REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE : Mathématiques et Informatique
DEPARTEMENT : Informatique
N° :.....

DOMAINE : Mathématiques et Informatique
FILIERE : Informatique
OPTION : SIGL

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par: BENBOUDINA Imane

Intitulé

Spécification et vérification d'un système d'ascenseur par la méthode B

Soutenu devant le jury composé de :

Mr. MOKHTARI Rabah	Université de M'sila	Président
Dr. BOURAHLA Mustapha	Université de M'sila	Rapporteur
Mme. MELIOUH Amel	Université de M'sila	Examinateur

Année universitaire : 2017/2018

Dédicace

Le premier remerciement est à Allah

A ceux qui mon offert la clé de la réussit sur un plat d'or ne cessent de donner sans jamais recevoir mes très cher parents dont je suis fière que Allah me garde pour toujours.

A tous mes frères Moustapha, Youcef, Bilal, Abd-elnor ;

A ma chère sœur Somia et mon beau-frère Khalil ;

A mes trois étoiles qui éclairassent notre famille Ritaj, Mohamed Ilyas et Abd-elmadjid ;

A mes oncles, tantes, cousins, cousines ;

A mes proches Ibtissam, Ilham, Hadjira, Affef, Abla Jihad, chayma, Khawla ;

A tous mes amis Anwar, Abd-elrahim ;

A tous mes enseignants depuis l'école primaire jusqu'à l'université

A tous la promotion de la post-graduation 2017/2018

Remerciements

Je tiens en premier à remercier ***Allah*** *le tout puissant et miséricordieux, qui m'a donné le savoir, la volonté et le courage pendant toute la période de mes études et d'accomplir ce modeste travail.*

Par le biais de ces quelques lignes, je souhaite ainsi exprimer ma gratitude aux personnes qui ont rendu ce travail possible.

Je remercie très vivement mon encadreur ***Mr.Bourahla Mustapha*** *docteur du Département de mathématiques et d'informatique de l'Université de M'sila pour ces conseils, pour avoir accepté d'examiner cette mémoire, et surtout pour leurs efforts effectifs et leur gentillesse pour la réalisation de ce travail.*

Mes plus grands remerciements s'adressent à ***mon père, ma mère, ma sœur*** *ainsi que toute* ***ma famille****, qui m'a témoigné un soutien inconditionnel tout au long de la préparation de cette mémoire. Qu'à travers ces quelques mots ils sachent toute ma reconnaissance.*

Et enfin je remercie toute personne ayant contribué de prés ou loin à l'élaboration de ce travail.

A toute la promotion de la post-graduation 2017-2018.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Grâce à l'utilisation toujours plus massive de l'informatique, dans tous les domaines de la vie quotidienne, les exigences accrues en qualité des applications rendent incontournable l'amélioration du processus de développement du logiciel. Les travaux s'orientent, de plus en plus, vers l'utilisation des méthodes formelles de spécification qui permettent de répondre à ces exigences. Cependant, les applications pour lesquelles les méthodes formelles ont été utilisées dans l'industrie sont encore limitées à des domaines bien particuliers.

Un des écueils pour l'acceptation des méthodes formelles dans l'industrie est l'absence d'un langage formel de spécification permettant d'exprimer et de vérifier de manière optimale tous les aspects d'un système informatique. En effet, différents langages sont proposés pour exprimer les spécifications relatives aux structures de données manipulées, au contrôle des différentes tâches à effectuer, aux contraintes temps réels, etc. Pour cela, l'emploi d'une méthode formelle inappropriée au domaine de l'application peut ajouter des difficultés notationnelles qui ne font souvent qu'obscurcir le problème. Il s'agit donc de choisir la méthode appropriée au domaine de l'application à développer, et aussi de choisir la manière de l'appliquer dans un cadre et une méthode de travail donnés.

De nombreuses méthodes de conception de systèmes ont été introduites dans ce domaine, en s'appuyant sur des techniques, des théories ainsi que des notations différentes. Parmi celles-ci, les méthodes formelles. Ces méthodes se distinguent par sa capacité de fournir des bases mathématiques pour la vérification et la correction de la solution logicielle obtenue.
Non seulement notre satisfaction réside dans l'application des techniques formelles pour produire des logiciels sûrs, mais nous sommes également persuadés que ses techniques formelles offrent à l'ingénieur logiciel un raisonnement abstrait sur les modèles, une lisibilité ainsi qu'une bonne compréhension de ces derniers. Notre travail dans cette mémoire s'inscrit donc, dans le cadre de l'étude des notations et méthodes formelles et de spécification formelle, et en particulier : la méthode B.

B est une méthode permettant de couvrir les différentes phases du développement de logiciel, de la spécification formelle jusqu'à l'implémentation en passant par des raffinements successifs. Elle permet la description d'une abstraction du système à développer puis par des approximations successives (raffinements) d'obtenir un modèle concret.

Cette méthode est principalement conçue afin de produire des logiciels sûrs, pour cette raison des preuves sont menées à chaque étape de construction. Plus les avantages que toutes

les méthodes formelles bénéficient, B possède d'autres intérêts que ceux classiquement évoqués pour celles-ci. Depuis sa création, elle a été introduite pour être pratique et applicable dans l'industrie. D'ailleurs elle comprend d'outils robustes, commercialisés et utilisés dans le monde académique et industriel. Bien que son langage est fondé sur des bases mathématiques solides, il est très proche des langages impératifs fréquemment utilisés par tous les informaticiens. Toutefois, malgré tous ces efforts, B subit en quelque sorte un certain nombre de carences héritées des méthodes formelles. Outre la difficulté d'écriture et d'utilisation des spécifications, il existe un manque de formation et de guides méthodologiques. Ainsi, malgré la simplification des preuves par les outils existants, elles demeurent laborieuses et notamment pour vérifier des raffinements. De plus, elles n'interdit pas d'écrire une spécification défavorable, car elles détectent les erreurs en identifiant les omissions ou les incohérences de ce qui a été spécifié, mais pas de ce qui a été exigé par l'utilisateur. Notre sujet s'inscrit dans le cadre du rapprochement des notations de langage formel (B), en s'appuyant sur leur complémentarité. Il sert à apporter des renforts aux concepteurs et aux développeurs dans leur démarche de développement de logiciel.

Notre travail concernant ce mémoire se dirige vers l'utilisation des méthodes formelles afin de concevoir et spécifier un système d'ascenseur. Comme point de départ, nous retenons la méthode de spécification formelle B [18]. Ce choix est motivé par ses principales caractéristiques et son utilisation de plus en plus répandue, notamment qu'elle couvre le cycle de vie d'un logiciel ainsi que son caractère récent et le fait qu'elle est particulièrement bien documentée et outillée par rapport aux autres méthodes.

Tout d'abord, nous modélisons les diagrammes de classes et d'états-transitions correspondant au système d'ascenseur, ensuite nous avons constitue les spécifications formelles B, après nous vérifions ses spécifications obtenues à l'aide des obligations de preuve et des vérifications de types. Afin de générer automatiquement le code C équivalent. En plus à cette introduction générale, notre mémoire est organisé comme suit.

Le premier chapitre présente la spécification et la vérification formelle des systèmes informatiques. Le but principal est d'introduire les méthodes de spécification formelle ses termes et ses notions de base dans ce domaine, ainsi que de mettre en évidence les intérêts et les avantages et les inconvénients de chaque méthode.

Le deuxième chapitre présente la spécification formelle d'un système avec la méthode B. Tout d'abord, nous introduisons la méthode B et ses concepts de base, ses notations, ses avantages et sa contribution à la spécification formelle des logiciels et des programmes.

Le troisième chapitre constitue notre contribution : nous modélisons les diagrammes de classes et d'états-transitions d'UML et la spécification formelle B du système de contrôle d'ascenseur afin d'appliquer la méthode B, en vérifiant par des preuves les résultats obtenues. Enfin, nous pouvons générer automatiquement un code C équivalent aux spécifications d'implémentation B. Dans ce but nous utilisons l'Atelier B.

Enfin, nous concluons ce mémoire par une conclusion générale et des perspectives.

CHAPITRE 1

SPECIFICATION ET VERIFICATION FORMELLE

1. Introduction

L'industrie développe maintenant des systèmes larges et complexes. Malgré le fait qu'un grand nombre de logiciels aient été développés, les processus utilisés et la qualité des résultats obtenus sont encore pauvres. Le coût et le temps nécessaires pour développer un logiciel sont imprévisibles et souvent très élevés. Le développement de systèmes larges et complexes est souvent achevé en retard par rapport au plan initial. Pour résoudre ces problèmes, il est indispensable d'apporter des améliorations au processus de développement. Plus particulièrement, ce chapitre s'intéresse à la phase de spécification formelle.

Pour l'informaticien, la question cruciale est plutôt de savoir si le programme effectue bien la tâche attendue ?

Une spécification peut être écrite à différents niveaux et prendre des formes diverses selon l'objectif exact qu'on lui assigne. Dans ce but, une description de cette tâche, précise et indépendante du programme, est nécessaire. On l'appelle une spécification de programme.

Dans ce chapitre nous présentons brièvement la spécification et la vérification formelle des logiciels. Ensuite, nous introduisons leur problématique générale, nous d'écrivons les méthodes de spécification formelle, en particulier ses techniques de la spécification, ses avantages et ses limites. Enfin, Nous présentons ses contributions à la spécification formelle des logiciels et des programmes et nous mettons en perspective les points communs entre elles.

2. Définitions

2.1 Test de logiciel

Le test est la méthode de vérification la plus largement employée pour le logiciel : tout programmeur, de l'étudiant débutant a l'ingénieur conformé, effectue des tests pour essayer de se convaincre que le logiciel qu'il est en train d'écrire réalise bien la fonction souhaitée. Ces tests sont généralement écrits « à la main » en choisissant des valeurs pertinentes pour l'application considérée, et en vérifiant que ce qui est calcule pour ces valeurs correspond bien à ce qu'on attend [3].

2.2 Vérification

La vérification est la réponse à la question « Are we building the product right ? ».

Cette définition sous-entend trois éléments de base à la vérification : « right » met en évidence une notion de référence qui énonce ce qui doit être correct. « Building the product » énonce un processus de conception : une partie du produit est en cours de réalisation. Enfin « Are we » met en évidence le besoin d'une méthode pour s'assurer que la réalisation en cours est correcte [3].

2.2.1 Vérification formelle

La vérification formelle est la réponse à la question : Est-ce que la réalisation satisfait la spécification quelles que soient les valeurs d'entrée ?

La vérification formelle introduit donc un quantificateur universel. Il apparaît donc clairement que ce problème ne peut être résolu par des méthodes probabilistes, mais que cela nécessite au contraire d'utiliser des méthodes formelles basées sur des mathématiques. Il s'agira de trouver un modèle mathématique permettant d'exprimer aussi bien la réalisation que la spécification, et fournissant des méthodes de preuve adaptées. Notons que même si la vérification formelle apporte une réponse plus fiable que la vérification, ce n'est pas une panacée [3].

2.2.2 Vérification formelle complète ou incomplète des programmes

La vérification formelle est complète si la vérification est effectuée en explorant de façon exhaustive tous les états successifs possibles du système. La vérification est incomplète si la vérification ne considère que des séquences d'état de longueur bornée [3].

2.2.3 Vérification formelle partielle ou totale des programmes

La vérification partielle consiste à vérifier le programme en supposant que le programme termine, tandis que la vérification totale nécessite de montrer en plus que le programme termine [3].

2.3 Spécification

Une spécification est une propriété qui sert de référence pour la vérification : c'est la propriété à vérifier pour le produit en cours de réalisation. Une spécification peut contenir des erreurs ou être incomplète. Une même propriété peut servir de spécification à une étape donnée, et être la réalisation qui doit vérifier une spécification pendant une autre étape du processus de vérification [3].

2.3.1 La spécification du logiciel

La spécification est un élément critique dans le processus de développement du logiciel ; elle est la documentation des exigences du système. Cette vision du système se situe à un niveau très abstrait. Cependant, elle doit être complète et précise, de sorte que tout système qui satisfait ces exigences documentées comble correctement les besoins des utilisateurs. La spécification est comme un véhicule qui transmet les besoins des utilisateurs aux

développeurs du système. Pour cela, la spécification doit être utilisée et comprise par toutes les personnes concernées par le développement du système. Des études ont démontré que les fautes de spécification sont les plus coûteuses à corriger [9].

2.3.2 La spécification informelle

La plupart des concepteurs écrivent les spécifications en langage naturel. Le langage naturel est compréhensible par les utilisateurs, les analystes et les développeurs. Chacun d'eux est habitué à lire et à écrire des documents en langage naturel, puisqu'il n'exige pas une spécialisation particulière. L'organisation de ces documents est bien familière : table des matières, chapitres, sections, etc. La spécification informelle s'accorde bien avec les méthodes actuelles utilisées pour développer un logiciel. Cependant, il est possible qu'une spécification en langage naturel ait plusieurs interprétations. Cela Signifié que les spécifications informelles sont prédisposées à être incomplètes et incohérentes, à cause de leur ambiguïté et de l'incapacité d'en faire une vérification méthodique et rigoureuse [1].

2.3.3 La spécification formelle

Dans le recueil de normes français en génie logiciel ISO/AFNOR 1997**,** la spécification formelle est définie comme suit :

« Spécification pouvant être utilisée afin de démontrer mathématiquement la validité de la mise en œuvre d'un système ou encore de dériver mathématiquement la mise en œuvre du système ».

« On parle de spécification informelle lorsque le langage utilisé n'a pas une syntaxe précise, de spécification semi-formelle lorsque la syntaxe du langage utilisé est définie de façon précise, et de spécification formelle lorsque la syntaxe et la sémantique du langage utilisé sont définies » [7].

Les méthodes de spécification formelles utilisent des techniques mathématiques pour décrire un problème. L'utilisation de notations formelles résous le problème de la variété d'interprétations par la rigueur du formalisme, l'abstraction, la syntaxe et la sémantique mathématiques bien définies. Elles engendrent des spécifications précises que l'on peut vérifier de manière systématique à l'aide d'outils. En identifiant tôt les problèmes du système, il est encore possible de les corriger avec un coût minimal. Ceci peut réduire le coût et la durée du développement. Des études [9] [10] ont montré que l'addition de méthodes formelles au cycle de développement améliore le processus de développement et la qualité du logiciel, en imposant un contrôle dès les premières phases du développement, notamment la spécification [1]. Les méthodes ne sont pas utiles si elles ne sont pas compréhensibles et

faciles à utiliser. Différentes notations formelles ont différents niveaux de compréhension, mais elles sont relativement faciles à apprendre [8].

2.3.4 Niveaux de spécifications

L'architecture est en général décrite par des assemblages de modules à différents niveaux. La démarche adoptée est à deux dimensions :

- **structuration horizontale :** à un même niveau d'abstraction, on a des modules hiérarchises ;
- **structuration verticale :** on passe d'un niveau à l'autre par des choix de représentation.

On distingue deux grandes sortes de spécifications de modules :

- **spécifications fonctionnelles :** elles décrivent l'effet du module sur son environnement;
- **spécifications d'exploitation :** elles donnent les contraintes en termes de vitesse, d'utilisation des ressources et de temps de réponse.

Ici, nous ne parlerons que de la spécification fonctionnelle des modules par des techniques formelles [2].

2.3.5 Avantage des spécifications formelles

Les spécifications formelles peuvent être étudiées de manière mathématique :

- des formes d'inconsistance ou d'incomplétude peuvent y être décelées ;
- elles servent à prouver des propriétés sur les objets décrits ;
- deux spécifications alternatives peuvent parfois être prouvées équivalentes ;
- elles peuvent servir à prouver la correction d'un programme ;
- elles peuvent être exécutées sur un ordinateur lors d'un prototypage rapide pour déceler des vices de comportement.
- elles encouragent la rigueur dans le développement.
- elles sont un outil de documentation de programme
- elles sont utile pour la maintenance et la modification : si l'implantation change alors que la spécification est encore valide, les modules appelants n'ont pas à être modifies [2].

2.3.6 Classification des spécifications formelles

- **Fonctions**

On peut spécifier une fonction de manière :

- opérationnelle : on donne alors un procède de calcul, ou algorithme ;

- pré-post : on définit précisément les entrées, les sorties, et les relations entre entrées et sorties.

Montrer qu'une spécification opérationnelle satisfait une spécification pré-post, c'est faire une preuve de correction de l'algorithme. C'est la base des preuves de programmes. Hoare à propose un système logique pour conduire de telles preuves avec des programmes impératifs.

- **Données dynamiques**

On utilise ici des automates, ou machines à états, avec :

- des variables d'états, ayant un état initial ;
- des modifications, ou transitions, sur ces états.

Les opérations agissant sur ces machines sont soit des observateurs d'états, soit des modifications décrites comme des séquences de transitions.

D'une certaine manière, les objets des langages orientes objets, avec leurs méthodes, sont des machines à états. Mais, on en décrit des types appelés classes.

- **Données statiques**

L'idée est ici que :

- les données, ou objets, sont regroupées en types ;
- une donnée ne subit jamais de modification, et, comme en mathématiques, les opérations sont des fonctions définies entre types.

Mais les idées divergent encore selon qu'on pense ou non a la représentation des données.

- **Spécification par modèle abstrait :** on choisit une représentation des données, et les types sont vus comme des ensembles de représentations.
- **Spécification axiomatique :** on ne choisit pas de représentation des données a priori, et l'on travaille à un niveau formel (ou syntaxique), à distinguer du niveau interprétatif, ou sémantique, qui est celui des modèles, ou des représentations.

- **Spécification de comportements**

Devant l'importance croissante prise par les systèmes distribues ou repartis, on cherche des méthodes et des langages pour décrire la concurrence ou la coopération de leurs composants.

L'objectif est de pouvoir dominer le processus de conception de systèmes, de prouver ou expérimentalement des propriétés sur leur comportement dans le temps, et de donner un guide aux développeurs.

Pour ce faire, de nombreux formalismes ont été proposes. Certains ont pour origine des représentations graphiques agrémentées d'équations que l'on tente de traiter. C'est le cas des automates d'états finis ou infinis, des systèmes de transitions ou des réseaux de Pétri, dont un nombre incalculable de dérives ont été proposes [2].

2.3.7 Les méthodes de spécification formelle

Def1 Les méthodes formelles ont évolué depuis les études de McCarthy, Floyd, Hoare et plusieurs autres pionniers [11]. Une tendance de cette évolution fut l'étude de méthodes pour des systèmes complexes, par exemple les systèmes distribués. Une autre tendance, peut être la plus déterminante de l'utilisation des méthodes formelles, a été le déplacement de l'intérêt des chercheurs des dernières phases aux premières phases du cycle de développement, soit de la vérification de programme vers la spécification de système [1].

Def2 On parle d'une méthode formelle si elle est fondée sur un langage à syntaxe et sémantique rigoureuse, créé en se basant sur des concepts théoriques. Ces concepts sont communément mathématiques, des démonstrations sur la spécification sont donc envisageables par le biais d'un système formel : syntaxe et sémantique sont conduites par des règles de déduction permettant de raisonner de manière précise et démontrer des propriétés d'une spécification.

Nous pouvons citer comme exemples de cette technique : les automates à états finis, les réseaux de Pétri, les grammaires formelles, l'algèbre, la logique formelle, la théorie des graphes et le ¸λ-calcul.

On distingue différentes classifications pour les méthodes formelles J.M.Wing [21], divise ces dernières selon trois approches figure 1.1 :

- approches orientées données décrivant les états du système,
- approches orientées opérations décrivant le comportement du système par des axiomes,
- approches hybrides qui combinent les deux orientations.

