple on a :

$$\frac{S_1 \sqsubseteq_J T_1 \qquad S_2 \sqsubseteq_J T_2}{S_1 \ ; \ S_2 \sqsubseteq_J T_1 \ ; \ T_2}$$

Le cas de l'opérateur || n'est pas à traiter ici, puisque ce constructeur n'est pas autorisé dans les implémentations.

## 4.3 Spécification incrémentale

Nous présentons dans cette section les principes adoptés dans la méthode B pour construire de manière incrémentale les spécifications, et pour composer les preuves d'invariant. Cette présentation n'a pas pour but de décrire en détail les différentes constructions offertes par la méthode B (voir [Pot01, Pot03] pour un développement plus complet). Dans la suite nous dénoterons par spécification ou composant, une machine, un raffinement ou une implémentation.

### 4.3.1 Principe de construction

La méthode B permet de construire de nouvelles spécifications, à partir de machines abstraites (les raffinements et les implémentations n'étant pas considérés comme des spécifications à part entière, ils ne peuvent être utilisés). Cette construction se fait par l'intermédiaire de *clauses d'assemblage*. Le principe est similaire à celui des langages de programmation (clause `WITH` du langage Ada par exemple). Une clause d'assemblage donne accès aux

définitions d'une machine et permet de définir de nouvelles opérations à partir des opérations offertes par les machines référencées. La méthode B offre plusieurs formes de clauses d'assemblage, qui vont permettre de maîtriser la composition des preuves. Ces clauses relèvent toutes du même principe de composition. Pour expliquer ce principe, nous utilisons dans un premier temps une clause générique, que nous appelons COMBINES, qui décrit la composition de machines abstraites. Les particularités des différentes clauses de la méthode B seront présentées rapidement dans la section 4.3.3. Le partage est introduit en composant des spécifications construites à partir d'une spécification commune.

Une spécification incrémentale se présente sous la forme suivante :

MACHINE $M$
COMBINES $M_1, \ldots, M_n$
VARIABLES $v$
INVARIANT $I$
OPERATIONS
$r \leftarrow op(p) \mathrel{\hat{=}} S$
...
END

Les opérations de $M$ décrivent de nouveaux traitements, qui peuvent être définis à l'aide de certaines opérations des machines $M_1, \ldots, M_n$. La substitution $S$ prend la forme d'une utilisation de ces machines, comme introduit dans la section 4.2.3. La machine $M$ peut contenir un invariant, associé au jeu d'opérations de $M$, qui établit une relation entre les variables des machines $M_i$ et les variables définies dans $M$. Par défaut les opérations des machines $M_i$ ne font pas partie de l'interface de $M$. Elles peuvent être promues par la clause PROMOTES. Cette possibilité est bien entendu conditionnée par le fait que les opérations promues vérifient l'invariant $I$.

Le principe adopté est de pouvoir composer les invariants en même temps que les spécifications. Ceci signifie que les opérations de $M$ doivent vérifier l'invariant $I$, mais aussi ceux des machines $M_1, , \ldots, M_n$. Ce choix permet de

construire de manière incrémentale non seulement les traitements, mais aussi les propriétés sur les données. La construction incrémentale de spécification est définie dans le B-Book (p. 312), par réduction syntaxique :

- les listes de variables des différentes machines sont concaténées avec la liste $v$ ;
- l'invariant est la conjonction des invariants des machines $M_i$ et de l'invariant de $M$ ;
- la définition des opérations de $M$ correspond au texte donné dans $M$ en expansant les appels par la définition des opérations des machines $M_1, \ldots M_n$.
- l'initialisation est construite en combinant les initialisations des machines $M_i$ et l'initialisation de la machine $M$.

Bien qu'une construction incrémentale soit possible, l'initialisation est un processus global à un développement dans son entier (similaire à l'élaboration des initialisations dans le langage Ada). Cette élaboration construit une initialisation, dans un ordre compatible avec les dépendances entre les machines. Différentes solutions sont possibles, qui peuvent d'ailleurs donner des valeurs différentes suivant l'algorithme choisi. Toutes ces solutions préservent les preuves [24].

### 4.3.2 Composition des preuves d'invariant

Le principe de composition des preuves consiste à garantir que les invariants des machines sont préservés par construction. Ceci impose des restrictions sur les opérations qui peuvent être composées dans les utilisations. En effet si les invariants portent sur des variables communes, toutes les opérations ne vérifient pas nécessairement ces invariants.

24. L'algorithme choisi dans l'atelier B est obtenu par un parcours en profondeur et gauche droite de l'arbre d'importation.

### Composition des machines

Le principe de compositionnalité des preuves repose sur une construction des spécifications incrémentales plus fine que celle décrite précédemment. Cette construction se découpe en deux étapes : les spécifications $M_i$ sont d'abord combinées entre-elles pour constituer un nouveau composant, puis la spécification incrémentale est construite à partir de cette combinaison. Ces constructions peuvent être décrites par des composants B, que nous donnons ci-dessous.

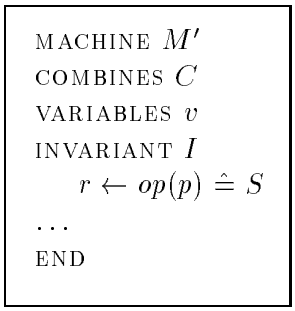
Le composant $C$ décrit la composition des machines. Ce composant va offrir à l'extérieur un sous-ensemble $Op_C$ des opérations des machines $M_1, \ldots, M_n$ qui ont la propriété de vérifier les invariants de chaque machine.

MACHINE $C$
COMBINES $M_1, \ldots, M_n$
PROMOTES $Op_C$
END

Le composant $C$ correspond à une opération classique sur les composants, qui consiste à composer les propriétés et les comportements. Cette composition est celle, par exemple, des programmes concurrents composés par entrelacement, pour lesquels la composition des preuves nécessite de vérifier la non-interférence entre les composants [OG76]. Les conditions sur les opérations qui peuvent être combinées vont garantir la non-interférence, par construction. Ces conditions seront détaillées dans la section 4.3.3.

La seconde étape consiste à construire le composant $M'$, correspondant au composant $M$. Il s'obtient à partir de $M$ en remplaçant la composition $M_1, \ldots, M_n$ par le composant $C$. Les opérations de $M$, et donc de $M'$, doivent être spécifiées comme des utilisations du composant $C$, elles ne peuvent donc

que référencer les opérations de la liste $Op_C$.




Le point important de cette construction est que les opérations des machines $M_i$ ne sont plus les mêmes que celles définies dans chaque composant $M_i$. Ces opérations sont maintenant étendues sur l'ensemble des variables de ces machines. L'extension considérée est celle qui correspond au principe d'expansion des appels : par défaut les variables ne sont pas modifiées. Pour préciser cela nous utilisons une notation introduite par R. Back et M. Butler [BB98] qui explicite à la fois les espaces de variables et les extensions d'espace d'état. Nous désignons par $env\ v\ .\ S$ la substitution $S$ définie sur l'espace de variables $v$ [25]. Soit $u$ un autre espace de variables. L'extension d'une substitution $S$ à l'espace de variables $u$ peut être notée $embed(S, u)$.

**Définition 9** *Extension des opérations*

$$embed((env\ v\ .\ S), u) = (env\ v\ .\ S)\ ||\ (env\ v - u\ .\ \mathsf{skip})$$

Dans la méthode B, l'opérateur *embed* n'a pas besoin d'être décrit explicitement, puisque l'extension par skip est celle qui correspond à l'interprétation d'une substitution par défaut, c'est-à-dire lorsqu'il n'y a pas de modification explicite. C'est donc le contexte, c'est-à-dire les formules, qui détermine l'espace de variables considéré. Néanmoins, pour éviter les ambiguïtés nous utiliserons cet opérateur.

25. Nous n'utilisons pas la notation VAR qui a déjà un sens dans la méthode B.

### Composition des preuves

La préservation des preuves va consister à établir la correction des deux composants $C$ et $M'$, par construction. Le point délicat est la correction du composant intermédiaire $C$. Soit $op_i$ une opération d'une machine $M_i$, définie par $P_i \mid S_i$ et appartenant à l'ensemble $Op_C$. Il faut montrer :

$$P_i \wedge \bigwedge_{j \in 1..n} I_j \Rightarrow [embed((env\ x_i\ .\ S_i), \bigcup_{j \in 1..n} x_j)] \bigwedge_{j \in 1..n} I_j$$

où $x_j$ désigne l'espace de variables de la machine $M_j$ et $I_j$ son invariant. La propriété sur la conjonction (chapitre 3, section 3.3.4) permet de couper la preuve. La condition de correction est donc :

**Condition 3** *Condition de préservation des invariants*

$$P_i \wedge \bigwedge_{j \in 1..n} I_j \Rightarrow [embed((env\ x_i\ .\ S_i), \bigcup_{j \in 1..n} x_j)] I_j$$

*pour* $j \in 1..n$.

Le cas $i = j$ correspond aux obligations de preuve de la machine $M_i$.

La correction du composant $M'$, et donc du composant $M$, découle directement de cette propriété. En effet, les opérations de $M$ sont définies en terme d'utilisations ne contenant que des appels d'opérations vérifiant la condition ci-dessus. Comme ceci a été établi par la propriété 9 (section 4.2.3, p. 106), les invariants sont préservés par la forme des utilisations. Pour chaque opération de $M'$, de la forme $P \mid S$, on a :

$$P \wedge \mathsf{trm}(S) \wedge \bigwedge_{j \in 1..n} I_j \Rightarrow [S] \bigwedge_{j \in 1..n} I_j$$

Les opérations du composant $M'$ préservent donc les invariants des machines $M_i$, sous la condition que ces opérations soient invoquées dans un état vérifiant leur précondition. Si la machine $M$ contient un invariant qui

lui est propre, la terminaison sera garantie par l'obligation de preuve relative à la préservation de cet invariant (propriété 4, chapitre 3, p. 56). Sinon l'obligation de preuve suivante doit être établie :

$$P \wedge \bigwedge_{j \in 1..n} I_j \Rightarrow \mathsf{trm}(S)$$

### 4.3.3 Mise en oeuvre dans la méthode B

#### Clauses d'assemblage

La méthode B permet donc de maîtriser les opérations composables, et les invariants, afin de garantir la condition 3 par construction. Ceci est implanté à l'aide d'un ensemble de clauses d'assemblage qui limite à la fois les définitions accessibles et la possibilité d'énoncer des propriétés invariantes.

- La clause INCLUDES $M$ permet d'appeler toute opération de la machine $M$ et d'énoncer des propriétés sur les variables incluses. Cette clause est autorisée dans les machines et les raffinements.

- La clause IMPORTS est l'équivalent de la clause INCLUDES au niveau des implémentations. Elle impose une vue fermée des composants importés.

- La clause SEES $M$ ne donne accès qu'aux opérations dites de lecture de la machine $M$, c'est-à-dire les opérations qui ne modifient aucune variable. De plus, les invariants ne peuvent pas porter sur les variables vues. Cette clause est autorisée dans les machines, les raffinements et les implémentations [26].

- La clause USES $M$ ne permet pas d'appeler les opérations, mais les variables peuvent être lues. Par contre, les invariants peuvent porter sur les variables utilisées. Cette clause est autorisée uniquement dans les machines.

- La clause PROMOTES *op*, avec *op* désignant une opération d'une machine incluse ou importée, a pour effet de compléter le jeu d'opérations

26. en vue fermée seulement pour les implémentations.

de la machine courante par l'opération *op*. Les obligations de preuve portent alors aussi sur cette opération.

- La clause EXTENDS est équivalente à l'inclusion ou l'importation d'une machine, en promouvant toutes ses opérations.

La clause USES propose un mécanisme sophistiqué de composition des preuves, dans le cas de partage au niveau des machines. Les restrictions qui lui sont associées font qu'elle n'est pas utilisée dans la pratique[27]. Nous ne la considérons pas plus dans la suite. Son principe est décrit dans [BPR96, Hab01]. On peut décrire les contraintes introduites par les clauses en terme du jeu d'opérations qu'elles offrent. Comme les clauses INCLUDES et IMPORTS permettent d'introduire de nouvelles propriétés sur des variables définies dans d'autres composants, les opérations des machines incluses ou importées sont masquées, sauf les opérations promues pour lesquelles l'invariant a été prouvé, et les opérations de lecture qui ont la propriété de préserver n'importe quel invariant. Dans le cas de la clause SEES, le mécanisme est plus simple. Puisqu'aucune contrainte supplémentaire n'est ajoutée sur les variables des machines vues, toutes les opérations sont implicitement promues.

Lors de la composition de machines, la préservation des invariants repose sur le principe *un écrivain/plusieurs lecteurs* : chaque machine est incluse ou importée au plus une fois. On peut montrer que ce principe permet d'établir la condition 3 par construction (la preuve est développée dans [Pot03]). La restriction à un seul écrivain impose, en cas de partage un peu complexe, que l'utilisateur maîtrise finement les invariants et les jeux d'opérations associés à ces invariants. La mise en place de telles constructions nécessite une certaine habitude, pour savoir où et comment exprimer les invariants. Nous ne détaillerons pas ceci ici (des exemples peuvent être trouvés dans [Pot01]). Remarquons que le choix fait de considérer que, par défaut, une substitution ne modifie pas les variables est complètement adapté à la préservation des invariants. Si nous avions pris le point de vue opposé, c'est-à-dire que par défaut les variables ne sont pas contraintes, alors une opération plongée dans un espace de variables plus grand ne préserverait pas les invariants.

