République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Ibn khaldoun -Tiaret
Faculté des Mathématiques et de l'Informatique
Laboratoire des recherches en Informatique et Mathématique (LIM)

THÈSE

pour obtenir le grade de docteur délivré par
l'Université Ibn khaldoun -Tiaret
Spécialité doctorale "Informatique"
Option : "Systèmes Embarqué et Temps Réel"
Préparée au "Laboratoire des recherches en Informatique et Mathématique(LIM)"
Dans le cadre de Doctorat de
"3 ème Cycle L.M.D"
présentée et soutenue publiquement par

Abdelhamid HARICHE

Approche à base d'opérateurs pour la validation formelle de systèmes micro-électroniques orientés NoC

Directeur de thèse : Mostefa BELARBI

Membres de Jury

M. Abdelkader SENOUCI	**Professeur ,Université** de Tiaret	Président
M. Abou El Hassan BENYAMINA,	**Professeur ,Université d'Oran**	Rapporteur
M. Mohamed SENOUCI,	**Professeur ,Université d'Oran**	Rapporteur

University of Tiaret
Mathematic and Computer Science Faculty
LIM Research Laboratory, MBox 78 Zaaroura ,Tiaret.

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la recherche Scientifique
Université IBN Khaldoun Tiaret-Algérie
Faculté de Mathématique et informatique
Laboratoire des recherches en Informatique et Mathématique (LIM)

THÈSE

pour obtenir le grade de docteur délivré par

l'Université Ibn khaldoun -Tiaret

Spécialité doctorale **"Informatique** *option* **: Systèmes Embarqué et Temps Réel"**

Préparée au **"Laboratoire des recherches en Informatique et Mathématique(LIM)"**

Dans le cadre de Doctorat de **"3**^{ème} **Cycle L.M.D"**

présentée et soutenue publiquement par

Abdelhamid HARICHE

le 15/07/2019

Approche à base d'opérateurs pour la validation formelle de systèmes micro-électroniques orientés NoC.

Directeur de thèse : **Mostefa BELARBI**

Membres de Jury

M. Abdelkader SENOUCI, Président	Professeur	Université Ibn khaldoun - Tiaret
M. Abou El Hassan BENYAMINA, Rapporteur	Professeur	Université Essenia - Oran
M. Mohamed SENOUCI, Rapporteur	Professeur	Université Essenia - Oran

University of Tiaret
Mathematic and Computer Science Faculty
LIM Research Laboratory, MBox 78 Zaaroura ,Tiaret.

Table des matières

Liste des figures

Liste des tableaux

Remerciements

Je tiens tout d'abord à remercier le directeur de cette thèse, Dr Mostefa BELARBI pour m'avoir fait confiance malgré les connaissances plutôt légères que j'avais en septembre 2012 sur les méthodes formelles, puis pour m'avoir guidé, encouragé, conseillé et en me faisant l'honneur de me laisser découvrir et réaliser des travaux dont j'espère avoir été à la hauteur.
Mes remerciements vont également à M. Prof. Abdallah CHOUARFIA, pour la gentillesse et la patience qu'il a manifestées à mon égard durant cette thèse, pour tous les conseils et les programmes qu'il a bien voulu m'envoyer et d'être mon ancien directeur de thèse.
Je remercie pour son soutien et son appui Monsieur Prof. Abdelkader SENOUCI, directeur du laboratoire de recherche en mathématique et informatique(LIM), pour m'avoir accueilli au sein de cette institution, et pour les conseils stimulants que j'ai eu l'honneur de recevoir de leur part et aussi qui de plus m'a fait l'honneur de présider le Jury de cette thèse.
Ainsi je remercie Monsieur le Prof. Abou El Hassan BENYAMINA qui a accepté d'être un rapporteur de cette thèse, et à cet effet je suis très reconnaissant.
Monsieur le Prof. Mohamed SENOUCI. m'a fait l'honneur de participer au Jury de soutenance ; je l'en remercie profondément.
Je tiens aussi à mentionner le plaisir que j'ai eu à travailler au sein de laboratoire LIM, et j'en remercie ici tous les membres.
Pour l'hospitalité et les encouragements qui m'ont permis de faire un séjour dans de bonnes conditions durant cette thèse je remercie M. Prof. Michael BUTLER et toute membre de son groupe.
Enfin, ces remerciements ne seraient pas complets sans mentionner tous ceux sans qui cette thèse ne serait pas ce qu'elle est et plus précisément : toute personne de la faculté des mathématiques et de l'informatique et de l'université Ibn Khaldoun qui m'ont largement rendu grâce à leurs encourageante présence quotidienne durant mon cursus pour cette thèse.

Dédicace

Tout d'abord je tiens à remercier le Bon dieu "ALLAH" pour m'avoir donner la puissance d'accomplir cette thèse.Ce fut pour moi un grand plaisir.
Ainsi chaudement mes sincères gratitudes pour mes parents et toute ma famille pour leurs encouragements et leur assistance aussi bien morale que d'autres.
Je tiens à remercier sincèrement Monsieur, BELARBI mostefa, qui, en tant que Directeur de thèse, à s'est toujours montré son savoir faire et sa disponibilité tout au long de la réalisation pour m'avoir inspiré et m'a aidé et a consacré son temps précieux, qui sans lui cette thèse n'aurait jamais vu le jour.
Ceci dit le proverbe anglais stipule :"*Many hands make light works.*" je tiens à remercier toute personne qui m'a poussé lentement et surement.

Introduction générale

Le nombre de fonctions logicielles contrôlées dans les systèmes embarqués ne cesse pas d'augmenter **ainsi que** la complexité de leur mise en œuvre. Par conséquent, les problèmes causés par des bugs dans la mise en œuvre ou par des spécifications incomplètes ou incorrectes peuvent conduire à un comportement involontaire **et** difficile à repérer et **l'éviter**. Face à une telle complexité, la conception d'un projet de système embarqué à base de Field Programmable Gate Array (FPGA) avec **un seul langage** est encore insuffisante. La vérification est utilisé pour démontrer si la fonctionnalité du projet en cours d'élaboration est conforme à leurs spécifications ou non [1], les types les plus populaires de vérification sont la vérification fonctionnelle (informelle), formelle et hybride. Selon [1, 2], environ 70% des ressources utilisées dans une conception de matériel sont utilisés dans l'étape de vérification fonctionnelle [1]. Les techniques fonctionnelles ultimes destinées à valider la mise en œuvre de solutions intégrées telles que les systèmes multiprocesseurs sur puce Multi-Processor System on Chips (MPSoC), qui ont l'avantage non seulement la fiabilité, mais aussi plus de gain de temps de mise en œuvre du système. Avoir ce travail est de commencer à améliorer **(en termes de réduction du temps de réalisation) la fiabilité de la phase** de validation d'une conception de haut niveau d'un système de MPSoC basé sur une architecture de réseau à base Network on Chips (NoC) en fournissant un résultat satisfiable pour la génération de code Very High Speed Integrated Circuit Hardware Description Language (VHDL).
Une approche en plein essor dans l'ingénierie des logiciels est celle des architectures logicielles **où** les descriptions d'architectures logicielles permettent de modéliser de façon assez intuitive les différentes unités de calcul et les interactions ou liens de communication entre elles. Elles ont donné lieu à de nombreux travaux ces dernières années, par exemple : [3–8]. D'autre part, un développement formel fournit un moyen sûr pour passer d'une spécification abstraite d'un système à un niveau de spécification plus détaillé, dit concret, afin de pouvoir, par la suite, générer automatiquement une implémentation **(le code)** pour une plate-forme et un langage de programmation donnés. Néanmoins, une méthode formelle de développement est à forte composante mathématique et son utilisation nécessite habituellement une formation spécifique.
Les méthodes formelles se réfèrent aux techniques mathématiques utilisées pour la spécification, le développement et la vérification des systèmes logiciels et matériels. Les méthodes formelles peuvent être classés sur la base de la méthodologie utilisée, en deux grandes catégories : ***les méthodes de vérification* et *les méthodes correcte par construction*. Dans l'approche basée sur la vérification, l'exactitude est établie à posteriori**, après que le programme en question est développé **et dans l'approche correcte par construction la mise au point du système est réalisée de façon progressive où chaque étape intermédiaire est vérifiée. On constate dans les méthodes par construction que la mise en œuvre d'une spécification formelle d'un système est évaluée selon un plan fournit**. Certaines méthodes formelles [9, 10] ont besoin **durant la modélisation** d'un processus pour concrétiser les spécifications abstraites en des spécifications plus détaillées ou de générer des implémentations appelé le raffinement. **En outre**, « raffiner » est de fournir une nouvelle formulation de la performance du système, qui ne doit pas contredire les propriétés ; cela se traduit souvent par une diminution du niveau d'abstraction ou indétermination, par exemple en ajoutant des détails à une spécification abstraite, de manière à approcher une mise en œuvre dans un langage de programmation. Mais ces approches ne tiennent pas **en** compte la description de l'architecture du système. Par ailleurs, certains langages de description d'architecture peuvent autoriser une compilation en code exécutable. Cependant, ils contraignent la spécification architecturale dès le départ avec des détails d'implémentation [11].
L'Event-B [12, 13] est une technique mathématique qui peut être incorporée dans le processus de développement des systèmes matériels et logiciels [14]. Event-B peut être utilisé pour modéliser des systèmes discrets et tombe dans la catégorie « correcte par construction ». Le formalisme est basé sur la méthode B [15], une méthode qui a déjà une bonne **expertise** industrielle [12]. La modélisation de l'Event-B est réalisée au moyen de deux composants : **les contextes et les machines. Les contextes définissent les aspects statiques d'un modèle ; ils peuvent inclure des ensembles et des constantes, ainsi que les axiomes et les théorèmes décrivant les ensembles et les constantes. Les machines, d'autre part, décrivent la partie dynamique d'un modèle ; elles incluent des variables et des invariants, et des événements (transitions)**. Event-B utilise la théorie des ensembles construit autour de la logique du premier

ordre comme un **atout** pour la modélisation. Les obligations de preuve sont générées à partir de modèles afin de vérifier leur cohérence par rapport à une sémantique de comportement [16]. La plate-forme RODIN [17] fournit un ensemble d'outils pour **bien établir** la spécification, le raffinement et la preuve en Event-B. RODIN propose un environnement de modélisation réactive qui le rend plus facile **à l'utilisateur pour relier** les modèles, les obligations de preuve et leurs preuves correspondantes. Etant donné que les preuves sont importantes pour l'activité de modélisation, **RODIN fournit une infrastructure de preuve extensible par l'ajout des prouveurs externes (par exemple, prouveurs Atelier-B [17,18]) qui peuvent également être utilisés en conjonction avec le prouveur interne RODIN.**
Le problème de recherche posé est **de comprendre durant l'opération de raffinement le moyen classique pour passer d'une description architecturale donnée à une spécification plus détaillée dans une méthode formelle**(Figure X.1). **L'objectif de ce travail est donc cerné d'automatiser ce passage classique, afin de l'exploiter systématiquement**. Ce raffinement doit être en relation avec une « plate-forme formelle » **et une description de code par un langage pour des architectures existantes**. De plus, il doit veiller à la conservation des biens entre les descriptions. A la suite de cette phase de spécification

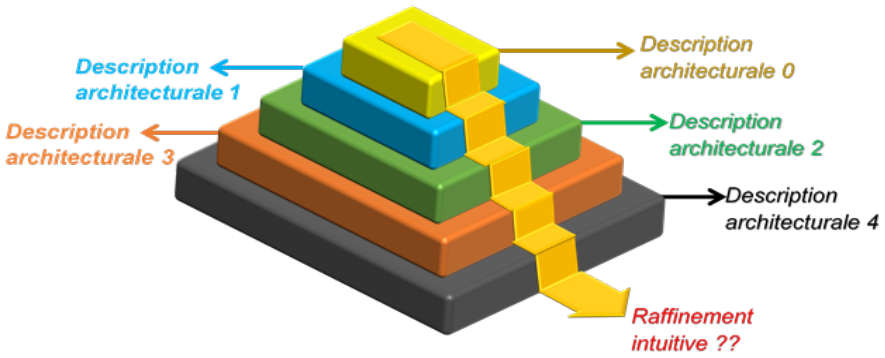


FIGURE X.1 – Le problème d'intuitivité de raffinement

architecturale, il n'aura qu'à "appuyer sur un bouton" afin qu'elle soit raffinée et que soit générée une spécification formelle conforme à la méthode de développement formel choisie [19]. **La spécification abstraite obtenue doit être traitée de façon conventionnelle en utilisant plusieurs raffinements dans le cadre d'un développement formel classique jusqu'à la phase d'implémentation**.
Le fait d'opérer cette phase de raffinement entre deux spécifications écrites dans des formalismes avec des pouvoirs d'expression comparables nous permettra, de plus, de poser les questions suivantes :

1. **Quel raffinement ?**
   Peu de langages et méthodes supportent explicitement un processus de "raffinement" en plusieurs étapes distinctes, à la différence de ceux qui ne permettent qu'une phase de compilation en code [20–24]. L'approche du raffinement est produite par ces deux types de mécanismes :
   - La décomposition(ou raffinement horizontal) qui ne provoque pas de changement de niveau d'abstraction, mais qui ajoute à la spécification de nouveaux composants, de façon plus détaillée **prenant l'exemple de l'Event-B : la décomposition d'un modèle en Event-B est effectuée par des contextes et des machines pour exprimer le comportement complexe d'un système et détailler ses opérations élémentaires**.
   - La transformation (ou raffinement vertical) de certaines parties de la spécification, décidées par des choix liés à la plate-forme d'implémentation ultérieure, et qui, de ce fait, diminue le niveau d'abstraction, sans remettre en cause la spécification (seules des précisions sont éventuellement ajoutées), **si toujours nous prenons Event-B comme exemple on peut dire que l'ensemble des variables du code VHDL sont raffinées par des invariants qui invoquent une théorie exprime le langage VHDL en Event-B**.
   - Ainsi, ce sont plutôt des mécanismes de raffinement vertical que nous devons mettre en place afin, notamment, de transformer la structure de contrôle d'une description architecturale (des

opérations de théories, ces opérations sont invoquées par des invariants et leurs axiomes apparait dans les contextes et les évènements qui déclenchent des traitements pour ces invariants) pour l'expliquer en des termes compréhensibles par la méthode de développement formel en Event-B.

2. **Quels formalismes**
   Notre attention est saisie pour le cadre du développement de logiciel où un système se définit formellement en Event-B par l'intermédiaire des raffinements décrits. Ce système doit permettre de mettre en place un "processus formel" de raffinements successifs, menant d'une description abstraite à une spécification suffisamment concrète pour qu'un développement formel de l'application soit possible. **Dans une aperçue plus grande pour notre point de vue, l'étape de modélisation est prise de façon que chacune des spécifications engendrées par le processus de raffinement correspond à un aspect bien précis de la transformation et sont collectés sous formes des théories et aussi démontrables (en Event-B on bien dit déployés), en relation avec soit le langage de cible d'architecture choisi (qui peut être aussi une théorie en Event-B à invoquer), soit la méthode de développement formel.** Ces niveaux d'abstraction intermédiaires facilitent la mise en œuvre du raffinement, mais resteront "invisibles" aux yeux de l'architecte.
   Cette suite de raffinements doit être décrite formellement. Certains langages de description d'architecture logicielle, tels **que** SADL [19,25–27] ou Rapide [28–30], prennent en compte le raffinement sur plusieurs niveaux d'abstraction, mais ils ne permettent pas d'achever le développement complet de systèmes logiciels complexes car ils n'offrent pas de support pour la génération du code. **Actuellement, les méthodes qui utilisent le raffinement se classent en deux catégories. La première nous oblige à faire prouver, après chaque étape de raffinement du système, la préservation de propriétés avec le niveau d'abstraction précédent (le raffinement est vérifié à posteriori). La seconde nous permet de construire la nouvelle spécification concrète à partir de la spécification abstraite (le raffinement s'inscrit dans un développement génératif). Cette dernière catégorie est la plus intéressante, puisqu'en garantissant que chaque étape de raffinement conserve les propriétés de la description architecturale de départ, indépendamment du système logiciel en cours de développement, nous pouvons nous abstenir d'une phase de vérification à posteriori.**
   L'espace de validation formelle qui nous intéresse, est Event-B, **une méthode qui** est fondé sur une algèbre de processus, le π-calcul ; ceci octroie à π-SPACE un pouvoir expressif important, qui permet notamment l'expression d'architectures dynamiques. Le langage de spécification formelle est celui de la méthode Event-B basée sur la méthode B [31] fournit la boîte à outils « Tool-set » RODIN [15] pour mener **à des bonnes spécifications**, des améliorations et des preuves dans l'Event-B. Cependant, l'un des objectifs de ce document est **d'améliorer le processus de description architecturale en Event-B de façon qui permet la génération de code exécutable et force** la spécification architecturale dès le début avec des détails de mise en œuvre. Après cette phase de spécification architecturale, il aura seulement raffiné et généré en tant que spécification formelle conforme à la méthodologie de développement formel choisi [32](voir Figure X.2). **L'objectif ultime pour le formalisme Event-B est fixé donc pour assurer qu'une spécification raffinée d'une spécification abstraite peut être raffinée à son tour conventionnellement mais non pas d'une manière classique**.