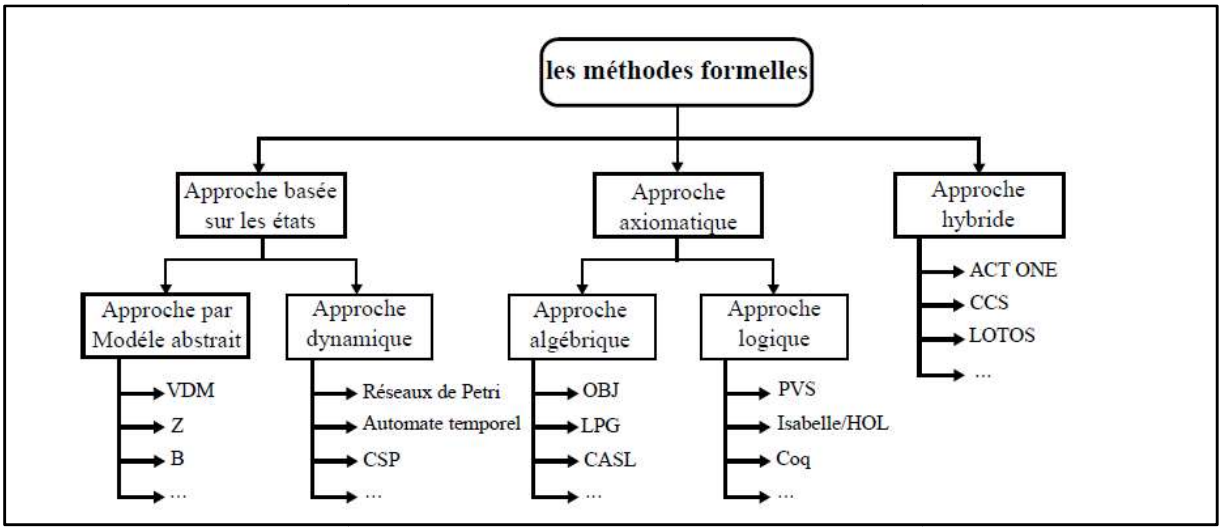


Figure1.1 : La classification des méthodes formelle

2.3.7.1 Les avantages des méthodes formelles

- L'avantage principal des méthodes formelles est l'utilisation de concepts de la logique et de la technique mathématique. Ces concepts fournissent des outils effectifs qui organisent les pensées des concepteurs et qui facilitent la communication entre toutes les personnes concernées par le développement.
- elles nous permettent de décrire de manière précise, non ambiguë, les demandes énoncées par l'utilisateur du système logiciel à réaliser. Les notions d'ensemble, de relation, de fonction et leurs différentes propriétés et opérations.
- elles nous permettent d'établir une spécification d'une manière simple et claire et de démontrer mathématiquement les propriétés de la spécification. Les termes de spécifications formelles ont une seule interprétation.
- les questions sont posées et répondues avec précision et d'une manière scientifique.
- elles fournissent des spécifications qui peuvent être rigoureusement vérifiées, analysées et testées dès les premières étapes du cycle de développement, Cela signifie qu'il possible de détecter et de corriger des fautes dès les premières étapes, ce qui réduit le coût et la durée du développement et améliore la qualité du logiciel.
- elles nous permettent de spécifier ce qui est nécessaire à un niveau d'abstraction particulier. Certains comportements et propriétés peuvent être volontairement exclus s'il est préférable que leurs élaborations soient remises aux prochaines phases du cycle de développement [1][9][12][13].

3. Les notations

3.1 La notation mathématique

Nous utilisons les structures mathématiques suivantes pour la spécification. Nous dénotons l'ensemble des sous-ensembles de X par P(X). Nous utilisons X x Y pour dénoter le produit cartésien de deux ensembles. Nous utilisons le tuple (x, y) pour dénoter un élément du produit cartésien. Nous dénotons les relations par *R*. L'expression x R y dénote que le tuple (x, y) est un élément de R (i.e., (x, y) ϵ R). Une fonction (partielle ou totale) d'un ensemble dans un autre ensemble est dénotée par f: X $\rightarrow$ Y. La composition de deux fonctions, dénotée par f $\circ$ g, est définie par f $\circ$ g (x) = f (g (x)). Nous présentons les notations des quatre méthodes formelles suivantes : relationnelle inductive, boîte noire par entités, langage B et assertions de traces.

3.2 La méthode relationnelle inductive

La méthode relationnelle inductive est une méthode de spécification formelle. Elle se situe à un niveau très abstrait (boîte noire). Elle se concentre seulement sur la description du comportement observable du système.

Les particularités de cette méthode sont : I ‘induction, le raffinement, la lisibilité et la possibilité de faire une vérification automatique. La méthode inductive s'articule autour de quatre concepts : la spécification relationnelle, la logique du premier ordre, la définition inductive et la sémantique donnée par le plus petit point fixe d'une fonction.

3.2.1 3.2.1. Spécification relationnelle

Ce formalisme consiste à identifier toutes les séquences d'entrées possibles du système et leurs sorties correspondantes.

3.2.1.1 La définition des axiomes

Lorsque les espaces d'entrée et de sortie ont été définis, la définition des axiomes doit être entamée. En utilisant la logique du premier ordre, une définition inductive est donnée par un ensemble d'axiomes**.** On peut diviser ces axiomes en trois classes :

- **Les axiomes de base**

Ils définissent toutes les sorties possibles pour les séquences de base

- **Les axiomes de réduction**

Ils servent à éliminer des symboles d'une séquence complexe donnée, afin de la réduire à une séquence de base.

- **Les axiomes de permutation**

Ils transforment une séquence donnée, qui n'est pas calculable par les axiomes de base ni par les axiomes de réduction, à une séquence calculable, en permutant des éléments.

3.3 La méthode boîte noire par entités

La méthode boîte noire par entités décrit formellement le comportement du système avec une relation d'entrée-sortie, comme dans la méthode relationnelle inductive. Elle s'inspire d'un concept issu de la méthode JSD (Jackson System Development), soit le diagramme de structure d'entité. De plus, elle est fondée sur l'algèbre de processus, les expressions régulières et la logique des prédicats du premier ordre. Ceux-ci nous permettent de décrire la relation R entre les entrées et les sorties du système.

3.3.1 Présentation de la méthode

Une spécification sous cette méthode se développe en quatre étapes :

- définition des espaces d'entrée et de sortie,
- la définition des entités du système en termes de séquences d'entrée,
- la définition des contraintes

- la définition de la relation les entrées et Les sorties.

Voici une description de ces étapes :

- **La définition des espaces d'entrée et de sortie**

Cette phase consiste à déterminer toutes les entrées du système et leurs attributs, et à définir les ensembles de sorties correspondant à chaque entrée.

- **La définition des entités du système**

L'objectif de cette phase est de définir toutes les entités du système et de décrire leurs comportements individuels en termes de séquences d'entrées.

- **La définition des contraintes**

Dans cette phase, on définit des contraintes sur les séquences d'entrées bien formées, afin de définir l'ensemble des séquences valides.

- **La définition de la relation d'entrée-sortie**

Cette dernière phase consiste à fixer toutes les sorties possibles pour chaque entrée.

3.4 La méthode B

La méthode B est une méthode pour la spécification et la conception de logiciels, fondée sur la logique du premier ordre, la théorie des ensembles et la théorie du raffinement.
Elle fait partie de la famille des méthodes dites orientées modèle qui représentent les logiciels par des données, caractérisées par leurs propriétés invariantes, et des services qui manipulent ces données.

Les caractéristiques de la méthode B sont les suivantes :

- elle dispose d'un seul cadre formel pour décrire abstraitement et concrètement les logiciels;
- elle autorise le développement progressif des modèles pour des transformations successives de leur spécification, appelées raffinements ;
- elle intègre les concepts d'encapsulation des données et de masquage de l'information, et a été conçue pour la construction structurée et modulaire de logiciels;
- elle fixe les conditions de vérification qui garantissent la cohérence de la spécification ainsi que la cohérence et la conformité à cette spécification, en ce qui concerne ces raffinements.

3.4.1 Les notations

La méthode B est composée de trois notations : la notation mathématique, la notation des substitutions généralisées et la notation des machines abstraites.

- **La notation mathématique**

Dans le langage de la méthode B**,** la notation mathématique permet de modéliser les données et les propriétés des données des logiciels. Cette notation est essentiellement la notation de la théorie des ensembles. Cependant, à la différence de la théorie classique des ensembles, la théorie des ensembles de B est typée: les ensembles sont constitués d'éléments ayant tous la même structure fondamentale.

- **La notation des substitutions généralisées**

La notation des substitutions généralisées permet de modéliser les services des logiciels, autrement dit, les actions que les logiciels peuvent effectuer pour remplir leurs fonctions. Une substitution généralisée est un transformateur de prédicats qui est appliqué aux prédicats qui caractérisent les données avant l'activation d'un service, produit les prédicats qui caractérisent les données après l'accomplissement de ce service [14].

- **La notation des machines abstraites**

La notation des machines abstraites définit les composants et les liens de composition des composants qui permettent de construire des modèles de logiciels structurés et modulaires. Il y a trois types de composants dans le langage de la méthode B : les machines abstraites, les raffinements et les implantations.

- **Machines abstraites**

Une machine abstraite définit les ensembles, les constantes, les variables et les opérations qui modélisent abstraitement les données et les services d'un logiciel. Ce faisant, elle définit l'interface opérationnelle du logiciel, parce qu'elle fixe le mode et les conditions d'utilisation de ses constituants.

- **Raffinements**

Généralement, une machine abstraite n'est pas un modèle exécutable du logiciel, pour obtenir un modèle exécutable, il est nécessaire d'enrichir et transformer la représentation abstraite des données et des opérations en une représentation concrète, le changement de représentation est réalisé par un ou plusieurs raffinements successifs.

- **Implantations**

Une implantation est le dernier raffinement d'une machine abstraite. C'est un modèle exécutable, le sous-ensemble du langage de la méthode B qui offre les structures de données fondamentales de programmation et les substitutions fondamentales de programmation.

3.5 La méthode des assertions de traces

La méthode des assertions de traces, proposée par Parnas [15] et améliorée par Bar-tussek [16] et puis par Wang [17]**,** permet de spécifier les comportements observables d'un système

sans référer aux structures internes de données. Une trace d'un module est définie comme une séquence d'appels des opérations appelées par les programmes d'accès.

3.5.1 Présentation

La méthode des assertions de traces est basée sur les concepts suivants: l'encapsulation de données, les séquences, les équations explicites et les machines à états. Pour modéliser un système avec cette méthode, on doit le décomposer en modules. Ces modules sont vus comme des boîtes noires ; on spécifie seulement leurs comportements observables.

Un module est considéré comme un ensemble de programmes qui fournissent l'accès à une structure de données. Le comportement d'un module est caractérisé par les événements reçus de l'environnement (appels de programmes d'accès). Trois types de programmes sont distingués : les programmes d'affectation qui changent les valeurs des données, les programmes d'interrogation qui consultent les données sans changer leurs valeurs et les programmes d'affectation et d'interrogation qui consultent les données et changent leurs valeurs.

La méthode des assertions de traces utilise des tableaux pour présenter la syntaxe des programmes d'accès, la fonction de transition d'état et les valeurs de sorties.

3.5.2 La structure de la spécification

La méthode propose un document de spécification bien structuré. Il est divisé en :

- la syntaxe,
- les traces canoniques,
- les traces équivalentes,
- les valeurs de sortie,
- le dictionnaire.

3.5.3 Le cas d'erreur

Une erreur a lieu lorsque l'élément d'extension d'une trace canonique définit un comportement indésirable, ce qu'on appelle la trace illégale.

4. Le processus des méthodes

4.1 La méthode inductive

Une spécification relationnelle inductive se présente comme une relation binaire entre les entrées et les sorties du système Intuitivement.

4.1.1 L'espace d'entrée et de sortie La stratégie utilisée est de sélectionner, à partir de cas réels du système toutes les entrées possibles et définir les espaces de sortie.

4.1.2 La génération des axiomes

Lorsqu'on a défini les espaces d'entrée et de sortie, il nous reste à générer les trois classes d'axiomes, ces axiomes doivent calculer les sorties correspondantes aux séquences d'entrées admissibles. , pour générer ces axiomes :

- déterminer tous les cas de base du système pour des séquences d'entrées : valides, invalides, retournant des messages d'erreur ;
- trouver les entrées qui n'ont aucun effet sur la valeur de sortie ;
- déterminer les entrées qui sont dépendantes entre elle;
- s'assurer que toutes les sorties peuvent être calculées par les axiomes générés.

La génération des axiomes est itérative. nous commençons par la génération des axiomes de base, puis des axiomes de réduction et enfin des axiomes de permutation.

- **Les axiomes de base** La génération commence par l'inventaire des entrées issues de la lecture du texte. Nous établissons une première version initiale et intuitive des axiomes, nous divisons les axiomes en trois classes: les axiomes qui calculent une commande facturée (cas valides), ceux qui calculent une commande refusée (cas invalides) et ceux qui produisent un message d'erreur (cas d'erreurs).
- **Les axiomes de réduction** La génération de ces axiomes met en évidence les relations entre entrées.
- **Les axiomes de permutation** En appliquant la méthode inductive, est de générer des axiomes qui servent à transformer, par permutation de symboles, des séquences d'entrée en d'autres séquences qui ont les mêmes comportements.

Remarque :

La méthode inductive se concentre seulement sur ce que le système doit faire et non sur comment le faire. Le cœur de cette méthode est basé sur la notion d'induction. Cette notion rend la spécification plus ou moins concise dépendamment du type de problème à résoudre.

4.2 La méthode boîte noire par entités

4.2.1 Les étapes de la spécification

La modélisation obtient à deux approches différentes l'une, relève de la spécification et repose sur la vision que se fait l'analyste de son système d'information. L'autre répond à une démarche systématique permettant de faciliter la validation du modèle obtenu. La manière la plus aisée de commencer la spécification consiste à travailler sur les quatre plans suivants :

- **la définition des espaces d'entrée et de sortie:**
- Établi la liste d'entrées
  - Définir les attributs de chaque entrée

- Pour chaque entrée, recueillir toutes les sorties possibles

- **la définition des entités et la description de leurs comportements individuels :**
  - Repérer les entités
  - Décrire le comportement individuel de chaque entité à l'aide des diagrammes de structure
  - Formaliser les diagrammes de structure des entités

- **la définition des contraintes du système :**

Cette étape débute par le recensement des contraintes et des conditions d'activation du système.

- **la définition de la relation d'entrée-sortie.**

Cette étape consiste à définir une relation entre les séquences d'entrées et les sorties du système. Cette relation est représentée par des axiomes qui seront appliqués aux séquences valides prédéfinies.

Pour modéliser, nous devons refaire les étapes utilisées et adapter la spécification.

- Ajout des entrées et des sorties
- Ajout des entités
- Ajout modification de contraintes
- Ajout des axiomes

Remarque :

La méthode boîte noire par entités couvre seulement la phase de spécification d'un logiciel. Elle est principalement fondée sur la notion de diagramme de structure d'entité de JSD, l'algèbre de processus et la logique des prédicats du premier ordre.

4.3 La méthode B

4.3.1 Représentation formelle des données définir :

- Les ensembles de base
  - Les ensembles différés
  - Les ensembles énumérés
- Les variables d'état

4.3.2 La spécification formelle définir la machine abstraite pour les variables prédéfinies :

- La machine
  - Les variables et leur initialisation
  - Les opérations de changement d'état
- Les opérations

- Les obligations de preuve

Une obligation de preuve est un théorème à démontrer. En fait, la démonstration des théorèmes n'est pas obligatoire, mais elle est souhaitable pour s'assurer que ce qui est écrit est parfaitement correct. Il existe deux modes pour démontrer les obligations de preuve : la preuve en mode automatique, la preuve en mode interactif.

Remarque :

La méthode B couvre la spécification et la conception des logiciels avec une notation homogène, et permet la traduction automatique en code exécutable, si on a suffisamment raffiné la spécification. La méthode B est fondée sur le principe d'obligations de preuve. La méthode B permet de construire des logiciels intégralement prouvés, c'est-à-dire conforme à leur spécification.

Contrairement aux méthodes relationnelle inductive et boite noire par entités, la méthode B oblige à réfléchir sur une construction modulaire du système ainsi que sur les variables d'état, les types et les opérations.

4.4 La méthode des assertions de traces

4.4.1 Les étapes de la spécification

- **La syntaxe** décomposer le système en modules et énumérer les programmes d'accès de chaque module ; pour chaque programme, définir les types des arguments et des valeurs de sortie, et la syntaxe pour chaque programme d'accès.
- **Les traces canoniques** identifier les programmes d'accès qui sont de type constructeur de données pour chaque module, définir les traces canoniques par les programmes identifiés.
- **Les équivalences de traces**
  - Énumérer tous les patrons possibles de traces canoniques ;
  - Formaliser toutes les conditions et les contraintes du système;
  - Définir les équivalences de traces pour toutes les traces canoniques étendues par chacun des programmes d'accès.
- **Les valeurs de sortie**
  - Lister les programmes d'accès ayant des sorties visibles ;
  - Déterminer et formaliser toutes les conditions et les contraintes du système relatives à ces programmes ;
  - Définir les sorties possibles pour toutes les traces étendues par chacun de ces programmes.

- **Le dictionnaire**

Définir toutes les fonctions auxiliaires utilisées dans les étapes précédentes.

Remarque :

La méthode des assertions de traces donne la structure du système en termes de modules et de liens entre ces modules. Un avantage de ce type de spécification est que la modification d'un module entraîne peu de modifications sur les autres modules. Une spécification par assertion est un automate où les états sont représentés par des traces [1].

5. Conclusion

Il est clair que la spécification joue une variété de rôles dans le cycle de développement. Une méthode qui fournit des spécifications précises, complètes et rigoureuses peut largement améliorer le processus de développement et la qualité du logiciel [1].

Dans ce chapitre, nous avons présentées la spécification et la vérification formelle sous quatre méthodes formelles.ses définitions, ses notations et ses méthodes d'utilisation, et nous avons fait une comparaison entre ses méthodes formelles a partir de citée ses étapes, avantages, et ses inconvénients. Dans le chapitre suivant nous verrons la spécification et la vérification formelle avec la méthode B en manière précise et on va spécifier un système informatique avec elle.

CHAPITRE 2

SPECIFICATION ET VERIFICATION FORMELLE AVEC LA METHODE B

1. Introduction

Spécification formelle d'un logiciel est la description précise mais abstraite de ce que le logiciel doit faire. La spécification aide au développement car elle vérifier formellement en partie automatiquement la correction de la conception, au but de La corriger par étapes avant d'investir dans l'implémentation, pour un résultat final d'implémentation correcte par construction. Plusieurs méthodes de spécification formelle, fondées sur : automates, réseaux de Pétri, logiques (équationnelles, des prédicats, temporelles etc.)

Parmi les méthodes de spécification formelle la méthode B. elle est Fondée sur la notion de machine abstraite, la méthode B est un Langage de spécification Logique, théorie des ensembles (langage de données), et elle est un langage des substitutions généralisées (langage des opérations), et un système formel de raisonnement (prouver des théorèmes)

2. La méthode B

2.1. Définitions

Méthode B

La méthode B a été présentée au milieu des années 80 par J.R. Abrial [18] auteur à L'origine de la notation Z [19]. C'est une technique de développement formel du logiciel dont tous les niveaux de construction sont pris en charge, de la phase de spécification jusqu'à la génération de code, en passant par la conception. Toute partie ou élément constitue une activité qui doit être approuvée par des preuves mathématiques. Ces preuves ont pour rôle ainsi de divulguer les erreurs et de guider la conception et la maintenance. La méthode B a été conçue au départ comme une méthode de développement de logiciel prouvé, toutefois tend actuellement à être plus généralement une méthode d'analyse et de construction système [20].

Développement formel

Développement formel de logiciel est la transformation systématique d'un modèle mathématique en code exécutable. Transformation de l'abstrait en concret Passage des structures mathématiques aux structures Informatiques Raffinement jusqu'au code dans un langage de programme [6]

Méthode formelle + théorie de raffinement (de machines abstraites)➔ développement formel.