27. Elle n'est d'ailleurs pas vraiment supportée par l'atelier B.

## 4.4 Développement incrémental

Dans cette section, nous allons décrire la construction et la correction des composants obtenus en combinant les raffinements.

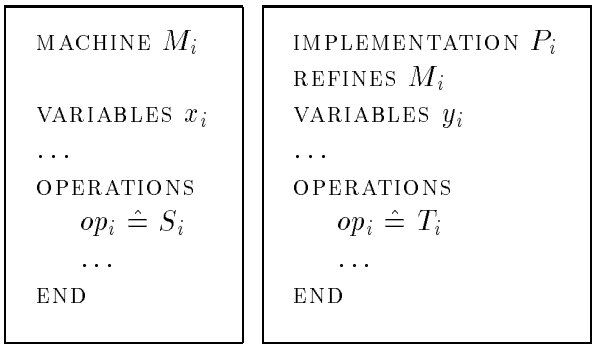
Dans la méthode B, une machine abstraite peut être implantée en plusieurs étapes, en raffinant chaque définition d'opération, comme décrit dans la section 4.2.2. Les implémentations sont des raffinements qui suivent des règles particulières : le langage de description des opérations est réduit au langage des instructions. Une implémentation n'est pas raffinable : elle correspond au dernier niveau d'une chaîne de raffinement. Une implémentation peut être construite à partir d'autres machines abstraites, par l'intermédiaire des clauses SEES et IMPORTS. Ces machines sont implémentées à leur tour. On parle de développement en couche. C'est à ce niveau qu'intervient la compositionnalité des raffinements. Soit une implémentation $P$ de la forme[28] :

IMPLEMENTATION $P$
REFINES $M$
COMBINES $M_1, \ldots, M_n$
INVARIANT $L$
OPERATIONS
$r \leftarrow op(p) \mathrel{\hat{=}} T$
...
END

où $T$ est une utilisation fermée, construite à partir des opérations des machines $M_1, \ldots, M_n$ autorisées par la composition. Soit $P_1, \ldots, P_n$ les im-

28. Nous ne considérons pas ici les variables concrètes.

plémentations de ces machines, de la forme :




Ces implémentations permettent de construire le code de $P$ en remplaçant dans $T$ les appels aux opérations $op_i$ définis en terme des spécifications $S_i$, par les mêmes appels, définis en terme de leur code $T_i$. La vue fermée permet cette substitution. Bien que cette construction ne soit pas visible par les utilisateurs de la méthode B, c'est elle qui sous-tend le processus de raffinement. Le résultat de la composition des implémentations constitue, par exemple, l'entrée de la phase de traduction vers un langage de programmation.

### 4.4.1 Composition des preuves de raffinement

Nous ne traiterons de la composition des raffinements qu'au niveau des implémentations, puisque c'est là que ce principe est utilisé dans la méthode B. Ce principe peut être étendu pour considérer des niveaux quelconques de raffinement [BP00].

#### Composition des raffinements

Dans cette section nous décrivons le principe de construction du composant qui contient le code associé à la machine. Intuitivement ce composant va être construit en ne considérant que le niveau des implémentations. Pour une machine $M$ nous désignerons par $C_M$ le code associé à $M$, et par $J_M$ l'invariant de raffinement entre $M$ et $C_M$.

Si l'implémentation $P$ de la machine $M$ n'est pas structurée (c'est-à-dire qu'elle ne contient pas de clause d'assemblage), le composant $C_M$ est la machine associée à $P$, construite suivant le principe donné dans la section 4.2.2 (p. 101). Si l'implémentation $P$ est structurée, la construction du composant $C_M$ se fait en plusieurs étapes, qui peuvent être décrites en terme de composants B[29].

La première étape consiste à construire le composant $C$ qui correspond à la combinaison des machines $M_1, \ldots, M_n$, comme décrit dans la section précédente. La seconde étape consiste à composer les codes $C_{M_i}$ associés aux composants $M_i$. Les machines $C_{M_i}$ se construisent par le principe donné dans la section 4.2.2, ainsi que les relations de raffinement $R_i$ entre les spécifications $M_i$ et $C_{M_i}$ :

REFINEMENT $R_i$
REFINES $M_i$
EXTENDS $C_{M_i}$
INVARIANT $J_{M_i}$
END

Les implémentations $C_{M_i}$ peuvent être composées ensemble dans le composant $C'$ suivant :

MACHINE $C'$
COMBINES $C_{M_1}, \ldots, C_{M_n}$
PROMOTES $Op_{C'}$
END

REFINEMENT $R$
REFINES $C$
EXTENDS $C'$
INVARIANT $\bigwedge_{j \in 1..n} J_{M_j}$
END

La relation de raffinement entre les composants $C$ et $C'$ s'obtient par la conjonction des invariants de raffinement entre chaque $M_i$ et $C_{M_i}$, comme décrit dans le raffinement $R$ ci-dessus.

29. en faisant abstraction des restrictions syntaxiques.

La dernière étape consiste à construire le composant $P'$, qui correspond au composant $P$ dans lequel les machines $M_i$ sont substituées par leur code. Ce composant a la forme suivante :

IMPLEMENTATION $P'$
REFINES $P$
COMBINES $C'$
INVARIANT $\bigwedge_{j \in 1..n} J_{M_j}$
OPERATIONS
$r \leftarrow op(p) \mathrel{\hat{=}} T$
...
END

Le résultat est un composant valide, car la vue fermée garantit qu'il n'y a pas de référence explicite aux états. La relation de raffinement est la même que entre $C$ et $C'$, c'est-à-dire la conjonction des invariants entre $M_i$ et $C_{M_i}$.

Le code $C_M$ associé à la machine $M$ s'obtient alors directement à partir de $M$, $P$ et $P'$, par le processus de construction de machine à partir de raffinement (section 4.2.2, p. 101). L'invariant de raffinement $J_M$ découle de la propriété de la transitivité du raffinement (voir section 4.2.2). La propriété ci-dessous caractérise la forme de cet invariant :

**Propriété 11** *Invariant de raffinement*

*Soit $L$ l'invariant entre une machine $M$ et son implémentation $P$, construite à partir des machines $M_1, \ldots, M_n$. Soit $J_{M_i}$ les invariants de raffinement établis entre les machines $M_i$ et leur code. L'invariant de raffinement entre la machine $M$ et son code $C_M$, noté $J_M$, est de la forme :*

$$J_M \mathrel{\hat{=}} \exists X \, (L \wedge \bigwedge_{i \in 1..n} J_{M_i})$$

*avec $X = \{x_i \mid M_i \in (combines(P) - combines(M))\}$.*

L'ensemble $combines(M)$ décrit la liste des machines qui sont utilisées pour construire la spécification $M$. Les variables qui sont communes à la

spécification $M$ et à l'implémentation $P$ ne sont pas quantifiées par la transitivité (sous-section raffinement par pas, p. 100). Ces variables correspondent aux machines qui sont utilisées à la fois dans $M$ et dans son implémentation.

### Composition des preuves

Le point délicat de la composition est la preuve du composant $R$, qui établit la relation de raffinement entre les composants $C$ et $C'$. Soit $op_i$ une opération d'une machine $M_i$ autorisée par la composition ($op_i \in OP_C$) et raffinée dans $M_{C_i}$ en $op'_i$. Il faut montrer que ce raffinement reste correct dans le composant $C'$, pour la conjonction des invariants de raffinement, c'est-à-dire :

**Condition 4** *Condition de préservation des raffinements*

$$\begin{array}{c} embed((env\ x_i\ .\ op_i), \bigcup_{j\in 1..n} x_j) \\ \sqsubseteq_{\bigwedge_{j\in 1..n} J_{M_j}} \\ embed((env\ y_i\ .\ op'_i), \bigcup_{j\in 1..n} y_j) \end{array}$$

*où $x_i$ désigne les variables des machines $M_i$ et $y_i$ les variables des implémentations $C_{M_i}$. Les relations de raffinement $J_{M_i}$ lient donc les variables $x_i$ et $y_i$.*

La correction du composant $P'$ découle de la condition ci-dessus, puisque toutes les opérations de $P$ sont des utilisations fermées définies sur les machines $M_1, \ldots, M_n$. Ces utilisations préservent donc, par construction, la relation de raffinement (section 4.2.3, propriété 10, p. 106).

Contrairement au cas des preuves d'invariant, il n'y a pas de théorème général qui permet de simplifier la condition 4. En effet les preuves de raffinement ne se composent pas nécessairement : si $S_1 \sqsubseteq_{L_1} S_2$ et $S_1 \sqsubseteq_{L_2} S_2$ alors $S_1 \sqsubseteq_{L_1 \wedge L_2} S_2$ n'est pas forcément valide. Par exemple la substitution non déterministe $x :\in \mathbb{N} - \{0\}$ a pour effet d'affecter à $x$ une valeur quelconque de l'ensemble $\mathbb{N} - \{0\}$[30]. Cette substitution peut se raffiner en $y := 1$, pour

30. Elle se ramène à la forme mathématique $@x'\ .\ (x' \in \mathbb{N} - \{0\} \Longrightarrow x := x')$.

tout invariant de la forme $x \neq 0 \Rightarrow x = y + n$, avec $n \in \mathbb{N}$. Par contre ce n'est pas vrai pour la conjonction de deux quelconques de ces invariants, qui se réduit à $x = 0$. En effet, la formule $x :\in \mathbb{N} - \{0\} \sqsubseteq_{x=0} y := 1$ est fausse, car elle revient à chercher un $x$ vérifiant à la fois $x = 0$ et $x \neq 0$.

De la même manière l'opérateur *embed* n'est pas transparent vis-à-vis du raffinement. Un raffinement correct peut devenir invalide si une substitution est plongée dans un environnement plus grand. Par exemple soit $S_1$ et $S_2$ les substitutions suivantes :

$$\begin{array}{rcl} S_1 & = & (env\ x\ .\ x := 1) \\ S_2 & = & (env\ y, z\ .\ y := 1\ ;\ z := 2) \end{array}$$

Le raffinement $S_1 \sqsubseteq_{x=y} S_2$ est correct mais on n'a pas $embed(S_1, z) \sqsubseteq_{x=y} S_2$ car $z := 2$ ne raffine pas $(env\ z\ .\ \mathsf{skip})$. Le problème provient du fait que le raffinement permet d'étendre l'espace de variables (ici $y$ et $z$), avec la seule contrainte que les nouvelles variables soient modifiées en accord avec la relation de raffinement. Les raffinements ne sont donc pas nécessairement compatibles avec l'opérateur *embed*, qui a pour effet d'étendre les substitutions par skip. La composition des raffinements va donc nécessiter un certain nombre de restrictions, afin de pouvoir garantir la condition 4 par construction. On a, par exemple :

**Théorème 8** *Préservation des raffinements sans partage*

$$\frac{(env\ x_1\ .\ S) \sqsubseteq_{J_1} (env\ y_1\ .\ T)}{embed((env\ x_1\ .\ S), x_2) \sqsubseteq_{J_1 \wedge J_2} embed((env\ y_1\ .\ T), y_2)}$$

*si $x_1 \cap x_2 = \emptyset$ et $y_1 \cap y_2 = \emptyset$. $J_1$ désigne un invariant de raffinement liant les variables $x_1$ et $y_1$ et $J_2$ un invariant de raffinement liant les variables $x_2$ et $y_2$.*

Ce théorème découle directement de la définition de l'opérateur *embed* (définition 9, p. 111), et de la monotonie du raffinement pour l'opérateur de substitution multiple, lorsqu'il n'y a pas de partage (B-Book p. 521) :

$$S_1 \sqsubseteq_{J_1} T_1 \ \wedge\ S_2 \sqsubseteq_{J_2} T_2 \ \Rightarrow\ S_1 \,||\, S_2 \sqsubseteq_{J_1 \wedge J_2} T_1 \,||\, T_2$$

si les espaces de variables de $S_1$, $S_2$, $T_1$ et $T_2$ sont disjoints deux à deux.

Le théorème précédent n'est pas suffisant pour la méthode B qui autorise l'introduction de certaines formes de partage au cours du raffinement. Cette dernière possibilité est intéressante du point de vue méthodologie car elle n'impose pas d'avoir spécifier complètement les partages au niveau des spécifications. En effet, comme nous l'avons évoqué dans la section 4.1.3, cette contrainte peut s'avérer gênante dans la pratique. Nous avons étendu le théorème précédent dans [Pot03] pour traiter les formes de partage autorisées par la méthode B.

**Théorème 9** *Préservation du raffinement avec partage*

*Soit $T'$ la substitution définie par :*

$$(env\ z\ .\ @\ \ y_1\ .\ (\exists x_1(\mathsf{trm}(S) \wedge J_1) \Longrightarrow T)$$

*Le théorème est le suivant :*

$$\frac{\begin{array}{ll} 1) & (env\ x_1\ .\ S) \sqsubseteq_{J_1} (env\ y_1 \cup z\ .\ T) \\ 2) & (env\ x_2\ .\ \mathsf{skip}) \sqsubseteq_{J_2} embed(T', y_2) \end{array}}{embed(env\ x_1\ .\ S, x_2) \sqsubseteq_{J_1 \wedge J_2} embed(env\ y_1 \cup z\ .\ T, y_2)}$$

*si $x_1 \cap x_2 = \emptyset$. $J_1$ désigne un invariant de raffinement entre les variables abstraites $x_1$ et les variables concrètes $y_1$ et $z$ et $J_2$ désigne un invariant de raffinement entre les variables abstraites $x_2$ et les variables concrètes $y_2$ et $z$.*

Ce théorème permet d'introduire du partage lors du raffinement chaque fois que l'hypothèse 2 est vérifiée. La condition introduite par cette hypothèse garantit que la substitution $T'$ est compatible avec l'extension par skip. La substitution $T'$ est obtenue à partir de la substitution $T$ en encapsulant les variables $y_1$ : elle décrit donc uniquement les modifications des variables $z$ communes entre $J_1$ et $J_2$. Cette condition permet de réconcilier le point de vue de la composition, qui suppose que les variables non référencées ne sont pas modifiées, et le point de vue du raffinement, qui suppose que les variables non contraintes peuvent varier de manière compatible avec le raffinement. Cette condition n'est à vérifier que pour les cas où la substitution $S$ termine.