3. **Pourquoi l'outil de l'Event-B RODIN**
   L'outil RODIN offre un environnement de modélisation réactive où le modélisateur est constamment informé sur les effets des changements apportés aux modèles [33]. Cela signifie que, lorsqu'un modèle est modifié automatiquement il est vérifié pour les erreurs de syntaxe et de type, ses obligations de preuve sont générées et le statut de ses preuves sont mises à jour. **L'activité de preuve est essentielle à la modélisation tant que le modélisateur peut avoir un aperçu considérable pour l'ensmeble des modèles (appelé *un projet*) en inspectant les preuves qui sont échouées** ; cela peut guider le modélisateur de modifier le modèle d'une manière telle que les preuves deviennent plus faciles à réaliser. **Cependant dans certains cas, les preuves peuvent être échouées à cause**

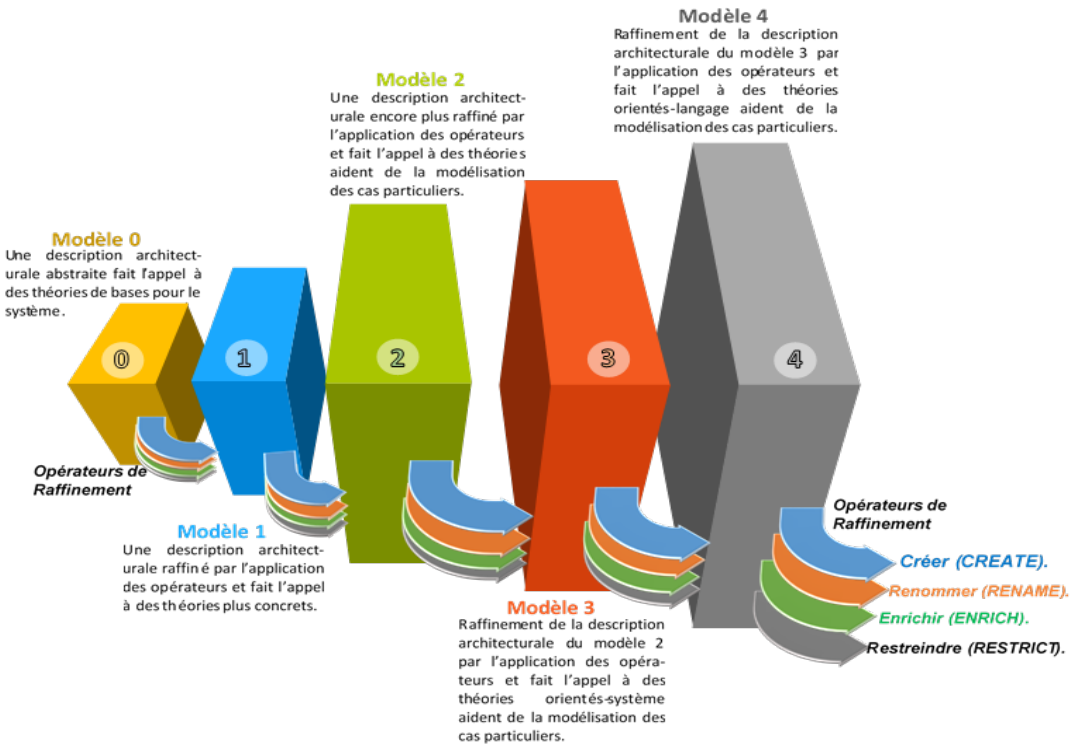


FIGURE X.2 – L'objectif de la thèse.

des limitations dans l'infrastructure de prouveur, par exemple, l'absence de certaines règles de preuve. En dépit pour un but d'optimiser pour la réutilisation de preuve [34]de l'architecture actuelle de RODIN nous présentons les limitations suivantes :

- afin d'ajouter une nouvelle règle de preuve, il est nécessaire de mettre en œuvre un schéma de règle dans Java. Par conséquent, **cette tâche a besoin un certain niveau de compétence avec le langage de programmation Java, ainsi que la connaissance de l'architecture RODIN.**
- **après qu'une nouvelle règle est ajoutée, la solidité du prouveur augmentée avec la nouvelle règle établie. Bien que les outils de vérification Java(par exemple Java Modellig Language (JML)) peuvent être utiles à cet effet pour qu'une telle validation ne doit pas être effectuée comme celle pour l'une des règles intégrées.**

Les prouveurs externes intégrés dans l'infrastructure RODIN **font une preuve en appliquant deux mode Mono-Lemma prover (ML) et Predicate Prover (PP) [18], ils** ne fournissent pas suffisamment d'informations sur la façon dont la preuve d'un séquent a été atteint**, citant l'exemple d'une information telle que l'ensemble des hypothèses requises qui est trés importante pour** la réutilisation et la régénération de preuve [34]. **Plus que l'extensibilité de preuve, nous cherchons un autre axe à résoudre en focalisant sur les problèmes liés à l'extensibilité de langages. Le langage mathématique de l'Event-B est basée sur la théorie des ensembles, comme celle-ci construit dans [12].** L'arbre de syntaxe abstraite représentant les formules dans Event-B ne peut pas être étendue avec une nouvelle syntaxe (l'ajout d'un nouvel opérateur, par exemple). Cela pose un problème majeur qui entrave la réutilisabilité dans l'Event-B. **Enfin, notre raison pour choisir d'améliorer l'outil RODIN est cristallisée sur le fait que l'extensibilité de langage et de prouveur sont intrinsèquement liés pour que la capacité de raisonner des propriétés d'un système qui sortent de ce qu'il existe dans un modèle en Event-B est efficace et cela par conséquence prend une importance primordiale.**

4. **Quelle est le bénéfice des extensions de théories dans RODIN**
   L'objectif durant cette thèse est d'utiliser l'extensibilité globale de l'Event-B afin d'améliorer la

convivialité et l'efficacité de la méthodologie et de faciliter l'ajout de nouveaux opérateurs **(l'élément de base pour les extensions en Event-B) pour faire convenir les besoins des utilisateurs finaux**. Il est essentiel de veiller à ce que toute technique qui permet d'atteindre les objectifs mentionnés ci-dessus :

— Maintenir l'aspect pratique d'utilisation et d'assurer la préservation de **la solidité et la praticité d'utilisation qui sont deux facteurs très important pour faciliter la tâche de formalisation appliquée par des utilisateurs finaux** au lieu d'avoir une grande compétence de l'écriture du code Java et ses langages de validation formelles JML avec une vaste connaissance de l'architecture interne de RODIN.

— La préservation de solidité garantit que toutes les extensions ne compromettent pas les fondements logiques du formalisme. La logique de l'Eventt-B est largement étudié dans [35] où une définition claire de la solidité est présentée, **et cette propriété (présentée comme un résumé des travaux de Schmalz [35]) est récemment intégrée dans l'Event-B [36, 37]**. Les points suivants résument les objectifs de cette propriété :

(a) Fournir un mécanisme par lequel les utilisateurs peuvent définir des opérateurs et des types de données d'une manière **usuelle (c'est à dire en conformité avec les pratiques existantes pour développer des modèles en utilisant l'outil RODIN) permet de modéliser les extensions du langage utilisé en Event-B.** Le nouveau mécanisme doit adhérer aux exigences : la pratique de l'utilisation et à la préservation de la solidité.

(b) Fournir un mécanisme par lequel l'infrastructure de preuve dans RODIN peut être augmentée avec de nouvelles règles de preuve. Toutes les règles nouvellement ajoutées devront être validées pour que la solidité du prouveur existant ne soit pas compromise. Les règles de Réécriture et les règles d'inférence sont utilisées dans RODIN pour exécuter des obligations de preuve. Les étapes suivantes sont importantes pour atteindre cet objectif :

i. Fournir une étude unificatrice de réécriture **des termes s'ils sont bien définis (Well-Definedness (WD)). Ceci permet d'une importance majeure puisque les logiques en Event-B sont traités avec des fonctions partielles qui peuvent donner lieu à des conditions potentiellement mal définies.**

ii. D'étudier comment la réécriture de termes peut être intégré comme **une étape qui prouve la bonne définition (WD) et préserve le calcul des séquents [34, 38].** Mehta [38] présente un calcul de raisonnement en présence de fonctions partielles. **Le calcul comprend un ensemble bien définie préserve des règles d'inférence qui peuvent être utilisés pour la déduction dans les étapes de preuves en Event-B. Le travail de Mehta est considéré comme une brique de base pour l'infrastructure de preuve dans l'outil RODIN [34, 38], et il décrit la manière d'ajouter des règles de réécriture en respectant les mesures de calcul préservés pour leurs bonne définition WD. En particulier, comment les règles de réécriture conditionnelles peuvent être utilisés avec les bonne définitionWD préservées par des règles d'inférence afin d'améliorer les capacités de prouveur de RODIN**.

(c) Montrer comment le soutien de l'outil RODIN est fourni pour atteindre les deux premiers objectifs. **La théorie plug-in présente une réponse pour ceux qu'il remet en question l'importance de l'extensibilité en Event-B et nous affirmons cette idée durant cette thèse par un exemple sur la réalisation des réseaux sur puces NoC par l'utilisation des propriétés de colloration des nœuds défaillants et la spécification du comportement exprimé en code VHDL dans l'environement réel et cela montre comment notre approche peut être incorporé dans la modélisation et l'activité de preuve en utilisant Event-B.**

5. **Les objectifs de la thèse**
Le formalisme fondamental durant cette thèse est l'Event-B manipulé l'ensemble des outils (toolset) appelée RODIN où le mécanisme de raffinement est tiré comme un critère majeur

de sélection. RODIN est également fondé sur la notion de la théorie (la théorie Plug-in) **et cela aide** pour exprimer **des** propriétés désirées **pour un système à base NoC et son comportement du code VHDL pour ce type de systèmes. L'intérêt d'utiliser une modélisation par théorie réside dans la facilité de la génération des preuves pour un modèle en Event-B, on affirme pour ce qui est déclaré par avant que les preuves pour ce genre de théories exprimées sont importantes pour l'activité de la modélisation et peut également être utilisé en conjonction avec le prouveur interne RODIN.**
Si on veut mettre une piste de comparaison entre une simple modélisation en Event-B et celle basée sur les théories on trouve dans la première que Chaque modèle contient des théories sous forme de différents types (entiers, ensembles, relations, fonctions⋯), quand les systèmes qui **nous avons entraîné de valider durant cette thèse** nécessitent des structures mathématiques supplémentaires (par exemple : listes, arbres, graphes, réels et autre liées au code VHDL) qui pourraient définis axiomatique dans les modèles, mais aucun polymorphisme est pris en charge , aucune extension de prouveurs directe est possible et aussi les définitions et les règles de preuve ne sont pas assurées, **par contre** notre travail conduit à utiliser une nouvelle extension de la modélisation en manipulant la théorie plug-in (sous forme de polymorphes) qui permet aux utilisateurs de : définir de nouveaux opérateurs mathématiques et des types de données, ajouter des règles de preuve (règles interactives ajoutées à des tactiques automatisées) au prouveur de RODIN et générer des OP de solidité pour les nouvelles définitions et les règles de preuve.
Le fait d'introduire des opérateurs durant cette phase de raffinement en Event-B présenté dans des travaux réalisés [39, 40] où deux spécifications écrites avec des règles [41] qui nous permettent d'assurer la compatibilité des propriétés éxprimés en utilisant les opérateurs de raffinement avec le comportement de système désiré. Dans notre approche on parle des propriétés que l'architecte souhaite de voir qui doivent être conservées, c'est à dire conserve par exemple, la vigilance, la sécurité, l'accessibilité de la modélisation de comportement de système et rapprocher ses modèles d'une forme à proximité du code VHDL en utilisant des opérateurs de raffinement. Notre approche casse l'intuitivté de raffinement en utilisant des type abstrait de données, elle repésente la réponse evident pour qu'un archtitect utilise l'outil RODIN mettre en question sur la différence entre notre approche par rapport l'extension de l'évènement (Event extension).La raison qui illustre l'utilisation de raffinement dans l'outil RODIN avec un type de données abstrait algèbrique [41–43] est de prendre une cible de vérification durant ce travail : le réseau à base de NoC auto-organisé (Self-Roganised Network on Chips (SONoC)) [44]. Nous proposons un ensemble de théories de base : **à partir de la théorie de NoC [45,46], ensuite une théorie étendue pour exprime l'exécution de l'auto-récupération des noeud défaillants. Après avoir vérifier tous les scénarios possibles que ce système peut rencontre à l'intérieur du réseau sans fil puis nous utilisons la théorie des graphes produit par JR Abrial [47], mais le grand nombre de caracteristiques ne pourra pas être pris en considération par la vérification exhaustive de toutes les propriétés, alors il est évident d'étendre la théorie des graphes en théorie des graphes colorés** pour animer et augmenter la complexité de la vérification. Dans la phase finale, et après avoir vérifié toutes les propriétés pour cette architecture, il peut être représenté avec la réalisation de son comportement VHDL en utilisant la théorie de VHDL. Cependant, **tous les étapes de développement pour notre système posé durant cette thèse** sera vérifié avec un mode incrémental basé sur des opérateurs spéciaux pour couvrir différents axes de validation d'une manière méthodologique.

6. **Plan de thèse**
Cette thèse fait des contributions importantes à l'Event-B en général comme il est décrit dans le paragraphe 1.2 et aussi pour le démarche de validation des systèmes micro-électroniques à base NoC. Le chapitre 1 fournit un historique des travaux de la validation des différente systèmes NoCs détaillés par des méthodes formelles et informelles **et la raison d'utiliser l'Event-B. Ainsi que** les différents concepts nécessaires **utilisé** pour le reste de la thèse. L'Event-B et l'outils RODIN sont introduits et ses limites **dans la version courante** sont décrites dans le chapitre 2. En outre,

les bases mathématiques utilisés dans l'Event-B est présenté avec un aperçu détaillé en terme des définitions. Les autres chapitres sont classés comme suit :

— **Chapitre 3 :** Ce chapitre présente une contribution de nature plus théorique. L'état de l'art du raffinement en génie logiciel est exposé dans ce chapitre. L'ensemble des bases théoriques pour l'approche qui nous permet d'automatiser le processus de raffinement par des opérateurs. Sachant que notre approche sera une base solide pour la validation des systèmes à base NoC présenté dans le chapitre suivant. Enfin, nous décrivons **une comparaison entre les opérateurs de raffinement et les autres outils utilisés** par Event-B pour rendre la validation d'un système (par exemple le réseau de capteurs à base NoC) plus convenable.

— **Chapitre 4 :** Une approche pratique de l'extensibilité l'Event-B en terme de prouveur et de langage dans ce chapitre, nous présentons le volet théorique qui sera utilisé pour définir les extensions dans l'Event-B. Nous montrons aussi comment la composante théorie peut être utilisée pour définir de nouveaux opérateurs polymorphes et types de données. Les obligations de preuve qui assurent la solidité des extensions sont discutées. Puis nous introduisons le plug-in de la théorie « Theory plug-in » qui incarne les différentes idées présentées dans cette thèse. Aussi nous présentons l'exemple d'application de l'idée présenté dans le chapitre 3 qui démontrent l'utilité du plug-in Theory par les différentes théories utilisés NoC, les graphes colorés et la théorie pour le code VHDL. Par la suite nous expliquons notre moyen de pensée pour la phase de génération de code VHDL sera ajoutée pour garantir que notre approche englobe le cycle de vie de développement des systèmes à base NoC.

— La dernière partie de ce document conclut la thèse et résume nos contributions et les domaines possibles pour les travaux de futur.

Chapitre 1

L'émergence de l'Event-B dans le développement micro-électronique

« In industry, people develop their products under precise guidelines, and usually, the introduction of such guidelines in an industry takes a significant time. In this context, managers are very reluctant to change the guidelines to incorporate the use of formal methods… »

JEAN-RAYMOND ABRIAL

Sommaire

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. Les performances d'une architecture implémentée dans un System on Chips (SoC) dépendent fortement du système d'interconnexion et du protocole de communication entre les unités de calcul. Avec la technologie d'intégration croissante, en particulier, l'augmentation permanente des densités de système à base de logique reconfigurable à grain fin de type FPGA (parallélisassions, intégration d'unités de calcul, des clusters étant un réseau de processeurs sur une même carte) le coût et le temps nécessaires pour le développement de systèmes larges et complexes est souvent achevé en retard par rapport au plan initial surtout pour les systèmes enfouis comme les réseaux sur puce (ou NoC) où le développement est basé sur la synthèse d'un matériel dédié.
Sur ce dans la suite, on présente brièvement les réseaux sur puce connus dans la littérature et les approches de validation (informelles et formelles) de ces systèmes à base NoC et leurs concepts de modélisation pendant le développement du système. Ensuite on aborde la validation formelle. Enfin, on introduit les principaux points pour la méthode Event-B utilisé durant ce mémoire et l'approche proposée pour son étude.