2.2. Présentation

A l'origine, la méthode B a été introduite en tant qu'une méthode pratique afin de l'appliquer dans l'industrie. Le but initial est d'offrir une technique et des outils associés aidant au développement de logiciels de la manière la plus sure possible. Par conséquent, cette méthode s'appuie sur la preuve, à la différence aux autres techniques classiques de développement fondées sur l'exécution. B est une méthode transformationnelle dont l'étape de conception se rapporte à la création des modèles par approximations successives (i.e. raffinements) : on commence par un modèle abstrait, puis on en diminue le niveau d'abstraction jusqu'à l'obtention d'un modèle concret. Pour chaque phase, des preuves de consistance et de raffinement sont engendrées. Pour cela, la certitude (sûreté) du système est garantie par la satisfaisabilité des preuves [4].

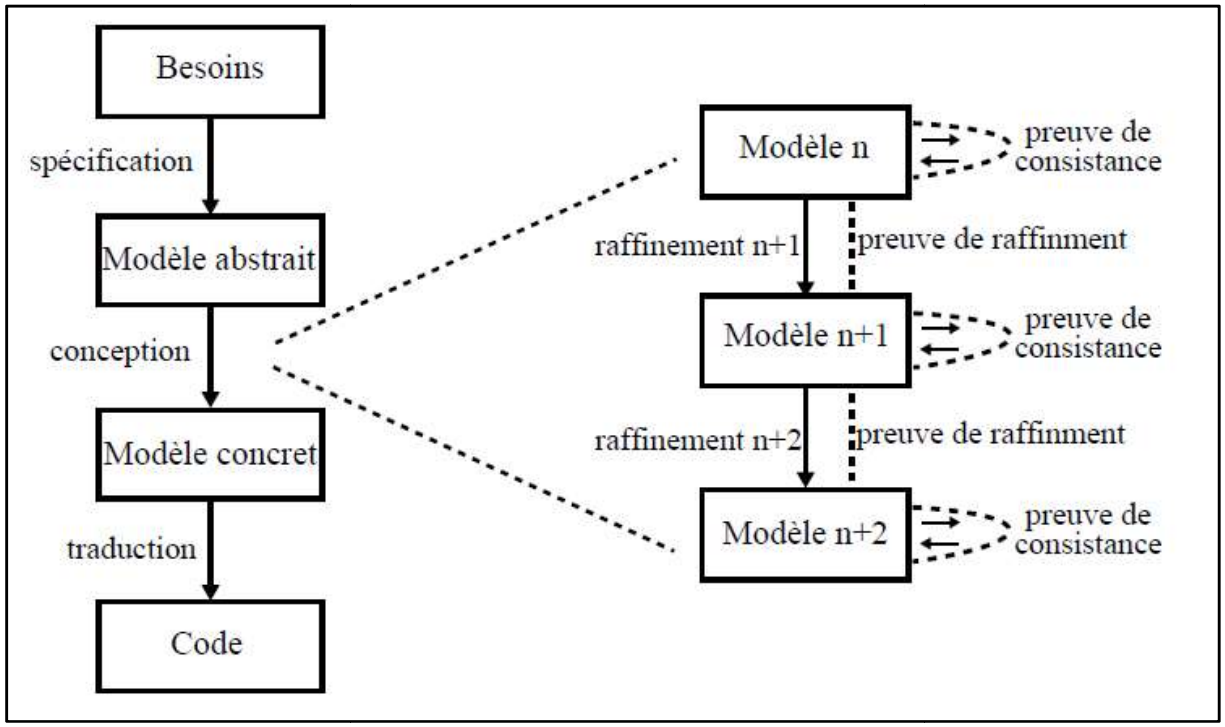


Figure 2.1: Cycle de développement de la méthode B

2.2.1. Langage B

Le langage B est le langage employé dans la méthode B. Il se base sur trois principes complémentaires : la logique des prédicats, la théorie des ensembles et les substitutions généralisées.

La logique de base du langage B s'appuie sur la logique des prédicats du premier ordre avec égalité et paires ordonnées. En ce qui concerne la théorie des ensembles de B, c'est une simplification du langage de la théorie des ensembles classique. Ce concept permet la spécification de l'aspect structurel ou statique des systèmes (les données). Cela a pour but d'une manière particulière de décrire l'invariant qu'on doit vérifier. On peut définir à partir

des ensembles : des entiers, des relations (ensemble de paires), des fonctions (relations particulières) et des séquences (fonctions particulières). Pour spécifier de façon très concise, divers opérateurs (notamment relationnels) sont employé. Les substitutions généralisées visent à représenter l'aspect comportemental ou dynamique des systèmes (les opérations). La sémantique de celle-ci est identifiée en fonction des transformateurs de prédicats. Les substitutions généralisées contiennent des instructions de spécification manipulant les notations de pré-condition et d'indéterminisme pour l'étape de spécification (modèle abstrait) ainsi que des instructions de programmation comme le séquencement et la boucle pour les étapes finales uniquement (modèle concret).

2.2.2. Modèle B

Un modèle mathématique en B est réalisé par un développeur tout en exprimant les propriétés aux quels le futur système doit suivre. Un modèle B est composé de machines abstraites spécifiant le système en tenant compte des aspects statique et dynamique :

- **Aspect statique.** Cette partie déclare des variables décrivant l'état du système modélisé. Ces déclarations sont enrichies par des conditions (i.e. invariants) spécifiant les propriétés que l'état du système (les variables) doit toujours satisfaire. Cette partie est représentée en utilisant la logique des prédicats et la théorie des ensembles.
- **Aspect dynamique.** Cette partie détermine les opérations représentant les changements d'état du système (l'évolution des machines abstraites) en utilisant des substitutions généralisées [4].

2.2.3. Raffinement

2.2.3.1. Définition

Def 1 Pour construire un système complexe en B, il est obligatoire de faire appel à des approximations successives. Par conséquent, sa réalisation ne peut s'effectuer en une seule fois. C'est la relation de raffinement qui joue le rôle pour lier les différentes approximations. Un raffinement est un composant qui garde la même interface et le même comportement que la machine abstraite mais qui reformule les données et les opérations de la machine à l'aide de données plus concrètes [22]. Ce concept est indispensable en B, car d'un côté, on va pouvoir l'employer pour décrire les détails de conception du cahier des charges qui n'étaient pas pris en compte auparavant. D'un autre côté, il est aussi employé pour concrétiser les modèles, celui-là afin de se diriger vers une implémentation [4].

Le modèle plus concret :

- contient plus de détails sur la spécification

- est plus proche d'une implantation
- réduit l'indéterminisme [5].

2.2.3.2. Types de raffinement

- **Raffinement de données**
  - Choix de l'implantation : Introduction de variables et d'ensembles **concrets**
  - Définition des opérations sur l'implantation : Réécriture des opérations en utilisant les variables concrètes
  - Définition d'une relation d'abstraction entre l'espace d'état concret et l'espace d'état abstrait: **l'invariant de collage**
- **Raffinement de contrôle**
  - Les opérations conservent la même signature
  - Élargissement des préconditions qui peut le plus, peut le moins
  - Réduction du non-déterminisme : choix de solutions ou d'options
- **Raffinement algorithmique**

Utilisation de structures de contrôle des langages de programmation : séquence et itération WHILE

Remarque

- Entre la machine abstraite et le raffinement, il y a changement de variable. L'invariant de la machine raffinement doit mettre en relation les variables abstraites et raffinées: c'est l'**invariant de liaison**.
- Entre la machine abstraite et le raffinement, il ne doit pas y avoir de changement de signature: mêmes opérations avec les mêmes paramètres **(**Pas de changement de signature). La machine abstraite peut être remplacée par la machine concrète sans que l'utilisateur de la machine ne puisse s'en rendre compte [5].

2.2.3.3. Approches du raffinement

Raffinement de substitutions:

Étant données deux substitutions S et T dans le contexte d'une machine abstraite M,**S** est raffiné par **T** si : T peut être utilisé à la place de S sans que l'utilisateur de la machine puisse s'en rendre compte. Pour cela, il faut que la précondition de T soit plus faible que celle de S, T soit plus déterministe que S, ce qui revient à renforcer la post-condition.

2.2.3.4. Propriétés du raffinement

Le raffinement B est correct: tout ce qui est vrai pour le composant raffiné l'est aussi pour le raffinement

Le raffinement B est transitif : SP1 ⊆ SP2 et SP2⊆ SP3 ➔ SP1 ⊆ SP3 Le raffinement peut se faire en plusieurs étapes élémentaires

Le raffinement B est monotone : SP1 ⊆ SP2 et SP3 ⊆ SP4➔ C (SP1;SP3) ⊆ C (SP2;SP4) Le raffinement peut se faire indépendamment des constructeurs ou de la composition choisie.

2.2.3.5. Définir un raffinement

- Les ensembles abstraits C sont implicitement présents dans MB
- Les variables abstraites a sont raffinées par les variables concrètes b
- Les variables concrètes b contiennent :
  - les variables abstraites conservées par le raffinement
  - des variables concrètes introduites par le raffinement
- L'invariant de collage J permet de :
  - exprimer typer les variables concrètes introduites par le raffinement
  - exprimer des propriétés sur les variables concrètes
  - la relation reliant les variables concrètes aux variables abstraites (d'où le nom d'invariant de collage)
- L'initialisation concrète Init0 est un raffinement de l'initialisation abstraite Init
- L'opération abstraite **Op** est raffinée par une opération concrète de même signature

2.2.3.6. Correction du raffinement

Définition Un raffinement est correct ssi à chaque effet de la spécification concrète correspond un effet de la spécification abstraite. Il faut vérifier cette définition pour :

- l'initialisation.
- chaque opération de la spécification abstraite.

2.2.3.7. Prouver un raffinement

```
MACHINE Abstraite
INVARIANT I
OPERATIONS
Oper =
PRE P THEN
S
END
END
```

```
REFINEMENT Concret
REFINES Abstraite
INVARIANT J
OPERATIONS
Oper =
PRE Q THEN
T
END
END
```

Obligation de preuve:

$I \wedge J \wedge P \rightarrow Q \wedge [T] \neg ([S] \neg J)$

Si I et J et P sont vérifiés .Alors Q est vérifiée et une fois effectuée la substitution concrète T.

Il faut prouver qu'il est faux que la substitution abstraite S si elle était appliquée établirait que l'invariant J de l'opération concrète ne soit pas vérifié (sachant par ailleurs que S établit l'invariant I).

2.2.3.8. Les clauses d'un raffinement

REFINEMENT NOM RAFFINEMENT
REFINES NOM MACHINE ou RAFFINEMENT /* Module raffiné */
SETS /* liste nouveaux ensembles abstraits et énumérés */
/* ceux de NOM MACHINE ou RAFFINEMENT sont visibles */
CONSTANTS /* ou ABSTRACT CONSTANTS */
CONCRETE CONSTANTS /* liste des constantes concrètes */
PROPERTIES /* type et propriétés des nouvelles constantes */
VARIABLES /* ou ABSTRACT VARIABLES */
CONCRETE VARIABLES /* liste des variables concrètes */
INVARIANT /* comprenant l'invariant de liaison */
/* le type et les propriétés des nouvelles variables */
ASSERTIONS
DEFINITIONS
INITIALISATION
OPERATIONS /* opérations avec la même signature */
END

Raffiner une opération

Dans un raffinement

- Il est interdit de définir de nouvelles opérations
- Les paramètres formels de l'opération doivent être identiques à ceux de l'opération raffinée
- Chaque paramètre conserve le même type que celui défini dans l'abstraction
- Le niveau d'indéterminisme doit diminuer
- Les préconditions doivent être affaiblies
- Les substitutions doivent être de plus en plus concrètes

2.2.3.9. Raffiner les substitutions: introduire les itérations

WHILE P Instruction boucle tant que
DO I
INVARIANT Q

VARIANT X

END

- Tant que le prédicat P est satisfait exécuter l'instruction I
- A chaque étape de l'itération, l'invariant Q doit être vérifié.
- La valeur entière de l'expression X du variant doit diminuer jusqu'à arriver à 0, sans jamais être négatif [5].

2.2.4. Implantation

- Raffinement : passage du quoi au comment
- Spécification abstraite : spécification des propriétés, pas de séquence, pas d'itération
- Implémentation : raffinement traduisible en un programme Ada, C, C++, ...
- Langage B : séquences, itérations, pas de non-déterminisme, pas de précondition, types de données concrets

2.2.4.1. Les clauses d'une implantation

IMPLEMENTATION NOM IMPLEMENTATION

REFINES NOM RAFFINEMENT /* Module raffiné*/

SETS

CONCRETE CONSTANTS /* liste des constantes concrètes */

PROPERTIES

VALUES /* valuer les ensembles abstraits et les constantes concrètes */

CONCRETE VARIABLES /* liste des variables concrètes */

INVARIANT /* comprenant l'invariant de liaison */

ASSERTIONS

DEFINITIONS

INITIALISATION /* expressions et substitutions concrètes */

OPERATIONS /* opérations avec la même signature */

/* expressions et substitutions concrètes */

END

Remarque:

Les clauses **ABSTRACT CONSTANTS** et **ABSTRACT VARIABLES** sont interdites [4].

2.2.4.2. Substitutions d'implémentations

- Substitutions déterministes = Instructions
- Pas de précondition
- Instructions :

I ; J	Instruction séquence
BEGIN I END	Instruction bloc
VAR a,b IN I END	Instruction variable locale
	Instructions devient égal pour:
a,b := x, y	- une liste de variables
f(x) := y	- un élément de fonction
ASSERT P THEN I END	Instruction assertion
IF P THEN I	Instruction conditionnelle IF
[ELSIF Q THEN J]_	
[ELSE K] END	
CASE X OF	Instruction condition par cas
EITHER a ; b THEN I	
[OR a'; b' THEN J]*	
[ELSE K] END	
[a; b <–] f [(x; y)]	Instruction appel d'opération
WHILE P DO I	Instruction boucle tant que
INVARIANT Q	
VARIANT X	
END [5]	

2.2.5. Preuve

Les preuves effectuées pendant la réalisation d'un système en utilisant la méthode B vérifient que le système construit dispose bien des propriétés qu'on attendait de lui (preuves de conservation d'invariants et de correction du raffinement).Concernant les premières preuves, on s'assure que les propriétés représentées dans les invariants sont maintenues après exécution des opérations. Les secondes garantissent que les propriétés précédemment prouvées dans l'abstraction demeurent valables dans le raffinement. On note que les preuves à satisfaire sont déterminées rigoureusement dans la méthode, le développeur n'a pas à les écrire. L'analyse des modèles ainsi que la génération des diverses conditions qu'il est obligatoire de prouver est réalisé au moyen d'un outil (générateur d'obligations de preuve). L'objectif de tout cela est d'assurer la cohérence de l'ensemble du système [4].

2.2.6. Domaines d'application

On distingue deux sortes d'ateliers génie logiciel reliés à la méthode B qui permettent sa mise en œuvre dans des grands projets industriels. Le primer atelier est l'Atelier B, qui a été introduit par la l'organisation STERIA (France) ; et le deuxième atelier est celui développé

par la société B-Core (Royaume Uni) qui est appelé B-Toolkit. Les outils caractérisant ces deux ateliers ont pour objectif de :

- Analyser lexicalement des machines (spécification), des raffinements ou des implémentations,
- Vérifier des types,
- Générer des obligations de preuve,
- Démontrer automatiquement des preuves,
- Démontrer interactivement des preuves,
- Traduire les modèles en langages de programmation impératifs (C, C++, ou AD A),
- Gérer et documenter des projets [4].

2.3. Fondements théoriques

Le formalisme de la machine abstraite décrit toutes les phases de la méthode B, allant de la spécification à l'implémentation, en passant par la conception (raffinement). Afin de montrer le principe de fonctionnement de cette méthode, il nous faut mettre l'accent sur les fondements théoriques de son formalisme. Nous commençons par la description des notations mathématiques élémentaires, et allons jusqu'aux concepts fondamentaux que sont les machines abstraites [4].

2.3.1. Notations mathématiques

Le langage B s'appuie sur les trois notations mathématiques : la logique prédicative du premier ordre, la théorie des ensembles et le langage des substitutions généralisées.

2.3.1.1. La théorie des ensembles et les types La théorie des ensembles se sert comme la base des modèles mathématiques en B. Les relations, fonctions et suites sont décrites en tant que des couples d'ensembles. Quoique les variables en langage B soient strictement typées, leur type n'est pas défini explicitement à la déclaration comme pour la majorité des langages de programmation mais par des prédicats au sein de l'invariant, les pré-conditions ou les contraintes des paramètres formels. Ces prédicats indiquent le typage des variables par une condition d'appartenance à un ensemble donné [4].

- **Langage de modélisation des données**
  - **Ensembles et typage**

A l'aide du langage des données

- On modélise l'état d'un système avec les données qui le caractérisent

- on explicite les propriétés invariantes d'un système

Modélisation de l'état :

- Abstraction, modélisation (ensembles abstraits, relations, fonctions, ...)
- Propriétés logiques, algébriques
- Lorsqu'on modélise un système (par l'ensemble de ces états) puis qu'on explicite ses (bonnes) propriétés, on s'assure après que le système ne parcourt que l'ensemble des états qui respectent les propriétés : c'est la cohérence du système.
- Pour montrer qu'il est possible d'avoir des états satisfaisant les propriétés énoncées, on exhibe au moins un état (l'état initial).
- Le système spécifié est correct si après chacune de ses opérations, l'état obtenu est un état respectant les propriétés invariantes [6].

▪ **Types de données concrets**

- Les ensembles énumérés
- Ensembles prédéfinis (statut de type)
- Produit cartésien E × F
- Ensemble des sous-ensembles P*E* noté POW(E)
- Les entiers: NAT, NAT1 ou INT compris entre MININT et MAXINT
- Les booléens**:** BOOL
- Les sous-ensembles d'éléments de types concrets
- Les intervalles d'entiers concrets
- Les tableaux
- Les records
- Les chaînes de caractères: STRING (uniquement comme paramètre d'opération)

▪ **Expressions arithmétiques du B**

m + n	m + n	Addition
m - n	m - n	Soustraction
m × n	m * n	Multiplication
m /n	m / n	Division entière
$(m)^n$	m**n	Puissance
m mod n	m mod n	Reste de la division entière
succ (n)	succ (n)	Fonction successeur
pred (n)	pred (n)	Fonction prédécesseur
m = n	m = n	Relation d'égalité

$m \neq n$ m /= n Relation d'inégalité

$m \leq n$ m <= n Relation inférieur ou égal

$m < n$ m < n Relation inférieur

$m \geq n$ m >= n Relation supérieur ou égal

$m > n$ m > n Relation supérieur [5]

Désignation	Notation	Ascii
et	$p \wedge q$	p & q
ou	$p \vee q$	p or q
non	$\neg p$	not p
implication	$p \Rightarrow q$	(p) ==> (q)
quantif. univ.	$\forall x.p(x)$	!x. (p(x))
quantif.exist.	$\exists x.p(x)$	#x. (p(x))

Table 2.1 Logique du premier ordre

Désignation	Notation	Ascii
union	$E \cup F$	E \/ F
intersection	$E \cap F$	E /\ F
appartenance	$x \in F$	x : F
différence	$E \setminus F$	E – F
inclusion	$E \subseteq F$	E <: F
sélection	choice(E)	__

Table 2.2 Les opérateurs ensemblistes classiques

Désignation	Notation	Ascii
non appartenance	$x \notin F$	x /: F
non inclusion	$E \not\subseteq F$	E /<: F
non égalité	$E \neq F$	E /= F

Table 2.3 Les notation ascii la négation est réalisée avec /.