### 4.4.2 Mise en oeuvre dans la méthode B

Dans la méthode B, la correction par construction de la composition des raffinements repose sur les choix suivants :

- le raffinement est localisé au niveau des composants : les espaces de variables sont raffinés composant par composant.
- la structuration est préservée par le raffinement ;
- le partage introduit par les raffinements est contrôlé.

Comme la structuration est préservée et que le partage est introduit uniquement par des machines communes, le partage va être préservé par raffinement. La méthode B impose de plus des contraintes architecturales, sur la forme des développements, qui vont permettre de contrôler les partages introduits au cours du raffinement. Ces contraintes, vérifiables statiquement, permettent d'assurer la correction de la composition des implémentations par construction. Cette section décrit les restrictions introduites par la méthode B et justifie leur caractère suffisant. La correction est établie en se basant à la fois sur des propriétés des invariants de raffinement et sur le théorème 9. Cette section est plus spécifiquement dédiée à la méthode B.

#### Propriétés des invariants de raffinement

La préservation de la structuration par le raffinement se traduit par le fait que les clauses qui introduisent du partage doivent obligatoirement être conservées lors du raffinement. Dans la méthode B, la seule clause qui introduise du partage, au niveau des machines, est la clause SEES[31]. Ceci signifie que si une machine $M$ contient la clause SEES $A$, alors tous ces raffinements contiendront cette même clause. Cette contrainte permet d'affiner la forme des invariants de raffinement, telle que définie par la propriété 11 (section 4.4.1, p. 118). En effet elle permet de partitionner les invariants de raffinement en fonction des espaces de variables de chaque machine. Dans la suite

31. La clause INCLUDES n'introduit pas de partage, elle pourrait mais elle devrait alors aussi être obligatoirement préservée par le raffinement. Ce n'est pas ce choix méthodologique qui a été fait.

nous notons $R_M$ l'invariant de raffinement propre à l'espace de variables de la machine $M$, c'est-à-dire aux variables déclarées dans $M$, et nous notons $J_M$ l'invariant de raffinement global, c'est-à-dire faisant intervenir toutes les variables d'une spécification structurée. On a alors :

**Propriété 12** *Partitionnement des invariants de raffinement*

$$J_M \;\hat{=}\; R_M \;\wedge \bigwedge_{M_i \in combines^+(M)} R_{M_i}$$

*où $combines^+(M)$ désigne les machines utilisées pour construire $M$, et ceci transitivement.*

Dans le cas d'un développement en couche, une implémentation peut être construire à partir d'autres machines, par les clauses SEES et IMPORTS. Comme les variables vues ne peuvent être référencées dans les invariants, ceci signifie que les variables d'une machine vue ne peuvent servir à représenter des variables abstraites. Une conséquence de cette restriction est que l'invariant propre d'une machine $M$ lie les variables de cette machine uniquement avec les variables introduites directement dans l'implémentation de cette machine et avec les variables concrètes accessibles à partir de M par une suite de liens d'importation. La forme des invariants $R_M$ est donc :

**Propriété 13** *Invariant de raffinement*

*Si $L$ est l'invariant de raffinement entre la machine $M$ et son implémentation, et si $R_{M_i}$ est l'invariant de raffinement propre à l'espace de variables de la machine $M_i$, l'invariant $R_M$ est de la forme :*

$$\exists X(L \wedge \bigwedge_{M_k \in imports(P)} R_{M_k})$$

*avec $X$ la liste des variables de l'ensemble $\{x_k \mid M_k \in imports(P)\}$.*

Seules les variables concrètes introduites par une suite d'importations sont liées aux variables abstraites. Les variables concrètes introduites par une suite

de clauses contenant un SEES ne sont contraintes que par l'invariant portant sur ces variables.

### Conditions pour le raffinement

Les propriétés structurelles sur les compositions et sur les invariants de raffinement vont permettre d'appliquer le théorème 9, afin d'établir la condition 4 de manière incrémentale. On va montrer :

$$embed((env\ x_i\ .\ op_i), x_j) \sqsubseteq_{J_i\ \wedge\ R_j} embed((env\ y_i\ .\ op'_i), y_j)$$

sachant $(env\ x_i\ .\ op_i) \sqsubseteq_{J_i} (env\ y_i\ .\ op'_i)$, et ceci itérativement pour chaque machine $M_j$ intervenant transitivement dans la composition.

Il y a deux cas à considérer suivant que $M_j$ appartient, ou non, à l'ensemble $combines^+(M_i)$. Si c'est le cas, l'invariant $R_j$ est nécessairement préservé puisque, par définition des invariants de raffinement (propriété 12), on a $J_{M_i} \Rightarrow R_{M_k}$ pour toute machine $M_k$ telle que $M_k \in combines^+(M_i)$. Si $M_j \notin combines^+(M_i)$, on a nécessairement $x_j \cap x_i = \emptyset$, puisque l'introduction de partage ne se fait que par partage de machine. On se trouve alors dans les hypothèses du théorème 9, puisque les variables $x_j$ et $x_i$ sont disjointes. Il suffit de vérifier l'hypothèse 2 de ce théorème, c'est-à-dire :

$$\begin{array}{c} (env\ x_j\ .\ \mathsf{skip}) \\ \sqsubseteq_{R_j} \\ embed((env\ y_i \cap y_j\ .@\ y_i - y_j\ .\ \exists x_i\ (\mathsf{trm}(S) \wedge J_i) \Longrightarrow\ op'_i), y_j - y_i) \end{array}$$

Cette propriété est, par exemple, trivialement vraie si $op'_i$ est une opération de lecture sur les variables $y_i \cap y_j$. Pour garantir par construction cette hypothèse, la méthode B impose des conditions architecturales sur les développements. Ces conditions sont introduites dans le B-Book et on été étendues dans [PR98], afin d'éliminer toutes les configurations invalides. Ces

restrictions sont les suivantes :

| | |
|---|---|
| ($Restri1$) | Une machine est importée au plus une fois dans un développement. |
| ($Restri2$) | Lorsqu'une implémentation $P$ importe ou voit les machines $M_1, \ldots, M_n$ alors les implémentations de ces machines ne doivent pas importer les machines $M_1, \ldots, M_n$ qui sont vues dans $P$. |

Ces contraintes structurelles vont garantir qu'aucune variable concrète ne peut à la fois représenter les variables $x_j$ et être modifiée dans $op'_i$ d'une manière observable par l'invariant $J_j$. Elles permettent donc de garantir la condition 2 du théorème 9 par construction. La preuve est détaillée dans [Pot03].

Les restrictions architecturales sont des conditions suffisantes. Elles éliminent de manière statique les configurations potentiellement non valides. Dans la pratique ces restrictions ne s'avèrent pas trop contraignantes. Elles permettent par exemple de partager des opérations qui modifient des variables communes, à condition que ce partage ne remonte pas au niveau des spécifications. Le théorème 9 est plus puissant que les restrictions architecturales. Il peut permettre de garantir, par la preuve, plus de raffinements.

### 4.4.3 En résumé

Dans la première partie de ce chapitre nous avons décrit différents aspects liés à la mise en œuvre de la compositionnalité dans un processus incrémental de développement ou de validation. Dans la section 4.1.2 nous nous sommes intéressés à la pertinence de ce principe, vis-à-vis de la preuve. Dans les sections 4.1.3 et 4.1.4 avons avons étudié le principe du raffinement compositionnel, ainsi que les restrictions théoriques et méthodologiques. Finalement, dans la section 4.1.5, nous avons caractérisé les différentes possibilités attendues des langages et environnements supportant une telle approche. La seconde partie a été dédiée au développement incrémental dans la méthode B, et à la compositionnalité des preuves. Cette partie décrit de plus le cadre

théorique validant les principes adoptés. La conclusion de ce chapitre pourrait donc être une évaluation des possibilités offertes par la méthode B, au vu des critères mis en évidence. Néanmoins ceci n'a pas encore vraiment de sens. En effet, du point de vue pratique, il n'y a eu encore que peu de gros développements mettant fortement en jeu les compositions horizontale et verticale, que ce soit avec la méthode B ou avec d'autres approches. D'autre part, comme nous l'avons évoqué, certains choix sont d'ordre méthodologique, comme par exemple le compromis entre la puissance des concepts offerts aux utilisateurs et ce qui est pris en charge par les outils. Il semble donc trop tôt pour pouvoir disposer de critères indiscutables, permettant de juger de la pertinence d'une approche. Il est néanmoins possible de donner quelques éléments d'évaluation, intrinsèques à l'approche choisie.

Globalement la méthode B offre un bon niveau de compositionnalité, que ce soit au niveau de la conception que du processus de validation par la preuve. La notion de composant, bien qu'assez classique, offre la notion d'invariant qui permet d'exprimer des propriétés des données, associées à une interface déterminée. Le principe de composition horizontale (section 4.3) permet de partager des spécifications et de composer les propriétés invariantes. La restriction la plus pénalisante semble être le principe un écrivain/plusieurs lecteurs, qui impose de construire des architectures dans lesquelles les modifications attachées à un jeu de variables sont localisées dans un composant. Remarquons que ce n'est pas le partage en écriture qui pose problème en lui-même, mais le fait d'énoncer localement des propriétés sur des variables partagées (ceci correspond exactement au problème d'interférence dans les systèmes concurrents [OG76]). Le choix retenu dans la méthode B, de pouvoir composer les preuves d'invariants indépendemment d'un contrôle particulier, c'est-à-dire sans raisonner sur les interférences possibles, rend difficile la conception d'architectures répondant aux restrictions. Les difficultés de mise en œuvre sont à peu près analogues à celles rencontrées lors de la vérification modulaire, où l'un des problèmes est de découper une propriété globale en différentes propriétés, vérifiables sur les sous-composants. Il semble que la restriction un écrivain/plusieurs lecteurs devrait être assouplie. Plusieurs pistes sont possibles, qui nécessitent soit d'abandonner en partie le principe de compositionnalité des preuves d'invariant soit de mieux localiser les partages, afin d'introduire la possibilité de raisonner localement sur les interférences.

Dans la méthode B, la composition des raffinements n'intervient qu'au niveau des implémentations et nécessite d'avoir fixé les interfaces, c'est-à-dire les opérations et leur profil. Le point de vue adopté dans la méthode B est donc un point de vue adapté aux phases de conception détaillée ou de codage. Les machines abstraites correspondent aux spécifications, et les implémentations au code. Les raffinements sont des pas intermédiaires de la construction du code. Les raffinements et les implémentations ne sont donc pas visibles, en tant que spécifications. Ces restrictions syntaxiques sont limitatives car elles imposent de décrire, dès le niveau le plus abstrait, toutes les informations attendues par les autres machines et leur développement [32]. Néanmoins ces restrictions, bien qu'incontournables au niveau des outils, ne sont que syntaxiques. Le principe du raffinement modulaire, comme décrit dans la section 4.4.1, permet de construire les composants correspondant aux implémentations, ainsi que les composants résultants de la composition des implémentations. Les restrictions architecturales liées au raffinement ne semblent pas poser de problème en elles-mêmes. Elles sont moins restrictives que celles liées aux preuves d'invariant, puisque des implémentations peuvent partager, dans une certaine mesure, des opérations de modification. La restriction *Restri*2, qui peut sembler *a priori* arbitraire, ne s'est pas avérée génante dans la pratique. Cette restriction élimine des cas dans lesquels le partage n'est, d'une certaine manière, pas bien maîtrisé par le spécifieur. Le problème peut se poser dans un processus de composition, c'est-à-dire lorsqu'on assemble des spécifications et des développements construits indépendemment. En phase de décomposition il est toujours possible de maîtriser le partage (ceci correspond par exemple à la notion de sous-composants de L. Lamport). Par exemple, les développements industriels ont, en général, leurs propres guides méthodologiques sur la manière de développer les modèles. Dans le cas de la méthode B, ces guides excluent par construction tous les cas pathologiques correspondant à la restriction *Restri*2.

Du point de vue du processus de validation par la preuve (section 4.1.2), la méthode B permet de mettre en place des structurations qui facilitent la maîtrise des preuves. La construction en couche permet de s'abstraire du contexte

32. Ceci est renforcé par les restrictions sur les substitutions en fonction des différents types de composant.

global d'un développement, par un processus de composition. Comme nous l'avons vu, ceci permet de réutiliser des développements, et donc des preuves, et de s'abstraire des détails non significatifs. Les préconditions, qui peuvent introduire des contraintes sur les variables partagées, permettent de prendre en compte, en partie, des hypothèses sur le contexte d'utilisation d'un composant. Le raffinement est aussi un outil puissant permettant de maîtriser la preuve, en construisant le modèle par niveaux successifs. Néanmoins, dans la pratique, construire une architecture demande une bonne expérience méthodologique. En effet, la structuration d'un développement B est le résultat d'un compromis entre l'architecture fonctionnelle du modèle et la structuration liée à la preuve. On retrouve ici la difficulté inhérente à l'activité de preuve, accrue par le fait qu'il faut aussi prendre en compte l'architecture logicielle et se plier aux différentes restrictions imposées par la méthode. La méthodologie doit donc être développée par exemple en fonction des domaines d'applications.