1.1 La nature des systèmes à base NoC

La nécessité d'une validation lors de la création de réseaux embarqués à base de NoC est une étape évidente, lorsqu'on parle de la vérification de réseau basé sur NoC on entend pour valider tant des propriétés pour ce genre de système, on le considère comme un système embarqué intégré très spécifique basé sur l'ensemble des unités de traitement connectés via un moyen de communication spécifique. Dans le suivi, on présente ce système avec les différents points de vue des validateurs.

1.1.1 Un système embarqué

Un système à base NoC est dans le premier lieu un système embarqué, on peut distinguer deux catégories de systèmes embarqués : les systèmes autonomes et les systèmes enfouis :

— **Un système autonome** correspond à un équipement autonome contenant une intelligence qui lui permet d'être en interaction directe avec l'environnement dans lequel il est placé. Il s'agit des téléphones portables, agendas personnels électroniques ou Global Positioning System (GPS).

— **Un système enfoui** (nous l'appelons en anglais **"Embedded softwares"** il est souvent invisible à l'utilisateur) est un ensemble cohérent de constituants informatiques (matériel et logiciel), d'un équipement auquel il donne la capacité de remplir un ensemble de missions spécifiques [48]. Il s'agit d'un système physique sous-adjacent avec lequel le logiciel interagit et qu'il contrôle.

Dans ce cas, les systèmes embarqués peuvent être vus comme des sous-systèmes de systèmes plus importants qu'ils sont chargés de commander et/ou de surveiller. Ils sont donc très étroitement liés à ces systèmes [48]. La question qui se pose est sans doute : comment peut-on maîtriser la complexité de ces systèmes au cours des phases de développement ? D'autant que les systèmes embarqués sont soumis à des contraintes techniques très sévères qui pèsent sur eux incluant l'optimisation du code, d'énergie, d'espace, etc.
Une comparaison des systèmes embarqués avec notre Personal Computer (PC) est sans doute difficile à établir (voir Tableau I.1), vu la complexité et la dispersion de ces systèmes. Toutefois, sur le plan technique sachant que les NoC sont des systèmes embarqués ainsi que sur le processus de développement de ses applications. Alors en quoi les systèmes embarqués diffèrent-ils de notre PC ? Une telle comparaison est présentée dans le tableau suivant et pour cela, les systèmes embarqués exigent des outils spécialisés et des méthodes de conception plus efficaces.

Système embarqué	PC
Dédié à des tâches spécifiques.	Des plateformes génériques.
Besoin moins de ressources.	Besoin plus de ressources.
Fonctionne dans des conditions environnementales extrêmes (des contraintes d'énergie et des contraintes temps-réel).	Fonctionne dans des conditions simples.
Possibilité d'utiliser un système d'exploitation temps-réel Real-Time Operating System (RTOS) ou sans system d'exploitation.	Utilise un système d'exploitation comme unix, windows etc.
Besoin une grande sélection de processeurs et architectures de processeurs.	Architecture Assi petit en terme de processeur et sélection de processeur.
plus de risque de défaillances logicielles.	Moins de risque de défaillances logicielles.

TABLEAU I.1 – PC vs. système Embarqué

1.1.2 Un système à base MPSoC

Les applications phares de la micro-électronique comme celles du multimédia sur systèmes mobiles sont difficiles à implémenter, à cause de fameux concept Time-To-Market qui représente une des contraintes sur le temps de développement,**à cet effet un nombre important des standards et des algorithmes de traitement sont nécessairemnt crées pour gèrer les systèmes largement parallélisés**,**ils** sont utilisés **pour ce genre des applications**. Dans les dernières années, l'introduction des MPSoC a apporté une solution qui permet la création des applications industriels existants : le CELL [49] développé par IBM, Toshiba et Sony, et l'architecture Davinci de Texas instruments [50]. Le besoin **de** communications efficaces s'accroıt, spécialement dans les systèmes massivement parallèles (MP2SoCs : Massively Parallel Multi Processors Systems-on-Chip) [51, 52]. **Ainsi que pour** des besoins de broadcast et de multicast apparaissent **on utilise** la notion de Quality of Service (QoS) **qui** est indispensable pour tout **système multimédia de type SoC** . La quantité de données échangées entre les différents composants d'un système **MPSoC** rend l'utilisation de connexions de type bus et point-à-point **très difficile**. Avec la mise en œuvre de plusieurs pro-

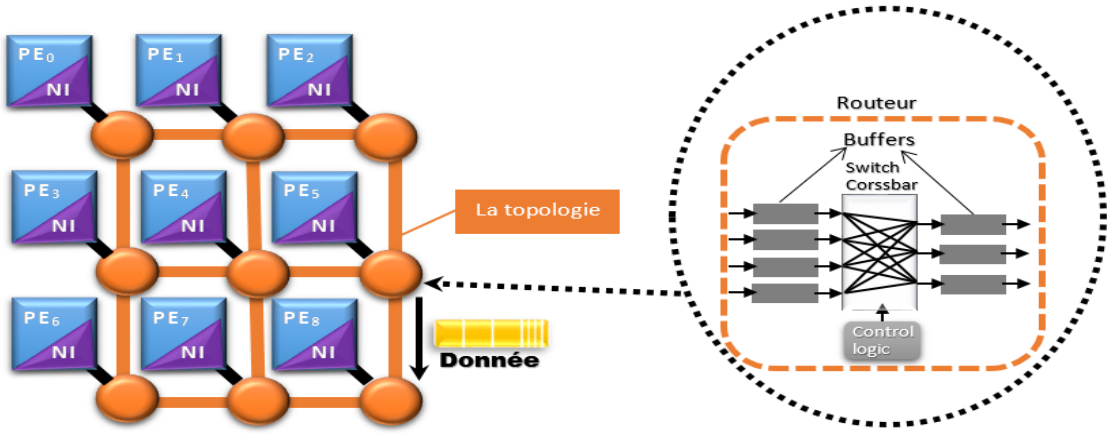


FIGURE I.1 – Exemple explicatif d'un réseau sur puce NoC.

cesseurs (Processing Element (PE)) sur la même **surface** de silicium, un réseau sur puce est une topologie de communication à l'instar d'un bus traditionnel. Cela étant dit, la tâche première d'un NoC (Figure I.1) est d'échanger des informations d'un point vers un autre point en offrant de meilleures performances que le bus, et ce non seulement au niveau de la bande passante, mais aussi du point de vue évolutif, tolérance aux pannes [53]. Un réseau sur puce est une fabrique de com-

munication multi-hop avec un système de commutation de paquet, dans la plupart des cas intégrée sur la puce [54, 55]. Les composants sont connectés au réseau à travers des connexions point-à-point à l'aide d'une interface de réseau (Network Interface (NI)). Des standards et des protocoles pour les NIs ont été proposés pour simplifier la création de systèmes communicants sur la puce. **Le protocole** Scalable Coherent Interface (SCI) [56, 57] a été initialement prévu pour les bus. Open Core Protocol (OCP) [58] est un autre exemple **des protocoles fréquement utilisés.Si on veut citer des exemples concrets pour les système à base MPSoC,** plusieurs fabriques d'interconnexion existent déjà : Hermes [59], Xpipes [60], SonicsMX 6, Nostrum [61], SPIN [62], et Spidergon [63] **qui sont réalisés par un** système permettant de créer un réseau sur puce paramètre **appelé** NetMaker [64] a été élaboré sous forme de bibliothèque de composants. Sur le marché des produits micro-électroniques, il est indispensable de proposer des produits innovants et performants dans les plus brefs délais pour exploiter la fenêtre de commercialisation. Le système crée doit respecter les spécifications attendues. Avec l'augmentation de la complexité des systèmes, de plus en plus **le temps consacré à la vérification est entraine d'augmenter considerablement** (jusqu'à 70% [65]). Dans le flot de conception matériel (Figure I.2), le passage d'un niveau au suivant

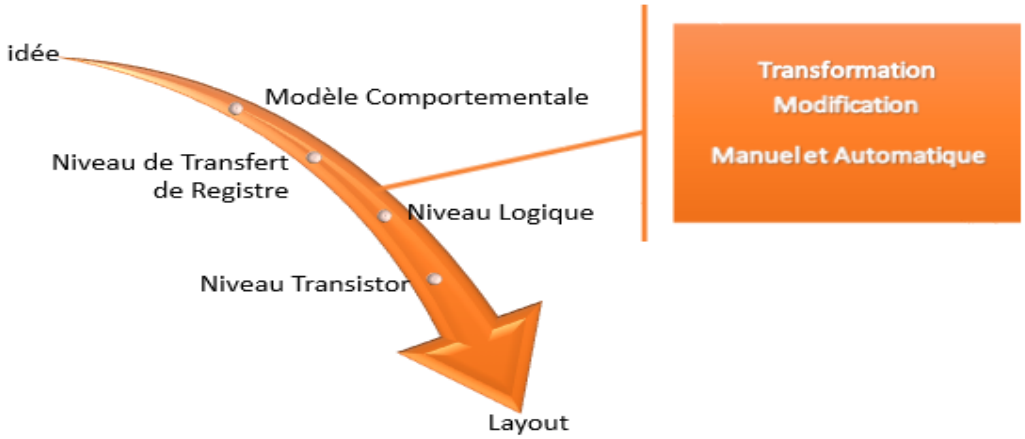


FIGURE I.2 – Flot de conception matériel.

est susceptible d'introduire des erreurs de conception. Même avec le support d'un large éventail d'outils, ce passage n'est pas toujours **établi sans problème et cela déclenche une phase de vérification vu qu'un SoC regroupe des blocs fonctionnels (compose d'un ensemble d'Intellectual Property (IP)) relies au moyen d'un système d'interconnexion, et de la même façon le système à base NoC est èlaboré ce que signifie qu'il pourra être une particularité des système MPSoC** [66]. Les IPs doivent avoir été vérifiés lors de leur conception. L'aspect crucial du point de vue de la validation reste donc celui du bon fonctionnement des communications. Cette thèse se place dans le contexte de la vérification des infrastructures de communication de type NoC.

1.1.3 Système Auto-organisé

Le problème majeur parmi toutes les définitions que l'on rencontre dans la littérature est souvent **de désigner le même terme à des propriétés qui sont** parfois très différentes, **à cet effet** on peut déduire que le concept d'auto-organisation caractérise la notion d'organisation interne d'un système par ses propres moyens et sans aucun contrôle extérieur.
La première apparition du terme d'auto-organisation a lieu dans les années 40 avec les travaux de *William R. Ashby*, psychiatre-ingénieur anglais [67, 68]. En effet, il a été introduit les premières définitions de l'auto-organisation et les termes relatifs associés. Dans ces travaux, il a été considéré que l'organisation d'un système présentait une dépendance fonctionnelle de ses états futurs par rapport aux états actuels et **les** entrées éventuelles du système. Nous pouvons y voir une analogie avec les systèmes numériques séquentiels par opposition aux systèmes combinatoires. Ainsi qu'un

système auto-organisé organise lui-même sa structure interne sans directives extérieures.
Une approche plus mathématique que celle proposée par *Ashby* a été présentée par *Lendaris* [69]. Cette approche est applicable pour tous les systèmes de traitement d'information possédant des entrées / sorties par lesquelles ces informations arrivent et sortent traitées du système.
Actuellement, les définitions récentes [70–72] du principe d'auto-organisation des systèmes rencontrées dans la littérature sont plus ou moins cohérentes. Les principales différences portent sur certaines caractéristiques de ce concept. D'une manière générale *On définit l'auto-organisation d'un système comme la capacité d'un système complexe et modulaire à se restructurer sans contrôle extérieur par l'interactivité des éléments le constituant dans l'objectif de s'adapter aux changements imprévus de son environnement ou d'optimiser son fonctionnement selon les critères préétablis*.

1.1.3.1 Modes de reconfiguration

On peut distinguer en fonction du nombre de reconfigurations (principalement lié au degré de couplage existant entre les éléments constituant le système reconfigurable), les systèmes statiques, dynamiques et ultra-dynamiques [73].

(a) **Mode Statique**
La grande majorité des architectures à base de FPGA utilise le mode de reconfiguration statique. Les FPGA sont configurés une seule fois pour exécuter un traitement unique sur une même série de données. Ces architectures restent actives tout au long des opérations (voir Figure I.3). Parmi les systèmes fonctionnant en reconfiguration statique nous trouvons principalement les accélérateurs reconfigurables dont le rôle est de réaliser des tâches gourmandes en temps de calcul nécessitant une exécution rapide et ne pouvant pas être effectuées sur les systèmes micro-programmés [74–77].

FIGURE I.3 – La reconfiguration statique d'un circuit FPGA.

(b) **Mode dynamique**
Dans le cas des systèmes reconfigurables dynamiquement, pour une même application le nombre de reconfiguration est au moins égal de deux au cours du traitement [78]. La mise en œuvre de cette stratégie de fonctionnement se traduit par la configuration du système reconfigurable évoluant tout au long de l'exécution des algorithmes et autant de fois que nécessaire. L'ensemble se comporte comme un processeur dont les ressources matérielles évoluent dans le temps à une fréquence liée à celle des données. La Figure I.4 illustre le procédé. Il existe plusieurs types de reconfiguration dynamique : reconfiguration totale ou partielle (locale et globale). Dans le procédé de la reconfiguration dynamique totale, la globalité du contenu de la mémoire de reconfiguration **reservée dans la carte** FPGA est remplacée par des nouvelles données de configuration. La figure I.5 illustre ce mode de reconfiguration sur un exemple de traitement avec quatre opérateurs A, B, C et D. La reconfiguration globale consiste alors à implanter les opérateurs (selon un ordonnancement opérateurs) en reconfigurant entièrement le système à chaque nouvelle étape. La reconfiguration globale est le mode de reconfiguration

FIGURE I.4 – La reconfiguration dynamique d'un circuit FPGA.

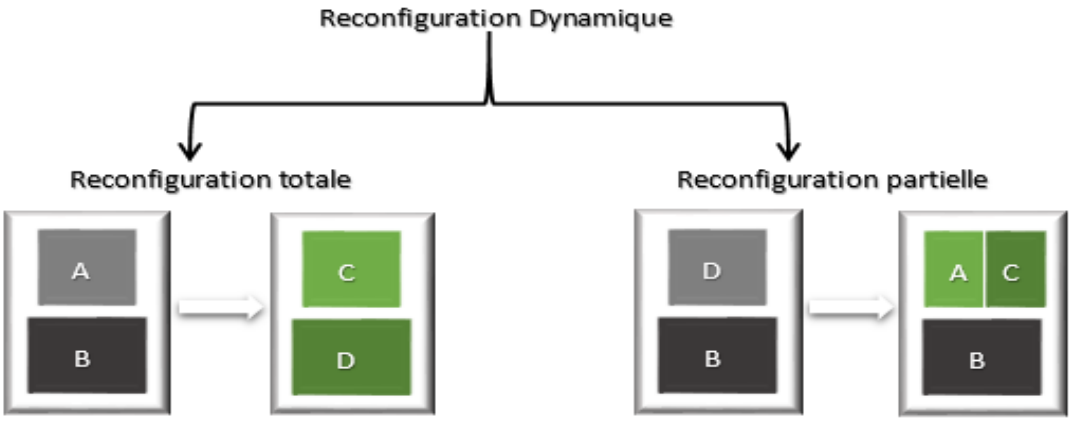


FIGURE I.5 – Le mode de reconfiguration dynamique d'un circuit FPGA.

dynamique le plus simple à réaliser. La reconfiguration dynamique partielle désigne la modification d'une partie des données de configuration du circuit reconfigurable [79]. En d'autres termes, une partie du circuit est reconfigurée pendant que le reste du circuit exécute des opérations. La figure I.5 illustre ce mode de reconfiguration sur un exemple de traitement. Ici seul l'opérateur D est reconfiguré en opérateurs A et C tandis que l'opérateur initial B est toujours en fonctionnement. Pendant la reconfiguration, la fonctionnalité du reste du circuit reste donc inchangée. La mise en œuvre du concept de la reconfiguration dynamique partielle nécessite des technologies FPGA permettant ces formes de reconfigurations. Parmi les systèmes fonctionnant en reconfiguration dynamique, nous trouvons principalement les co-processeurs reconfigurables ayant pour but d'accélérer des tâches spécifiques pour lesquelles un processeur n'est pas performant et adapté [80–82].

(c) **Mode Ultra-dynamique**

La reconfiguration ultra-dynamique d'un système reconfigurable à base de FPGA présente un type de reconfiguration dynamique prenant en compte **le** temps de reconfiguration du circuit. Ce type de reconfiguration dynamique s'utilise notamment dans les applications où la notion du temps réel est fondamentale. Afin d'utiliser le procédé de la reconfiguration ultra-dynamique, les temps perdus pendant les phases de reconfiguration doivent être réduits. Plusieurs solutions ont été proposées mettant en œuvre le masquage des temps de reconfiguration [83, 84]. Parmi les systèmes qui aspirent à évoluer vers des systèmes ultra-dynamiques, on trouve les unités reconfigurables. Dans ce cas, la partie reconfigurable est directement placée en tant qu'une unité particulière sur le chemin de données de l'unité de calcul et est généralement intégrée sur la même puce [85].