Désignation	Notation	Ascii
relation	$r : S \leftrightarrow T$	r : S <–> T
domaine	$dom(r) \subseteq S$	dom(r) <: S
image	$ran(r) \subseteq T$	ran(r) <: T
composition	$r ; s$	r ; s
composition r(s)	$r \circ s$	r(s)
identité	$id(S)$	id(S)
restrictition domaine	$S \lhd r$	S <| r
restriction codomaine	$r \rhd T$	r |> T
antirestriction domaine	$S \lhd r$	S <<| r
antirestriction codomaine	$r \rhd T$	r |>> T
inverse	r^{-1}	r ~
image relationnelle	r[S]	r[S]
écrasement	$r1 \oplus r2$	r1 <+ r2
produit direct de rel.	$r1 \otimes r2$	r1 >< r2

fermeture fermeture réflexive trans.	closure(r) closure1(r)	closure(r) closure1(r)

Table 2.4 Les relations

Désignation	Notation	Ascii
fonction partielle	$S \nrightarrow T$	S +-> T
fonction totale	$S \rightarrow T$	S --> T
injection partielle	$S \rightarrowtail T$	S >+-> T
injection totale	$S \rightarrowtail T$	S >–>
surjection partielle	$S \twoheadrightarrow T$	S +->> T
surjection totale	$S \twoheadrightarrow T$	S -->> T
bijection totale	$S \rightarrowtail T$	S >->>
lambda abstraction	$\%x.\ (P \mid E)$	

Table 2.5 Les fonctions

Désignation	Notation
suite d'éléments de T	$seq(T) = union(n).\ (n\ 2\ IN \mid 1..n\ !\ T)$
suite vide	[]
suite inj. d'élém. de T	$iseq(T)$
suite bij. d'élém. de T	$perm(T)$
taille d'1 séq. s	$size(s) = card\ (dom(s))$
premier élém. d'une séq.s	$first(s) = s(1)$
dernier élém. d'une séq.s	$last(s) = s(size(s))$
restric. de s à ses n prem. éléments	$s \uparrow n$
élimination de n premiers éléments de s	$s \downarrow n$

Table 2.6 Les séquences

2.3.1.2. Le langage des substitutions généralisées L'ensemble composé de modules, appelés ainsi machines abstraites, constitue des spécifications B. Chaque machine comprend un état et des opérations qui permettent d'accéder à celui-ci ou de le changer. Les substitutions constituent les processus déterminant et changeant l'état dont leur sémantique est donnée par les substitutions généralisées. Le concept de substitution est proche au mécanisme d'affectation des langages de programmation. Ce concept a été étendu en B afin de modéliser la transformation de prédicats [4].

- **Substitutions généralisées**
  - **Concepts de base pour la partie dynamique**

Les plus faibles préconditions

Contexte : Logique de Hoare/Floyd/Dijkstra triplet de Hoare (Etat, espace d'état, commandes, exécution, triplet de Hoare)

$$\{P\}\ S\ \{R\}$$

S une commande et ***R*** un prédicat décrivant le résultat de ***S.***

$\omega p(S, R)$, prédicat qui représente : l'ensemble de tous les états | exécution de S commençant par un d'entre eux **se termine** en un temps fini dans un état satisfaisant R, $\omega p(S, R)$ est la plus faible précondition de S par rapport à R.

Le sens de $\omega p(S, R)$ peut être précisé par deux propriétés : $\omega p(S,R)$ est une précondition garantissant R après l'exécution de S, c'est à dire que :

$$\{\omega p(S,R)\}\ S\ \{R\}$$

$\omega p(S,R)$ est la plus faible de telles préconditions, c'est à dire que : si $\{P\}\ S\ \{R\}$ alors $P \rightarrow \omega p(S, R)$

En pratique un programme S établit une postcondition R. Intérêt pour les préconditions qui permettent d'établir R.

ωp est une fonction à deux arguments : une instruction (ou programme) S et un prédicat R. Pour un S fixé, on peut voir $\omega p(S, R)$ comme une fonction à un seul argument $\omega pS(R)$.
La fonction ωpS est appelé transformateur de prédicats. C'est la fonction qui associe à tout prédicat R la plus faible précondition telle que $\{P\}\ S\ \{R\}$.

- **Axiomes**

Généralisation de la substitution simple de la logique classique (pour modéliser des comportements d'opérations).
Soit R un prédicat à établir, la sémantique des substitutions généralisées est définie par le transformateur de prédicat.

- **Substitution simple** S sémantique **[S] R** se lit : S établit R
- **Substitution multiple** $x, y := E, F$ Sémantique **[x, y := E, F] R**

Jeu de base

Le langage de syntaxe abstraite pour spécifier les opérations : Soit R l'invariant, S, T des substitutions

Nom	Synt. Abs	définition	équivalent logique
neutre (id.)	*skip* [*skip*]	R	R
Pré-condition	$P \mid S$	$[P \mid S]\ R$	$P \wedge [S]\ R$
Choix borné	$S\ []\ T$	$[S\ []\ T]\ R$	$[S]\ R \wedge [T]\ R$
Garde	$P \rightarrow T$	$[P \rightarrow T]\ R$	$P \rightarrow [T]\ R$
non borné	$@x.S$	$[@x.S]\ R$	$x.[S]R$
	x non libre dans R		

Table 2.7 Substitutions

Non-déterminisme substitutions

- **Abstraction** =>non déterminisme possible. OK pour spécifier
- **concrétisation** => raffinement vers code
- Extension du jeu de base à d'autres substitutions proches de la programmation

Extension syntaxique des substitutions :

choix borné *S* [] *T* Extension syntaxique CHOICE S OR T END	**Substitution simple** notée *S* Extension syntaxique BEGIN S END	**Substitution avec garde** (*P* => *T*) [] (¬ *P* => *S*) Extension syntaxique IF P THEN T ELSE S END

Subst. avec précondition *P* | *S* Extension syntaxique PRE P THEN S END	**Substitution de choix non borné** *@x.Sx* Extension syntaxique VAR x IN Sx END

Extension du jeu de base : non-déterminisme

Nondéterminisme *x* :∈ *U* (Devient appartient) x:: U *@y*. (*y* ∈ *U* => *x* := *y*) Extension syntaxique ANY y WHERE y : U THEN x := y END	**Nondéterminisme** *x* : *P*(*x*) (x tel que P) x: P(x) BEGIN x : (x < 30) END

2.3.2. Machine abstraite B

La notion de machine abstraite est similaire aux notions de classe, module, paquetage ou autres objets des langages de programmation. Elle constitue l'unité de structuration d'une spécification B. Celle-ci vise à réaliser les logiciels de façon modulaire et structurée. Elle peut être réalisée ainsi, à partir d'autres machines. L'encapsulation des données et les opérations qui les manipulent est un concept très important pour une machine abstraite. Plus précisément, cette dernière est un module, constitué d'un côté d'un état : représenté par les données et l'invariant que ces données doivent respecter ; et d'un autre côté, de la description des transformateurs de cet état : la dynamique du système est modélisée par les opérations.

La notion d'encapsulation est un moyen de protéger l'état d'une machine abstraite. De ce fait, il est crucial d'introduire non seulement des opérations de modification mais aussi des opérations permettant aux utilisateurs extérieurs de consulter l'état d'une machine. **Structure générale d'une machine abstraite.** Une machine abstraite est généralement composée par des clauses. Nous distinguons des clauses déterminant les données et d'autres les opérations. Une brève description des principales clauses d'une machine abstraite :

MACHINE M(X,x) //en-tête de la machine : nom et paramètres formels
CONSTRAINTS //définition du type et des propriétés des paramètres formels
C
SETS //déclaration d'ensembles abstraits et définition d'ensembles énumérés
S ;
T = {a,b}
CONSTANTS // déclaration des constantes
C
PROPERTIES //définition du type et des propriétés des constantes et ensembles
P
VARIABLES //déclaration des variables
V
INVARIANT //définition du type et des propriétés des variables
I
INITIALISATION //initialisation des variables
U
OPERATIONS //déclaration et définition des opérations
U←Op(w) =
PRE

Q

THEN

V

END

DEFINITIONS //alias utilisables dans le texte des autres clauses

D(z)=X

END

- La clause MACHINE définit le nom identifiant la machine abstraite *M*, suivi d'une liste de paramètres *(X,x)* ; où *X* décrit des ensembles abstraits et *x* des scalaires.
- La clause CONSTRAINTS définit un prédicat C permettant de typer les scalaires des paramètres et de spécifier des contraintes sur ces scalaires ainsi que sur les ensembles des paramètres.
- La clause SETS décrit les ensembles définis par la machine. Ce sont soit des ensembles abstraits (*S*), soit des ensembles énumérés (*T*) déterminés par la liste de leurs éléments. L'utilisation des ensembles abstraits a pour but de désigner des objets dont on ne veut pas définir la structure au niveau de la spécification. Tout ensemble abstrait est déterminé non vide et fini. Les ensembles énumérés sont définis par les éléments de leur énumération *(a,b)*.
- La clause CONSTANTS déclare une liste d'éléments dont la valeur ne peut être modifiée *(c)*.
- La clause PROPERTIES désigne un prédicat *P* permettant, d'un côté, de typer les constantes, et d'un autre côté de spécifier des conditions sur les ensembles et les constantes.
- La clause VARIABLES définit la liste des variables de la machine *(v)*. Celles-ci décrivent l'état de la machine. Ce sont les seuls objets qui peuvent être modifiés au sein de l'initialisation et les opérations de la machine.
- La clause INVARIANT collecte un ensemble de prédicats *(I)*. Ces prédicats déterminent les propriétés mathématiques des variables et permettent de typer celles-ci. Ils définissent aussi les contraintes (autre que le typage) que doit vérifier l'état de la machine à tout moment. Cette clause est inévitable si la clause VARIABLES est présente.
- La clause INITIALISATION est constituée des substitutions *(U)* définissant les valeurs initiales pour chaque variable propre à la machine. Toute variable propre à la

machine doit être initialisée. L'initialisation doit satisfaire l'invariant de la machine. Cette clause est inévitable si la clause VARIABLES est présente.

- La clause OPERATIONS définit une liste d'opération. Ces opérations peuvent être paramétrées en entrée *(w)* et en sortie *(u)* par une liste de scalaires. Lorsqu'une opération est paramétrée en entrée, une pré-condition *Q* est employée pour typer les paramètres. La substitution *V* définit le comportement de cette opération par rapport à l'état de la machine. Une machine abstraite offre des opérations qui sont appelables à partir d'autres opérations/programmes externes.

Quelques règles de spécification en B

- Une opération d'une machine ne peut pas appeler une autre opération de la même machine (violation de la PRE) ;
- On ne peut pas (de l'extérieur) appeler en parallèle, deux opérations d'une même machine;
- Une machine doit comporter des opérations auxiliaires pour tester les préconditions des principales opérations fournies ;
- L'appelant d'une opération doit vérifier ses préconditions avant l'appel ("On ne doit pas diviser par 0") ;
- Lors des raffinements, on affaiblit les PREconditions jusqu'à les faire disparaitre (ce n'est pas le cas en B événementiel) ;
- Finalement, la clause DEFINITION définit des alias *(D)*, éventuellement paramétrés *(z)*, sous la forme d'une expression *X* qui peut être utilisée dans le corps de toutes les autres clauses.
- On peut utiliser les constituants des différentes clauses dans d'autres, à l'intérieur d'une même machine abstraite. Nous pouvons noter que les opérations d'une machine ne sont pas visibles au sein des autres clauses de la même machine. Des contraintes sont imposées par la méthode et alors imposées également au niveau des machines abstraites :
- Le séquencement des substitutions n'est pas permis (ceci est réservé au raffinement), seule la substitution parallèle doit être employée.
- La même chose pour les substitutions de boucle, elles ne sont pas autorisées au niveau de la spécification abstraite.

2.3.3. Développement B par raffinement successif

Dans la méthode B, le raffinement est le processus transformateur graduel des spécifications vers des implémentations qui peuvent être traduites en langage de

programmation. C'est le mécanisme qui lève le non-déterminisme et précise des choix de réalisation. Son principe est montré dans la Figure 2.2.

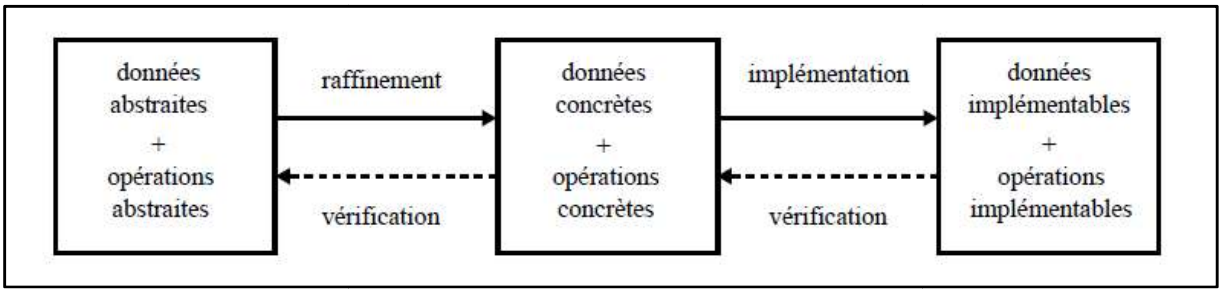


Figure 2.2 : Principe général de développement B par raffinement successif

Soit *M* une machine abstraite, son raffinement *R* est un nouveau composant plus déterministe que *M*, il modifie éventuellement l'état ou les opérations de *M*, cependant, perçu de l'extérieur, il dispose d'un comportement compatible avec celui de *M*. En général, deux types de raffinement sont distingués :

- **Le raffinement de données :** il sert à transformer des variables d'état en variables plus concrètes (plus proches des types de données classiques) ou à ajouter de nouvelles variables pour introduire des détails de réalisation ;
- **Le raffinement procédural :** il consiste en la modification du corps des opérations d'une machine en diminuant le non-déterminisme, affaiblissant les pré-conditions ou ajoutant plus de détails de conception particulièrement en utilisant des structures de contrôle plus concrètes.

B discerne l'implémentation comme un ultime niveau de raffinement. Celle-ci est considéré en tant que le niveau le plus concret et directement traduisible en langage de programmation. Toutes les variables à ce niveau doivent être des structures de données programmables, les ensembles et les constantes sont remplacées par des valeurs, des structures de contrôle comme les itérations peuvent être utilisées. Le non-déterminisme, la pré-condition ainsi que le parallélisme sont éliminés.

Le raffinement peut être effectué soit en une seule étape par le passage d'une machine abstraite à une implémentation, soit par une suite d'étapes successives en passant d'une machine abstraite à un raffinement, d'un raffinement à un autre raffinement ou encore directement d'un raffinement à une implémentation. Un raffinement se diffère d'une machine abstraite en se focalisant sur les points suivants :

- la clause MACHINE est remplacée par REFINEMENT si c'est un raffinement intermédiaire ou IMPLEMENTATION si c'est un raffinement ultime ;

- il dispose d'une clause REFINES spécifiant le nom de la machine ou du raffinement qui est raffiné ;
- il déclare les mêmes opérations (elles ont la même signature) que celles du composant qu'il raffine, mais seul le corps qui peut être modifié ;
- Pas de spécifications explicites pour les nouveaux paramètres de composant, ceux-ci sont décrits de manière implicite par le composant raffiné ;
- l'invariant d'un raffinement contient des prédicats de liaison entre les nouvelles variables et les variables du niveau précédent (composant raffiné) ; il est baptisé invariant de liaison (ou invariant de collage).
- un raffinement est transitif et monotone. Il est équivalent d'utiliser une machine abstraite ou son raffinement, et un raffinement peut utiliser d'autres machines abstraites. Mais la machine abstraite reste toujours l'interface.

L'implémentation en B est un raffinement particulier et unique contraint par certaines restrictions. Le but de est d'assurer d'un côté que l'implémentation est concrète et d'autre part qu'elle corresponde uniquement à la spécification d'autres machines abstraites et non à leur implémentation :

- L'implémentation ne dispose pas d'état propre ;
- Les opérations sont implantées par le mécanisme d'importation d'opérations d'autres machines ; les variables des composants importés ne peuvent pas être modifiées ou référencées par les opérations de l'implémentation, seules les opérations importées y sont autorisées;
- L'instanciation de paramètres des composants importés est effectuée dans la clause IMPORTS ;
- Interdiction des substitutions non-déterministes et de la composition parallèle.

2.3.4. Obligations de preuves

Puisque la méthode B supporte le processus de raffinement, par conséquent un mécanisme de preuve lui est associé. Cela sert à démontrer que le modèle raffiné est cohérent avec le modèle abstrait.

La généralisation d'un raffinement pour une machine abstraite est effectuée par le raffinement de ses opérations. De façon informelle, une substitution *S* est raffinée par une substitution *T* si le comportement de *T* correspond au comportement attendu de *S* à la condition que les deux agissent sur le même état (variables). Donc intuitivement, la substitution *T* est le raffinement de *S* où *S* est l'abstraction de *T*.

Les prédicats à prouver pour s'assurer de la cohérence (et de la correction) du modèle mathématique défini par la machine abstraite. Le développeur de la machine abstraite a deux types d'obligations de preuve :

- Prouver que l'INITIALISATION établit l'invariant ;
- Prouver que chaque OPERATION, lorsqu'elle est appelée sous sa précondition, préserve l'invariant.

$$I \wedge P \Rightarrow [Subst]I$$

Dans la pratique, on s'équipe d'outils ou d'ateliers qui aident à décharger ces preuves.

Sémantique d'une machine - Cohérence

Pour garantir la correction d'une machine, on doit s'affranchir des principales obligations de preuve :

- L'initialisation établit l'invariant
- Il est possible d'avoir des valeurs de paramètres satisfaisant les contraintes
- Il y a des ensembles et des constantes qui satisfont les propriétés de la machine
- Il y a un état satisfaisant l'invariant
- Chaque opération de la machine, lorsqu'elle est appelée sous sa précondition, préserve l'invariant.

Ce sont là des expressions logiques, des prédicats, qui sont prouvés.

2.3.5. Composition de spécifications B

Principe général De même que pour le développement logiciel, B offre la capacité de décomposer un système en plusieurs sous systèmes plus simple à spécifier, développer et prouver. Afin d'écrire des spécifications et partager des informations, différentes possibilités sont fournies pour décomposer. Ces possibilités sont désignées par des termes clause de liaison ou clause d'assemblage.

Nous décrivons au départ les clauses INCLUDES, IMPORTS, SEES et USES pour entamer ensuite d'autres clauses qui ont pour but, soit d'étendre les capacités de composition des précédentes clauses, soit de servir de «sucre syntaxique» afin de combiner différentes clauses. Un projet B représente les dépendances comme un graphe acyclique. Nous pouvons exprimer ce dernier sous forme d'un arbre, mais nous ne tenons compte qu'à des liens IMPORTS et REFINES. Par conséquent, cette situation limitée nous permet d'utiliser le terme « arborescence des dépendances ».