Nous terminons en résumant rapidement nos différents travaux sur la compositionnalité en B, ainsi que des travaux abordés par d'autres auteurs.

Dans [BPR96] nous nous sommes intéressés à la primitive USES du langage B, qui permet d'introduire une certaine forme de partage au niveau des spécifications. Nous avons repris cette construction afin d'obtenir à chaque pas des composants B validés, ce qui a nécessité de modifier les obligations de preuve associées à cette construction. Nous avons aussi défini une algèbre de composants, adaptés aux composants B. Ce travail a permis de donner une sémantique aux différentes clauses de la méthode B, en terme de construction de composants équivalents, et de justifier les restrictions et les obligations de preuve associées à ces clauses. Le chapitre *Composition des machines et des raffinements* du livre *Spécification formelle avec B* [Hab01] présente, à travers les exemples, les différentes clauses d'assemblage de la méthode B et la manière de les utiliser. Des formes de structuration sont proposées pour développer certaines formes d'architecture, tout en respectant les restrictions imposées.

Dans [PR98] et [Rou99] nous avons étudié la monotonie du raffinement, par rapport aux clauses d'assemblage SEES et IMPORTS. Nous avons intro-

duit le principe de composition des implémentations, comme décrit dans la section 4.4.1, introduit une condition suffisante (analogue à la condition 2 du théorème 9, p. 121) et proposer des restrictions vérifiables sur la structure des développements B. Ces conditions ont été implantées dans l'AtelierB (version 3.5). Dans [BP00], nous avons rendu effective la théorie sous-jacente à la composition des raffinements, en proposant des constructions syntaxiques, et validées, des composants déductibles de la structuration des développements. Ces constructions correspondent à celles que nous avons décrites dans la section 4.4.1, étendues à tout niveau de raffinement. Elles on été implantées, en partie, dans un outil. La correction de ces composants nécessite d'introduire certaines restrictions afin d'assurer le principe de subtitutivité, d'étendre les restrictions architecturales aux compositions envisagées et d'étudier les propriétés de compositionnalité du raffinement vis-à-vis de l'opération ||. L'objectif pratique est de pouvoir exhiber les spécifications globales, ainsi que les propriétés des variables, correspondant à différents niveaux de raffinement. En plus de pouvoir visualiser de telles spécifications, le but est de pouvoir énoncer et valider des propriétés sur ces spécifications intermédiaires. Cette possibilité peut s'avérer intéressante en particulier pour vérifier, *a posteriori*, des propriétés. L'intérêt pratique est en cours d'évaluation. Enfin, dans [Pot03], les différents résultats sur la composition des preuves d'invariant et de raffinement ont été repris et complétés.

Dans [BB99], M. Büchi et R. Back proposent une extension des mécanismes de composition en B, afin d'admettre certaines formes de partage en écriture entre les spécifications. Cette extension est basée sur la notion de rôles, proche de la notion de *rely/guarantee* proposée par Cliff Jones [Jon83]. Cette extension assouplit les contraintes actuelles de la méthode B et permet, dans certains cas, de composer des spécifications modifiant des variables communes. La contrepartie est que le partage soit bien maîtrisé lors du raffinement. M. Büchi et R. Back étendent les conditions que nous avons proposées dans [PR98], pour garantir la monotonie du raffinement. Ces règles s'avèrent assez complexes et il faudrait pouvoir évaluer l'intérêt des architectures finalement autorisées. Dans [DBMM00], les auteurs s'intéressent à la préservation de la consistance interne, par les primitives de composition de la méthode B. Les preuves de consistance interne, non introduite dans le B-Book, ont été proposées par K. Lano [Lan96]. Elles permettent, dès la spécification, de vérifier qu'il existe des instanciations des paramètres et des

constantes ayant les propriétés attendues. Dans le B-Book ces contraintes sont vérifiées lors de la valuation effective des paramètres et des constantes. Faire ces preuves en amont peut permettre de vérifier le plus tôt possible si les propriétés demandées sont cohérentes. Les auteurs proposent de nouvelles obligations de preuve, liées aux clauses d'assemblage, permettant de garantir la consistance interne de manière incrémentale.

Finalement des études sont en cours pour proposer une approche de la décomposition, dans le cadre du B événementiel[33]. Le principe retenu est, *a priori*, proche du principe de décomposition proposé par L. Lamport (section 4.1.2), qui permet de raffiner un sous-composant en exploitant les propriétés des autres sous-composants. En effet, lorsqu'on s'intéresse à la modélisation de comportement, les propriétés sont généralement intrinsèquement liées aux comportements. Il est donc nécessaire de connaître, au moins en partie, les propriétés garanties par les autres composants. L'approche par décomposition permet, de plus, d'offrir des mécanismes plus fins pour maîtriser le partage, qui, comme nous l'avons vu, est un point délicat du raisonnement compositionnel. On peut, par exemple, imposer de localiser le partage dans une même spécification, afin de pouvoir vérifier et utiliser localement les propriétés globales de ces variables. Du point de vue langage, des expérimentations sont donc actuellement en cours pour définir un mécanisme de décomposition. Le but est de préciser la forme des interfaces admises entre les sous-composants, les possibilités de raffinement et la mise en œuvre dans l'outil. Ce nouveau principe de compositionnalité devrait aussi être applicable à l'approche B classique, relative au développement logiciel. L'approche par composition a aussi son sens dans le cas du B événementiel, mais ceci change le niveau de complexité des spécifications. Elles doivent alors être exprimées sous une forme *rely-garantee*, afin de préciser les interférences possibles entre les composants et leur environnement. La compositionnalité dans toute sa généralité pose des problèmes intéressants, mais ne semble pas encore vraiment à l'ordre du jour, en raison de la difficulté de sa mise en œuvre, et du fait que les besoins ne sont pas encore à la réutilisation de développements formels. Peut-être le deviendront-ils ?

33. Ces études sont principalement menées par Jean-Raymond Abrial.

# Conclusion

L'évolution des techniques et des besoins a permis l'émergence du logiciel dans des systèmes de taille conséquente, et qui mettent en jeu des fonctionnalités complexes. Cette évolution est à la fois liée aux demandes technologiques, sans cesse croissantes, et aux particularités du logiciel, qui en font une technique qui offre une grande souplesse et qui, d'une certaine manière, peut s'avérer bon marché. En effet il n'y a pas de coût de fabrication, donc pas de dépendance vis-à-vis d'un fournisseur de matériel, et pas d'usure. D'autre part un logiciel, est, a priori, un produit adaptable, c'est-à-dire qui peut évoluer sans remettre en cause la technologie utilisée. La contrepartie est bien entendu la difficulté et les coûts potentiels liés à la conception.

Les premières évolutions techniques ont été le passage de la programmation empirique à la programmation structurée et aux méthodologies de programmation. Cette évolution s'est accompagnée du développement d'outils supportant ces concepts, c'est-à-dire des langages de programmation et des environnements associés. L'étape d'après, qui a permis un second passage à l'échelle, a été l'évolution de la maîtrise des aspects fonctionnels à l'introduction des aspects structuraux. Un logiciel est maintenant développé à l'aide d'entités logicielles (modules, objet ou composant) encapsulant des données et des traitements. Une dimension supplémentaire est relative à l'organisation de ces entités : l'architecture et les communications entre ces entités. L'importance de l'aspect structuration se retrouve par exemple dans les différents diagrammes proposés par l'approche UML, comme les diagrammes de classes, les diagrammes de déploiement ou de collaboration. La technologie associée à cette évolution est le domaine de l'architecture logicielle et le développement de middlewares supportant une telle approche. Ces évolutions

permettent d'envisager une étape dans la technologie du développement de logiciels : il devient possible de construire des systèmes par assemblage de composants pré-existants et de modifier ou de faire évoluer les systèmes en remplaçant des sous-parties, sans remettre en cause la structuration globale.

Le développement des méthodes formelles s'inscrit dans ces évolutions. En effet, les méthodes formelles ont déjà montré leur applicabilité à des développements *in the small* et, pour certains types d'applications, pour des systèmes de taille moyenne. C'est par exemple le cas dans le domaine de la vérification de protocoles et dans le domaine de la vérification hardware. Les logiciels ferroviaires développés à l'aide de la méthode B ont aussi montré l'applicabilité d'un processus formel de construction à des logiciels déjà complexes. Néanmoins le passage à l'échelle nécessite de développer des concepts et des techniques adaptées aux approches basées composants. On retrouve bien sûr les mêmes besoins de modélisation, de raisonnement formel, et de techniques de développement vers du code correct. Les particularités sont relatives au fait que l'objectif est maintenant de passer de la maîtrise des aspects fonctionnels à la maîtrise des aspects structuraux.

Les spécifications doivent donc maintenant permettre de modéliser d'une part des entités logicielles, comme des objets ou des composants adaptables et réutilisables, en incluant leur mode d'interaction avec l'extérieur, et de modéliser d'autre part le comportement global des systèmes, c'est-à-dire la manière dont ces entités collaborent. La seconde difficulté est de pouvoir raisonner sur ces systèmes en exploitant leur structuration, afin de rendre possible le processus de vérification formelle. Ceci nécessite de pouvoir découpler, au moins en partie, la structuration des aspects fonctionnels. La difficulté repose sur le fait que, dans le cas du logiciel, les communications ne se limitent pas à mettre des fils entre des composants, mais aussi à raisonner sur l'information qui circule dans ces fils, information qui peut être complexe. La possibilité de pouvoir raisonner finement sur les communications, tout en ayant un bon niveau d'abstraction sur les aspects fonctionnels, est un des verrous actuels. Il fait d'ailleurs se rejoindre les approches de vérification et de développement formel, puisqu'il devient indispensable de raisonner sur des abstractions et de développer des techniques de vérification dépendantes d'une structuration imposée. On rejoint alors le problème de la composition-

nalité du raisonnement formel, qui permet de découper une propriété globale ou de recomposer des propriétés locales, en fonction de l'architecture des systèmes.

Il est bien évident que le fait de s'attaquer à des systèmes de plus grande complexité fait que les méthodes formelles vont buter sur certains verrous, soit du point de vue théorique, soit du point de vue de la complexité des outils. La question de leur applicabilité peut alors se poser. Néanmoins, une des facettes de la complexité est l'accumulation de traitements simples. L'intérêt des outils n'est pas nécessairement de réaliser des tâches intrinsèquement complexes mais de maîtriser des tâches relativement simples, rendues complexes par leur répétitivité. Les difficultés et les erreurs sont souvent liées à l'accumulation de détails simples, qui peuvent être en grande partie traités automatiquement. Ceci signifie que les techniques formelles actuelles, étendues de manière à passer à l'échelle des applications, seront a priori applicables et utiles. Pour les autres facettes de la complexité, reste à faire la part entre ce qui pourra être pris en charge par des outils, et ce qui restera à charge des développeurs. Ces études vont bien entendu dépendre des types de systèmes considérés, et des exigences formelles visées.

Les travaux que j'ai développés et présentés dans ce manuscrit s'inscrivent, à leur niveau, dans cette évolution. Dans le cadre de la théorie du raffinement, et dans une approche basée sur la programmation séquentielle, nous avons étudier le principe de compositionnalité des différents constructeurs du langage des substitutions généralisées de la méthode B, ainsi que les propriétés de compositionnalité de l'assemblage de composants. L'assemblage de composants permet de bâtir de nouveaux composants, en s'appuyant sur des composants déjà existants, validés et implémentés. Comme nous l'avons vu, le principe adopté dans la méthode B est, du point de vue des preuves, équivalent à la composition des traitements par entrelacement. Il n'est donc pas restreint au cas de la programmation séquentielle. Le travail présenté ici montre que l'étude de la compositionnalité est une tâche délicate qui, de plus, peut sembler fournir des résultats laborieux. Elle nécessite de restreindre à la fois les propriétés qui peuvent être composées et la forme des assemblages. Ces difficultés sont néanmoins inhérentes à la compositionnalité, particulièrement dans les langages basés sur la notion d'état, qui, par définition, n'est

pas une notion modulaire. Toute approche doit maîtriser le partage et les possibilités d'aliasing. Des problèmes similaires sont par exemple rencontrés dans les approches objets où le raisonnement formel nécessite de restreindre les possibilités offertes par le langage.

Les difficultés propres posées par la méthode B sont liées, au niveau des invariants, à la généralité de ces propriétés. En effet les invariants sont attachés à des jeux d'opérations, ce qui correspond à un niveau de granularité d'observation des états. Ce choix nécessite donc de maîtriser finement les comportements qui peuvent être entrelacés. Dans le cas du raffinement, les possibilités offertes par la méthode B sont, au vu des résultats actuels de la théorie du raffinement, un bon compromis entre la pratique et les difficultés de mise en œuvre. En effet, en dehors des restrictions liées aux invariants, il est possible d'introduire du partage au cours du raffinement, tant que ce partage reste compatible avec la composition des spécifications. Les restrictions introduites par la méthode B portent sur la souplesse d'utilisation du raffinement, au niveau des composants. En effet, l'approche retenue encode de manière très syntaxique les différents types de composants, et seules les machines correspondent réellement à des spécifications à part entière. Finalement, le dernier choix propre à cette méthode est d'adopter le point de vue que la compositionnalité doit se faire de manière transparente pour l'utilisateur, c'est-à-dire n'impliquer aucune vérification supplémentaire, autre que structurelle. Une des implications de ce choix est bien sûr de maîtriser le processus de preuves, qui reste ainsi linéaire. Par contre les utilisateurs doivent développer des architectures qui respectent les contraintes structurelles, ce qui n'est pas toujours aisé.