1.2 L'état de l'art des NoCs réalisés

Dans cette section, on doit présenter les différents projets réalisés pour les NoCs suivant deux grandes classes en raison de faire une classification des NoC en fonction des changements possibles de leurs paramètres au cours de fonctionnement peut être établie comme soit des *NoC statiques* ou des *NoC dynamiques (reconfigurable)*. Les réseaux sur puce dont les topologies, le placement d'éléments de calcul PE, les fonctions de routage et d'aiguillage et les routeurs sont fixes et inchangeables dans le temps sont considérés comme statiques. Dans cette classe de NoC nous trouvons les architectures Dans le cas où, les réseaux sur puce dont les paramètres mentionnés ci-dessous peuvent changer au cours du fonctionnement sont considérés comme réseaux dynamiques.

1.2.1 Les réseaux sur puce Statiques

Les travaux sont classés selon leurs caractéristiques en trois types suivants :

— *les réseaux de meilleur-effort* qui sont des réseaux conçus dans le but d'avoir des bonnes performances moyennes. Tous les objets communicants dans le réseau se partagent la capacité de trafic du réseau, aucune priorité est accordée,

— *les réseaux avec qualité de service* des réseaux qui intègrent des services garantis permettent de réserver des ressources de communication pour différents flux afin de garantir leurs débits et/ou leurs latences. Ce type de réseau est très favorable pour les applications temps-réel telles que le traitement de la vidéo et de la voix,

— *les réseaux asynchrones* qui sont des réseaux conçus et implémentés en utilisant les méthodologies de conception asynchrones. Il n'existe donc pas d'horloge globale dans ces réseaux. L'avantage de ce type de réseau est d'éviter des problématiques concernant à l'horloge dans la conception des SoC complexes.

1.2.1.1 Réseaux sur puce de Meilleur-effort

Dans cette classe, on présente deux architectures de réseaux sur puce typiques, le réseau SPIN [**?**,86] et le réseau OCTAGON.

— **OCTAGON**
Le réseau OCTAGON (voir Figure I.6) a été proposé par STMicroelectronics et l'Université de Californie à San Diego en 2001 [87]. La topologie du réseau est un anneau octogonal avec des liens supplémentaires entre routeurs diamétralement opposés. Le mécanisme de commutation peut être circuit ou paquet suivant les choix de l'arbitre de connexion. Avec la commutation de paquets avec un algorithme de routage distribué et adaptatif Plus de détails sur cette architecture et les comparaisons de cette architecture avec le modèle crossbar sur plusieurs caractéristiques (débit, probabilité de perte de paquets, la capacité de mettre à l'échelle de l'architecture) sont présentés dans [88].

1.2.1.2 Réseaux sur puce avec Qualité de Service

Pour montrer des architectures de type « Qualité de Service », on peut présenter deux architectures de NoC typiques : QNOC(Figure I.7) et ÆTHEREAL [89, 90].

— **QNOC**
L'architecture est proposée par l'Institut Technologie du Technion [91]. La topologie du réseau est une maille à deux dimensions, elle peut être irrégulière (voir Figure I.7(a)). L'algorithme de routage est de type distribué en fonction des positions de l'émetteur et du récepteur de manière à ce que le chemin soit le même lors d'échanges bidirectionnels. Le contrôle de flux de message est basé sur un mécanisme de crédit. Afin d'implémenter la QoS, le trafic est

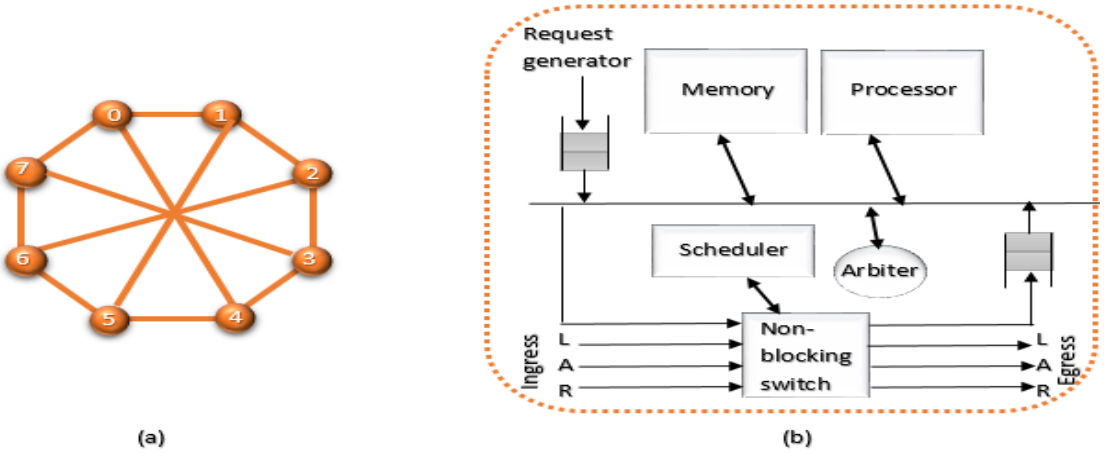


FIGURE I.6 – Le Réseau OCTAGON : (a) Topologie du réseau ; (b) Architecture du routeur.

classé en quatre niveaux de services correspondant à des types de messages et des contraintes de latence et de débit souhaitées :

— *Signaling* pour les messages très courts avec la priorité la plus élevée, qui nécessitent une faible latence (par exemple, les messages de contrôle ou les messages d'interruption).

— *Real-Time* garantit la bande passante et la latence pour les applications temps-réel (audio, vidéo, etc.).

— *Read/Write* est utilisé pour gérer des échanges de données tels qu'ils se font habituellement sur un bus.

— *Block-Transfer* est employé pour transférer les messages longs et les gros blocs de données.

Pour implémenter ces quatre niveaux de service, le routeur inclut également quatre buffers séparés pour chaque niveau de service aux entrées et aux sorties, La taille des buffers d'entrées est variable tandis que la taille des buffers de sortie est d'une position.

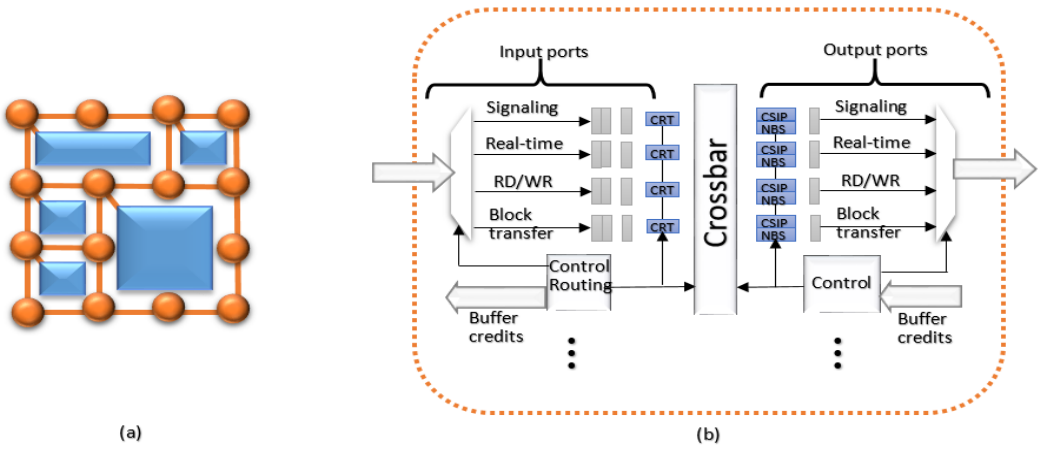


FIGURE I.7 – Le Réseau QNOC : (a) Topologie du réseau ; (b) Architecture du routeur.

1.2.1.3 Réseaux sur puce asynchrones

Les architectures de réseaux sur puce asynchrones ont été proposées pour améliorer la propagation d'horloge. Dans cette sous-section, on **peut citer comme des** exemples concrets d'architectures de NoC asynchrones : NEXUS [92] et QNOC asynchrone **et ce dernier est détaillé par la suite.**

— **Asynchronous QNOC**
L'Institut Technologie du Technion à proposé une implémentation asynchrone de [93]. La topologie du réseau est aussi une maille à deux dimensions et l'algorithme de routage est aussi de type source, déterministe. Le routeur asynchrone a été comparé avec le routeur synchrone avec la même fonctionnalité, et la même bibliothèque de cellules. Donc, la surface de routeur asynchrone est inférieure à la surface du routeur synchrone tandis la performance est supérieure. Par exemple, un routeur asynchrone avec 4 niveaux de services occupe une surface $0,47mm^2$ et il a un débit maximal de 75,2 M flits/s tandis qu'un routeur synchrone avec le même nombre de services occupe $0,96mm^2$ et possède un débit maximal de 67,6 M flits/s.

1.2.2 Les réseaux sur puce reconfigurables

Les NoCs dynamiques sont souvent confondus et associés avec les NoC reconfigurables. Par la suite, nous donnons un résumé des travaux existants dans la littérature sur les NoC reconfigurables. La plupart des approches NoC reconfigurables présentées dans la littérature définissent la reconfigurabilité des NoC comme un moyen de modifier leurs configurations au cours de leur fonctionnement. Ces configurations de réseau correspondent des changements de paramètres de réseau tels que l'algorithme de routage, les techniques d'aiguillage, la Qualité de Service - QoS, adaptés à une application donnée et pouvant être changés au cours du fonctionnement du réseau. Cependant, dans le cadre de la conception de système reconfigurable à base de technologie FPGA, une architecture reconfigurable est considérée comme une architecture permettant une modification de sa structure architecturale ou de traitement de données à tout moment. Dans ce qui suit on donne des exemples des NoC reconfigurable liées aux changement des paramètres (algorithme de routage [94], topologie [95]), destinés MPSoC (ReNoC [96], NOVA [97]) ou basés sur FPGA (CoNoChi [98], DyNoC [99, 100] et CuNoC [101, 102]) . Dans ces approches, pour une application donnée, soit les PE sont reconfigurés en d'autres PE, soit on reconfigure le réseau NoC pour l'adapter à l'ensemble de PE utilisés pour l'application considérée(Figure I.8).

FIGURE I.8 – Illustration de deux couches distinctes (NoC et PE) dans les approches NoC reconfigurables.

1.2.2.1 CUNoC

Ce type de réseaux sur puce est destiné à la conception de systèmes sur puce reconfigurables à base de FPGA [102–104]. Cette architecture réseau nommé CuNoC(voir Figure I.9) permet la communication entre des modules placés dynamiquement sur puce au cours de leur fonctionnement. Il correspond à un réseau sur puce de type **commutateur par paquet** « packet switching » composé de routeurs intelligents appelés Communication Unit (CU) (Unité de communication).

Le rôle principal de ces routeurs est le routage des paquets, à partir de PE sources vers des PE destinations selon des adresses contenues dans les paquets de données à transmettre, au sein du réseau dont la structure peut évoluer au cours du temps. Un CU est caractérisé à la fois par une technique d'arbitrage basée sur un principe de priorité à droite, par un nouvel algorithme de routage, par sa spécificité architecturale et la manière dont il est connecté avec les éléments de calcul PE.

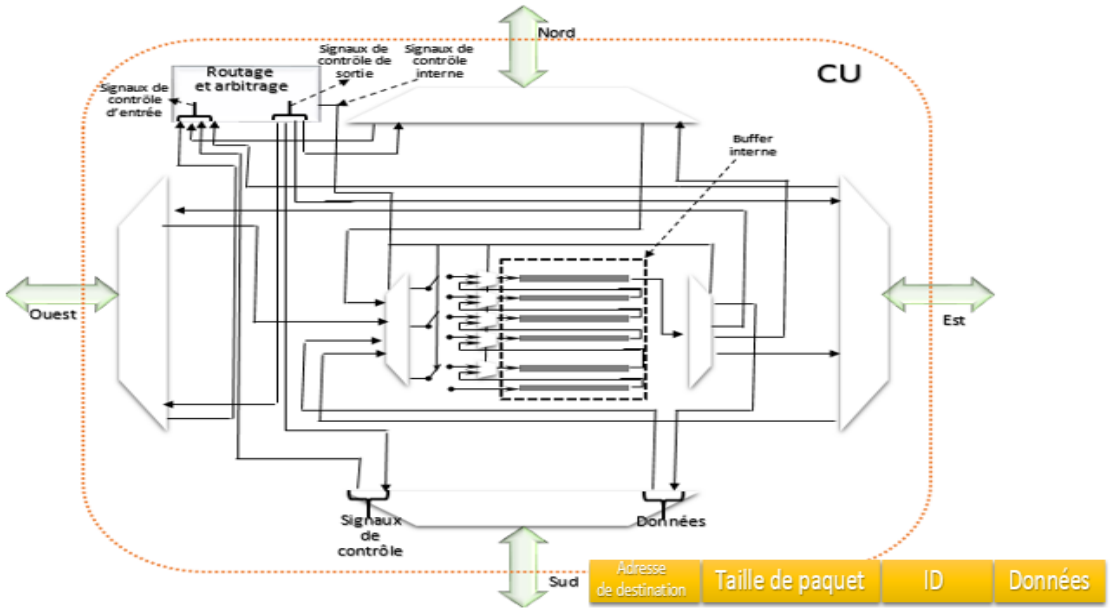


FIGURE I.9 – L'architecture de l'unité de communication CU et Le format d'un paquet dans le CuNoC.

1.3 La validation des systèmes à base NoC

Avec l'augmentation de la complexité des réseaux sur puce NoC, il existe un besoin croissant de méthodes d'aide à la conception qui opèrent à des niveaux d'abstraction élevés. Ces systèmes enfouis nécessitent des recherches de compromis **pour les facteurs : *performances/surface de silicium* et */consommation d'énergie*** de plus en plus précis avec une pression croissante sur le temps de conception (time-to-market). Les critères sont liés à l'ensemble des ressources matérielles et logicielles composés des systèmes à base NoC. Plusieurs approches ont apparu pour assurer et être à l'échelle les complexités liées au ces systèmes, ces approches peuvent être informelles, formelles ou bien hybrides. On intéresse de détailler les approches fonctionnelles et formelles et leur utilisation pendant le développement des réseaux sur puce

1.3.1 Les critères matériels des NoCs

Les réseaux sur puce sont des systèmes composés d'unités de traitement, de contrôle, de mémorisation et de communication. Cependant les constructeurs à considérer **durant leur développement ces derniers composants comme une cible de validation et surtout** pour chacune de ces unités surtout ces cœurs de processeurs disponibles parmi les composant à valider fonctionnellement celles présentés dans cette suite.

1.3.1.1 Surface silicium

Maximiser les rapports *performances/surface de silicium* et *performances/consommation d'énergie*, par exemple système a pipeline de 3 à 6 étages pour les processeurs Reduced Instruction Set Computing (RISC) de la famille ARM(Des architectures externes de type RISC développées par

ARM Ltd depuis 1990) [105], des fréquences d'horloge et des surfaces de silicium environ dix fois plus faibles que celles des super scalaires hautes performances.

1.3.1.2 Mémoire de stockage

Des solutions exigent de synthétiser une fonction matérielle **pour résoudre** le problème de l'allocation des données dans la structure mémoire du système NoC **mais cette réservation de memoire est** largement inférieur aux besoin,**l'objectif de validation est donc fixé** afin de réduire les pénalités des temps d'accès à la mémoire externe et les surcoûts en consommation d'énergie [106].

1.3.1.3 Topologie de communication des IPs

Pratant vers l'objectif d'améliorer le facteur *performances/délais* des IP de communications avec la même rapidité [107] par le regroupement sur un même bus**de façon que** toutes les unités ayant des caractéristiques de performances **identiques**. Cependant, la complexité de la conception de l'interconnexion reste **fortement limitée** par des modifications matérielles ultérieures. **Par conséquence** des architectures **basé sur** des réseaux sur puce sont proposées [107] dans le but de faciliter le travail de conception en utilisant une approche plate-forme et ainsi favoriser la réutilisation.