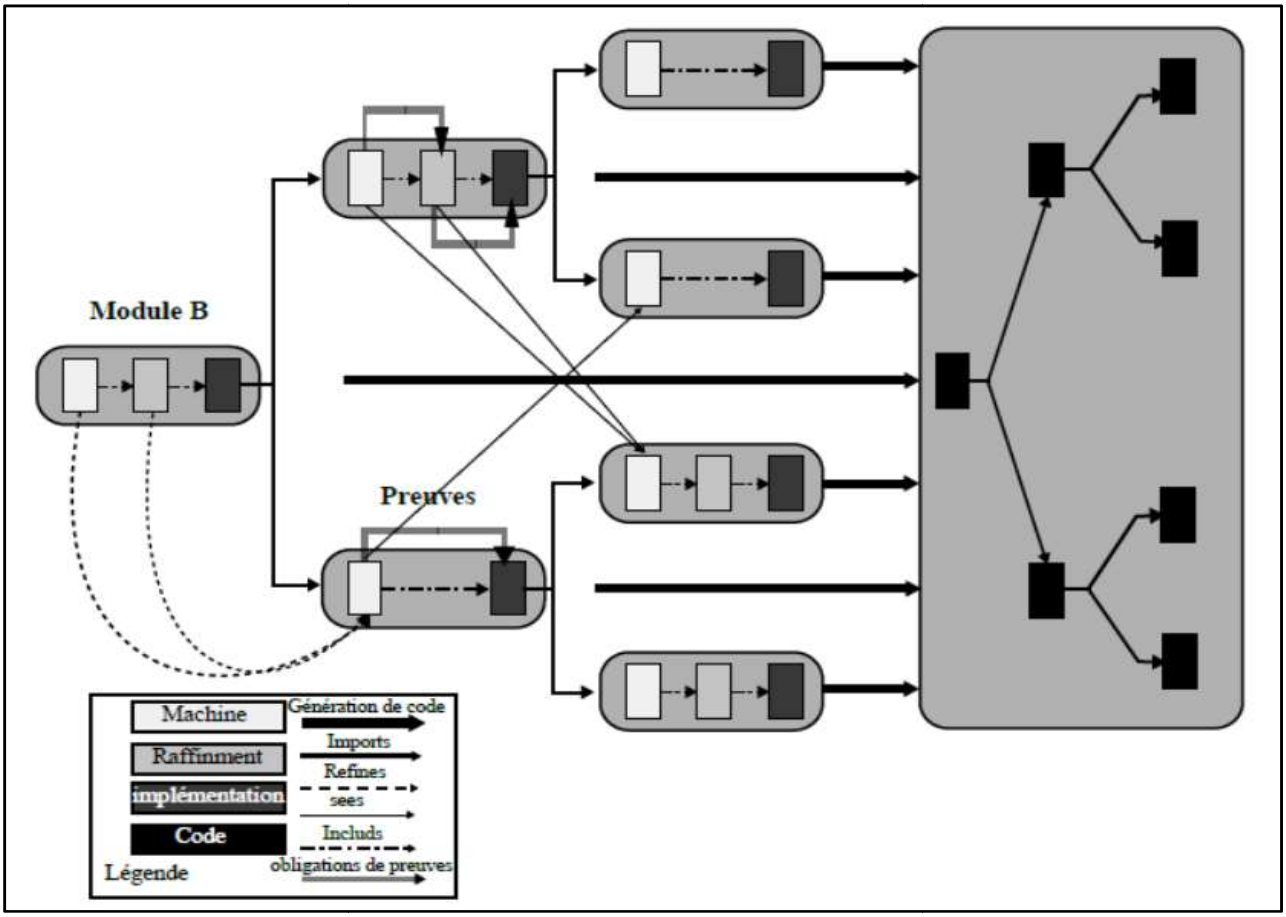


Figure 2.3 : Décomposition d'un projet B

3. Conclusion

La méthode B couvre toutes les phases amont du cycle de vie pour le développement d'un logiciel à savoir la spécification, la conception (raffinement) et l'implémentation dont toute phase étant validée par des preuves. Le grand avantage, est que celle-ci est soutenue par des outils performants employés dans l'industrie. En réalisant un système en utilisant le développement B, implique la construction des modèles mathématiques décrivant les propriétés auxquelles le futur système doit obéir. Ces modèles sont réalisés progressivement commençant par des composants plus abstraits jusqu'à l'obtention d'un modèle concret traduisible en un programme exécutable. Au cours de ce développement, des preuves sont effectuées pour l'objectif de s'assurer que le système construit possède et vérifie bien les propriétés souhaitées. Les preuves ici, servent non seulement à prouver les différents modèles réalisés mais aussi à vérifier leur cohérence au cours des raffinements.

Dans ce chapitre nous avons présenté la méthode B et ses concepts de base, ses notations, ses avantages et sa contribution à la spécification formelle des logiciels et des programmes.

Dans le chapitre suivant, nous allons spécifier un système de contrôle d'ascenseur avec la méthode B.

CHAPITRE 3

SPECIFICATION ET VERIFICATION FORMELLE D'UN SYSTEME D'ASCENSEUR

1. Introduction

Pour expliquer l'utilité de la méthode B dans le processus du développement des systèmes, nous présentons la spécification d'un système d'ascenseur qui peut être trouvé dans la littérature [4,23].

Premièrement, sa conception est présentée par des diagrammes UML. Le premier diagramme est le diagramme de classe où deux calasses sont présentées : la première est la classe « lift » (représentant le cabinet de l'ascenseur) et la deuxième classe est pour représenter le « button ». Les opérations sont spécifiées à l'aide d'un diagramme d'états transitions. Ensuite, nous donnons les spécifications formelles B appropriées.

Pour s'assurer de la fiabilité et de la sûreté de fonctionnement du système, nous vérifions ces spécifications. Ces vérifications syntaxiques et sémantiques sont réalisées au moyen des prouveurs. Enfin, nous générons automatiquement le code C à partir de cette spécification B. L'Atelier B est utilisé au long de ces taches.

2. Système de contrôle d'ascenseur

La figure 3.1 présente une vue générale d'Astah community. Dans cette figure, Astah a été utilisé pour modéliser en UML un système de contrôle d'ascenseurs en respectant les règles suivantes :

– les ascenseurs desservent les étages d'un bâtiment ;

– les ascenseurs et les étages disposent d'un ensemble de boutons. Chacun de ces boutons symbolise un étage. Les boutons à l'intérieur des ascenseurs sont des boutons d'indication, ils précisent lorsqu'un usager est dans l'ascenseur, l'étage où il souhaite se rendre. Les boutons situés à chaque étage sont des boutons d'appel de l'ascenseur pour cet étage ;

– demander la visite d'un ascenseur à un étage donné correspond à un appel d'un ascenseur en appuyant sur un des boutons associé à cet étage. Quand un bouton a été activé, il s'allume et reste allumé jusqu'à ce que la visite de l'étage correspondant soit effective ;

– un ascenseur en mouvement a toujours un étage de destination prioritaire. Celui-ci est changé lorsqu'il a été visité et qu'il y a une demande de visite à un autre étage. Ainsi, le mouvement de l'ascenseur dépend de l'ordre des demandes. On ne traite pas l'optimisation

du mouvement. Lorsque l'ascenseur passe par l'étage de destination, il s'arrête, ouvre sa porte pendant un certain temps puis la referme automatiquement ;

– l'ascenseur est mis en attente au dernier étage desservi lorsque toutes les demandes ont été satisfaites. Une demande provenant de l'étage où l'ascenseur est en attente provoque l'ouverture de la porte pendant un certain temps puis sa fermeture automatique.

Dans cette étude, nous modélisons ce système par deux sortes de diagrammes UML : le diagramme de classes et les diagrammes d'états-transitions liés à chaque classe.

2.1 Interface graphique

L'interface graphique de l'outil de modélisation avec UML est comme suit :

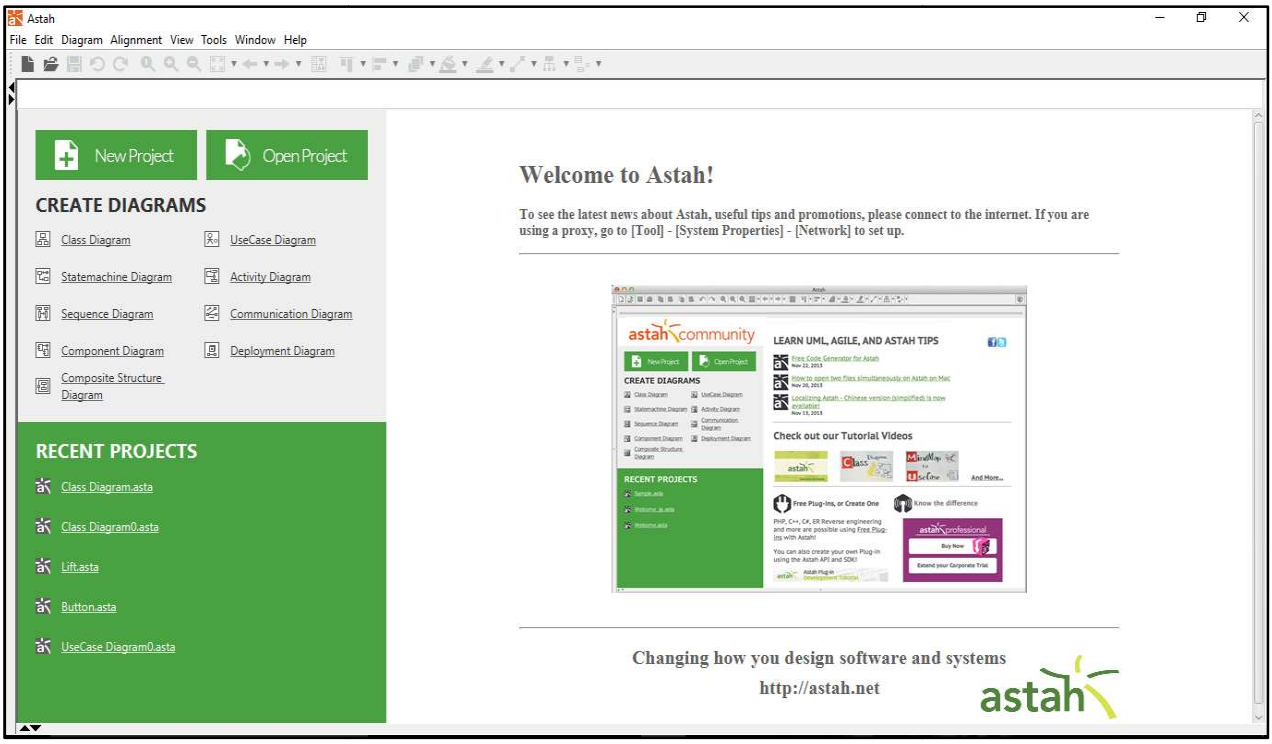


Figure 3.1: L'interface graphique d'Astah community

2.2 Création des diagrammes UML

Notre étude commence par la spécification de l'aspect statique du système décrit par un diagramme de classes, puis de l'aspect dynamique de ce dernier décrit par des diagrammes d'états-transitions.

2.2.1 Le diagramme de classes

La figure 3.2 présente le diagramme de classes du système de contrôle d'ascenseurs. Ce diagramme se compose de deux classes : Lift pour les ascenseurs et Button pour les boutons.

La classe « Lift » spécifie les attributs pour les informations concernant la direction courante de l'ascenseur (dir : DIRECTION), l'étage de destination courante de l'ascenseur (curDestFloor : FLOOR) et l'état de la porte de l'ascenseur (doorStatus : STATUS).

La classe Button spécifie seulement un attribut floor qui modélise l'étage associé à chaque bouton ; cet attribut est de type FLOOR.

Le type énuméré DIRECTION = {*up*, *down*} est défini comme un type avec deux attributs up et down modélisant les deux valeurs possibles du type. Il en est de même pour le type STATUS.

En revanche, le type intervalle FLOOR= [ground …top] est défini par un type avec un stéréotype "intervalle" pour indiquer que c'est un type intervalle. Deux attributs de FLOOR servent à borner l'intervalle.

Les méthodes UML dans les deux classes sont groupées en deux catégories : les méthodes de base locales à chaque classe et les méthodes avec le stéréotype "événement" ; il s'agit des événements au sein des diagrammes d'état-transition dont nous allons parler plus tard.

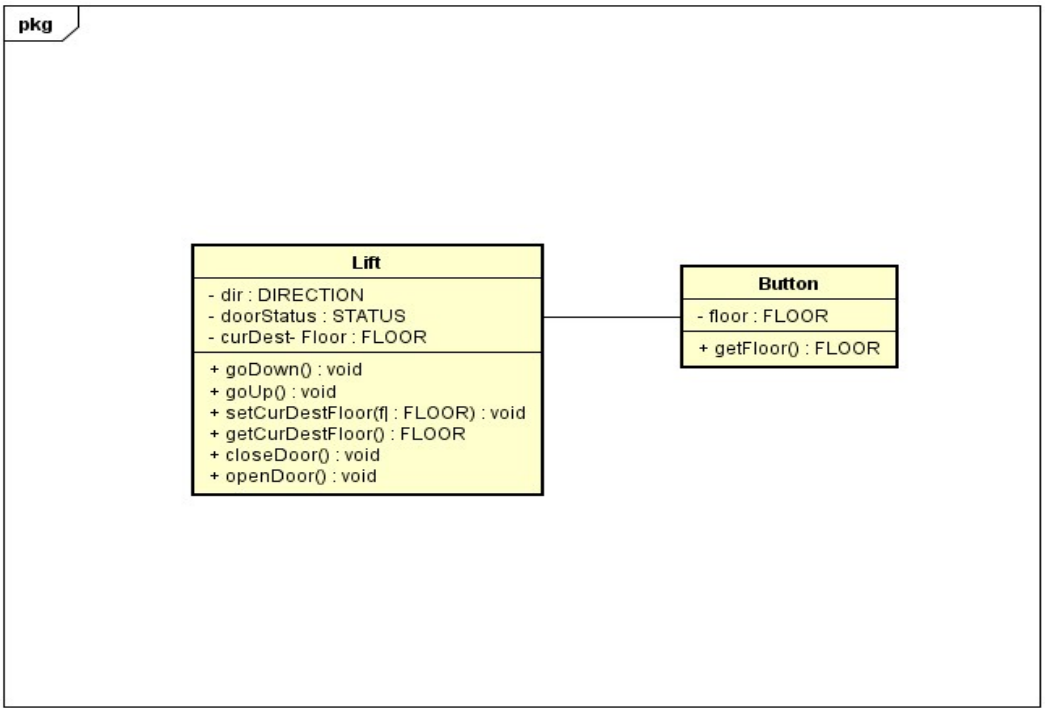


Figure3. 2 : Le diagramme de classes

2.2.2 Les diagrammes d'état-transition

Le diagramme d'état-transition d'une classe peut être créé par l'opération "Statemachine Diagram" du sous menu "Create Diagram" en sélectionnant l'élément classe en question dans le diagramme de classes.

La figure 3.3 présente le diagramme d'état-transition de la classe **Button** créé. Chaque bouton a deux états *on* et *off*. Quand un bouton est activé, il passe dans l'état *on* ; quand un ascenseur s'arrête à un étage, les deux boutons associés sont dans l'état *off*. Lorsqu'un bouton dans l'état *off* est activé, un événement *call* est envoyé de façon asynchrone (le message est préfixé par "asyn") à l'ascenseur associé au bouton ; le paramètre de *call* est l'étage du bouton.

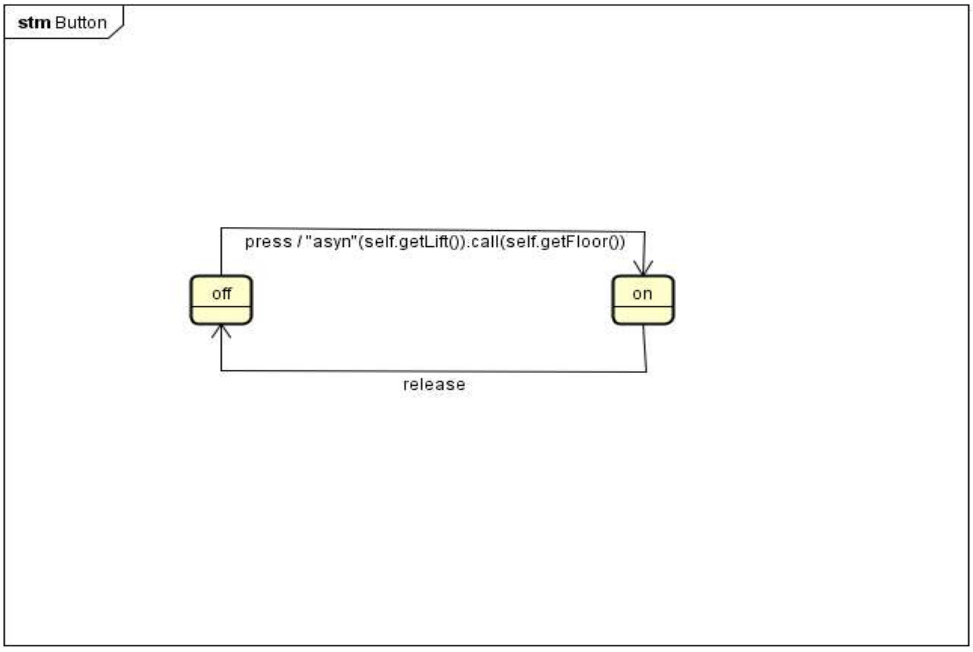


Figure 3.3 : Le diagramme d'état-transition de la classe Button

La figure 3.4 présente le diagramme d'état-transition de la classe **Lift** a trois états *ready*, *visit* et *movement*. C'est seulement dans l'état *ready* que l'ascenseur est prêt à servir un éventuel événement *call*.

Si un ascenseur est dans l'état *ready* et le bouton d'appel à l'étage courant de l'ascenseur est activé, l'ascenseur passe dans l'état *visit*. La porte d'un ascenseur passant dans l'état *visit* doit s'ouvrir, cela veut dire que l'appel à l'opération *openDoor()* est l'action d'entrée de l'état *visit*.

Si l'étage du bouton d'appel n'est pas celui de l'ascenseur, l'ascenseur passe dans l'état *movement* ; il peut monter ou descendre, en fonction de l'étage du bouton d'appel vis à vis de l'étage où se trouve l'ascenseur. C'est pourquoi il y a deux transitions reliant l'état *ready* et l'état *movement*.

Un ascenseur dans l'état *visit* doit passer dans l'état *ready* avant de traiter un autre événement *call*. A la sortie de l'état *visit*, la porte de l'ascenseur doit se fermer, cela veut dire

que l'appel à l'opération *closeDoor()* est l'action de sortie de l'état *visit*. Dans l'état *movement*, lorsque l'ascenseur arrive à l'étage de destination courante (événement *arrive*), l'événement *release* est envoyé de façon synchrone pour éteindre les deux boutons associés à l'étage visité.