Les développements futurs des travaux que nous avons présentés dans ce manuscrit s'inscrivent dans les perspectives des évolutions générales évoquées précédemment. Du point de vue de la théorie du raffinement, l'étude d'opérateurs permettant de fusionner des spécifications doit être développée, en considérant à la fois les propriétés de ces opérateurs en tant que transformateurs de prédicats, et leur intérêt du point de vue de la spécification et de la structuration. De la même manière la notion de raffinement appliquée à des entités logicielles plus complexes doit être développée. Un développement intéressant consiste à étendre la notion de machine à état, telle que proposée

dans la méthode B, pour aller vers des entités correspondant à des objets. La difficulté est alors de maîtriser le partage de manière moins statique que celle adoptée dans la méthode B, qui consiste à vérifier des contraintes structurelles sur la structuration figée des composants. Une autre dimension consiste à intégrer plus fortement la structuration et le raffinement afin d'une part de proposer des possibilités de raffinement plus souples que la simple compositionnalité et afin, d'autre part, de pouvoir raisonner sur la structuration, sans rentrer dans les détails des composants. Ceci nécessite de modéliser la structuration en décrivant les relations d'assemblage et de raffinement entre les composants. Ces études doivent être menées en lien avec le développement de la méthodologie sous-jacente à la construction de logiciels à base de composants, afin de combiner les décisions de conception et les raisonnements sous-jacents.

# Annexes

# Annexe A

# Complément sur les substitutions généralisées

## A.1 Prédicat trm et prd

### A.1.1 Substitutions mathématiques

$$
\begin{array}{lcll}
\mathsf{prd}_x(x := E) & \Leftrightarrow & x' = E & \\
\mathsf{prd}_{x,y}(x := E) & \Leftrightarrow & x', y' = E, y & \\
\mathsf{prd}_x(\mathsf{skip}) & \Leftrightarrow & x' = x & \\
\mathsf{prd}_x(P \mid S) & \Leftrightarrow & P \Rightarrow \mathsf{prd}_x(S) & \\
\mathsf{prd}_x(S \,[\!]\, T) & \Leftrightarrow & \mathsf{prd}_x(S) \ \vee \ \mathsf{prd}_x(T) & \\
\mathsf{prd}_x(P \Longrightarrow S) & \Leftrightarrow & P \ \wedge \ \mathsf{prd}_x(S) & \\
\mathsf{prd}_x(@z \cdot S) & \Leftrightarrow & \exists z \cdot \mathsf{prd}_x(S) & \text{si } z \backslash x' \quad (*) \\
\mathsf{prd}_x(@z \cdot S) & \Leftrightarrow & \exists (z, z') \cdot \mathsf{prd}_{x,z}(S) & \text{si } z \backslash x' \quad (**)
\end{array}
$$

Le prédicat avant-après d'une substitution de choix non borné "$@z \cdot S$" est divisé en deux : dans le cas marqué (*), la variable $z$ n'est pas modifiée par $S$. Dans le cas marqué (**), la variable $z$ est modifiée dans $S$. Le prédicat avant-après de la substitution $S$ porte donc aussi sur $z$.

$$
\begin{array}{lcl}
\mathsf{trm}(x := E) & \Leftrightarrow & true \\
\mathsf{trm}(\mathsf{skip}) & \Leftrightarrow & true \\
\mathsf{trm}(P \mid S) & \Leftrightarrow & P \ \wedge \ \mathsf{trm}(S) \\
\mathsf{trm}(S \ [\!] \ T) & \Leftrightarrow & \mathsf{trm}(S) \ \wedge \ \mathsf{trm}(T) \\
\mathsf{trm}(P \Longrightarrow S) & \Leftrightarrow & P \Rightarrow \mathsf{trm}(S) \\
\mathsf{trm}(@z \cdot S) & \Leftrightarrow & \forall z \cdot \mathsf{trm}(S)
\end{array}
$$

### A.1.2 Séquencement

$$
\mathsf{trm}(S \ ; T) \quad \Leftrightarrow \quad (\mathsf{trm}(S) \ \wedge \ \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']\,\mathsf{trm}(T)))
$$

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

### A.1.3 Itération

Le prédicat de terminaison doit contenir l'information que l'invariant de boucle est préservé à chaque pas :

$$
\begin{array}{l}
\mathsf{trm}\ (\textsc{while}\ P\ \textsc{do}\ S\ \textsc{invariant}\ I\ \textsc{variant}\ V\ \textsc{end}) \\
\Leftrightarrow \\
\quad I \ \wedge \\
\quad \forall x \cdot (I \ \wedge \ P \ \Rightarrow \ [S]\,I) \ \wedge \\
\quad \forall x \cdot (I \ \Rightarrow \ V \in \mathbb{N}) \ \wedge \\
\quad \forall x \cdot (I \ \wedge \ P \ \Rightarrow \ [n := V][S](V < n))
\end{array}
$$

Le prédicat $\mathsf{prd}_x$ donné ci-dessous ne correspond pas exactement à celui donné dans le B-Book. Dans ce dernier la condition $[x := x'](I \ \wedge \ \neg P)$ n'est pas conditionnée par la terminaison. En toute rigueur cette condition

est nécessaire. La formule ci-dessous correspond exactement à l'instanciation de la définition générale du prédicat $\mathsf{prd}_x$ ($\mathsf{prd}_x(S) = \neg[S](x' \neq x)$).

$$
\begin{array}{l}
\mathsf{prd}_x\ (\text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END}) \\
\Leftrightarrow \\
\quad (\mathsf{trm}\ (\text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END}) \\
\quad\ \Rightarrow [x := x']\,(I\ \wedge\ \neg\, P))
\end{array}
$$

## A.2 Élimination du séquencement

Les égalités ci-dessous vont permettre d'éliminer le séquencement, lorsqu'il se combine avec les substitutions mathématiques (B-Book pages 375 et 376) :

| | | | |
|---|---|---|---|
| 1) | $\mathsf{skip}\ ;\ S$ | $=$ | $S$ |
| 2) | $(P \mid S)\ ;\ T$ | $=$ | $P \mid (S\ ;\ T)$ |
| 3) | $(P \Longrightarrow S)\ ;\ T$ | $=$ | $P \Longrightarrow (S\ ;\ T)$ |
| 4) | $(S \,[\!]\, T)\ ;\ U$ | $=$ | $(S\ ;\ U) \,[\!]\, (T\ ;\ U)$ |
| 5) | $(@z \cdot S)\ ;\ T$ | $=$ | $@z \cdot (S\ ;\ T) \quad if\ z \backslash T$ |
| 6) | $S\ ;\ \mathsf{skip}$ | $=$ | $S$ |
| 7) | $S\ ;\ (P \mid T)$ | $=$ | $[S]P \mid (S\ ;\ T)$ |
| 8) | $(x := E)\ ;\ (P \Longrightarrow T)$ | $=$ | $[x := E]P \Longrightarrow (x := E\ ;\ T)$ |
| 9) | $S\ ;\ (T \,[\!]\, U)$ | $=$ | $(S\ ;\ T) \,[\!]\, (S\ ;\ U)$ |
| 10) | $S\ ;\ @z \cdot T$ | $=$ | $@ \cdot (S\ ;\ T) \quad if\ z \backslash S$ |

L'application des règles 1 à 5 permet de ramener un sous-terme de la forme $S\ ;\ T$ à un sous-terme de la forme $x := e\ ;\ T$. La règle 8 devient donc appliquable puisque la première substitution est bien une affectation. Les règles 6 à 10 permettent alors de se ramener à un sous-terme de la forme $x := e\ ;\ y := f$. Il reste à éliminer le séquencement lorsqu'il porte sur deux

affectations. On peut ajouter la règle suivante :

$$x := e \ ; \ x := f \quad = \quad x := [x := e]f$$

Cette égalité peut être établie en prouvant $[x := e \ ; \ x := f]R \Leftrightarrow [x := [x := e]f]R$ pour tout $R$, par induction sur $R$. L'application de cette règle nécessite que les n-uplets en partie gauche des deux affectations soient identiques (à l'ordre près). Il est toujours possible d'étendre une affectation de la forme $x := e$ en $x, y := e, y$ avec $x$ et $y$ des n-uplets disjoints.

## A.3 Preuves des propriétés de l'opérateur ||

Nous montrons ici que les définitions des prédicats $\mathsf{trm}$ et $\mathsf{prd}$ sont cohérentes avec les égalités définissant l'opérateur || (chapitre 3, section 3.4.1, page 58). L'égalité entre substitutions $S_1$ et $S_2$ va être établie en montrant :

$$\begin{array}{ll} (a) & \mathsf{trm}(S_1) \Leftrightarrow \mathsf{trm}(S_2) \\ (b) & \mathsf{trm}(S_1) \Rightarrow (\mathsf{prd}(S_1) \Leftrightarrow \mathsf{prd}(S_2)) \end{array}$$

En effet, sous ces hypothèses on a $[S_1]R \Leftrightarrow [S_2]R$ :

$$\begin{array}{lcll} [S_1]R & \Leftrightarrow & \mathsf{trm}(S_1) \wedge \forall x' \,.\, (\mathsf{prd}_x(S_1) \Rightarrow [x := x']R) & \textit{propriété 1 chapitre3} \\ & \Leftrightarrow & \mathsf{trm}(S_1) \wedge \forall x' \,.\, (\mathsf{prd}_x(S_2) \Rightarrow [x := x']R) & (b) \\ & \Leftrightarrow & \mathsf{trm}(S_2) \wedge \forall x' \,.\, (\mathsf{prd}_x(S_2) \Rightarrow [x := x']R) & (a) \\ & \Leftrightarrow & [S_2]R & \end{array}$$

### A.3.1 Cas sans partage

La preuve se fait par cas sur les différentes égalités. Pour toutes les égalités on désignera par $S_1$ le terme de gauche et par $S_2$ le terme de droite. De plus pour toute instance de $S_1$ de la forme $T \ || \ S$ on suppose que $T$ est définie sur l'espace de variables $w$ et $S$ sur l'espace de variables $z$, ces espaces étant disjoints ($z \cap w = \emptyset$).

**Egalité** (1) $x := e \parallel y := f \;=\; x, y := e, f$

On pose $u = w - x$ et $v = z - y$, qui correspondent respectivement aux variables de la première et de la seconde affectation, qui ne sont pas modifiées. On a :

$$\begin{array}{lcl} \mathsf{trm}(S_1) & \Leftrightarrow & \mathsf{trm}(x := e) \wedge \mathsf{trm}(y := f) \\ \mathsf{trm}(S_2) & \Leftrightarrow & \mathsf{trm}(x, y := e, f) \end{array}$$

Ces formules se réduisent toutes deux à $true$.

$$\begin{array}{lcl} \mathsf{prd}_{z \cup w}(S_1) & \Leftrightarrow & (x' = e \wedge u' = u) \wedge (y' = f \wedge v' = v) \\ \mathsf{prd}_{z \cup w}(S_2) & \Leftrightarrow & x', y' := e, f \wedge u', v' = u, v \end{array}$$

Ces deux prédicats sont équivalents.

**Egalité** (2) $\mathsf{skip} \parallel S \;=\; S$

| | | |
|---|---|---|
| $\mathsf{trm}(S_1)$ | $\Leftrightarrow$ | $true \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{z \cup w}(S_1)$ | $\Leftrightarrow$ | $w' = w \wedge \mathsf{prd}_z(S)$ |

On a bien $\mathsf{trm}(S_1) \Leftrightarrow \mathsf{trm}(S)$. Il reste à montrer :

$$\mathsf{trm}(S_1) \Rightarrow (w' = w \wedge \mathsf{prd}_z(S) \Leftrightarrow \mathsf{prd}_{z \cup w}(S))$$

Cette équivalence peut être établie par récurrence sur la substitution $S$, sous l'hypothèse que $S$ ne modifie pas les variables $w$. Cette propriété découle de la définition $\mathsf{prd}_{x,y}(x := e) \Leftrightarrow x', y' = e, y$ (section A.1.1).

**Egalité** (3) $(P \mid T) \parallel S \;=\; P \mid (T \parallel S)$

| | | |
|---|---|---|
| $\mathsf{trm}(S_1)$ | $\Leftrightarrow$ | $\mathsf{trm}(P) \wedge \mathsf{trm}(T) \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_2)$ | $\Leftrightarrow$ | $\mathsf{trm}(P) \wedge \mathsf{trm}(T) \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{z \cup w}(S_1)$ | $\Leftrightarrow$ | $(P \Rightarrow \mathsf{prd}_w(T)) \wedge (P \Rightarrow \mathsf{prd}_z(S))$ |
| $\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{z \cup w}(S_2)$ | $\Leftrightarrow$ | $P \Rightarrow (\mathsf{prd}_w(T) \wedge \mathsf{prd}_z(S))$ |

Dans les deux cas, les prédicats sont équivalents.

**Egalité** (4) $(T \llbracket U) \mid\mid S = (T \mid\mid S) \llbracket (U \mid\mid S)$

| | | |
|---|---|---|
| $\mathsf{trm}(S_1)$ | $\Leftrightarrow$ | $\mathsf{trm}(T) \wedge \mathsf{trm}(U) \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_2)$ | $\Leftrightarrow$ | $\mathsf{trm}(T) \wedge \mathsf{trm}(S) \wedge \mathsf{trm}(U) \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{z \cup w}(S_1)$ | $\Leftrightarrow$ | $(\mathsf{prd}_w(T) \vee \mathsf{prd}_w(U)) \wedge \mathsf{prd}_z(S)$ |
| $\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{z \cup w}(S_2)$ | $\Leftrightarrow$ | $(\mathsf{prd}_w(T) \wedge \mathsf{prd}_z(S)) \vee (\mathsf{prd}_w(U) \wedge \mathsf{prd}_z(S))$ |

Dans les deux cas, les prédicats sont équivalents.