1.3.1.4 La base matérielle FPGA

La **notion** d'un NoC reconfigurable **peut être envisagé par un exemple sur** le problème identifié sous le terme radio logicielle [108] **où** les propriétés de reconfiguration dynamique de circuit FPGA sont appropriées à la conception de systèmes NoC auto-reconfigurable (ou simplement SONoC). Le choix des technologies reconfigurables(voir Figure I.10) est d'apporter une solution intermédiaire entre des circuits spécifiques comme : Application-Specific Integrated Circuit (ASIC) (degré d'intégration et de calcule toujours maximale, degré de flexibilité dépends de temps de développement et taux de calcule), Digital Signal Processor (DSP)(puissance de calcul supérieur aux processeurs Microprocessing Unit (MPU), et degré de flexibilité supérieur à celui des ASIC), FPGA programmables (développements et prototypage rapides par rapport à ceux spécifiques au circuits ASIC, comparables à ceux du MPU micro-programmé). Les systèmes reconfigurables (Figure I.11) les

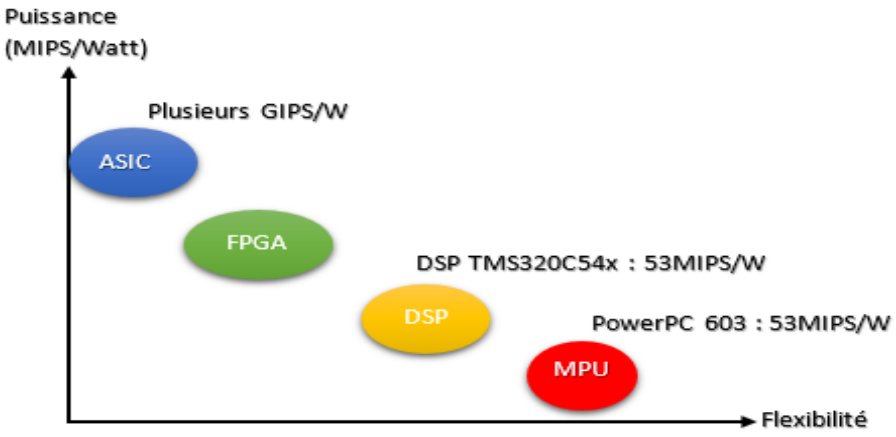


FIGURE I.10 – Rapport performances / flexibilité pour les principales technologies.

plus souvent utilisés sont les systèmes à base de FPGA (Field Programmable Gate Array : réseau prédiffusé programmable par l'utilisateur. Réseau prédiffusé de portes logiques qui est destiné à être programmé sur place, avant d'être utilisé pour une fonction particulière). Les systèmes reconfigurables à base de FPGA sont principalement réalisés par des combinaisons de couplage entre des matrices de FPGA et des processeurs. Un circuit FPGA (selon [109]) est généralement composé

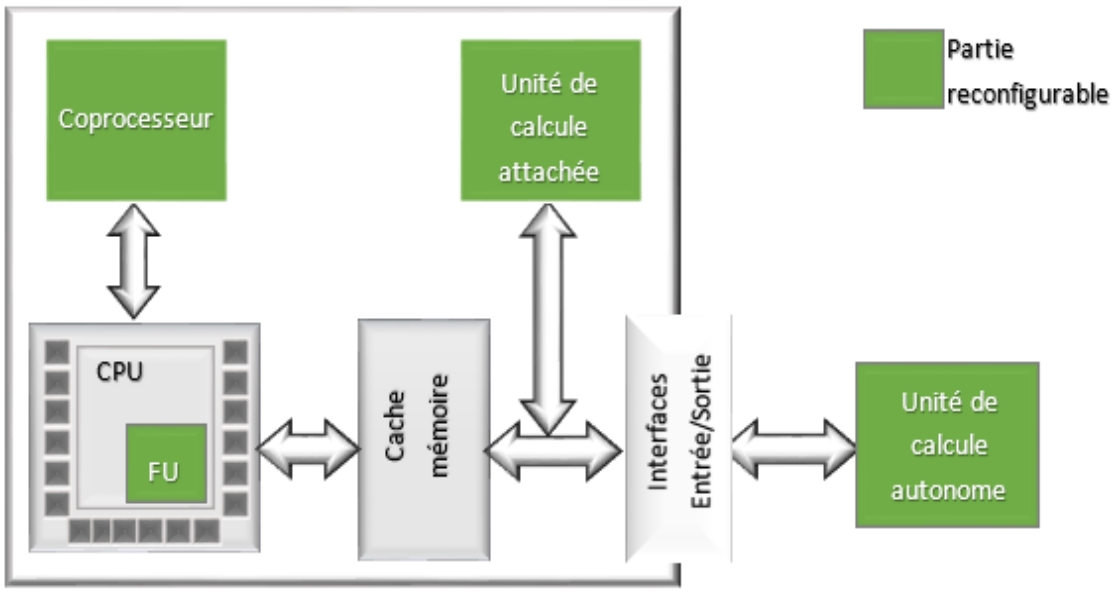


FIGURE I.11 – Le système reconfigurable.

de réseaux d'éléments de calcul souvent appelés « blocs logiques »(logic blocks) dont la fonctionnalité est déterminée par la programmation de bits de configuration (Figure I.12(a)). Ces unités de calcul sont connectées à travers des canaux de routage configurables (Figure I.12(b)). La structure

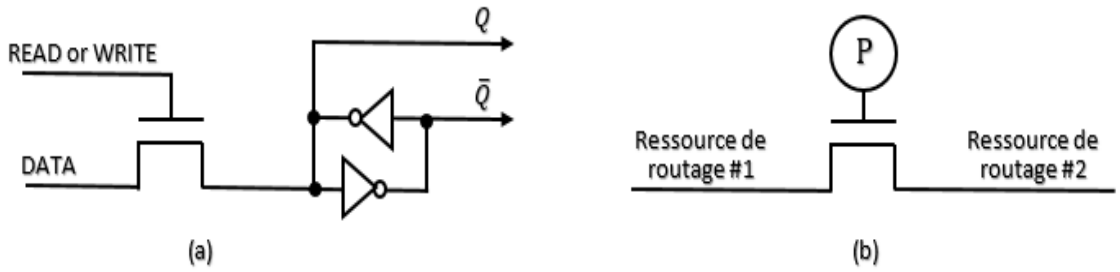


FIGURE I.12 – (a) Bit de programmation d'un FPGA à mémoire statique SRAM (b) Connexion de routage programmable.

classique d'un FPGA à mémoire statique (Static Random Access Memory (SRAM)) est présentée sur la figure I.13 [109]. La plupart des architectures FPGA sont organisées de telle sorte que la communication entre les blocs logiques le long des lignes et des colonnes de la logique de routage soit rapide et efficace. Une organisation en une matrice de blocs logiques distincts, reliés par des ressources d'interconnexions, constitue la solution la plus générale. Un bloc ou une cellule logique abrite une ou plusieurs fonctions programmables réalisées par des tables de transcodage (Look Up Table (LUT)) et des éléments de mémorisation (registres). Suivant les technologies proposées, la granularité de ces cellules logiques varie entre des LUT à 2, à 4 ou à 5 entrées [110] et des blocs de calcul intégrés plus ou moins complexes. En fonction de la taille et de la complexité des blocs logiques leur structure est classée en fonction de leur granularité fine, moyenne et grosse (FPGA, Systolic Ring [111], etc.). Plusieurs familles de circuits FPGA sont commercialisées (Xilinx, Altera, Atmel, etc.) [109, 112, 113]. Elles se distinguent les unes des autres par leur mode de fonctionnement (fonctionnalités des blocs logiques), leur architecture interne, leurs ressources de routage, etc.

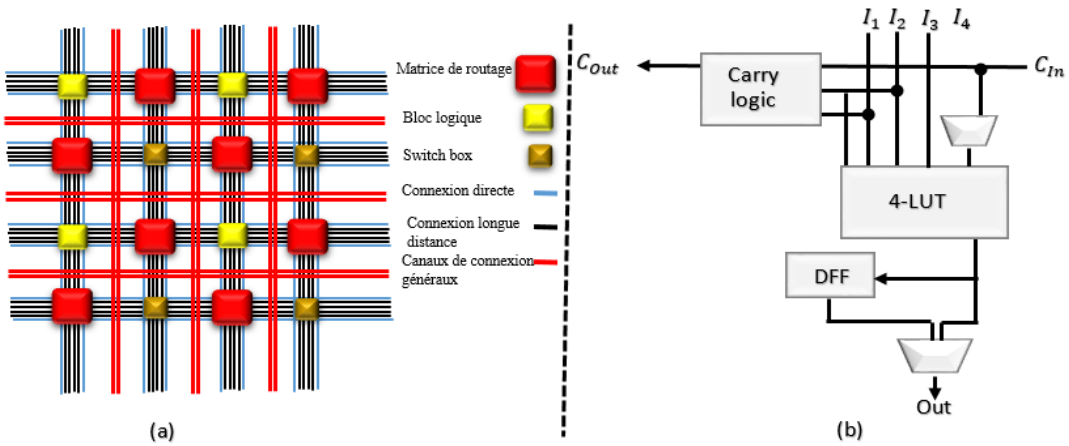


FIGURE I.13 – (a) Architecture globale d'un circuit FPGA, (b) d'un bloc logique.

1.3.2 Critères logiciels des NoCs

Le concepteur d'un NoC est confronté à de multiple choix pour architecturer son système tout en optimisant les critères suivants : performances, coûts, consommation, time-to-market. Le développement de ce genre des systèmes embarqués possède aussi des caractéristiques et des besoins qui les distinguent particulièrement des autres systèmes. Significativement, ce type de système comme toutes les systèmes embarqués diffèrent des autres systèmes dans plusieurs aspects d'après [114], en exigeant :

- Un haut degré d'intégration des composants software et hardware.
- Un besoin temps-réel rigide.
- Un comportement très complexe, caractérisé par de nombreux scénarios compliqués.
- Un besoin en fiabilité et sécurité résistant, surtout pour les systèmes avec des missions critiques (safety-critical and/or mission-critical).
- Un traitement parallèle et souvent distribué.

Egalement, le développement de tels systèmes inclus l'utilisation de :

- Méthodes formelles pour la spécification, conception, vérification et test.
- Outils et méthodes de communication, synchronisation, etc.
- Co-conception (co-design) hardware/software.
- Langages de programmation spécialisés.
- Outils (des environnements) de développement spécialisés.
- Contraintes d'optimisation.

De point de vue de travaux réalises, des études consacrées aux NoCs sont majoritairement concentrées sur les performances [115–117], la latence [118], la bande passante [119], l'estimation de consommation [120], la détection et la correction d'erreurs [121,122], et la surface utilisée. D'autres proposent des méthodes de routage tendant à éviter des problèmes de deadlock [123–125] en visant la caractérisation du trafic [126].

1.3.3 La validation informelle des systèmes à base NoC

Les coûts élevés des masques d'un circuit de type NoC renforcent le problème de la vérification pour éviter les erreurs de conception. Par conséquent, il est nécessaire de tendre vers des approches classique et autres moderne pendant la conception et l'optimisation multicritères pour ce type de système complexes.

1.3.3.1 Validation informelle Classique

À la différence du développement et de la conception d'une application logicielle sur une plateforme standard, la conception d'un système embarqué **est faite pour** que le software et le hardware **d'un système** soient conçus en parallèle. En effet, le développement et la conception sont réalisés **(comme il est montré dans la figure I.14)** en décomposant et en assignant le système embarqué en software(logiciel) et hardware(matériel). **Ensuite,** une conception séparée des composants software et hardware **est déclenchée**, et finalement **on passe vers l'étape de l'intégration**. Dans un flot classique de conception d'un NoC et selon [127,128], l'application cible est dans un premier temps décrite fonctionnellement (Figure I.14). Un processus de conception est alors appliqué à chaque bloc identifié par le concepteur, ce qui conduit à un modèle de réalisation du bloc. Chaque modèle de réalisation est validé individuellement en y apportant des corrections locales au bloc. Une fois chaque bloc validé, l'ensemble des blocs est assemblé ou intégré pour former le système complet.

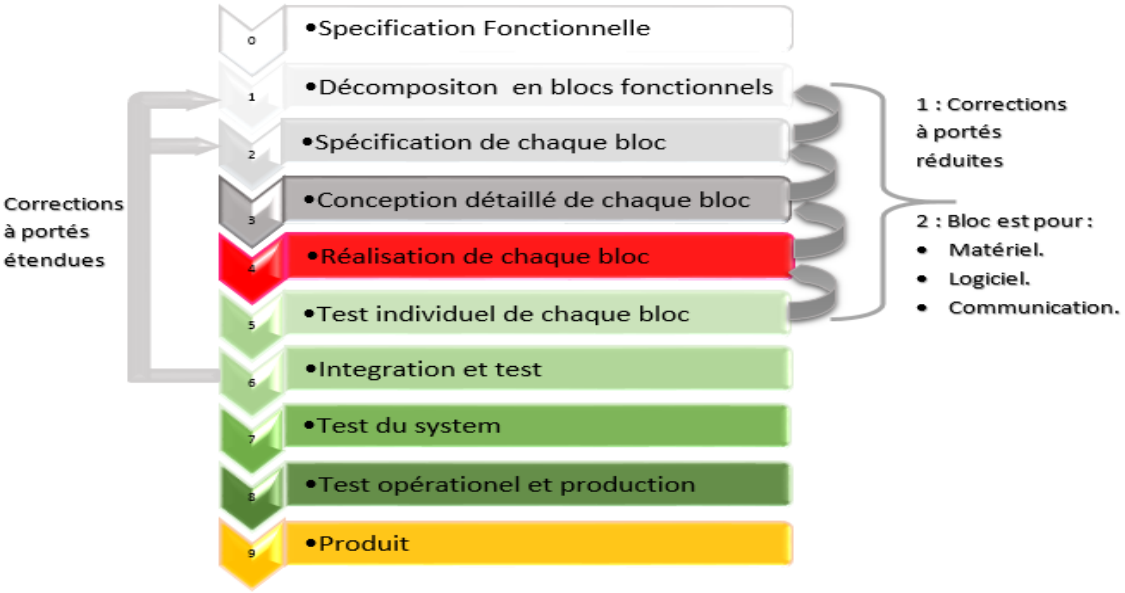


FIGURE I.14 – Cycle de flot classique.

Les erreurs identifiées à cette étape d'intégration ont en général des portées du fait qu'elles peuvent remettre en cause les spécifications de plusieurs blocs : par exemple, des incompatibilités temporelles entre les signaux utilisés dans les échanges de données entre blocs. Il est alors nécessaire de reprendre le processus de conception des blocs concernés et ce tant que l'intégration globale des blocs n'est pas jugée correcte. Il peut être nécessaire d'itérer plusieurs fois avant d'arriver à une solution fonctionnellement correcte ce qui allonge d'autant les temps de conception. Par ailleurs, les objectifs de performances, de surface de silicium et de consommation sont directement dépendants de la décomposition en blocs de la spécification et des choix de réalisation de ces blocs. Si ces objectifs ne sont pas atteints par des optimisations locales dans les blocs il peut être nécessaire de remettre en cause la décomposition et/ou les choix de réalisation. Ces changements ont en général des conséquences importantes sur les temps de conception et peuvent même conduire à l'abandon du projet. Ces difficultés ont principalement pour origine le fait que la vérification du système complet intervient tardivement dans le processus de conception. La vérification d'un système complexe, composé d'unités hétérogènes pose des difficultés de vérification. Dans la pratique, celle-ci est abordée principalement par simulation ou co-simulation [129].

1.3.3.2 Validation informelle moderne

Les systèmes embarqués ont généralement besoin des spécificités qui orientent le concepteur à développer un système complet avec une vue globale et unifiée pour une architecture matérielle composée de ressources (processeurs, mémoires, bus de communication, etc.) et une partie applicative exprimant le code des tâches du système dans un langage de programmation. La complexité du processus de conception de tels systèmes est déterminée par la sémantique et la syntaxe du langage de conception niveau système System Level Design Language (SLDL) adoptée pour véhiculer l'implémentation et exige en générale de supporter la modélisation du comportement de système à tous les niveaux d'abstraction et que les modèles doivent être exécutables et simulables, de sorte que les fonctionnalités et les contraintes peuvent être validées. Les approches de SLDL peuvent être classifiées [130] dans quatre groupes : orientée langages, orientée plateforme, orientée modèle, orientée architecture.

(a) **Orientée langages**
Dans cette approche, la conception commence par décrire le comportement du système dans un langage haut niveau comme C, C++, SDL. Cette description est ensuite transformée vers une description structurelle sous un format qualifié par transferts de registres

. Le niveau Register Transfer Level (RTL) utilise des langages de description de matériels (Hardware Description Langages (HDL)) comme VHDL, Verilog ou HardwareC [131]. Ces langages de description intègrent la capacité d'exprimer le temps, par une ou plusieurs horloges, et la notion de concurrence matérielle. Ils sont différents des langages de description comportementale en combinant la construction d'une architecture pour les calculs, les communications et le stockage du code et des données et la compilation de cette architecture, où la compilation sous-entend la synthèse d'un matériel dédié [131,132] en terme l'accès convenable aux caractéristiques particulières du matériel cible, les particularités de l'architecture cible. Les langages de conception niveau système ou SLDL les plus utilisés dans l'ingénierie des systèmes embarqués sont :

- *SystemC :* destiné pour modéliser, analyser et simuler un système embarqué lié aux outils commercialisés tels que Synopsys.
- *SpecC :* c'est une notation formelle inclue la hiérarchie comportementale et structurale, la concurrence, la communication, la synchronisation, les transitions d'états et les manipulations d'exception,
- *Esterel :* un langage peut s'appuyer sur une sémantique formelle [133]).Cette approche **peut avoir** (Figure I.15) **un premier inconvénient stipule que les optimisations effectuées localement ne doit pas évaluer leur impact sur les caractéristiques du système global**. Un autre inconvénient est relatif à la rupture sémantique entre représentations internes.

Après **une suite des** raffinements et transformations on obtient pour chaque sous-système une représentation interne qui possède sa propre sémantique (Figure I.16). La validation du système complet peut alors révéler des modifications nécessaires dans l'ensemble des sous-systèmes. Cette approche allonger les temps de conception et limiter l'évaluation dans le processus de conception basé sur la sémantique d'accueil utilisée dans la co-simulation du fait de l'absence d'une sémantique précise permettant de décrire des communications et synchronisations entre les descriptions des sous-systèmes.