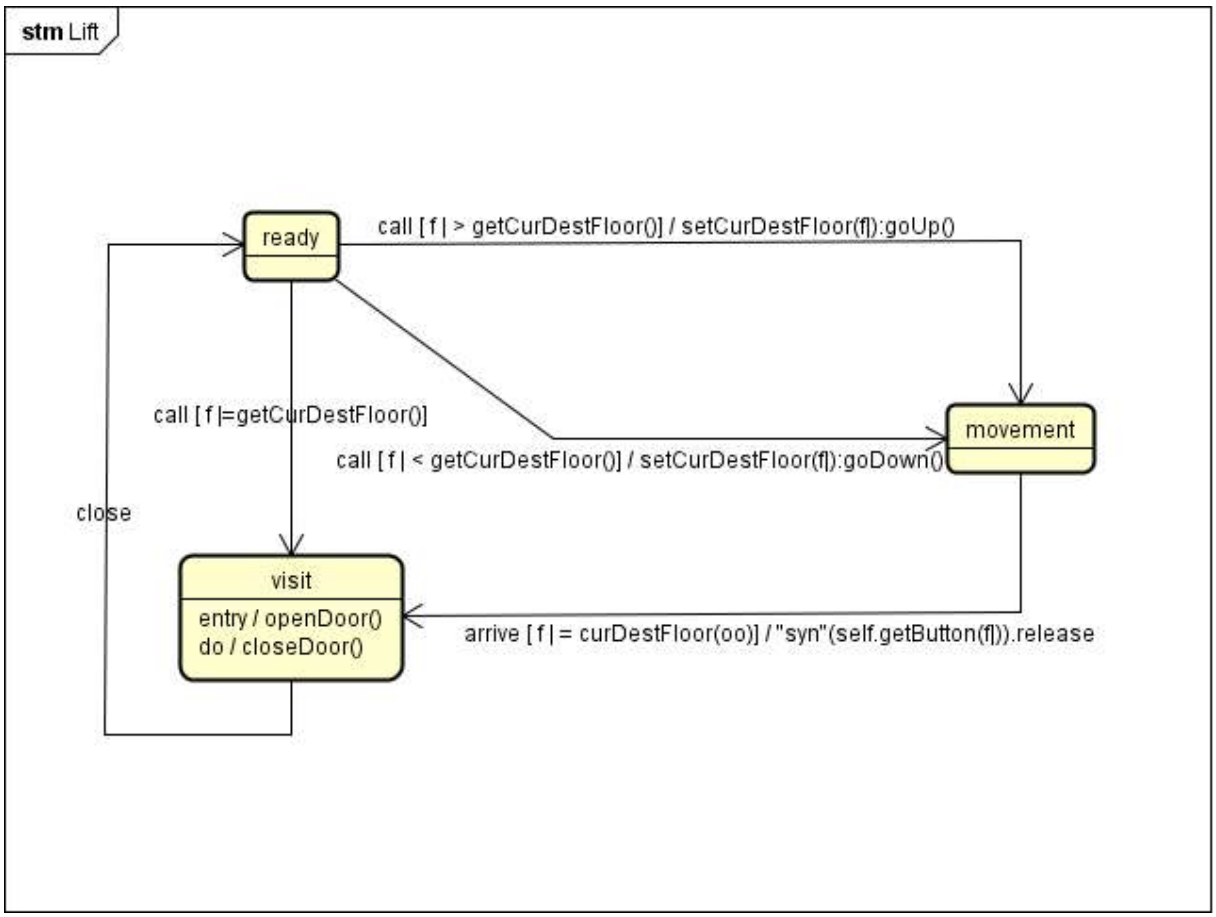


Figure 3.4: Le diagramme d'état-transition de la classe Lift

2.3 Spécification avec la méthode B

La figure 3.5 montre la hiérarchie des composants de la spécification B correspondante au modèle UML présenté ci-dessus et qui est composé des diagrammes de classes et d'état transition. Cette spécification du système d'ascenseur est prise de [4,23].

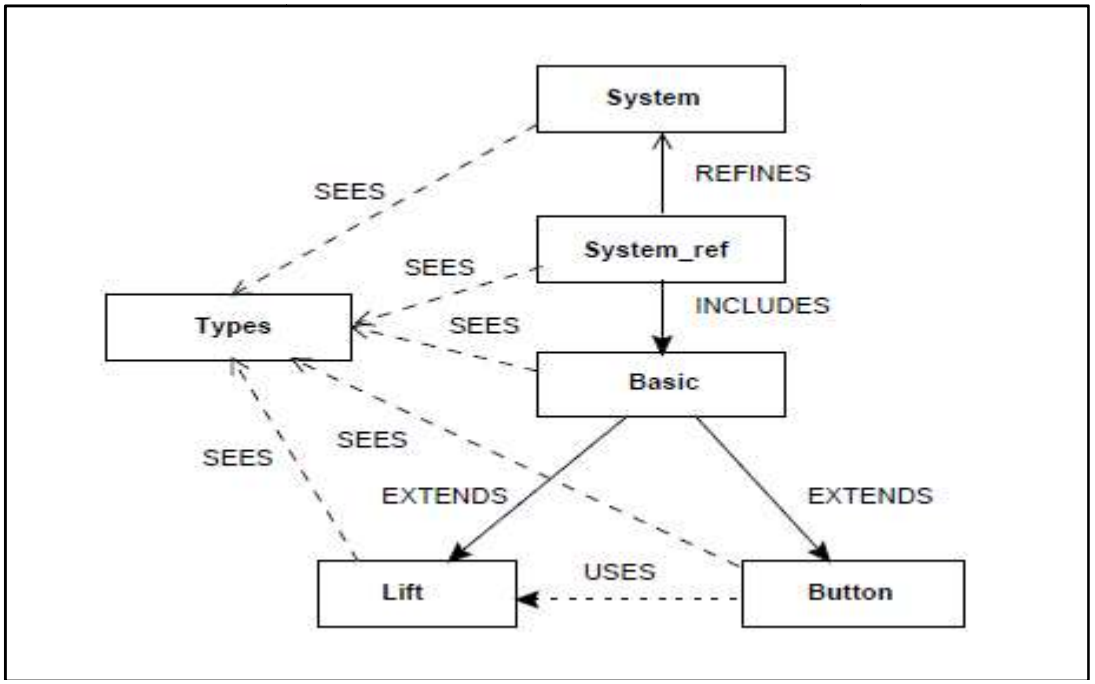


Figure3.5 : Les composants de la spécification B

Les composants de la spécification B sont comme suit :

- **Types** est une machine abstraite pour déclarer les constantes et les ensembles utilisés par les autres composants. L'ensemble B abstrait OBJECTS est pour modéliser des instances possibles de tous les objets. En plus, il y a des constantes modélisant l'ensemble des instances possibles pour chaque machine et le constant définissant les bornes des types intervalle. La machine **Types** est vue par tous les autres composants de la spécification B grâce à la clause **SEES**.
- **System** est la machine abstraite dont chaque opération correspond à la spécification d'un comportement du système d'ascenseur. Les variables de cette machine sont des identificateurs des concepts structurels de la spécification.
- **Basic** est la machine abstraite dont les opérations modélisent les transitions et les opérations correspondant aux actions. Les données de la machine Basic sont identiques aux données de la machine System. De plus, par la clause **EXTENDS**, la machine **Basic** se compose des machines abstraites. il s'agit de la machine **Lift** et de la machine **Button** : la machine **Basic** déclare seulement les variables correspondant à la communication asynchrone éventuelle ainsi que les opérations pour manipuler ces variables.

- **Lift** est la machine abstraite qui modélise le comportement de cabinet de l'ascenseur et également les transitions et l'état courant ;
- **Button** est la machine abstraite qui modélise le comportement du bouton et déclare les opérations B modélisant les transitions et la variable B modélisant l'état courant. Ceci permet à **Basic** de déléguer les opérations pour les actions et pour les transitions aux machines incluses.
- **System_ref** est un raffinement de la machine abstraite System. System_ref inclut la machine Basic par la clause **INCLUDES**. Ceci permet de réaliser une opération abstraite B d'un événement en appelant les opérations abstraites B modélisant les transitions et les opérations B pour les actions.

Remarque technique : La machine **Basic** est indispensable. Si elle est omise, le composant de raffinement **System_ref** doit avoir les liens "**INCLUDES**" aux deux machines **Lift** et **Button**. Ceci est impossible en B à cause du lien **USES** de la machine **Button** à la machine **Lift**.

3. La spécification B d'Ascenseur

Dans cette section, nous présentons une spécification B de l'exemple de l'ascenseur modélisé par les diagrammes UML présentés ci-dessus. Cette spécification B est inspirée des travaux [4,23].

Considérons la machine **Button**, on déclare une constante BUTTON et une variable B button. La constante BUTTON modélise l'ensemble des instances possibles de la machine **Button** ; elle est considérée comme un sous ensemble de OBJECTS ; rappelons que l'ensemble B abstrait OBJECTS modélise l'ensemble des instances possibles de toutes les machines.

La variable button modélise l'ensemble des instances effectives de la machine **Button** ; par conséquent button est un sous ensemble de BUTTON. La constante BUTTON est déclarée une fois pour toute dans la machine **Types** tandis que la variable button est déclarée à la fois dans la machine **System** et dans la machine abstraite **Button**.

La variable Button_floor définie comme une fonction de button vers FLOOR qui est un intervalle floor_ground... floor_top. La variable Lift_dir définie comme une fonction de lift vers DIRECTION qui est un ensemble énuméré {direction_up, direction_down}. La variable B est déclarée dans la machine **System** et dans les autre machines B. Ainsi, Button_floor est déclarée dans **System** et **Button** et Lift_dir est déclarée dans **System** et **Lift.**

Dans la machine **Types** l'ensemble B DIRECTION = {direction_up, direction_down} est énuméré. Si c'est un type générique, l'ensemble B correspondant est abstrait. En revanche, si c'est un type intervalle, les deux bornes de l'intervalle sont modélisées comme deux constantes dans la machine **Types** et le type intervalle est modélisé comme une définition dans tous les composants qui voient **Types**. Pour le type FLOOR ground… _ top, on déclare deux constantes floor_ground et floor_top dans **Types** et on déclare dans tous les composants **System**, **System_ref**, **Basic**, **Lift** et **Button** une définition FLOOR == (floor_ground … floor_top).

La modélisation en B des associations binaires est similaire à la variable. On crée une variable B liftButton définie comme une relation entre lift et button. L'inverse liftButton^{-1} de liftButton est défini comme une fonction de button vers lift.

La variable B d'une association est déclarée dans la machine **System** et dans la machine B de base dans laquelle l'association est modélisée. La machine **Button** déclare la variable B liftButton modélisant l'association entre les deux machines **Lift** et **Button**.

3.2 Spécification des opérations B

Les opérations B sont spécifiées à partir des événements, des actions, des transitions et de la communication asynchrone cités dans les diagrammes UML.

L'opération B Lift_transVisitReady modélise la transition de l'état *visit* à l'état *ready* dans le diagramme d'état-transition de la classe Lift. Cette opération est dans la machine **Lift**. Le paramètre ll de Lift_transVisitReady modélise l'objet de la classe Lift sur lequel la transition se produit.

La variable Lift_state modélise les états des objets de la classe Lift ; Lift_state (ll) modélise ainsi l'état courant de l'objet cible de la transition. La pré-condition de l'opération vérifie le fait que l'objet cible est parmi les instances effectives de la classe Lift et l'état de objet est bien *visit*. Le corps de la substitution **pre** est une affectation pour mettre à jour l'état de Lift_state.

L'opération Lift_closeDoor de la machine **Lift** correspond à l'action de sortie closeDoor de l'état visit.

Chaque événement est modélisé par deux opérations B : une opération abstraite dans la machine **System** et son raffinement dans le composant de raffinement **System_ ref**. Le nom des opérations B est préfixé par le nom de la machine. Les opérations B sont paramétrées par l'objet cible de l'événement ainsi que par des paramètres éventuels de l'événement.

L'opération abstraite spécifie les effets souhaités de l'événement directement sur les données modélisées.

L'opération B de raffinement fait appel aux opérations B abstraites modélisant les transitions et les opérations B abstraites modélisant les opérations des actions. Ceci explique pourquoi la machine **Basic**, qui inclut les machines, est incluse dans le raffinement **System_ref**. L'opération B abstraite et l'opération B de raffinement pour l'événement *close*.

A noter également que si l'opération B de raffinement peut être spécifié à partir des diagrammes d'état-transition, l'opération B abstraite est construite à partir des contraintes associées à l'événement en tant qu'opération déclarée au sein d'une classe , comme on utilise les notations B pour écrire les contraintes, il suffit de copier les contraintes d'un événement dans l'opération B abstraite pour l'événement.

L'événement *arrive* dans Lift provoque deux événements *release* dans un message synchrone à deux objets Button. On incorpore ainsi les effets de ces deux événements *release* dans les opérations B de l'événement *arrive*.

Dans le système de contrôle d'ascenseurs, il s'agit de l'événement *call* survenu lors d'un message asynchrone envoyé par l'événement *press*. La file d'événements *call* est modélisée par une variable B call_Queue dans la machine **System**. L'envoi de messages asynchrones *call* est modélisé par l'écriture d'un élément dans la file d'événements call_Queue au sein de l'opération Button_press. Parce que l'opération Button_press doit être raffinée, il faut déclarer call_Queue dans la machine **Basic**. Une opération auxiliaire insert_Call (...) a été introduite dans **Basic**, car cette opération est appelée lors du raffinement de Button_press.

Contrairement à l'événement *release*, l'événement résultat *call* doit être modélisé par les opérations B. On peut noter la présence des substitutions pour enlever un élément de call_Queue. On omet ici l'opération de raffinement *call* dans laquelle on fait appel à l'opération remove_Call, qui est une opération auxiliaire introduite dans la machine **Basic.**

3.3 Le code B de la spécification du système d'ascenseur

La machine abstraite ***System*** englobe les opérations correspondantes aux événements. Et les variables qui modélisent les files de messages asynchrones échangés. Le code de cette machine est :

```
MACHINE System
SEES Types
VARIABLES
    lift, lift_dir, lift_curDestFloor, lift_doorStatus, lift_State,
    button ,button_floor, button_State, liftButton, call_Queue
INVARIANT
```

```
    lift <: LIFT &
    lift_dir : lift --> DIRECTION &
    lift_curDestFloor : lift --> FLOOR &
    lift_doorStatus : lift --> STATUS &
    lift_State : lift --> LIFT_STATE &
    button <: BUTTON &
    button_floor : button --> FLOOR &
    ran(button_floor)= FLOOR &
    button_State : button --> BUTTON_STATE &
    liftButton : button --> lift &
    ran(liftButton)=lift &
    !(ll,fl).(ll : lift & fl :FLOOR =>card(liftButton~ [{ll}]/\
button_floor~ [{fl}])=2) &
    call_Queue <: lift * FLOOR
INITIALISATION
    ANY ll,bb,ff,lb WHERE
        ll <: LIFT &
        ll /= {} &
        bb <: BUTTON &
        ff : bb --> FLOOR &
        ran (ff) = FLOOR &
        lb :bb --> ll &
        ran(lb) = ll&
      !(li,fl).(li : ll & fl :FLOOR =>card(lb~ [{li}]/\ ff~[{fl}]) =
2)& bb*{button_off} : bb +-> BUTTON_STATE
    THEN
        lift := ll ||
        lift_dir := ll * {direction_up} ||
        lift_curDestFloor := ll * {floor_ground} ||
        lift_doorStatus := ll * {status_closed} ||
        lift_State := ll * {lift_ready} ||
        button := bb ||
        button_floor := ff ||
        button_State:= bb*{button_off}||
        liftButton:=lb
    END ||
    call_Queue := {}
DEFINITIONS
    FLOOR == ( floor_ground..floor_top)
OPERATIONS
    lift_close(ll)=
    PRE
        ll:lift
    THEN
        SELECT lift_State(ll) =lift_visit THEN
            lift_doorStatus(ll) := status_closed ||
            lift_State(ll):= lift_ready
        ELSE
            skip
        END
    END;
    lift_call(ll,fl)=
    PRE
        ll:lift &
        fl : FLOOR &
        ll |-> fl : call_Queue
    THEN
```

```
        SELECT lift_State(ll)=lift_ready & fl = lift_curDestFloor(ll)
THEN
            lift_doorStatus(ll):= status_open ||
            call_Queue:= call_Queue - {ll|-> fl} ||
            lift_State(ll) := lift_visit
        WHEN lift_State(ll)= lift_ready & fl > lift_curDestFloor(ll)
THEN
            lift_curDestFloor(ll):=fl ||
            lift_dir(ll):=direction_up ||
            call_Queue:= call_Queue - {ll |-> fl}||
            lift_State(ll):=lift_movement
        WHEN lift_State(ll)=lift_ready & fl < lift_curDestFloor(ll)
THEN
            lift_curDestFloor(ll):=fl ||
            lift_dir(ll):= direction_down ||
            call_Queue:= call_Queue - {ll |-> fl}||
            lift_State(ll):=lift_movement
        ELSE
            skip
        END
    END;
    lift_arrive(ll,fl)=
    PRE
        ll:lift &
        fl:FLOOR
    THEN
        SELECT lift_State(ll)= lift_movement & fl
=lift_curDestFloor(ll) THEN
            ANY b1,b2 WHERE
                b1:button &
                b2:button &
                {b1,b2}=liftButton~[{ll}]/\button_floor~[{fl}]
            THEN
                button_State:= button_State <+ {b1 |-> button_off,b2 |-
> button_off}
            END ||
            lift_State(ll):=lift_visit ||
            lift_doorStatus(ll):=status_open
        ELSE skip
        END
    END;
    button_press(bt)=
    PRE
        bt:button
    THEN
        SELECT button_State(bt)=button_off THEN
            button_State(bt):=button_on ||
            call_Queue:=call_Queue\/{liftButton(bt)|->button_floor(bt)}
        ELSE
            skip
        END
    END
END
```

Le raffinement ***System_ref*** est un composant qui raffine la machine ***System*** en incluant la machine ***Basic*** au moyen de la clause *INCLUDES*. Cela a pour objet de permettre la

réalisation d'une opération abstraite B d'un événement (défini dans la machine abstraite à raffiner ***System***) qui fait appel aux opérations abstraites B spécifiant les transitions et les opérations B pour les actions (définies dans la machine abstraite incluse *Basic*). Le code de ce raffinement est :

```
REFINEMENT System_ref
REFINES System
INCLUDES Basic
SEES Types
DEFINITIONS
    FLOOR==(floor_ground..floor_top)
OPERATIONS
    lift_close(ll)=
    IF lift_State(ll)=lift_visit THEN
        lift_closeDoor(ll);
        lift_transVisitReady(ll)
    ELSE
        skip
    END;
    lift_call(ll,fl)=
    IF lift_State(ll)=lift_ready & fl = lift_curDestFloor(ll)THEN
        BEGIN
            lift_openDoor(ll);
            remove_call(ll,fl);
            lift_transReadyVisit(ll)
        END
    ELSIF lift_State(ll)=lift_ready & fl > lift_curDestFloor(ll)THEN
        BEGIN
            lift_setCurDestFloor(ll,fl);
            lift_goUp(ll);
            remove_call(ll,fl);
            lift_transReadyMovement(ll)
        END
    ELSIF lift_State(ll)=lift_ready & fl < lift_curDestFloor(ll) THEN
        BEGIN
            lift_setCurDestFloor(ll,fl);
            lift_goDown(ll);
            remove_call(ll,fl);
            lift_transReadyMovement(ll)
        END
    ELSE
        skip
    END;
    lift_arrive(ll,fl)=
    IF lift_State(ll)=lift_movement & fl = lift_curDestFloor(ll)THEN
        lift_transMovementVisit(ll);
        lift_openDoor(ll);
        VAR b1,b2 IN
            b1,b2 <-- getButtons(ll,fl);
            IF button_State(b1)= button_on THEN
                button_transOnOff(b1)
            END;
            IF button_State(b2) = button_on THEN
                button_transOnOff(b2)
            END
        END
```

```
    ELSE
        skip
    END;
    button_press(bt)=
    IF button_State(bt)=button_off THEN
        button_transOffOn(bt);
        insert_call(liftButton(bt),button_floor(bt))
    ELSE
        skip
    END
END
```

La machine abstraite ***Basic*** comporte des opérations spécifiant les transitions et les opérations associées aux actions. Dans cette machine, nous définissons les mêmes données que celles déclarées au sein de la machine ***System***. Ainsi, la machine ***Basic*** étend deux machines abstraite ***Button*** et ***Lift*** et de leur association en utilisant la clause *EXTENDS*. Ces machines abstraites ont les caractéristiques suivantes : l'association *liftButton* entre ses deux machines *Lift* et *Button*.

La spécification des transitions (par des opérations) et de l'état courant (par une variable) est réalisée par la machine *Lift* ; tandis que la spécification de ceux de la classe *Button* est effectuée par la machine *Button*. Donc, les opérations pour les actions et pour les transitions définies au sein des machines incluses (*Lift* et *Button*) peuvent être toutes déléguées par la machine *Basic*.