**Egalité** (5) $(P \Longrightarrow T) \mid\mid S = \mathsf{trm}(S) \mid P \Longrightarrow (T \mid\mid S)$

| | | |
|---|---|---|
| $\mathsf{trm}(S_1)$ | $\Leftrightarrow$ | $(P \Rightarrow \mathsf{trm}(T)) \wedge \mathsf{trm}(S)$ |
| $\mathsf{trm}(S_2)$ | $\Leftrightarrow$ | $\mathsf{trm}(S) \wedge (P \Rightarrow \mathsf{trm}(T) \wedge \mathsf{trm}(S))$ |
| $\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{z \cup w}(S_1)$ | $\Leftrightarrow$ | $P \wedge \mathsf{prd}_w(T) \wedge ((P \Rightarrow \mathsf{trm}(T)) \Rightarrow \mathsf{prd}_z(S))$ |
| $\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{z \cup w}(S_2)$ | $\Leftrightarrow$ | $P \wedge \mathsf{prd}_w(T) \wedge (\mathsf{trm}(T) \Rightarrow \mathsf{prd}_z(S))$ |

$P \Rightarrow \mathsf{trm}(T)$ se déduit de $\mathsf{trm}(S_1)$. Ceci permet de ramener les deux prédicats $\mathsf{prd}_{z \cup w}(S_1)$ et $\mathsf{prd}_{z \cup w}(S_2)$ à la formule $P \wedge \mathsf{prd}_w(T) \wedge \mathsf{prd}_z(S)$.

**Egalité** (6) $(@x \cdot T) \mid\mid S = @x \cdot (T \mid\mid S)$

La condition associée à cette égalité est $x$ non libre dans $S$.

| | | |
|---|---|---|
| $\mathsf{trm}(S_1)$ | $\Leftrightarrow$ | $\mathsf{trm}(S) \wedge \forall x \,.\, \mathsf{trm}(T)$ |
| $\mathsf{trm}(S_2)$ | $\Leftrightarrow$ | $\forall x \,.\, (\mathsf{trm}(T) \wedge \mathsf{trm}(S))$ |
| $\mathsf{trm}(S_1) \Rightarrow \mathsf{prd}_{z \cup w}(S_1)$ | $\Leftrightarrow$ | $(\exists x\ \mathsf{prd}_w(T)) \wedge \mathsf{prd}_z(S)$ |
| $\mathsf{trm}(S_2) \Rightarrow \mathsf{prd}_{z \cup w}(S_2)$ | $\Leftrightarrow$ | $\exists x\ (\mathsf{prd}_w(T) \wedge \mathsf{prd}_z(S))$ |

Dans les deux cas ces formules sont équivalentes, puisque $z$ est non libre dans $S$. Ceci termine la preuve.

### A.3.2 Cas avec partage en lecture

**Egalité** (1) $x := e \parallel y := f \;=\; x, y := e, f$

Soient $S_1 = x := e \parallel y := f$ et $S_2 = x, y := e, f$. On suppose que l'affectation $x := e$ est définie pour l'espace de variables $\{x, z, u\}$ et que l'affectation $y := f$ est définie pour l'espace de variables $\{y, w, u\}$, avec $x \notin u \cup z \cup w$ et $y \notin u \cup z \cup w$ et $z \cap w = \emptyset$.

$$\begin{array}{lcl} \mathsf{prd}_{u \cup x \cup z \cup y \cup w}(S_1) & \Leftrightarrow & (x' = e \wedge z' = z \wedge u' = u) \wedge (y' = f \wedge w' = w \wedge u' = u) \\ \mathsf{prd}_{u \cup x \cup z \cup y \cup w}(S_2) & \Leftrightarrow & x', y' := e, f \wedge z', w', u' = z, w, u \end{array}$$

Ces deux prédicats sont équivalents.

**Egalité** (2) $\mathsf{skip} \parallel S \;=\; S$

On suppose que dans la substitution $\mathsf{skip} \parallel S$, $S$ est définie sur l'espace de variables $z \cup u$ et $\mathsf{skip}$ sur $w \cup u$. On a :

$$\mathsf{trm}(S) \Rightarrow \mathsf{prd}_{z \cup w \cup u}(\mathsf{skip} \parallel S) \;\Leftrightarrow\; w' = w \wedge u' = u \wedge \mathsf{prd}_{z \cup u}(S)$$

Il reste à montrer que cette formule est équivalente à $\mathsf{prd}_{z \cup w \cup u}(S)$, sachant $z \cap w = \emptyset$, $(\mathsf{trm}(S) \wedge \mathsf{prd}_{z \cup u}(S) \Rightarrow u' = u)$ et $\mathsf{trm}(S)$. D'après le cas sans partage, on a $\mathsf{prd}_{z \cup w \cup u}(S) = \mathsf{prd}_{z \cup u}(S) \wedge w' = w$. L'hypothèse $\mathsf{prd}_{z \cup u}(S) \Rightarrow u' = u$ permet donc de déduire la formule attendue.

### A.3.3 Preuve de la propriété 7

Dans cette section nous montrons :

$$[S_1]P_1 \wedge [S_2]P_2 \Rightarrow [S_1 \parallel S_2](P_1 \wedge P_2)$$

avec $x_1$ l'espace de variables de la substitution $S_1$, $x_2$ l'espace de variables de la substitution $S_2$, ces variables vérifiant les propriétés suivantes : les variables $x_1 \cap x_2$ sont partagées en lecture par $S_1$ et $S_2$, les variables $x_2 - x_1$

sont non libres dans $P_1$ et les variables $x_1 - x_2$ sont non libres dans $P_2$.

Soit $u = x_1 \cap x_2$, $y_1 = x_1 - u$ et $y_2 = x_2 - u$. Il faut donc montrer :

$$\mathsf{trm}(S_1 \mid\mid S_2) \wedge \forall y_1', y_2', u' \; (\mathsf{prd}_{y_1 \cup u}(S_1) \wedge \mathsf{prd}_{y_2 \cup u}(S_2) \Rightarrow [y_1, y_2, u := y_1', y_2', u'](P_1 \wedge P_2))$$

sachant :

$$\mathsf{trm}(S_1) \wedge \forall y_1', u' \; (\mathsf{prd}_{y_1 \cup u}(S_1) \Rightarrow [y_1, u := y_1', u']P_1)$$
$$\mathsf{trm}(S_2) \wedge \forall y_2', u' \; (\mathsf{prd}_{y_2 \cup u}(S_2) \Rightarrow [y_2, u := y_2', u']P_2)$$

Or, sous les hypothèses, on a :

$$[y_1, u := y_1', u']P_1 \wedge [y_2, u := y_2', u']P_2 \Rightarrow [y_1, y_2, u := y_1', y_2', u'](P_1 \wedge P_2)$$

puisque $y_1$ est non libre dans $P_2$ et $y_2$ non libre dans $P_1$.

# Annexe B

# Rapport Projet RNTL BOM

Cette annexe est le texte du rapport du projet BOM intitulé *Etude du passage de paramètres dans la méthode B en vue d'optimiser la mémoire.*

## B.1 Introduction

Le but de ce document est de proposer des règles sous lesquelles il est possible d'optimiser la mémoire, lors des passages de paramètres. La première partie de ce document explique comment le passage de paramètre est actuellement traité dans la méthode B. La seconde partie propose des conditions d'optimisations. La dernière section introduit des calculs utilisés dans les sections précédentes.

Cette étude ne s'applique pas aux opérations locales. Celles-ci pouvant être ramenées à un modèle équivalent sans opération locale, il suffira d'appliquer cette équivalence pour trouver les conditions.

## B.2 Règles d'appel des opérations

Le principe retenu est l'appel par copie. Ceci signifie que les paramètres d'entrée sont copiés à l'appel et que l'opération travaille sur la copie. De la

même manière les résultats sont élaborés dans une copie qui est affectée, à la fin de l'appel, aux variables effectives. Dans cette section nous présentons différentes formulations de ce principe et montrons que, dans le cadre de B, ces formulations sont équivalentes.

### B.2.1 Appel par copie

La première définition décrit exactement comment l'appel par copie est traité dans les langages de programmation.

**Définition 10** *Appel par copie*

*Soit* $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$ *la définition d'une opération de nom op et soit l'appel* $v \leftarrow op(e)$*. Le principe retenu est le passage par copie. Le résultat est :*

$$([p := e]P) \mid var\ p, r\ in\ p := e\ ;\ S\ ;\ v := r\ end$$

Les conditions sont les suivantes :

**Condition 5** *conditions sur la définition d'opération*

*Soit op une opération définie par* $r \leftarrow op(p) \mathrel{\hat{=}} P \mid S$ *et travaillant sur la liste de variables x. Les restrictions sont les suivantes :*

$$\boxed{\begin{array}{l} p \cap r = \emptyset \\ p \cap x = \emptyset \\ r \cap x = \emptyset \end{array}}$$

Ces conditions sont vérifiées par les règles de portée du langage B. De plus $p$ et $r$ sont des listes de noms distincts deux à deux.

**Condition 6** *conditions sur l'appel d'opération*

*Soit un appel de la forme* $v \leftarrow op(e)$*. La restriction est la suivante :*

$$\boxed{v \text{ ne contient pas de doublon}}$$

Cette condition est facile à vérifier car $v$ est une liste de noms. On n'admet pas de terme de la forme $t(i)$.

### B.2.2 Substitution des paramètres résultats

Dans le cas de la méthode B on peut simplifier la définition précédente. Cette simplification revient à montrer que les résultats n'ont pas besoin d'être traités en copie, mais peuvent être directement substitués.

La suite ne sera correcte que si $v$ est une variable et non un terme dénotant une variable (par exemple $t[i]$).

**Définition 11** *Substitution des paramètres résultats*

$$([p := e]P) \mid p := e \; ; \; ([r := v]S)$$

**Condition 7** *Condition de la règle de substitution des paramètres résultats*

| *La liste de variables $v$ ne contient pas de variables modifiées dans $S$* $(v \cap x = \emptyset)$ |
|---|

Dans la méthode B cette condition est vérifiée, car $v$ ne peut contenir une variable de *op* puisque ces variables ne peuvent être affectées directement à l'extérieur.

Montrons que, sous cette condition, $S \; ; \; v := r = [r := v]S$. La forme normalisée de $S$ est :

$$@\, x', r' \,.\, Q \Longrightarrow x := x' \,||\, r := r'$$

avec $r$ non libre dans $Q$ ($r$ n'a pas de valeur en entrée). La forme normalisée de $v := r$ est :

$$@\, v' \,.\, (v' = r \Longrightarrow v := v')$$

Par B.5.2 (définition de la substitution dans une substitution) $[r := v]S$ devient :

$$@\, x', r' \,.\, [r := v]Q \Longrightarrow [r := v](x := x' \,||\, r := r')$$

ce qui se simplifie en :

$$\boxed{@\ x', r'\ .\ Q \Longrightarrow x := x' \,||\, v := r' \qquad (a)}$$

car :

- $r$ est non libre dans $Q$ ;
- $r$ est non libre dans $x$ donc $[r := v]x := x'$ se réduit en $x := x'$ ;
- $v$ est non libre dans $x$. La substitution multiple $x := x' \,||\, v := r'$ a bien un sens ;
- $v$ ne contient pas de doublon. La substitution $v := r'$ a bien un sens.

Par B.5.1 (forme normalisée d'un séquencement) la substitution généralisée $S$ ; $v := r$ est équivalente à :

$$@\ x', v'\ .\ \exists\ r'\ (Q \wedge [r := r']v' = r) \Longrightarrow x := x' \,||\, v := v'$$

c'est à dire :

$$@\ x', v'\ .\ \exists\ r'\ (Q \wedge v' = r') \Longrightarrow x := x' \,||\, v := v'$$

Ceci se réduit à :

$$\boxed{@\ x', v'\ .\ [r' := v']Q \Longrightarrow x := x' \,||\, v := v' \qquad (b)}$$

car $v'$ est non libre dans $Q$ ($v$ non modifiée par $S$).

Les deux formules $(a)$ et $(b)$ sont donc équivalentes, sous la condition 7. Remarquons que si la variable $v$ peut être modifiée dans $S$, alors l'équivalence est fausse. Soit $S \hat{=} (r := e_1 \,||\, v := e_2)$. La forme substituée donne $[r := v]S$, c'est à dire $v := e_1 \,||\, v := e_2$. La forme avec affectation donne $(r := e_1 \,||\, v := e_2)$ ; $v := r$, c'est à dire $v := e_1$.

### B.2.3 Substitution des paramètres entrées

**Définition 12** *Substitution des paramètres entrées*

$$[p := e]P \ \mid \ [p := e || r := v]S$$

Pour établir ce résultat il faut prouver l'équivalence :

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

Soit $S$ de la forme $@\ x', r' \ .\ Q \Longrightarrow x := x' \ || \ r := r'$. Le terme $[p, r := e, v]S$ s'écrit donc :

$$@\ x', r' \ .\ [p := e]Q \Longrightarrow [p, r := e, v](x := x' \ || \ r := r')$$

c'est à dire :

$$@\ x', r' \ .\ [p := e]Q \Longrightarrow x := x' \ || \ v := r'$$

car $p$ est non libre dans $x$, $x'$, $v$ et $r'$.