(b) **Orientée plateforme**
Le comportement dans cette approche du système est mappé sur une architecture prédéfini du système, au lieu d'être généré à partir du comportement comme dans l'approche de synthèse niveau système. Cette approche est adoptée, dans l'industrie des systèmes embarqués **prennant l'exemple du plateforme fournit** par Sun Microsystems et son langage Java. Sun Microsystems propose l'environnement de spécification Java pour les systèmes embarqués,

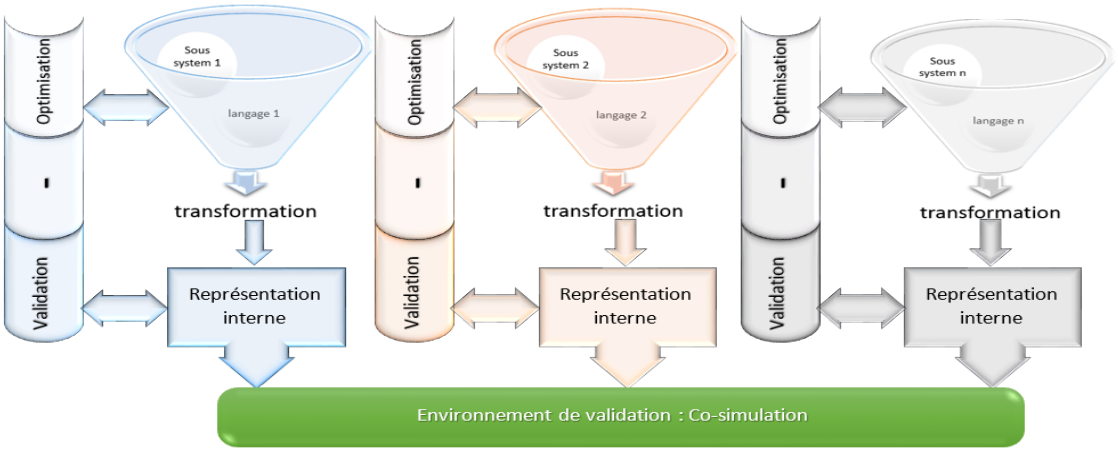


FIGURE I.15 – Conception système suivant une approche orientée langage.

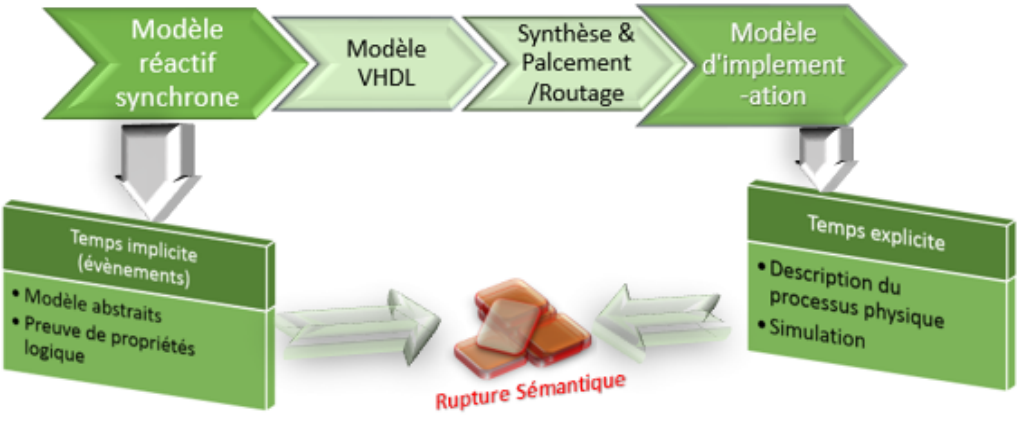


FIGURE I.16 – Exemple de rupture sémantique dans le processus de raffinement.

où la machine Java Virtual Machine (JVM) est adaptée aux besoins de chaque plateforme par le développeur de systèmes [134].

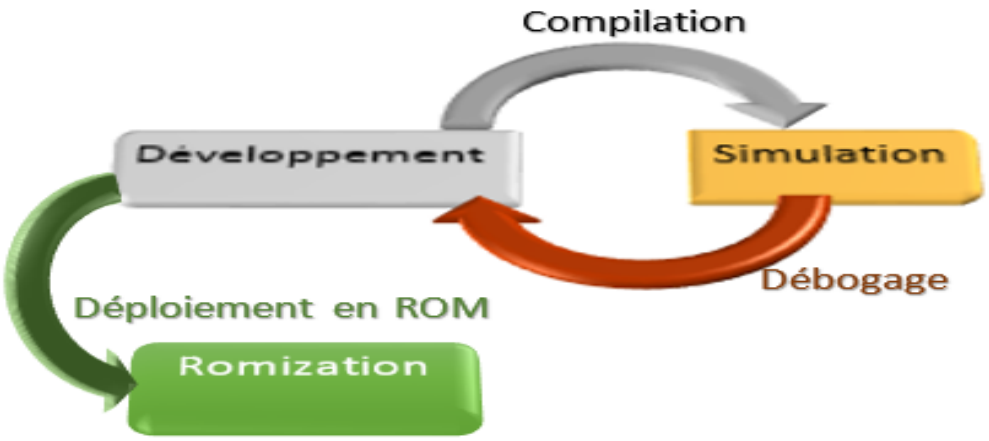


FIGURE I.17 – Cycle de vie des logiciels embarqués dans l'approche orientée plateforme.

Ce processus de développement doit être suivi, comme le montre la figure I.17, par plusieurs phases de compilation, simulation, débogage avant d'être déployé en ROM (ou Romization) [135]. La compilation Java en code natif par des compilateurs Java est faite directement pour la machine cible, comme par exemple GNU Compiler for Java (GCJ). Pour la phase de simulation, les concepteurs du hardware peuvent utiliser des outils de simulation, pour exécuter probablement le code benchmark sur l'ordinateur single-board utilisant le microprocesseur cible. Ces ordinateurs single-board évaluent la performance du microprocesseur en lui faisant subir plusieurs tests.**Cependant, le temps consacré pour les tests reste toujours influencer sur le facteur *temps de réalisation / qualité des produits* et la réalisation des tests est dépendant des caractéristiques de système à simuler.**

(c) **Orientée modèle**

Une autre approche dite orientée modèle consiste à décrire chaque sous-système suivant un modèle de calcul ayant une sémantique clairement définie et utiliser un environnement d'accueil de ces modèles de calcul qui détermine des règles sémantiques d'interactions entre les types de composants de chacun des modèles [136]. Les transformations ou les migrations de fonctionnalités s'opèrent alors dans cet environnement ce qui facilite la vérification et limite le travail de conception et d'interfaçage pour obtenir un modèle global de simulation (figure I.18).

Dans le cas de développement des systèmes embarqués, le profil Unified Modeling Language (UML) pour la modélisation et l'analyse des systèmes embarqués temps réel Modeling and Analysis of Real Time and Embedded systems (MARTE) [137, 138], son profil est structuré autour d'un noyau nommé Time, Concurrency and Resources Modeling (TCRM), décrivant les propriétés non-fonctionnelles NFPs (ou Non Functional Property (NFP)) [139] et raffiné pour donner deux sous-profils Real-Time and Embedded Modeling (RTEM) et Generic Quantitative Analysis Modeling (GQAM). Cette extension concerne la modélisation et l'analyse des systèmes embarqués temps réel en tenant compte de leurs propriétés fonctionnelles, non-fonctionnelles (la puissance d'énergie ou la capacité de la mémoire), l'analyse de l'ordonnancement (contraintes temporelles pour un ensemble de tâches logicielles), et l'analyse de performance (la détermination du système avec un comportement non-déterministe). Un autre exemple décrit par une présentation de modèles de calcul, encore appelés domaines, pour la description et la validation de systèmes est disponible dans [140]. Sans environnement d'accueil, la vérification inter-domaines nécessite de développer des interfaces entre les modèles [136] décrits dans les domaines concernés pour expliciter les communications et syn-

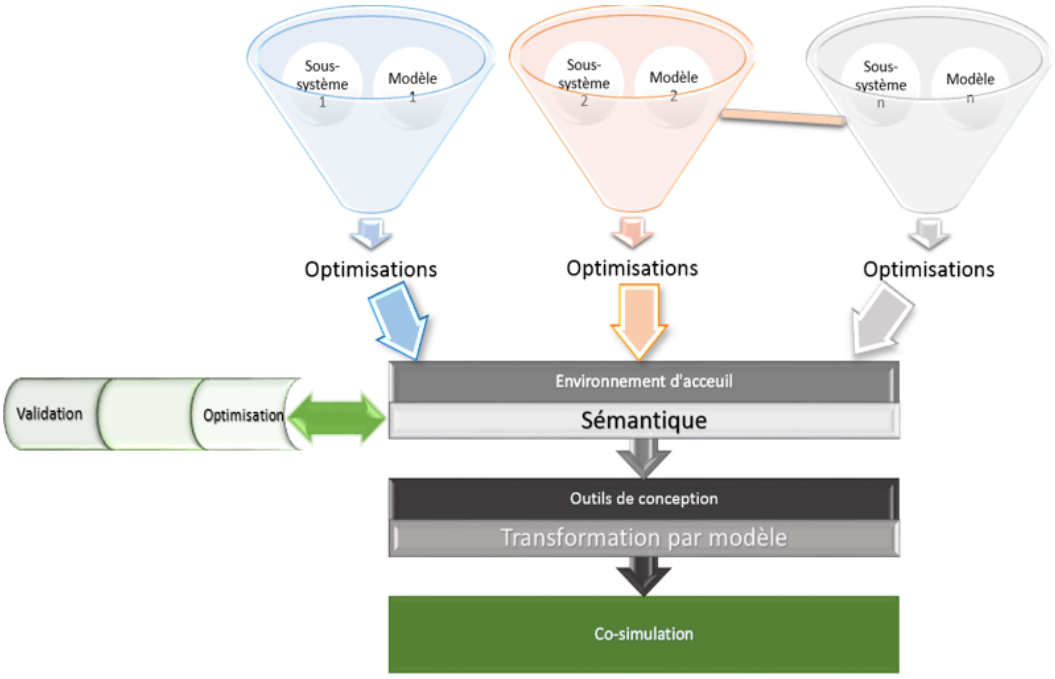


FIGURE I.18 – Conception système suivant une approche orientée modèle.

chronisations (Ptolemy développé à l'Université de Berkeley [141] et le système Polis [142]) pour faire face à la complexité et à l'hétérogénéité croissantes des systèmes sur puce.

(d) **Orientée architecture**
L'approche basée architecture ou l'adoption de la description architecturale logicielle d'un système définit son organisation, à un niveau élevé d'abstraction, comme une collection de composants, de connecteurs, et de contraintes d'interactions avec leurs propriétés additionnelles définissant le comportement prévu. Les méthodes formelles s'imposent pour appréhender la complexité et la fiabilité de ces systèmes [143]. Elles entraînent le développement de techniques d'analyse de systèmes et de génération automatique de tests à partir de leurs spécifications, ce qui réduit à la fois le temps et les coûts de validation, et augmente la fiabilité du système en utilisant un formalisme capable de décrire les composants, les interactions entre les composants, et de spécifier des propriétés non fonctionnelles concernant la configuration, le déploiement … etc.
Les langages de description d'architectures (ou ADL) offrent une manière formelle de spécifier les architectures logicielles des systèmes en décrivant les interfaces fonctionnelles des composants (flots de contrôle et de données) et les aspects non fonctionnels (aspects temporels, sûreté, fiabilité) et vérifier ces propriétés. Ces langages sont classés dans la catégorie des ADLs formels abstraits (sans entrer dans la nature de système) et des ADLs concrets (système comme une entité du monde réel logiciel et matériel et la génération automatique du système). Comme exemples d'ADL de premier classe, on peut citer : Wright [144], Rapid [145], et ACME [146]. Parmi les ADLs concrets, on trouve UNICON [147], MetaH [148] et par la suite AADL [149].

1.3.3.3 Critiques de la validation informelle

La simulation concernée par le de développement classique repose en général sur l'animation d'un modèle à événements discrets du composant. Un composant décrit en VHDL peut être simulé suivant ce principe. Le problème se complique lorsqu'il s'agit de simuler un système constitué de plusieurs composants logiciels et matériels qui communiquent. Cela nécessite de faire coopérer

plusieurs de modèles de simulation [150] associés aux composants contenus dans le système. Par exemple, les outils commerciaux comme N2C de CoWare connectent un simulateur du jeu d'instruction d'un processeur (simulateur Instruction Set Simulator (ISS)) et un simulateur de langage de description de matériel comme VHDL. Les temps de simulation sont directement dépendants de la précision des modèles utilisés. On distingue la simulation précise au cycle près de celle précise au bit près. Dans le premier cas on suppose que toutes les variations des signaux se produisent de façon synchrone avec les fronts d'horloge alors que dans le deuxième cas les comportements temporels de chaque signal sont modélisés à l'intérieur de chaque cycle d'horloge. La simulation au cycle près est plus rapide mais ne permet pas d'identifier des erreurs de timing entre signaux. Les temps de simulation deviennent également prohibitifs si on souhaite atteindre des taux de couverture élevés du système.
La compilation, dans l'approche de synthèse niveau système SLDL est de transformer le programme source en un programme optimisé en tenant compte des paramètres de chaque composant de l'architecture cible [151, 152]. Les techniques d'optimisations concernent le parallélisme de données, le parallélisme de flot, l'ordonnancement des instructions, la transformation de boucles, etc. [153]. Ces problèmes de transformations de boucles et les combinaisons de transformations sont le plus souvent posés sous forme d'équations linéaires et résolus par des programmes linéaires [154–156].
Le concept basé sur la machine virtuelle Java JVM permet la portabilité dans une machine virtuelle sur la cible. Il facilite ainsi le développement et le test indépendamment dans la machine cible. Cependant, Java présente également des limitations pour le développement d'applications critiques. La portabilité théorique du langage Java se heurte au masquage du matériel et des couches basses (par exemple : le RTOS) issues de fournisseurs différents. La préconisation sur la qualité de service des plateformes Java pourrait permettre de réduire cette contrainte. En plus, une JVM nécessite plusieurs centaines de K-octets pour permettre le fonctionnement d'une application Java. Des techniques de compilation "intelligentes" ou de pré-compilation ont permis de réduire de manière significative les ressources mémoires nécessaires.
MARTE définit les constructions d'un langage seulement mais ne couvre pas les aspects méthodologiques pour offrir une démarche à suivre pour la modélisation. Le profil souffre de l'absence d'une sémantique formelle des modèles UML décrivant le comportement du système. Il est aussi difficile de disposer d'une vue globale du système à modéliser, il faut faire appel à plusieurs types de diagrammes. Les contraintes d'optimisation, dans MARTE, sont spécifiées par les propriétés non-fonctionnelles, incluant le temps et l'utilisation des ressources, nécessaires pour différentes sortes d'analyses quantitatives. Les sous profils d'analyse de l'ordonnaçabilité et de performance permettent juste de faire des prévisions de conception.
De ceux qui est posés en question précédemment on peut dire que les techniques de vérification formelle peuvent prendre le relais pour valider certains comportements du système difficilement mis en évidence par la vérification fonctionnelle. Les approches de vérification formelle diffèrent suivant que l'on vise à vérifier la spécification elle-même ou l'équivalence entre une spécification et un raffinement de cette spécification [157, 158] permet de vérifier des propriétés particulières, par exemple qu'un comportement erroné ne se produira jamais. Dans cette approche, des techniques de type model checking (vérification de modèles en logique temporelle) ou preuves de théorèmes (logiques d'ordre supérieur) sont utilisées. Ainsi de prouver l'équivalence de comportements de deux modèles représentés par des machines d'états. Des outils commerciaux sont disponibles, ils opèrent par équivalence entre deux modèles décrits en VHDL ou Verilog. La vérification d'un système complet est un point clé dans la conception d'un système sur puce. Avec l'accroissement des prix des masques, on recherche de plus en plus une réalisation zero-défaut alors que la complexité des composants ne fait qu'augmenter.

1.3.4 La validation formelle des systèmes à base NoC

Dans le recueil de normes français en génie logiciel ISO/AFNOR(Créée en 1926, ISO(International Standardisation Oganisation)AFNOR(Association Française de NORmalisation)) 1997 [159], 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 ; ou bien Spécification écrite en notation formelle, souvent utilisée pour une démonstration d'exactitude.
Selon Gaudel, Marre, Bernot et Schlienger [160], une spécification est dite formelle si : elle est écrite en suivant une syntaxe bien définie, comme celle d'un langage de programmation ; et la syntaxe est accompagnée de sémantique rigoureuse qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte. Selon Choppy [161], la spécification formelle est définie comme suit : 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.