En outre, seules les variables correspondant à la communication asynchrone éventuelle entre les diagrammes d'états-transition ainsi que les opérations manipulant ces variables, sont spécifiées au sein de la machine *Basic*. Les codes respectifs des machines *Basic*, *Lift* et *Button* sont comme suit :

```
MACHINE Basic
SEES Types
EXTENDS
    Lift, Button
VARIABLES
    call_Queue
INVARIANT
    call_Queue <: lift * FLOOR
INITIALISATION
    call_Queue:= {}
DEFINITIONS
    FLOOR ==(floor_ground .. floor_top)
OPERATIONS
    insert_call(ll,fl)=
    PRE
        ll : lift &
        fl : FLOOR
    THEN
        call_Queue:=call_Queue \/ {ll |-> fl}
    END;
```

```
    remove_call(ll,fl)=
    PRE
        ll : lift &
        fl : FLOOR
    THEN
        call_Queue := call_Queue-{ll|-> fl}
    END
END

MACHINE Lift
SEES Types
VARIABLES
    lift,lift_dir,lift_curDestFloor,lift_doorStatus,lift_State
INVARIANT
    lift <: LIFT &
    lift_dir : lift --> DIRECTION &
    lift_curDestFloor : lift --> FLOOR &
    lift_doorStatus : lift --> STATUS &
    lift_State : lift --> LIFT_STATE
INITIALISATION
    ANY ll WHERE
        ll <: LIFT &
        ll /= {}
    THEN
        lift:= ll ||
        lift_dir:= ll * {direction_up} ||
        lift_curDestFloor:= ll * {floor_ground} ||
        lift_doorStatus:= ll * {status_closed} ||
        lift_State:= ll * {lift_ready}
    END
DEFINITIONS
    FLOOR == (floor_ground .. floor_top)
OPERATIONS
    lift_goDown(ll)=
    PRE
        ll : lift
    THEN
        lift_dir(ll):= direction_down
    END;
    lift_goUp(ll)=
    PRE
        ll : lift
    THEN
        lift_dir(ll):= direction_up
    END;
    lift_setCurDestFloor(ll,fl) =
    PRE
        ll : lift &
        fl : FLOOR
    THEN
        lift_curDestFloor(ll):= fl
    END;
    fl <-- lift_getCurDestFloor(ll) =
    PRE
        ll : lift
    THEN
        fl:=lift_curDestFloor(ll)
    END;
```

```
    lift_closeDoor(ll) =
    PRE
        ll : lift
    THEN
        lift_doorStatus(ll):= status_closed
    END;
    lift_openDoor(ll) =
    PRE
        ll : lift
    THEN
        lift_doorStatus(ll):= status_open
    END;
    lift_transReadyMovement(ll) =
    PRE
        ll : lift &
        lift_State(ll) = lift_ready
    THEN
        lift_State(ll):= lift_movement
    END;
    lift_transReadyVisit(ll) =
    PRE
        ll : lift &
        lift_State(ll) = lift_ready
    THEN
        lift_State(ll):= lift_visit
    END;
    lift_transVisitReady(ll) =
    PRE
        ll : lift &
        lift_State(ll) = lift_visit
    THEN
        lift_State(ll):= lift_ready
    END;
    lift_transMovementVisit(ll) =
    PRE
        ll : lift &
        lift_State(ll) = lift_movement
    THEN
        lift_State(ll):= lift_visit
    END
END

MACHINE Button
SEES Types
USES Lift
VARIABLES
    button,button_floor,button_State,liftButton
INVARIANT
    button <: BUTTON &
    button_floor : button --> FLOOR &
    ran(button_floor) =FLOOR &
    button_State : button --> BUTTON_STATE &
    liftButton : button --> lift &
    ran(liftButton)=lift &
    !(ll,fl).(ll : lift & fl : FLOOR => card(liftButton~ [{ll}]/\
button_floor~ [{fl}])=2)
INITIALISATION
    ANY bb,ff,lb WHERE
```

```
        bb <: BUTTON &
        ff : bb --> FLOOR &
        ran(ff)= FLOOR &
        lb : bb --> lift &
        ran(lb)= lift &
        !(ll,fl).(ll : lift & fl : FLOOR => card(lb~ [{ll}]/\ ff~
[{fl}])=2)& bb*{button_off} : bb +-> BUTTON_STATE

    THEN
        button := bb||
        button_floor := ff ||
        button_State := bb * {button_off} ||
        liftButton := lb
    END
DEFINITIONS
    FLOOR ==(floor_ground .. floor_top)
OPERATIONS
    fl <-- button_getFloor(bt)=
    PRE
        bt : BUTTON
    THEN
        fl := button_floor(bt)
    END;
    button_transOffOn(bt)=
    PRE
        bt : button &
        button_State(bt)=button_off
    THEN
        button_State(bt):=button_on
    END;
    button_transOnOff(bt)=
    PRE
        bt : button &
        button_State(bt)= button_on
    THEN
        button_State(bt):=button_off
    END;
    bt1,bt2 <-- getButtons(ll,fl)=
    PRE
        ll : lift &
        fl : FLOOR
    THEN
        ANY b1,b2 WHERE
            b1 : button &
            b2 : button &
            {b1,b2}=liftButton~ [{ll}]/\ button_floor~[{fl}]
        THEN
            bt1:=b1 ||
            bt2:=b2
        END
    END;
    ll <-- getLift(bt)=
    PRE
        bt : button
    THEN
        ll:=liftButton(bt)
    END
END
```

La machine abstraite ***Types*** a pour but de déclarer les constantes et les ensembles utilisés par les autres composants B. L'ensemble B abstrait *OBJECTS* qui modélise l'ensemble des instances possibles de toutes les machines, est un exemple faisant partie de ces ensembles définis dans *Types*. Celle-ci, déclare également les constantes identifiant les bornes des types intervalle ainsi que les constantes spécifiant l'ensemble des instances possibles. Enfin, la clause *SEES* permet à tous les autres composants de la spécification B générée, de voir la machine *Types*. La machine *Types* est décrite comme suit :

```
MACHINE Types
SETS
    OBJECTS ;
    DIRECTION = {direction_up,direction_down};
    STATUS = {status_open,status_closed};
    BUTTON_STATE = {button_on,button_off};
    LIFT_STATE = {lift_ready,lift_movement,lift_visit}
CONSTANTS
    floor_ground, floor_top, LIFT, BUTTON
PROPERTIES
    floor_ground : NAT &
    floor_top : NAT &
    floor_ground < floor_top &
    LIFT <: OBJECTS &
    BUTTON <: OBJECTS
END
```

Une fois la spécification B décrite, l'outil Atelier B peut être utilisé pour la vérification des propriétés.

Remarque il est impossible de ne pas utiliser la machine *Basic*, en raison du raffinement *System_ref* qui ne peut pas disposer des liens *INCLUDES* vers les deux machines *Lift* et *Button*. Cela n'est pas autorisé en langage B vu que le lien *USES* de la machine *Button* vers la machine *Lift* est présent.

3.4 Vérification formelle des propriétés

L'outil Atelier B est utilisé pour écrire la spécification B et la vérification des propriétés. Dans cette section nous allons prouver et effectuer des vérifications sur les résultats de la spécification B du système de contrôle d'ascenseur. Ces vérifications sont fondées sur des preuves mathématiques en utilisant l'Atelier B.

La spécification à vérifier est constituée des six composants B (les machines System, Basic, Lift, Button et Types ; ainsi que le raffinement System réf*)*.

3.4.1 Application de la méthode B en utilisant l'Atelier B

Nous pouvons rappeler que pour développer des systèmes suivant la méthode B, l'Atelier B propose un ensemble de commandes, qui sont illustré dans la figure 3.6. Ces commandes permettent :

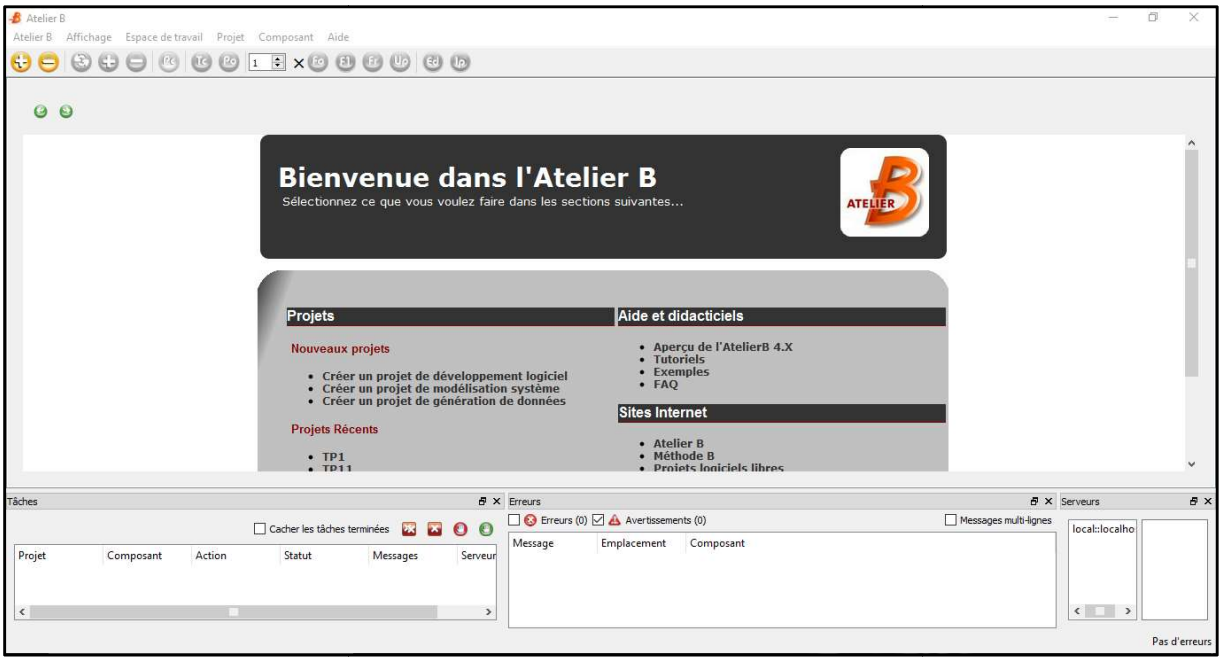


Figure3.6 : Commandes principales de l'outil Atelier B pour la vérification

– l'analyse syntaxique et le contrôle de type des machines d'un projet ;

– la génération automatique des obligations de preuve (OPs) ;

– la démonstration automatique d'OPs ;

– la démonstration interactive des OPs non démontrées automatiquement ;

– le contrôle du langage d'implantation ;

– et enfin, la traduction du dernier niveau de raffinement des spécifications vers un langage informatique (code C).

Dans notre travail, nous nous intéressons aux six étapes, car l'objectif est de vérifier les résultats de la spécification et de générer du code impératif équivalent.

3.4.2 Preuves des résultats B du système d'ascenseur

3.4.2.1 Analyse syntaxique et contrôle de type

Notre analyse syntaxique du système d'ascenseur étudié vise à vérifier que les règles de construction du langage B sont respectées par les sources des composants sélectionnées. Le contrôle de type de ce système sert à détecter :

– Les confits d'identificateurs

– L'omission ou l'absence de déclarations

– Les erreurs de type au sein de l'utilisation des constructions

– La mauvaise utilisation des opérateurs

– Les violations de règles de visibilité, etc.

Ce contrôle est très important car il nous permet de produire des obligations de preuve. Le contrôle de type que nous appliquons à un composant B (e.g. le raffinement *System_ref)* s'applique également de manière systématique à tous les composants requis par le composant courant, via des liens *SEES, USES, INCLUDES, IMPORTS, EXTENDS* et *REFINES*.

Dans notre cas, le composant *System_ref* dispose d'un lien avec la machine *System* par la clause *REFINES*, d'un lien avec la machine *Basic* par la clause *INCLUDES*, la machine *Basic* à son tour comporte de liens avec les machines *Lift* et Button par la clause *EXTENDS*, la machine *Button* comporte un lien avec la machine *Lift* via la clause *USES* et enfin toutes les machines sont liées avec la machine *Types* par la clause *SEES* .

Dans la deuxième colonne de la figure 3.7 (Typage vérifié), la valeur OK indique que l'analyse syntaxique et le contrôle de type ont été effectués avec succès sur tous les composants, et qu'il n'existe pas d'erreurs. Pour les tirets dans toutes les autres colonnes, ils signifient que nous n'avons réalisé en ce moment aucune fonction de vérification.

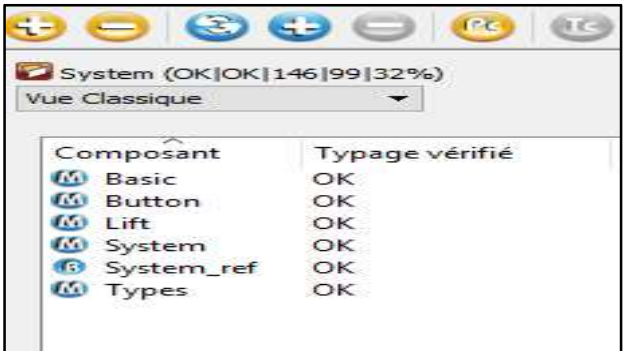


Figure 3.7: Analyse syntaxique et typage vérifié

3.4.2.2 Des obligations de preuves (OPs)

Avant la génération des OPs d'un composant, l'Atelier B revient à vérifier que nous avons bien effectué le contrôle de type sur tous les composants dont il procède de prouver. Si ce n'est pas le cas, le contrôle de type va se réaliser de façon systématique dans l'étape qui suit.

La méthode B fixe les OPs, et celles-ci diffèrent selon le stade de développement du système :

Dans la phase de spécification du système, nous devons démontrer que le modèle mathématique retenu est cohérent (avant de raffiner les composants).

Dans les phases suivantes, nous devons démontrer que les raffinements (transformations) préservent les propriétés du modèle du stade précédent.

Théoriquement, nous distinguons deux sortes d'OPs, une pour l'initialisation et l'autre pour chacune des opérations. Pratiquement, ces OPs pouvant être des formules grandes et complexes, et pour les simplifier et produire des formules plus facilement démontrables, nous effectuons la fonction générer les OPs. Par contre, le nombre de formules initialement prévu augmente. Quelques OPs, sont automatiquement exclus par l'Atelier B. La forme générale d'une obligation de preuve est : $H \rightarrow P$ Où P et H sont des prédicats.

Cette formule qui est illustrée par la figure 3.8, indique que nous devons démontrer le but P (opération signifiant la direction vers le haut de l'ascenseur) sous l'hypothèse H. Pour chaque composant, lors de la génération des OPs, quatre fichiers sont créés dans la BDP :

- un fichier dispose de toutes les OPs ;
- un fichier dispose des OPs détruites par le générateur ;
- un fichier dispose des états des OPs (démontrées/non démontrées) ainsi que les démonstrations interactives ;
- et enfin, le fichier contient la description du composant.

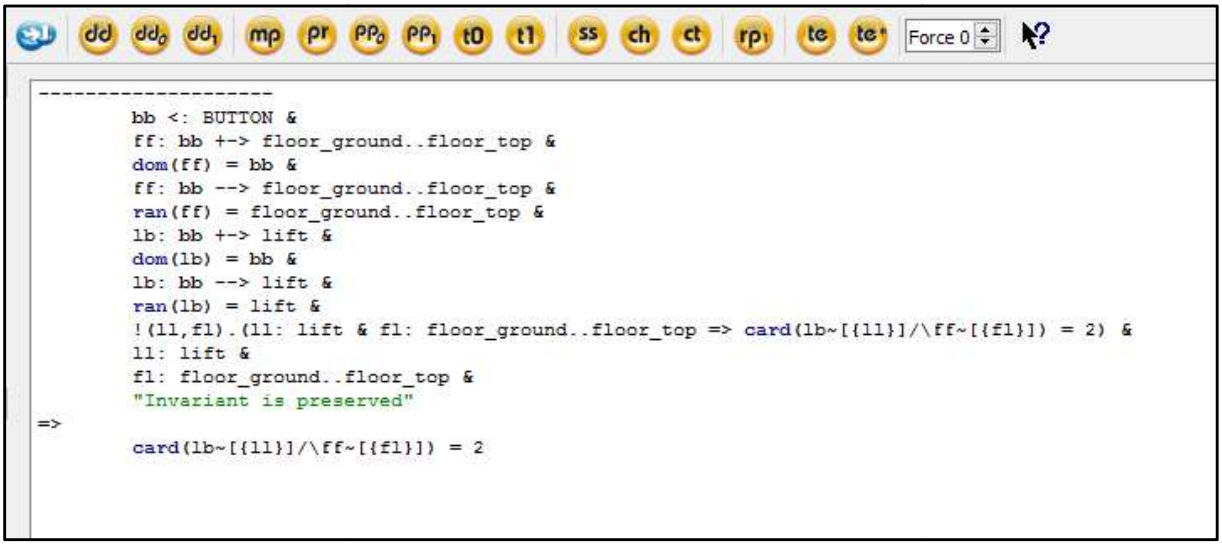
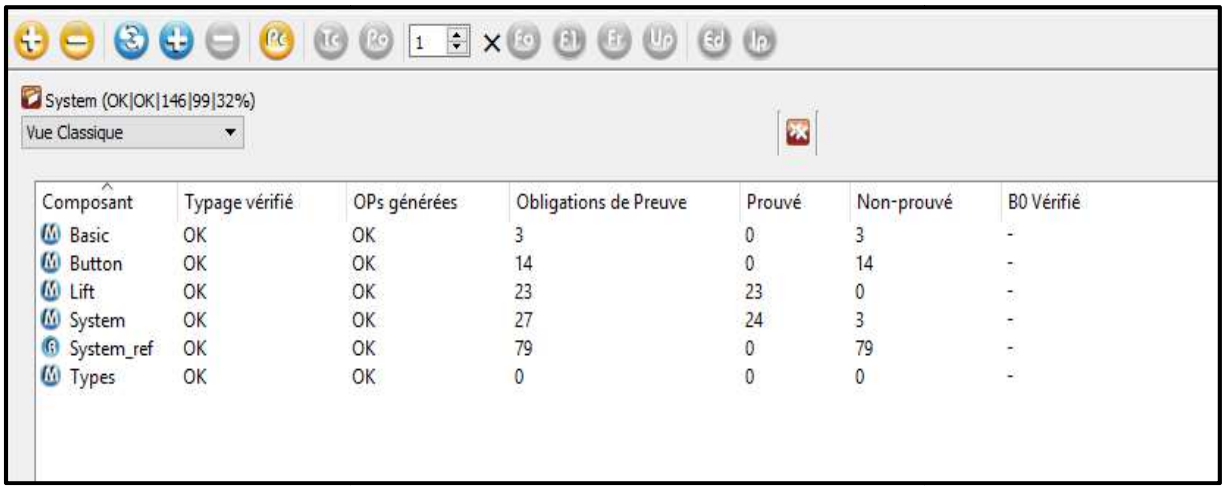


Figure 3.8: Forme détaillée d'une OP du système d'ascenseur

La figure 3.9 présente la génération des OPs des composants du système d'ascenseur étudié. Dans la troisième colonne de celle-ci (OPs générées), la valeur OK signifie que les OPs ont été générées sur le composant. La quatrième colonne (OPs) indique le nombre d'obligations générées. Par exemple, pour la machine *Lift* nous avons vingt-trois (23) OPs générées, alors que pour la machine *Types* le nombre d'OPs générées est égal à zéro (0), car celle-ci ne contient ni variable ni opération à vérifier. La cinquième colonne (Prouvé) contient le nombre d'OPs démontrées pour chaque composant. Enfin, la sixième colonne (Non-prouvé) contient le nombre d'OPs non encore démontrées de chacun des composants du système.



Composant	Typage vérifié	OPs générées	Obligations de Preuve	Prouvé	Non-prouvé	B0 Vérifié
Basic	OK	OK	3	0	3	-
Button	OK	OK	14	0	14	-
Lift	OK	OK	23	23	0	-
System	OK	OK	27	24	3	-
System_ref	OK	OK	79	0	79	-
Types	OK	OK	0	0	0	-

Figure 3.9: Génération des OPs du système d'ascenseur

3.4.2.3 Démonstration automatique d'OPs

L'Atelier B nous offre un prouveur automatique de force variable (force 0 jusqu'à force 3). Plus la force est augmentée, plus le temps de démonstration s'accroit et risque de boucler, mais plus elle est puissante.