La forme normalisée de l'affectation $p := e$ est $@\ p' \ .\ (p' = e) \Longrightarrow p := p'$. Le terme $p := e \ ; \ [r := v]S$ s'écrit donc :

$$@\ x', r' \ .\ \exists\ p' \ (p' = e \wedge [p := p']Q) \Longrightarrow x := x' \ || \ v := r'$$

La formule $\exists\ p' \ (p' = e \wedge [p := p']Q)$ est équivalente à $[p := e]Q$. La simplification 2 est donc correcte, sans hypothèse particulière.

## B.3 Implantation des règles d'appel

### B.3.1 Règles des langages de programmation

La définition 10 correspond au mode de passage par copie des langages de programmation (passage par valeur). Elle est donc directement reproductible dans les langages de programmation. Le seul inconvénient est son coût en mémoire.

Les définitions 11 et 12 ne sont pas usuelles dans les langages de programmation. Nous allons montrer qu'elles s'assimilent au passage par référence, sous certaines conditions. Cette preuve nécessite de montrer que le code de l'appel est équivalent au code de la définition de l'opération, en remplaçant partout les noms des paramètres formels par les paramètres effectifs. Ceci ne sera possible bien sûr que sous certaines conditions. Remarquons que les définitions précédentes ne posent pas de problème pour les preuves, pour lesquelles on raisonne sur les formes normalisées. La difficulté est liée à la mise en correspondance avec les mécanismes de passage de paramètres des langages de programmation. On pourrait donc introduire le séquencement en spécification sans que ceci n'introduise de difficulté pour la preuve.

Les règles SUB23 à SUB26 (B-Book, p. 756) permettent d'effectuer le remplacement dans les substitutions de base, en préservant la structure. Ce remplacement correspond bien à une simulation du passage par référence. On peut étendre ceci à la plupart des constructeurs de substitutions (conditionnelles, ...). Le cas qui pose problème est le séquencement, pour lequel il faut trouver les règles et les conditions d'application. Ces conditions correspondront au cas où on peut assimiler le passage par copie au passage par nom (ou par référence si on n'admet pas les termes de variables).

Nous donnons ci-dessous les conditions de validité de la règle SUB27 (B-Book, p. 756) dans le cas de la substitution des paramètres résultats, c'est à dire pour la substitution $r := v$.

### B.3.2 Cas des paramètres résultat

Conditions pour que :

$$[r := v](S_1 \ ; \ S_2) = ([r := v]S_1) \ ; \ ([r := v]S_2)$$

On suppose dans la suite que les formes normalisées de $S_1$ et $S_2$ sont respectivement :

$$\begin{array}{l} P_1 \mid \ @\ x', r' \ .\ Q_1 \Longrightarrow x := x' \ || \ r := r' \\ P_2 \mid \ @\ x", r" \ .\ Q_2 \Longrightarrow x := x" \ || \ r := r" \end{array}$$

**Calcul du membre gauche :** $[r := v](S_1 \ ; \ S_2)$

En utilisant la forme normalisée de $S_1 \ ; \ S_2$ (section B.5.1), ce terme s'écrit :

$$[r := v](P_1 \wedge \forall\ x', r'\ (Q_1 \Rightarrow [x := x' \parallel r := r']P_2))$$
$$\mid\ @\ x", r"\ .\ [r := v](P_1 \Rightarrow \exists\ x', r'\ (Q_1 \wedge [x := x' \parallel r := r']Q_2)) \Longrightarrow [r := v](x := x" \parallel r := r")$$

Ceci se simplifie en (1) :

$$[r := v]P_1 \wedge \forall\ x', r'\ ([r := v]Q_1 \Rightarrow [r := v][x := x' \parallel r := r']P_2)$$
$$\mid\ @\ x", r"\ .\ ([r := v]P_1 \Rightarrow \exists\ x', r'\ ([r := v]Q_1 \wedge [r := v][x := x' \parallel r := r']Q_2))$$
$$\Longrightarrow x := x" \parallel v := r" \qquad (a)$$

**Calcul du membre droit :** $([r := v]S_1) \ ; \ ([r := v]S_2)$

Calcul de $[r := v]S_1$ :

$$[r := v]P_1 \ \mid\ @\ x', r'\ .\ [r := v]Q_1 \Longrightarrow x := x' \parallel v := r'$$

Calcul de $[r := v]S_2$ :

$$[r := v]P_2 \ \mid\ @\ x", r"\ .\ [r := v]Q_2 \Longrightarrow x := x" \parallel v := r"$$

Calcul du séquencement (section B.5.1) :

$$[r := v]P_1 \wedge \forall\ x', r'\ ([r := v]Q_1 \Rightarrow [x := x' \parallel v := r'][r := v]P_2)$$
$$\mid\ @\ x", r"\ .\ ([r := v]P_1 \Rightarrow \exists\ x', r'\ ([r := v]Q_1 \wedge [x := x' \parallel v := r'][r := v]Q_2))$$
$$\Longrightarrow x := x" \parallel v := r" \qquad (b)$$

La substitution $[r := v]S_2$ travaille sur les variables $x$ et $v$. Le renommage est donc de la forme $[x := x' \parallel v := r']$.

### Conditions

Pour que les deux formes normalisées $(a)$ et $(b)$ soient égales les deux conditions suivantes doivent être vérifiées :

$$(1) \quad [r := v][x := x' \mid\mid r := r']P_2 \equiv [x := x' \mid\mid v := r'][r := v]P_2$$
$$(2) \quad [r := v][x := x' \mid\mid r := r']Q_2 \equiv [x := x' \mid\mid v := r'][r := v]Q_2$$

Cherchons les conditions pour que :

$$[r := v][x := x' \mid\mid r := r']R \equiv [x := x' \mid\mid v := r'][r := v]R$$

Puisque la variable $x$ est non libre dans $r'$ cette équivalence se réécrit en :

$$[r := v][x := x'][r := r']R \equiv [x := x'][v := r'][r := v]R$$

Et, puisque $r$ est non libre dans $r'$ et $x'$ :

$$[x := x'][r := r']R \equiv [x := x'][v := r'][r := v]R$$

Pour que cette équivalence soit correcte une condition suffisante est que $v$ soit non libre dans $R$. En effet, dans ce cas, le second terme devient $[x := x'][r := r']R$, et l'équivalence est correcte.

Instanciée sur les prédicats $P_2$ et $Q_2$ la condition est :

**Condition 8** *Condition sur les paramètres effectifs résultats*

| *Les paramètres effectifs résultats ne peuvent être des variables sur lesquelles portent la précondition $P_2$ et la postcondition $Q_2$.* |
|---|

Ceci revient à dire que $S_2$ ne référence pas $v$. Cette condition est vraie dans la méthode B. Les variables de $v$ ne sont directement modifiables qu'à l'endroit où elles sont définies, c'est à dire ici là où l'appel est présent. Elles ne peuvent donc être référencées dans la définition de l'opération, sinon il y aurait un cycle dans l'architecture.

Voici un exemple qui poserait problème. Soit une opération définie par $r := r + 1 \ ; \ r := v$. Soit la substitution $r := v$, avec $v$ valant initialement 0. En appliquant la substitution sur la forme normalisée on obtient $v := v$, c'est à dire $v = 0$. En substituant dans chaque sous-terme du séquencement on obtient $v := v + 1 \ ; \ v := v$, c'est à dire $v = 1$.

### B.3.3 Cas des paramètres d'entrée

#### Calcul

Conditions pour que :

$$[p := e][r := v](S_1 \ ; \ S_2) = ([p := e][r := v]S_1) \ ; \ ([p := e][r := v]S_2)$$

Le calcul du membre gauche s'obtient en appliquant la substitution $p := e$ à la formule $(a)$ de la section précédente B.3.2. Le second terme s'obtient à partir de la formule $(b)$ de la même section, en remplaçant les formules $[r := v]R$ par $[p := e][r := v]R$.

Les conditions à vérifier sont donc :

(1) $[p := e][r := v][x := x'][r := r']P_2 \equiv [x := x'][v := r'][p := e][r := v]P_2$
(2) $[p := e][r := v][x := x'][r := r']Q_2 \equiv [x := x'][v := r'][p := e][r := v]Q_2$

#### Conditions

Cherchons les conditions sous lesquelles l'équivalence suivante est correcte :

$$[p := e][r := v][x := x'][r := r']R \equiv [x := x'][v := r'][p := e][r := v]R$$

Le premier terme se simplifie en $[p := e][x := x'][r := r']R$ puisque $r$ est non libre dans $x'$ et $r'$. Il reste donc à trouver les conditions telles que :

$$[p := e][x := x'][r := r']R \equiv [x := x'][v := r'][p := e][r := v]R$$

La première condition est :

**Condition 9** *Condition entre les paramètres effectifs résultats et les paramètres effectifs entrée.*

| *v est non libre dans e* |
|---|

En effet, par les conditions 8 et 9, la formule $[v := r'][p := e][r := v]R$ se simplifie en $[p := e][r := r']R$. Il ne reste qu'à trouver la condition pour que :

$$[p := e][x := x'][r := r']R \equiv [x := x'][p := e][r := r']R$$

**Condition 10** *Condition entre les paramètres effectifs résultats et les variables de l'opération.*

| $x$ *est non libre dans* $e$ |
|---|

car $p$ est non libre dans $x'$.

Un exemple similaire à celui de la section B.3.2 peut être donné en prenant un corps d'opération de la forme $x := p+1$ ; $x := p$ et la substitution $p := x$.

## B.4 Résultats

Les résultats obtenus sont les suivants :

- La substitution des paramètres formels par les paramètres effectifs est légale sur les formes normalisées. La substitution correspond bien au passage par copie.
- La substitution des paramètres formels résultats par les paramètres effectifs est toujours possible sur le code (les cas d'utilisation de la règle SUB27 sont corrects). Ceci est valide dans la méthode B pour la raison suivante :

  | Les paramètres effectifs ne peuvent être directement référencés dans l'opération. |
  |---|

- La substitution des paramètres formels entrée par les paramètres effectifs n'est pas toujours possible sur le code (la règle SUB27 n'est pas toujours correcte). Les conditions à vérifier sont les suivantes :

  | Les paramètres effectifs entrée ne doivent dépendre ni des paramètres effectifs résultats de l'appel de l'opération ni des variables de l'opération. |
  |---|

Il faut aussi garantir que les paramètres entrée ne sont pas modifiés directement par l'opération, ce qui est le cas dans la méthode B.

Remarquons que les conditions obtenues sont exactement celles décrites dans Berlioux & Bizard [BB89] permettant de garantir l'équivalence entre le passage par nom et le passage par valeur (paramètre d'entrée de la méthode B) et le passage par référence (paramètre résultat de B).

## B.5 Quelques calculs

Dans cette section nous introduisons quelques calculs sur les formes normalisées.

### B.5.1 Calcul de $S_1 \ ; \ S_2$

Soient $S_1$ et $S_2$ deux substitutions généralisées ayant pour formes normalisées respectives :

$$\begin{array}{rcl} S_1 & = & P_1 \ | \ @\, x' \,.\, (Q_1 \Longrightarrow x := x') \\ S_2 & = & P_2 \ | \ @\, x" \,.\, (Q_2 \Longrightarrow x := x") \end{array}$$

La forme normalisée de $S_1 \ ; \ S_2$ s'obtient à partir des prédicats trm et prd (B-Book, p. 375) :

$$P_1 \wedge \forall\, x' \,(Q_1 \Rightarrow [x := x']P_2) \ | \ @\, x" \,.\, ((P_1 \Rightarrow \exists\, x' \,(Q_1 \wedge [x := x']Q_2)) \Longrightarrow x := x")$$

### B.5.2 Calcul de la forme normalisée d'une substitution substituée

Utilisation des règles SUB23 à SUB 26 (B-Book, p. 756) :

$$\begin{array}{rcl} [w := e]S & = & [w := e](P \ | \ @\, x' \,.\, (Q \Longrightarrow x := x')) \\ & = & ([w := e]P) \ | \ [w := e](@\, x' \,.\, (Q \Longrightarrow x := x')) \\ & = & ([w := e]P) \ | \ @\, x' \,.\, [w := e](Q \Longrightarrow x := x') \\ & = & ([w := e]P) \ | \ @\, x' \,.\, (([w := e]Q) \Longrightarrow ([w := e]x := x')) \end{array}$$

### B.5.3 Affectation

La forme normalisée de $p := e$ est : @ $p'(p' = e \Longrightarrow p := p')$.

[BB89] P. Berlioux, Ph. Bizard,
*Algorithmique - Construction, preuve et évaluation de programmes.*
DUNOD-Informatique, Bordas, Paris, 1989.

[Abrial96] J-R. Abrial
*The B-Book*
Cambridge University Press, 1996.

# Bibliographie


[ABL96] J.R. Abrial, E. Boerger, and H. Langmaack. *The Steam-boiler Case Study: Competition of formal program specification and development methods*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Abr96a] J.R. Abrial. Extending B Without Changing it. In Nantes H. Habrias, editor, *First B conference*, 1996.

[Abr96b] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, may 1991.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, may 1995.

[AM98] J.R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *Proceedings of the Second International B Conference*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Ba99] P. Behm and all. Météor: A Successful Application of B in a Large Project. In *FM'99 - Formal Methods -volume 1*, number 1708 in Lecture Notes in Computer Science, pages 348–387. Springer-Verlag, september 1999.

[Bac80] R.J.R Back. Correctness Preserving Program Refinements: Proofs and Application. Technical Report Tract 131, Mathematical Centrum, Amsterdam, 1980.