1.3.4.1 Les avantages

L'avantage principal des méthodes formelles est l'utilisation de concepts de la logique et de la technique mathématique par des outils effectifs qui organisent les pensées des concepteurs et qui facilitent la communication entre toutes les personnes concernées par le développement. De plus, ils nous permettent de décrire de manière précise, non ambiguë, les demandes énoncées par l'utilisateur du système **et spécialement les propriétés logicielles** à réaliser. Les notions d'ensemble, de relation, de fonction et leurs différentes propriétés et opérations, avec les quantifications universelles et existentielles, nous permettent d'établir une spécification d'une manière simple et claire et de démontrer mathématiquement les propriétés de la spécification.
Les principaux avantages techniques d'une spécification formelle [162–167], par rapport à une spécification informelle, sont la précision et la clarté. Des imprécisions et des ambiguïtés peuvent facilement se glisser dans les spécifications informelles. Ceci peut ouvrir la voie à plusieurs interprétations. Par contre, les termes de spécifications formelles n'ont qu'une seule interprétation. Un autre avantage des spécifications formelles est que les questions sont posées et répondues avec précision et d'une manière scientifique. De plus, les méthodes formelles fournissent des spécifications qui peuvent être rigoureusement vérifiées, analysées et testées dès les premières étapes du cycle de développement, ce qui n'est pas le cas dans les méthodes informelles. Cela signifie qu'il **est** 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.
Les méthodes formelles 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, Selon Hoffman et Stooper [168], les spécifications rigoureuses jouent un triple rôle dans le développement du logiciel. D'abord, les spécifications documentent avec précision les décisions de conception, indépendamment de l'implantation, et servent de base pour la revue de cette conception. Durant l'implantation, les mêmes spécifications supportent le développement **en** parallèle. D'une part, elles renseignent les utilisateurs de ce qu'ils peuvent s'attendre et, d'autre part, elles renseignent les programmeurs de ce qu'ils doivent faire, et servent de base pour la phase de test. Finalement, durant la maintenance, ces mêmes spécifications supportent l'analyse de changements et aident à la formation de nouveaux personnels.

1.3.4.2 Usage industriel

Dans cette section nous présentons brièvement quelques utilisations documentées des méthodes formelles dans l'industrie.En 1993, Craigen, Gerhart et Ralston [169] ont rédigé un rapport sur l'utilisation de méthodes formelles dans l'industrie, notamment dans le domaine de l'énergie nucléaire en délivrant une étude des effets de méthodes formelles pendant développement d'un projet, en particulier la satisfaction du client, le coût et la qualité du produit. Plusieurs entrevues ont été faites pour examiner l'utilisation des méthodes formelles dans un contexte industriel pour les applications commerciales. Les domaines de ces applications vont de l'énergie nucléaire au transport, en passant par l'aéronautique et l'espace, la physique et le domaine médical.
L'article de Hall [170] constitue une bonne introduction sur l'utilisation des méthodes formelles dans l'industrie. C'est un des meilleurs articles non techniques disponibles sur les méthodes formelles. Hall cite nombre d'applications qui ont été développées avec succès, en utilisant la notation Z. Parmi ces applications la modélisation des oscilloscopes de Tektronix [171].Une autre application qui a connu un succès important consiste à développer une nouvelle version du moniteur de télétraitement CICS [172, 173] en utilisant la notation Z qui a réduit le coût de 9%.
La méthode formelle SCR, dérivée de la méthodologie A7 [174] a été utilisée par David Pamas [175] pour vérifier que l'implantation du logiciel satisfaisait parfaitement les exigences pour arrêter les réacteurs en cas d'urgence de la station nucléaire Darlégnton, gérée par Hydro-Ontario [10,169]. Le projet a été complété avec succès et la station a eu l'autorisation pour la mise en opération de la centrale, l'Atomic Energy Control Board (AECB). Un autre exemple qui englobe l'utilisation de méthodes formelles est le développement du TRANSPUTER [10, 31, 174, 176] par le manufacturier de semi-conducteur Inmos qui a permis de réduire le temps de développement de moitié par rapport aux methodes traditionnelles. Parmi les applications les plus connues dans le domaine du transport, est celle du Métro de Paris, en utilisant la méthode B de Abrial [177, 178].
Dans les facteurs de succès figurent, bien sûr : un support technique suffisant, mais également une réflexion sur l'intégration de la méthode choisie au sein du processus de développement requis par l'entreprise et la prise en compte des contraintes qui en découlent. Les applications pour lesquelles les techniques formelles ont été utilisées sont variées, avec un accent sur les domaines où la sûreté de fonctionnement est primordiale.
Les résultats obtenus sont très positifs et les applications aux systèmes développés dans l'industrie sont généralement facteurs de progrès, en ce sens que la qualité des logiciels est très nettement améliorée. Mais une mauvaise utilisation des méthodes formelles, en choisissant des objectifs inconvenables, en sélectionnant des langages et des techniques inappropriés, sont de bonnes raisons pour annihiler les avantages des méthodes formelles. Notre étude par la suite est consacrée pour détailler le bénéfice d'utiliser les méthodes formelles en générale et la méthode Event-B comme une base d'explication de la raison la proposition de notre approche de validation.

1.4 Les concepts des méthodes formelles

Utiliser une méthode formelle pour le développement d'un système consiste à *spécifier* **(ce qu'on appelle une spécification)** de façon formelle le comportement attendu du système par des propriétés et à *prouver* **(ce qu'on appelle une vérification)** par la suite que le système satisfait cette spécification.

1.4.1 La spécification formelle

La spécification formelle consiste à établir une description formelle d'un système et de ses propriétés de comportement. Elle utilise un langage formel dont la syntaxe et la sémantique sont définies mathématiquement. L'intérêt des méthodes formelles est la possibilité de vérifier de manière rigoureuse des propriétés sur un modèle formel. Pour assurer qu'un système respecte certaines propriétés,

on lui associe **un modèle formel plus ou moins vérifiable** .
Dans les spécifications on parle toujours sur les notions suivantes :

1.4.1.1 Raffinement

Le raffinement [179] est une approche de plus en plus répandue pour construire progressivement des systèmes complexes : il consiste à dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du système préserve bien sa correction vis-à-vis de la spécification précédente.

1.4.1.2 Composition

La composition [180] est une technique de développement horizontal qui permet de maîtriser la complexité des systèmes de grande taille. Elle permet une spécification structurée avec des composants réutilisables. De façon générale, la composition dans le développement consiste à assembler des composants vérifiés en introduisant éventuellement des contraintes pour restreindre le comportement des composants.

1.4.2 La vérification formelle

L'utilisation de méthodes formelles permet une vérification rigoureuse de systèmes informatiques. Il existe deux techniques de vérification formelle : le *model checking* [181] et le *theorem proving* [182].

1.4.2.1 La preuve par modèle "model checking"

C'est une approche algorithmique [181] qui consiste à effectuer la vérification du modèle d'un système en s'assurant que tout comportement satisfait les propriétés attendues. Il y a trois étapes dans la mise en œuvre d'un model checker : modéliser le système par un automate d'états finis, puis définir les propriétés à vérifier sur le modèle dans une logique temporelle, et enfin vérifier les propriétés définies sur le modèle.
Le model checking présente l'avantage d'être une technique complètement automatisée et rapide et ne requiert aucune aide extérieure pendant l'analyse. Cependant, l'inconvénient du *model checking* reste le problème d'explosion combinatoire des états à couvrir. Il est, en effet, impossible de faire une énumération exhaustive d'un nombre d'états infinis. En effet, l'effort de calcul requis pour la vérification d'une propriété sur un ensemble d'états augmente rapidement avec le nombre des variables décrivant ces états. On peut citer comme exemple de model checker les outils, comme TLC [183], UPPAAL [184], SPIN [185], SMV [186], Kronos [187] et HyTech [188].

1.4.2.2 La preuve par théorème "Theorem proving"

Theorm proving Ou bien La vérification par preuve s'appuie sur un système formel contenant des axiomes et des règles d'inférence. Elle consiste à dériver un théorème à partir d'axiomes et/ou de théorèmes prouvés par application des règles d'inférence. Il existe de nombreux prouveurs de théorèmes, tels que Coq((initialement CoC : Calculus of Constructions) et changé pour Coq comme une référence indirecte à Thierry *Coq*uand) [189], Isabelle/HOL(Isabelle/High Order Logic) [190], PVS (Prototype Verification System) [191] et l'Atelier B [192].
Les techniques de preuve ont l'avantage de pouvoir traiter des systèmes à nombre d'états infini contrairement à la technique de model checking qui repose sur des systèmes finis et décidables. Néanmoins, l'une des limitations de l'utilisation de la preuve en industrie est le fait que les prouveurs de théorèmes ont besoin d'être assistés par des ingénieurs expérimentés. En effet, quel que

soit leur niveau d'automatisation, tous les outils de preuve nécessitent l'intervention d'un spécialiste, car leur usage requiert une très grande connaissance aussi bien au niveau du modèle à vérifier que des techniques de preuves utilisées.Cette section propose de définir ce qu'on entend par la modélisation formelle de systèmes. on s'intéresse plus particulièrement à la modélisation de logiciels et de systèmes. Cette partie décrit également une classification des modèles en différents types de ces différentes techniques pour prouver ou assurer que les propriétés du système sont respectées par le modèle construit. Parmi les différentes méthodes formelles disponibles aujourd'hui, on a choisi la méthode Event-B pour la réalisation du travail proposé dans **cette thèse**.

1.4.3 Méthodes formelles et cycle de vie

Les méthodes formelles peuvent être utilisées à différentes étapes du cycle de vie, et pour des objectifs différents. On liste ces utilisations possibles en cherchant, dans la mesure du possible, à donner des critères sur l'impact sur le cycle de vie. **Nous pouvons énoncés leurs usagés durant les différents étapes de la cycle de vie comme suit :**

1.4.3.1 Phase de spécification

Cette phase est souvent la plus critique au niveau de **la phase de** la spécification, c'est à dire **facilite le** passage des besoins au cahier des charges, l'intérêt d'utiliser les méthodes formelles est clair. Elles obligent à un effort de *formalisation* qui, *a priori*, fournit une aide à la modélisation. En effet, la formalisation peut obliger à un certain degré de questionnement, d'autant plus si cette formalisation est assistée par des outils.

1.4.3.2 Conception et codage

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

— ***Processus automatique :*** Le premier type d'outils est adapté lorsqu'il existe un processus systématique de traduction qui, de plus, fournit un code ayant les qualités requises. Ceci signifie d'une part que la spécification est, d'une certaine manière, déjà exécutable et d'autre part qu'elle décrit effectivement le comportement désiré au niveau de l'exécution. Lorsque l'automatisation est possible, le gain est indéniable puisqu'il supprime un niveau de développement. Ceci revient à offrir au développeur un langage de plus haut niveau.

— ***Processus assisté :*** Lorsque la spécification est plus abstraite, ou bien n'a pas été conçue comme **le comportement attendu à l'exécution est décrit**, le processus de transformation peut être assisté formellement. C'est la technique de raffinement. Le développeur doit alors écrire lui-même les spécifications plus précises et la correction sera vérifiée formellement. Le gain est ici moins tangible car l'activité de développement devient plus coûteuse. Néanmoins, si cette activité est mise en place dans une approche globale de la qualité, ce gain peut être évaluable. Ceci a été le cas, par exemple, dans le projet Météor où les tests unitaires ont été supprimés pour les composants développés formellement.

1.4.4 Niveaux d'utilisation des méthodes formelles

On peut distinguer plusieurs niveaux (voir Figure I.19) d'implication des méthodes formelles dans le processus de développement d'une application. Il existe un niveau adapté à chaque cas de figure, prenant en compte les exigences de délais, de coûts, de qualité, de sécurité et l'état de la technologie **à développer , par la suite on explique les niveaux d'utilisation en donnant des exemples pour chacun**.

1.4.4.1 Niveau 0

Aucune utilisation de méthodes formelles. Cela correspond à la pratique standard actuelle dans laquelle la vérification est un processus manuel d'examen et d'inspection appliqué aux documents écrits en langage naturel, pseudocode, ou un langage de programmation, peut-être augmentée avec des diagrammes (et les équations dans le cas des lois de contrôle). La validation se fonde sur des tests, qui peut être entraîné par les exigences et les spécifications (test en boîte noire ou fonctionnelle) ou par la structure du programme. Par exemple les soluces structuré ou en anglais "structured

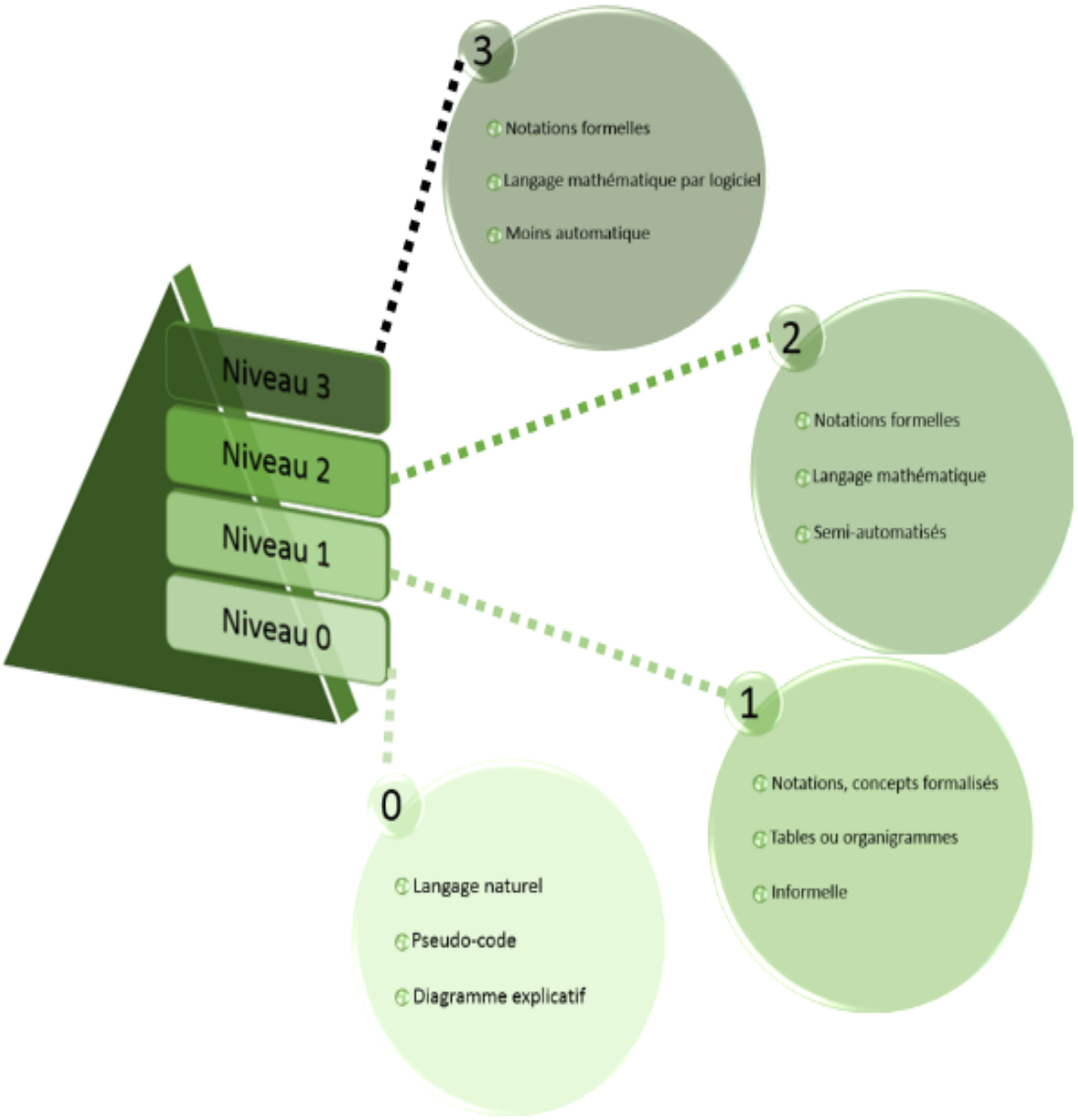


FIGURE I.19 – Les niveaux d'utilisation des méthodes formelles.

walk-throughs" et les inspections **formelles ou** en anglais "formal inspections" sont deux méthodes très structurées pour l'examen manuel de modèles et le code programme. Comme d'abord décrit par Fagan [193] (voir [194] et [195] pour les traitements les plus récents, et [196, 197] pour les études de cas), l'un des principaux avantages de la première méthode par rapport aux autres formes de vérification et de validation est qu'il ne nécessite pas un programme exécutable ; il peut donc être appliqué au début du cycle de conception pour aider à découvrir les défauts et les oublis avant qu'ils ne deviennent enracinées. L'efficacité considérable est signalée pour les inspections formelles : suppression de 50% des erreurs de conception et de mise en œuvre "à travers une vaste gamme des applications" et 70% à 80% des erreurs retraient lors des inspections sont pratiquées avec une plus

grande rigueur et la fréquence [198]. Récemment, il a été suggéré que le processus peut être rendu plus fiable si les équipes d'inspection N (coordonnés par un seul modérateur) sont utilisés [199].