La force 0 constitue le meilleur choix (performance-rapidité) par rapport aux autres forces dont nous devons utiliser en premier pour notre vérification. Elle requiert approximativement jusqu'à 10 secondes par OP. Pour quelques OPs complexes, la preuve peut nécessiter plusieurs minutes.

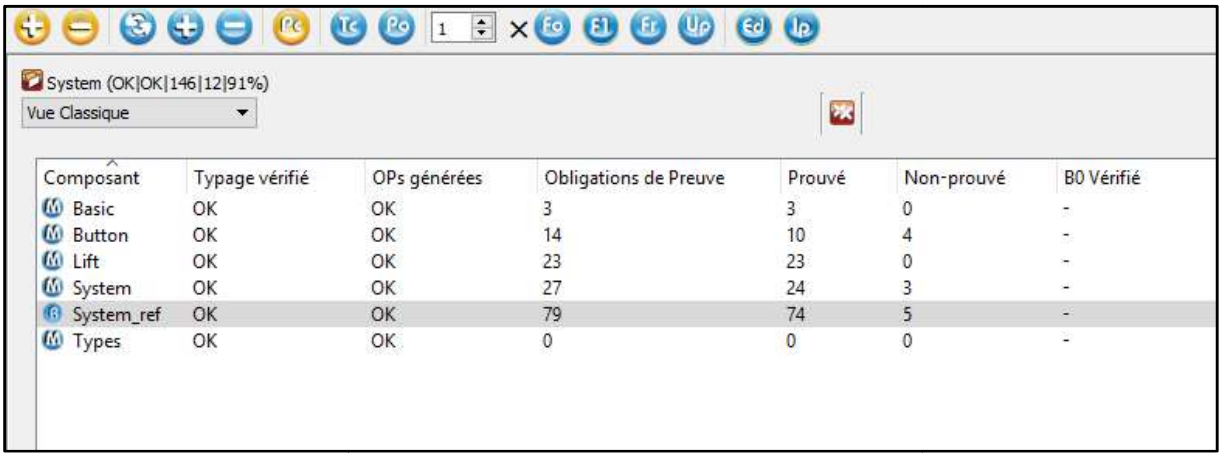
Pour notre étude, la vérification du système d'ascenseur en utilisant la force 0 de l'Atelier B, donne lieu aux résultats décrits dans la figure 3.10. Nous pouvons constater à partir de la cinquième colonne « Prouvé » (ou même en se référant à la sixième colonne « Non-prouvé ») que les composants *Basic* et *Lift* sont totalement prouvés.

Dans la colonne « Prouvé », la valeur désignant le nombre d'OPs prouvés est identique à celle des OPs générées pour ces deux composants. Par conséquent, dans la colonne « Non-prouvé » leur valeur est égale à 0. Ce qui signifie que toutes les OPs générées pour ces deux machines abstraites sont vérifiées avec succès et démontrées mathématiquement à l'aide de la preuve de théorème de l'Atelier B.

Par contre les autres composants (*Button, System* et *System_ref)* restent à vérifier pour une seule (1) OP des deux composants *Button* et *System* ; et sept (7) OPs du composant *System_ ref*. Nous pouvons dans ce cas appliquer la force 1 qui peut être longue, mais peut éviter une démonstration interactive. Les résultats de celle-ci peuvent être obtenus à partir de la figure 3.11.

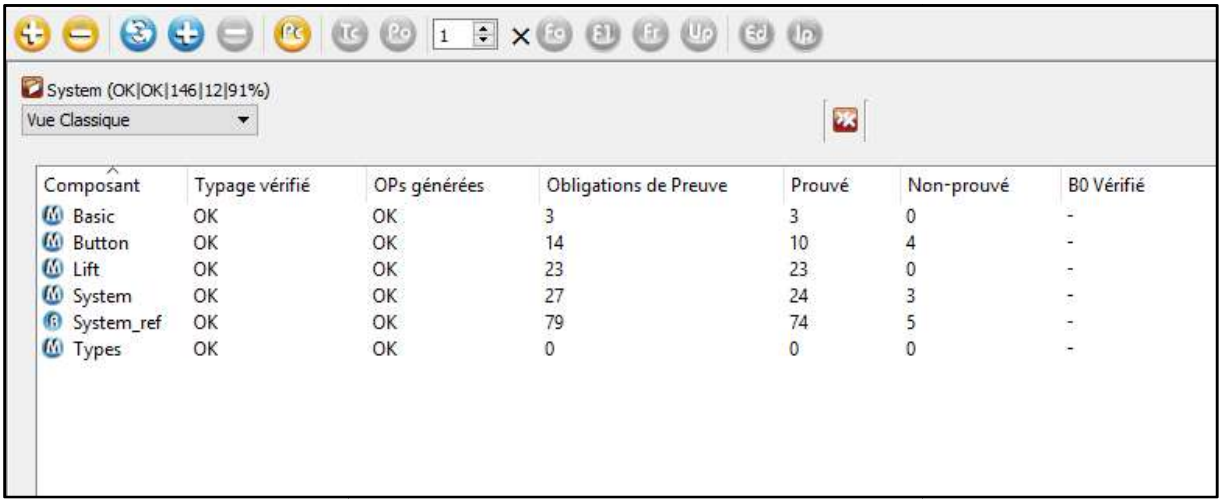
Nous pouvons remarquer à partir de la colonne « Prouvé » (ou « Non-prouvé ») qu'une (1) OP du composant *System_ref* est encore prouvée en force 1. Ce qui fait que nous devions

passer maintenant à la démonstration interactive afin de prouver le reste des OPs non démontrées.



Composant	Typage vérifié	OPs générées	Obligations de Preuve	Prouvé	Non-prouvé	B0 Vérifié
Basic	OK	OK	3	3	0	-
Button	OK	OK	14	10	4	-
Lift	OK	OK	23	23	0	-
System	OK	OK	27	24	3	-
System_ref	OK	OK	79	74	5	-
Types	OK	OK	0	0	0	-

Figure 3.10 : Démonstration automatique des OPs du système d'ascenseur par la force 0



Composant	Typage vérifié	OPs générées	Obligations de Preuve	Prouvé	Non-prouvé	B0 Vérifié
Basic	OK	OK	3	3	0	-
Button	OK	OK	14	10	4	-
Lift	OK	OK	23	23	0	-
System	OK	OK	27	24	3	-
System_ref	OK	OK	79	74	5	-
Types	OK	OK	0	0	0	-

Figure 3.11: Démonstration automatique des OPs du système d'ascenseur par la force 1

3.4.2.4 Démonstration interactive d'OPs non démontrées automatiquement

Le but principal du prouveur interactif est de nous permettre de démontrer manuellement les OPs qui n'ont pu l'être automatiquement. Il s'agit que cette preuve interactive vient juste après l'échec de l'exécution de la preuve automatique. Pour cela, nous décrivons ci-dessous ces démonstrations pour les trois composants restants (*Button, System* et *System_ref)*.

– **Preuve interactive de la machine Button.** Deux OPs sont générées pour l'initialisation (INITIALISATION), et une de celles-ci (PO1) n'est pas prouvée automatiquement. Pour cela, nous utilisons la commande pp en force 0 (les mécanismes du cœur de preuve peuvent échouer sur un tel lemme car ils cherchent d'abord à simplifier le but), qui fait appel au prouveur de prédicats sur le lemme courant. Le prouveur de prédicats agit

souvent avec succès sur des buts composés d'expressions ensemblistes ou fonctionnelles (comme notre exemple de cas d'étude).

La syntaxe de pp est : pp (rp.n), tel que rp.n indique que l'on utilise le prouveur de prédicats sur les hypothèses réduites.

1. rp : est un mot clef.
2. n : est un entier positif ou nul indiquant le niveau des hypothèses prises en compte.

pp (rp.0) : S'il y a peu d'hypothèses (c'est le cas de notre OP non prouvée dans cette machine).

pp(rp.1) : Si les hypothèses nécessaires sont celles qui ont un symbole commun avec le but.

Donc, nous pouvons résoudre le problème détecté au niveau de la propriété de l'invariant button_State : button --> BUTTON_STATE pour la machine abstraite *Button*. L'Atelier B détecte qu'il existe une absence d'initialisation, et résolve le problème de cette omission par l'instauration de l'initialisation suivante : bb*(button_off) : bb +-> BUTTON_STATE.

La figure 3.12 illustre que ce problème a été résolu avec succès en indiquant la zone de but en vert pour un taux de démonstration terminée en bas à 100 % [4,23].

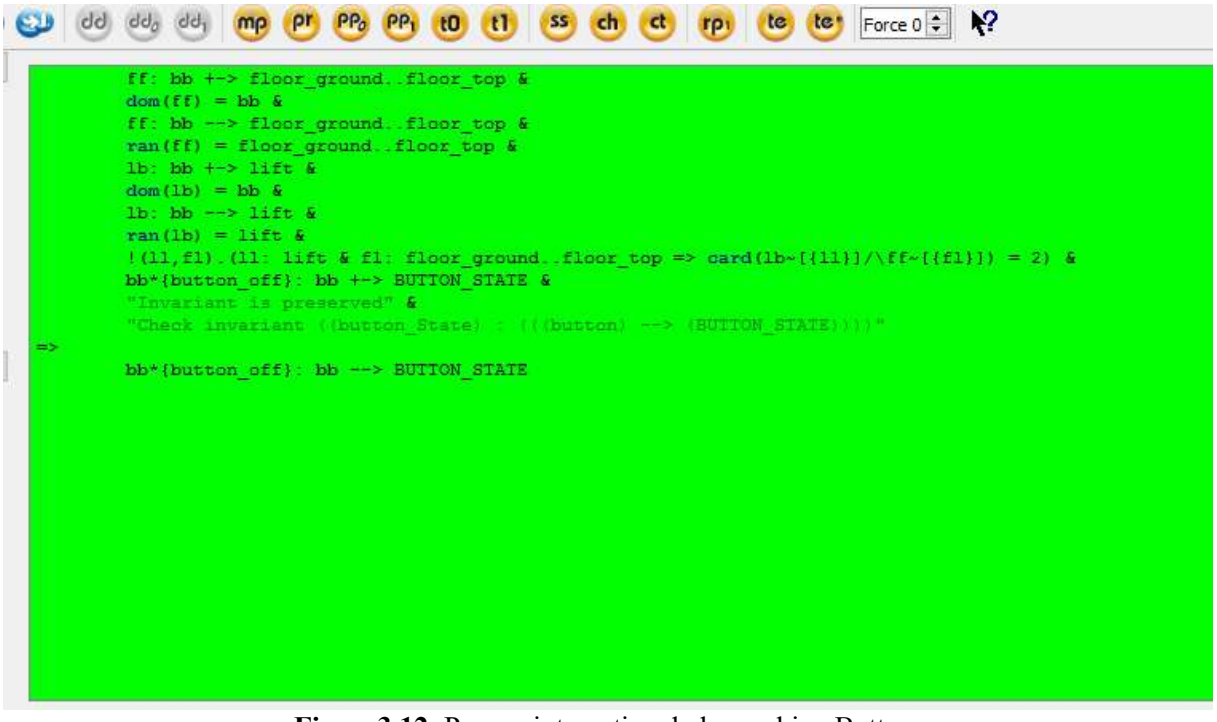


Figure3.12: Preuve interactive de la machine Button

Remarque : Il est possible de générer le code C à partir d'une spécification d'implémentation. L'écriture des spécifications d'implémentation suit des règles théoriques bien fondées.

4. Conclusion

Nous avons choisis comme exemple à étudier un système de contrôle d'ascenseur, au début, nous avons modélisé les diagrammes de classes et d'états-transitions d'UML et la spécification formelle B correspondante du système, ensuite nous avons vérifié par des preuves les résultats obtenues, après nous avons pu détecter et rectifier des erreurs par la démonstration mathématique des obligations de preuve générées et les corrigés. Enfin, nous pouvons générer automatiquement le code C à partir de nos spécifications B. Grâce à l'outil de vérification Atelier B.

Conclusion générale

Dans ce mémoire, nous avons étudié et mis en œuvre des méthodes formelles pour le développement de logiciel, afin de faciliter des taches d'analyse et de spécification pour le développement formel. Plus précisément, nous avons utilisée la méthode de spécification formelle B.

D'un côté, la notation formelle de la méthode B fournis des bases pour la vérification et la validation des systèmes. De part sa fourniture d'un cadre sémantique rigoureux et précis pour les modèles des systèmes étudiés, cette méthode dispose de nombreux outils de preuve et d'animation de spécifications, en particulier l'Atelier B.

D'autre côté, la méthode formelle B, quoiqu'elle ne soit pas orientée objet, nécessite l'utilisation des techniques objets pour ses qualités de description. La spécification UML favorise la compréhension du système conçu et la communication entre les différents acteurs du développement.

Nous avons étudié dans ce mémoire un système en B. le système traite la phase de spécification du développement par l'étude du contrôle d'ascenseur. Il se focalise également sur les deux aspects, dont l'aspect statique est exprimé par le diagramme de classes, alors que l'aspect dynamique est décrit par les diagrammes d'états-transitions. Nous avons spécifié la spécification B qui décrit le système de manière précise et rigoureuse à travers des composants B permettant une analyse fine et une compréhension sans ambiguïté ou incohérence du modèle spécifié.

Le raffinement et l'implantation sont un appui fournis pour la spécification. Ensuite, à l'aide des preuves, elles assurent la fiabilité et la consistance de la spécification en facilitant la détection des erreurs et leur correction du système ; et enfin en peut obtenant automatiquement un code C généré.

Comme perspectives, nous souhaitons que cette méthodologie de développement soit adaptée dans le développement de logiciel par un processus d'automatisation.

Bibliographie

[1] Hassan Diab, ÉVALUATION DE MÉTHODES FORMELLES DE SPÉCIFICATION, Sherbrooke, Québec, Canada, mai 1999

[2] Jean-Fran¸cois Dufourd Spécifications formelles, preuves et programmation Université de Strasbourg UFR de Mathématique et d'Informatique Département d'Informatique, 7 rue Rêne Descartes, 67084 Strasbourg Cedex Et Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (LSIIT, UMR CNRS-ULP 7005), Pole Technologique, Boulevard Sébastien Brant, 67400 Illkirch, France 18 aout 2009

[3] Helene Collavizza, Contribution à la vérification formelle et programmation par contraintes, Soutenue publiquement le jeudi 3 décembre 2009

[4] Seidali REHAB Une approche de transformation des diagrammes
UML vers les spécifications B Thèse de Doctorat en Sciences Département d'Informatique Fondamentale et ses Applications, Faculté des NTIC, Université Constantine 2 ABDELHAMID MEHRI 2015

[5] Nicole Levy La méthode B Le raffinement CNAM NFP 209 2014-2015

[6] J. Christian Attiogbé La méthode B pour la construction rigoureuse de logiciels Novembre 2008, maj 11/2009, 10/2010

[7] C. Choppy. Spécifications algébriques: validation et prototypage. Habilitation à diriger des recherches, Université Paris-Sud, Orsay, 1994

[8] M. A. Ardis et al. A Framework for evaluating specification methods for reactive systems : experience report. IEEE Transactions on Softzuare Engineering, 22 (6), pages 378-389, Juin 1996.

[9] E. M. Clark and J. M. Wing. Formal Methods : State of the Art and Future Directions. ACM Workshop on Strategic Directions in Computing Research-Group
Report: Formal Methods, pages 1415, Juin 1996, Cambridge, MA, USA.

[10] D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applications of Forma1 Methods. Volume 1 : Purpose, approach, analysis and conclusions ;
Volume 2 : Case studies. Technical Report NIST GCR 931626, National hstitute of Standards and Technology, Gaithersburg, MD, Avril 1993.

[11] *C.* B. Jones. The Search for Tractable Ways of Reasonings About Programs. Technical Report, UMCS-92-44, Department of Computer Science, UniversiS. of Manchester, Manchester, UK, Mars 1992-

[12] J. P. Bowen and M. G. Hinchey. Seven more rnyths of formal methods. IEEE Softurare, pages 34-41, 4(l2), Juillet 1995.

[13] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. IEEE Cornputer, pages 56-63, 4(28), Avril 1995.

[14] E. W. Dijkstra. Discipline Programming. Prentice-Hall, 1976

[15] D. L. Parnas, éditeur. Information Distributions Aspects of Design Methodology, *Proceedings of IFIP Congress* 1971, pages 26-30, 1972.

[16] W. Bartussek and D. L. Parnas, éditeurs. Using Assertions About Traces to Write Abstract Specifications for Software Modules, *Proceedings of 2nd Con f erence of Evropean Comporation in In* f *ormatics,* Venice, 1978, Volume 65 of Lecture Notes in Computer Science, Springer-Verlag, pages 21 1-236, 1978

[17] Y. Wang and D. L. Parnas. Specifjring and Simulating the Externally Observable Behavior of Modules. Report/Manuscript 292, TRIO Cm, McMaster University, Ontario, Canada, Août 1994.

[18] J.R. Abrial. The B-Book : Assigning Programs toMeanings. CambridgeUniversity Press, 1996.

[19] J. M. Spivey. The Z Notation : A Reference Manual. Prentice Hail International Series in Computer Science, 2nd edition, 1992. ISBN 013-978529-9.

[20] J.R. Abrial. B : 2000 et plus. Ecole jeunes chercheurs en programmation, Ecole Normale Supérieure de Lyon - 46, Allée d'Italie 69364 LYONCedex 07,March 2000.

[21] J.M.Wing, A Specifier's Introduction to FormalMethods, Computer, vol. 23(9) :8-23, 1990.

[22] Clearsy System Engineering, "Manuel de référence du langage B", version 1.8.8, www.tools.clearsy.com/resources/Manrefb.pdf, France, 2009.

[23] H. Ledang, J. Souquières et S. Charles, ArgoUML+B : un outil de transformation systématique de spécifications UML en B, 2003

ملخص

عملنا هو جزء من هندسة البرمجيات. تركز هذه الذاكرة على المواصفات والتحقق الرسمي من البرنامج. استخدمنا الطرق الرسمية لتطوير نظام كمبيوتر.

استخدمنا على وجه الخصوص الطريقة B و Atelier B لبناء مواصفات رسمية B لنظام التحكم في المصعد. بعد ذلك، أجرينا التحقق من المواصفات التي تم الحصول عليها للنظام للتحقق من بعض الخصائص (التزامات الإثبات) ولضمان اتساقها وتسهيل اكتشاف الأخطاء وتصحيحها قبل توليد الشفرة المكافئة C.

الكلمات المفتاحية :

الطريقة B ; التطوير الرسمي ;التحقيق الرسمي ;Atelier B ; الصقل ; آلة مجردة.

Abstract

Our work is a part of software engineering. This memory focuses on the formal specification and verification of softwares. We used formal methods to develop a computer system.

In particular, we used the method B and its workshop Atelier B to construct a formal specification B of an elevator control system. Then, we performed a verification B of the specification obtained of the system to verify certain properties (proof obligations) and to ensure its consistency and to facilitate the detection and correction of errors before generating the equivalent C code.

Keywords :

Method B; formal development; formal verification; Workshop B; Refinement, abstract machine.

Résume

Notre travail entre dans le cadre de génie logiciel. Cette mémoire s'intéresse à la spécification et vérification formelle des logiciels. Nous avons utilisé les méthodes formelles pour développer un système informatique.

Nous avons utilisé en particulier la méthode B et son atelier B pour construire une spécification formelle B d'un système de contrôle d'ascenseur. Ensuite, nous avons effectuée une vérification B de la spécification obtenue du système afin de vérifier certaines propriétés (les obligations de preuve) et pour garantir sa consistance et faciliter la détection et la correction des erreurs avant de générer le code C équivalent.

Mots Clés :

Méthode B; développement formel ; vérification formelle ; Atelier B ; Raffinement ; Machine abstraite.