[BB89] P. Berlioux and Ph. Bizard. *Algorithmique - Construction, preuve et évaluation de programmes.* DUNOD-Informatique, Bordas, Paris, 1989.

[BB98] R.J. Back and M. Butler. Fusion and simultaneous execution in the refinement calculus. *Acta Informatica*, 35:921–949, 1998.

[BB99] M. Büchi and R. Back. Compositional Symmetric Sharing in B. In J. Davies J.M. Wing, J. Woodcoock, editor, *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.

[BBa92] M. Franzle B. Buth, K-H. Buth and all. Provably Correct Compiler Development and Implementation. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction'02*, volume 641 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[BC99] R.E. Bloomfield and D. Craigen. Formal Methods Diffusion : Past Lessons and Future Prospects. Technical Report D/167/6101/1 v2.0, Adelard, 1999.

[BC00] D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[BCR03] L. Burdy, L. Casset, and A. Requet. Développement formel d'un vérificateur embarqué de byte-code java. In V. Donzeau-Gouge et H. Habrias D. Bert, editor, *Développement rigoureux de logiciel avec la méthode B*, volume 22. Technique et Science Informatiques, 2003.

[BDB+03] F. Badeau, D.Bert, S. Boulmé, M-L. Potet, N. Stouls, and L. Voisin. Traduction de B vers des langages de programmation : points de vue du projet BOM. In *Actes des Journées AFADL, IRISA, RENNES*, Janvier 2003.

[BDM97] P. Behm, P. Desforges, and F. Mejia. Application de la méthode B dans l'industrie du ferroviaire. In *Application des techniques formelles au logiciel.* Observatoire Français des Techniques Avancées, 5 rue Descartes 75005 PARIS France, 1997.

[Beh96] Patrick Behm. Développement formel des logiciels sécuritaires de Météor. In Henri Habrias, editor, *Proceedings of the 1st Conference on the B method*, pages 3–10, November 1996.

[BEJ+95] D. Bert, R. Echahed, P. Jacquet, M-L. Potet, and J-C. Reynaud. Spécification, Généricité, Prototypage : Aspects du langage LPG. *Technique et Science Informatiques*, 14(9), 1995.

[BlML01] D. Bolignano, D. le Métayer, and C. Loiseaux. Besoins en méthodes formelles dans le contexte de la sécurité des systèmes d'information. In Y. Ledru and M-L Potet, editors, *Approches formelles pour l'aide au développement de logiciels*, volume 20. Technique et Science Informatiques, 2001.

[BM99] Lilian Burdy and Jean-Marc Meynadier. Automatic Refinement. In *FM'99 – B Users Group Meeting – Applying B in an industrial context: Tools, Lessons and Techniques*, pages 3–15. Springer-Verlag, 1999.

[Boe99] E. Boerger. High Level System Design and Analysis Using Abstract State Machines. In Dieter Hutter et all, editor, *Applied Formal Methods: FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[Bon99] P. Bontron. Modélisation des développements de la méthode B. Rapport de magistère, Université Joseph Fourier, Grenoble, France, 1999.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design*. Addison-Wesley, 1994.

[BP00] P. Bontron and M-L. Potet. Automatic Construction of Validated B Components from structured Developments. In J. Bowen, S. Dunne, and all, editors, *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BP90] D. Bert and M-L Potet. Fonctionnalités d'un démonstrateur de théorèmes pour la dérivation assistée de programmes. Rapport de contrat SYSECA-INPG, MRT 88.5.1104, LIFIA, décembre 90.

[BPR96] D. Bert, M-L. Potet, and Y. Rouzaud. A study on Components and Assembly Primitives in B. In H. Habrias, editor, *Proc. of First B Conference*, Novembre 1996.

[BS96] R.J. Back and K. Sere. From Action system to Modular Systems. *Software - Concepts and Tools (version préliminaire dans FME'94)*, pages 26–39, 1996.

[CGR93] D. Craigen, S.L. Gerhart, and T.J. Ralston. An International Survey of Industrial Applications of Formal Methods. Technical Report NISTGCR 93/626, U.S. National Institute of Standards and Technology, 1993.

[Cha98] P. Chartier. Formalisation of B in Isabelle/HOL. In D. Bert, editor, *Proceedings of the Second International B Conference*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Cha02] P. Chartier. ABS Project: Merging the Best Practices in Software Design from Railway and Aircraft Industries. In D. Bert, J.P. Bowen, and all, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[CU89] W. Chen and J.T. Udding. Towards a Calculus of Data Refinement. *Mathematics of Program Construction, LNCS 375*, pages 197–218, 1989.

[CW96] E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. In *Workshop on Strategic Directions in computing Research*. ACM, june 96.

[DBMM00] T. Dimitrakos, J. Bicarregui, B. Matthews, and T. Maibaum. Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context. In A. Galloway J.P. Bowen, S. Dunne and S. King, editors, *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[DEF03] D.Dollé, D. Essamé, and J. Falampin. B à Siemens Transportation Systems- Une expérience industrielle. In D. Bert,

V. Donzeau-Gouge, and H. Habrias, editors, *Développement rigoureux de logiciel avec la méthode B*, volume 22. Technique et Science Informatiques, 2003.

[Dij76] E.W. Dijkstra. *A discipline of Programming.* Prentice-Hall, 1976.

[dR98] Willem-Paul de Roever. The Need for Compositional Proof Systems: A Survey. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, 1998.

[dRLP98] Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, volume 1536 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

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

[EJPS91] R. Echahed, P. Jacquet, M-L Potet, and S. Sebar. Equational Reasoning and the completion procedure: a comparative study in program transformation. In *Workshop on statics analysis of equational, functional and logic programs*, Bordeaux, France, octobre 1991.

[For89] P. Forin. Vital Coded Microprocessor: Principles and Application for Various Transit Systems. In *Proc. IFAC-GCCT*, 1989.

[Gar96] David Garlan. Style-Based Refinement for Software Architecture. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW2) and the International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*, San Francisco, CA, October 1996. ACM Press.

[GM93] P.H.B. Gardiner and C. Morgan. A Single Complete Rule for Data Refinement. *Formal Aspects of Computing*, 5:367–392, 1993.

[GP85] D. Gries and J. Prins. A New Notion of Encapsulation. In *Proceedings of Symposium on Languages Issues in Programming Environments, SIGLPAN*, 1985.

[Gue96] Y. Guerte. Dérivation de programmes impératifs à partir de spécifications algébriques. Doctorat d'Université, Univ. Claude Bernard, Lyon, Octobre 1996.

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

[Hay96] I.J. Hayes. Supporting module reuse in refinement. *Science of Computer Programming*, 27:175–184, 1996.

[HM01] P. Hartel and L. Moreau. Formalizing the Safety of Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4), 2001.

[Hoa84] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1984.

[ITS93] ITSEC. Evaluation criteria for IT security - part 3 : Assurance of IT systems. Technical Report version 1.2, INFOSEC central office, Brussels, Belgium, 1993.

[JLNP97] P. Jacquet, Y. Ledru, X. Nicollin, and M.-L. Potet. Logiciels critiques : catalogue d'une exposition bibliographique. *Technique et Science Informatiques*, 16(6), 1997.

[Jon81] C. B. Jones. Development methods for computer programs including a notion of interference. Technical report, D. Phil. Thesis, Oxford University computing Laboratory, juin 1981.

[Jon83] C. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83, North Holland*, 1983.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, London, 1986.

[KS99] R. Kurki-Suonio. Component and interface refinement in closed-system specifications. In J.M. Wing, J. Woodcoock, and J. Davies, editors, *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[Lam83] L. Lamport. What good is temporal logic. *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, septembre 1983.

[Lam98] L. Lamport. Composition: A Way to Make Proofs Harder. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, volume 1536 of *Lecture Notes in Computer Science*, pages 402–423. Springer-Verlag, 1998.

[Lan96] Kevin Lano. *The B language and Method.* Springer-Verlag, 1996.

[Lan01] J-L. Lanet. Cartes à puce et méthodes formelles, une lente intégration .... In Y. Ledru and M-L Potet, editors, *Approches formelles pour l'aide au développement de logiciels*, volume 20. Technique et Science Informatiques, 2001.

[Leb00] J. Lebray. Modélisation de systèmes en B : Proposition de guides méthodologiques pour la décomposition d'événements. Rapport de DEA, Institut National Polytechnique de Grenoble, France, 2000.

[LL98] J-L Lanet and P Lartigue. The Use of Formal Methods for SmartCards, a Comparison between B ans SDL to Model the T=1 Protocol. In *Proceedings of International Workshop on Comparing Systems Specification Techniques*, Nantes, March 1998.

[LOP98] Y. Ledru, C. Oriat, and M-L. Potet. Le raffinement vu comme primitive de spécification - une comparaison de VDM, B et Specware. In *Actes des Journées AFADL, LISI/ENSMA, Futuroscope Poitiers*, Septembre 1998.

[LP96] Y. Ledru and M-L. Potet. A VDM specification of the steam-boiler problem. In *Steam-Boiler Case Study*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

[LR98] J-L. Lanet and A. Requet. Formal proof of smart card applets correctness. In *proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[LW94] B. Liskov and J. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, novembre 1994.

[Mar91] C.E. Martin. Preordered Categories and Predicate Transformers. Technical report, D. Phil. Thesis, Programming Research Group, Oxford University, 1991.

[MD94] Fernando Mejia and Babak Dehbonei. Formal methods in the railways signalling industry. In M. Bertran, M. Naftalin, and T. Denvir, editors, *FME'94: Industrial Benefit of Formal Methods*, pages 26–34. Springer-Verlag, October 1994.

[Mey87] B. Meyer. Reusability: the case for object-oriented design. *IEEE Software*, march 1987.

[MG90] C. Morgan and P.H.B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481–503, 1990.

[MoD91] MoD. The Procurement of Safety Critical Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance). Interim Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, 5 April 1991.

[Mor87] J. Morris. A theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9, 1987.

[Mor88] C.C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3), 1988.

[Mor92] C. Morgan. *On the Refinement Calculus*. Springer-Verlag, 1992.

[Mot00] S. Motré. A B automaton for Authentification Process. In *WITS: Workshop on Issues in the Theory of Security*, Genève, Suisse, 2000.

[MQR95] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[MT00] S. Motré and C. Téri. Using Formal and Semi-Formal Methods for a Common Criteria Evaluation. In *EUROSMART*, Marseille, France, juin 2000.

[MV92] C. Morgan and T. Vickers. Types and Invariants in the Refinement Calculus. In *On the Refinement Calculus*. Springer-Verlag, 1992.

[Nah01] J. Nahoum. Outils d'assistance à la construction de systèmes dans la méthode B. Rapport de DEA, Institut National Polytechnique de Grenoble, France, 2001.

[Nau92] D.A. Naumann. Two-Categories and Program Structure : Data types, Refinement Calculi, and Predicate Transformers. Technical report, Ph. D. Thesis, University of Texas at Austin, 1992.

[Nec97] G.C. Necula. Proof-carrying code. In *In 24th Principles of Programming Languages (POPL)*, Paris, France, 1997.

[OFT97] OFTA. *Application des techniques formelles au logiciel.* Observatoire Français des Techniques Avancées, 5 rue Descartes 75005 PARIS France, 1997.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[PMP94] C. Paulin-Mohring and M-L. Potet. Logique intuitionniste : un cadre pour la construction de programmes. Technical report, LIFIA - Rapport de Recherche 931 -I-, 1994.

[Pot93] M-L Potet. How informal reasoning in the design process may match formal reasoning: a case study. In INRIA, editor, *Proc. of ERCIM workshop on Development and Transformation of Programs*, Nancy, novembre 1993.

[Pot01] M-L. Potet. Composition des machines et raffinements (chapitre 14). In *Spécification formelle avec B.* Hermès Science Publications, 2001.

[Pot02] M.-L. Potet. Étude du passage de paramètres dans la méthode B - Optimisation mémoire. Technical report, Rapport du projet RNTL BOM, 2002.

[Pot03] M-L. Potet. Spécifications et développements structurés dans la méthode B. In D. Bert, V. Donzeau-Gouge, and H. Habrias, editors, *Développement rigoureux de logiciel avec la méthode B*, volume 22. Technique et Science Informatiques, 2003.

[PR98] M.-L. Potet and Y. Rouzaud. Composition and Refinement in the B-Method. In D. Bert, editor, *Proceedings of the Second International B Conference*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[RBB02] H. Ruiz-Barradas and D. Bert. Specification and Proof of liveness Properties under Fairness Assumptions in B event Systems. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Req00] A Requet. A B Model for Ensuring Soundness of the Java Card Virtual Machine. In *FMICS'2000*, Berlin, March 2000.

[Rou99] Y. Rouzaud. Interpreting the B-Method in the Refinement Calculus. In J. Davies J.M. Wing, J. Woodcoock, editor, *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[SJ94] Y.V. Srinivas and R. Jüllig. Specware(TM): Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, december 1994.

[SJ95] Y.V. Srinivas and R. Jüllig. *Specware Language Manual*. Kestrel Institute, November 1995.

[SL99] D. Sabatier and P. Lartigue. The Use of the B Formal Method for the Design and the Validation of the Transaction Mechanism for Smart Card Applications. In *FM'99 - Formal Methods - volume 1*, volume 1708 of *Lecture Notes in Computer Science*, pages 348–387. Springer-Verlag, september 1999.

[ST99] National Institute Of Standards and Technology. Common criteria for information technology security evaluation. Technical report, U.S. Dept. of Commerce, National Bureau of Standards and Technology, Aout 1999.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *CACM*, 14(4), 1971.

[Zwi89] J. Zwiers. Compositionality, Concurrency and Partial Correctness. volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.