1.4.4.2 Niveau 1

L'utilisation de concepts et de la notation des mathématiques discrètes. L'idée ici est de remplacer une partie du langage naturel utilisé dans des énoncés de besoins et les spécifications avec des notations et des concepts issus de la logique et de mathématiques discrètes. Les **méthodes** incluent **des** notions et la terminologie de la théorie des ensembles, et les différents types particuliers de relations et fonctions (telles que les relations transitifs ou fonctions one-to-one). D'une manière générale, les applications à ce niveau ne sont pas trop pré-occupés par les détails précis du système formel utilisé (par exemple, personne ne va se soucier de savoir si la théorie des ensembles est utilisé Zermelo Fraenkel ou celle de Gödel, von Neumann, et Bernays), et les preuves **dans le cas échéant sont effectuées dans le style informel typique d'un manuel de mathématiques**. Ce style informel correspond à la façon dont les mathématiques sont utilisées dans la plupart des autres disciplines, en fait, la façon dont la plupart des mathématiques lui-même est réalisée.
Pendant l'utilisation de ce genre des méthodes, les mathématiques discrètes contribuent au logiciel et matériel l'ingénierie **sont appliquées selon** trois façons :

- *Des collections de blocs de construction mentale* qui aident dans le développement et la présentation des déclarations claires, précises et systématiques des besoins, des hypothèses, les spécifications et des conceptions.
- *Des notations compactes* qui permettent à ces énoncés de besoins, les hypothèses, les spécifications et des conceptions à être écrit et communiqué avec moins d'ambiguïté que le langage naturel **d'une manière** que l'auteur et le lecteur se partageront la même compréhension de la notation.
- *Certaines «lois»* (c.à.d , les théorèmes ou règles dérivées de l'inférence) qui peuvent systématiser ou guider l'élaboration du cahier des charges ou de la conception d'un niveau à l'autre.

le niveau 1 peut être présenté par la méthodologie Cleanroom lancée par Mills [200] et Dyer [198] et moins révolutionnairen : l'A7 (également connu sous le nom "Réduction des coûts du logiciel" ou Software Cost Reduction (SCR)) et la méthodologie mise au point par Parnas et d'autres au Naval Research Laboratory [201, 202] (voir [203, 204] pour les traitements les plus récents).

1.4.4.3 Niveau 2

L'utilisation des langages de spécification formalisées avec quelques outils de soutien mécanisé. Simplement l'utilisation des mathématiques discrètes est probablement pas une technique de gestion efficace pour les projets impliquant plus de quelques membres ; normes et conventions sont généralement souhaitable, et ceux-ci peuvent être systématisées comme informel "langage de spécification.". Les méthodes formelles sur les échelons inférieurs de niveau 2 peut fournir des outils, tels que les vérificateurs de syntaxe et des « pretty-printers » pour la bonne représentation, qui examinent uniquement les caractéristiques de surface des spécifications ; **des meilleurs** méthodes prise en charge **qui** peuvent **être** des outils plus profonds, tels que les contrôleurs de type, qui examinent un peu plus de leur contenu. Toutes les applications de niveau 2 des méthodes formelles devraient conserver ou renforcer les avantages des applications de niveau 1, et peuvent offrir des avantages supplémentaires :

- Les langages de spécification fournissent généralement plus que juste une notation standardisée pour les mathématiques discrètes : en général, ils traitent également des logiciels d'ingénierie et de préoccupations permettent spécifications à être structurés en **des** unités (par exemple, des modules, des types abstraits, ou objets) avec des interfaces explicitement indiquées. Certains langages de spécification font partie **de ce type de** méthodologie d'ingénierie de logiciel entièrement développé (par exemple, Vienna Development Method (VDM) [205]).

- Les outils de contrôle mécanisés permettent une détection précise de certains types des défauts ; d'autres outils peuvent le rendre simple pour générer une mise à jour de documentation et de rapports de synthèse, et de tracer les dépendances à travers de grandes spécifications.
- **les propriétés sont**, soit générées automatiquement, soit avec une orientation humaine [206, 207]. Ceux-ci peuvent être des moyens très utiles pour explorer certaines propriétés d'un cahier des charges. Certains langages utilisés pour la spécification sont directement exécutables (par exemple, OBJ [208]), mais ce qui compromet plutôt leur statut en tant que **des atouts** pour la spécification et les rend plus proche de haut niveau des langages de programmation [209].

1.4.4.4 Niveau 3

L'utilisation de la spécification complète des langages formels avec des environnements de soutien complets, y compris de théorèmes mécanisé ou vérification de preuve. Le plus vraiment formel de méthodes sont celles qui emploient un langage de spécification avec une interprétation très directe dans la logique, et avec des méthodes de preuve correspondante formelles. Une fois que les méthodes de preuve sont complètement formalisées (c.à.d , la réduction à la manipulation de symboles), il devient possible de les mécaniser. **Cette** mécanisation peut prendre la forme d'un vérificateur de preuve (par exemple un programme informatique qui vérifie les étapes de la preuve proposée par l'utilisateur humain), ou d'un prouveur (un programme informatique qui tente de découvrir des preuves sans orientation humaine), ou le plus souvent, de quelque chose entre les deux. Les avantages de cette approche pleinement formelle sont que les exigences, les spécifications et les conceptions peuvent être soumis à la recherche de la validation , et que la mécanisation élimine un raisonnement erroné avec **une** certitude presque complète ; **alors que** ses inconvénients sont que les spécifications formelles pleinement et **les** preuves vérifiées à la machine peuvent être coûteux à développer, **et que la plupart des langages de spécification sont soutenu complètement avec un mécanisme et des notations qui sont plutôt pauvres et fondées sur des logiques restreintes.** Cela signifie que les spécifications peuvent doivent être tordu pour répondre aux restrictions du **langage**, et peuvent donc être difficiles à écrire et à lire.
Les coûts de développement des vérifications pleinement formels sont souvent **évalués** par la nécessité de développer des formalisations des théories de soutien ou mathématique "connaissances de base" (par exemple, les propriétés de la moyenne arithmétique, ou de permutations) qui sont pris pour acquis dans les approches moins formelles. Comme la vérification formelle devient plus largement pratiquée, les bibliothèques vérifiées de ces théories deviennent disponibles, **elles réduisent** ainsi le coût des développements ultérieurs.
Nous pouvons dire que les méthodes de niveau 3 sont de plus en plus rigoureuses et puissamment mécanisées (par exemple, le proveur Boyer-Moore "Nqthm" [210], Eves [211], PVS [191], **ou** SDVS [212] ainsi **que** le système britannique appelé HOL [213] **qui** est généralement utilisé pour les applications de ce niveau).

1.4.5 La modélisation formelle de systèmes

Un modèle formel est une représentation mathématique d'un système, utilisé pour vérifier ou assurer les propriétés du système considéré. Il **représente** une abstraction mathématique du monde réel obtenue après suppression de certains détails d'implémentation ou après le choix de certaines hypothèses et de caractéristiques essentielles du système. La modélisation mathématique est considéré comme une précision dans la spécification formelle supérieure à celle fournie par une description informelle en langage naturel qui est souvent ambiguë. Elle a des fondements permettant d'énoncer des propriétés et d'étudier le comportement du système dans son ensemble.
Les modèles peuvent être base sure plusieurs fondements qui seront cité par la suite.

1.4.5.1 Les automates

Ce type de modélisation s'appuie sur une description comportementale du système. Cette description donne les actions et les réactions du système face à différents stimuli. Un automate est un système de transitions d'états finis. Il est constitué d'un ensemble d'états et d'un ensemble de transitions décrivant le flot de contrôle du système. Parmi ces modèles : LOTOS [214]., SDL [215], **les diagramme d'état *statecharts*** [216–218]. On peut expliquer encore plus les automates temporisés et BIP.

(a) **Automates temporisés**
Les automates temporisés [219] constituent un des modèles de systèmes réactifs temps continu proposé par Alur et Dill en 1991(Voir Figure I.20). Un automate temporisé est un automate avec des horloges, c'est-à-dire des variables à valeurs réelles, positives ou nulles. Les transitions entre places sont instantanées, dans chaque place, le temps peut s'écouler : à tout instant, la valeur d'une horloge x est le temps écoulé depuis la dernière mise à 0 de x. Les transitions entre places sont conditionnées par des contraintes sur les horloges, appelés gardes, et peuvent remettre certaines horloges à 0. A chaque place est associée une contrainte sur les horloges, appelée invariant.

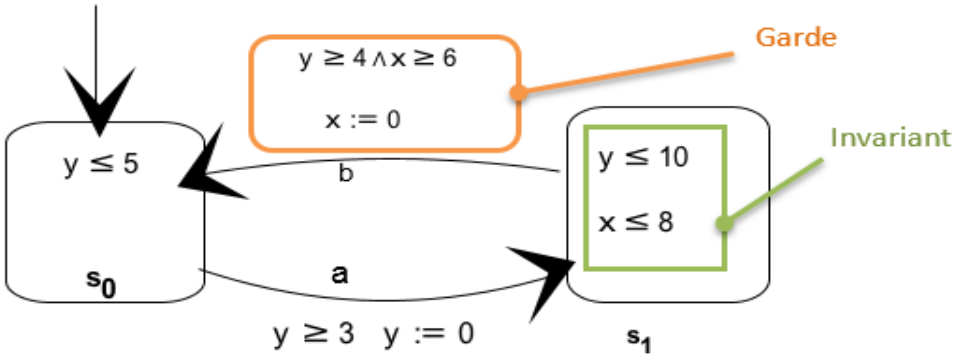


FIGURE I.20 – Exemple d'un automate simple.

(b) **BIP**
la méthode Behaviour Interactive Property (BIP) permet de modéliser et d'analyser des systèmes temps réel à partir de composants hétérogènes. Le langage offre la conception et la description comportementale d'architectures comme un ensemble de transitions, les connecteurs décrivent les interactions entre les transitions et un ensemble de règles de priorité pour décrire les politiques de planification des interactions. Les modèles BIP sont obtenus par la superposition de trois couches de modélisation :

- *La couche inférieure* décrit le comportement d'un composant comme un ensemble de transitions (c'est-à-dire un automate à états finis étendus avec des données) ;
- *La couche intermédiaire* inclue les connecteurs décrivant les interactions entre les transitions de la couche de comportement ;
- *La couche supérieure* est constituée d'un ensemble de règles de priorité utilisées pour décrire les politiques de planification des interactions.

Le modèle BIP(voir Figure I.21) offre une séparation claire entre le comportement des composants et la structure d'un système (interactions et priorités). Le modèle BIP permet une construction hiérarchique [220] des composants composites à partir des composants atomiques en utilisant les connecteurs et les priorités. Le langage **de la méthode** BIP fournit des constructions syntaxiques structurelles pour la définition du comportement de composants, en précisant la coordination par l'intermédiaire des connecteurs et des priorités. BIP permet

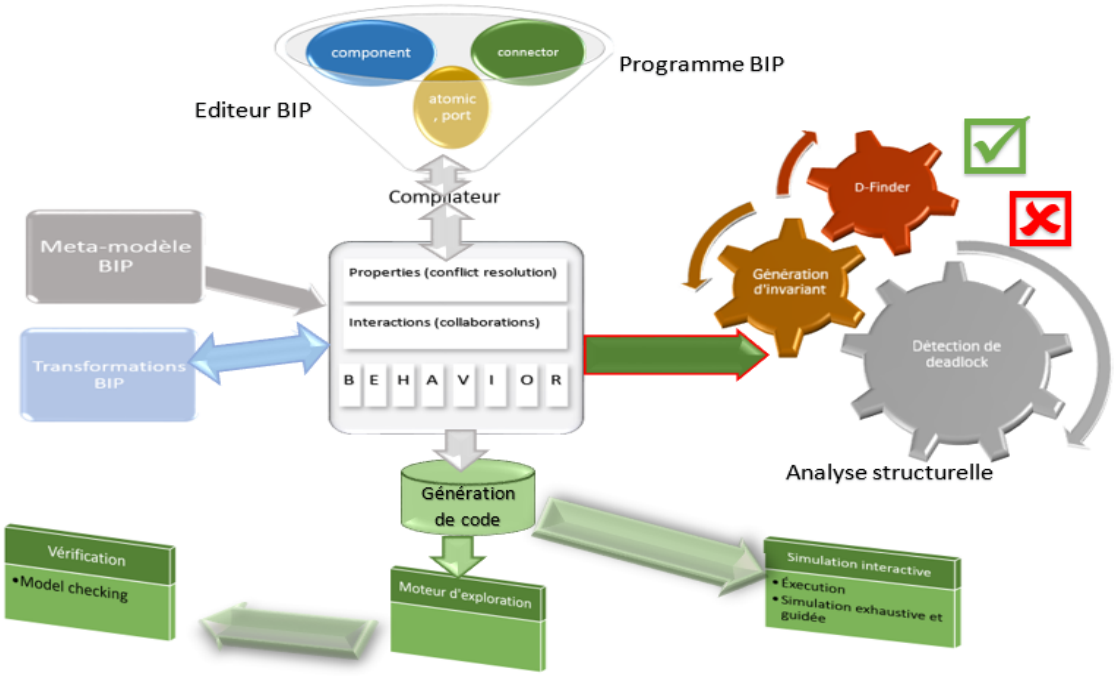


FIGURE I.21 – Explication de la méthode BIP et sa chaine à outil.

d'exprimer la concurrence et le comportement séquentiel des systèmes, comme l'interconnexion de composants. Un système BIP peut être décrit hiérarchiquement et peut également exprimer des informations temporisées dans la même description. Les principales constructions du langage BIP sont :

— **Atome :** pour préciser le comportement avec une interface composée de ports. Le comportement est décrit comme un ensemble de transitions. Les transitions sont étiquetées par des ports.

— **Connecteur :** pour spécifier les coordinations entre les ports des composants et les actions.

— **Priorité :** pour restreindre les interactions possibles basées sur des conditions, en fonction de l'état des composants intégrés.

— **Composite :** pour spécifier la hiérarchie du système à partir des atomes ou des composites, avec l'utilisation des connecteurs et des priorités.

— **Modèle :** pour spécifier le plus haut niveau du système.

BIP [221] possède une grande puissance d'expression et une sémantique formelle qui assure que pour chaque modèle, **qu'aucune** ambiguïté ou erreur d'interprétation est possible. Plusieurs études de cas ont été menées comme MPEG4 encoder [222], TinyOS [223], et DALA [224]. Les outils de vérification BIP permettent l'exploration exhaustive de l'espace des états du système, la détection des blocages potentiels et la vérification de certaines propriétés dans les modèles. Ces derniers éléments jouent en faveur de l'utilisation de telles méthodes dans le cycle de développement. BIP joue un rôle central au sein de plusieurs projets tels que ITEA/Spices [225], OpenEMBeDD [226].

1.4.5.2 Les langages synchrones

Les langages synchrones possèdent un modèle d'exécution qui suppose une horloge logique globale. Trois principaux langages synchrones ont été développés : Estérel [227, 228], Lustre [229] et Signal [230]. Ces langages se basent sur un modèle synchrone à réaction instantanée ayant une

hypothèse de synchronisme qui permet de considérer le système de façon isolée de son environnement et, par conséquent, simplifie le développement et l'analyse du système. Cette hypothèse se résume à :
Les sorties sont simultanées aux entrées qui les provoquent ou autrement dit le temps de réaction du système est négligeable par rapport au temps d'action de l'environnement.
Dans ces langages, des primitives ont été introduites permettant de raisonner comme si le programme réagissait instantanément aux événements externes. Chaque événement interne du programme est précisément daté par rapport au flot des événements externes, et le comportement du programme est complètement déterministe, tant du point de vue fonctionnel que temporel.
En fait, la notion du temps physique (chronomètre) est remplacée par une simple notion d'ordre entre événements : les seules notions importantes sont celles de simultanéité et de précédence entre événements. Le temps physique n'a pas un statut particulier ; il sera perçu comme un événement externe, au même titre que les autres événements en provenance de l'environnement.
La notion d'instant devra être comprise comme celle d'instant logique : l'histoire d'un système est une succession totalement ordonnée d'instants logiques, à chacun desquels 0, 1 ou plusieurs événements surviennent. Les occurrences d'événements survenant au même instant logique seront considérées comme simultanées, celles survenant à des instants différents seront considérées comme survenant à l'ordre de leurs instants d'occurrence. Tous les processus intervenant dans le système ont la même perception des événements présents et absents à un instant donné. L'hypothèse de synchronisme revient à supposer que le programme réagisse assez vite pour percevoir tous les événements externes au bon ordre. Ces approches s'adaptent bien aux systèmes réactifs synchrones. Ces langages, bien qu'ils utilisent la notion du temps, ce temps est physique (chronomètre) et il sera considéré comme un événement externe, provenant de l'environnement. C'est la notion de temps multiforme.

1.4.5.3 La logique temporelle

Ces méthodes permettent de décrire le système à l'aide d'un ensemble de règles logiques permettant de spécifier comment le système doit évoluer à partir de certaines conditions. Elles sont inadéquates pour la représentation des aspects structurels du système, mais sont très appropriées pour la description des propriétés du système. Ainsi, elles contribuent à la définition d'un langage de type assertionnel. La validation consistera à prouver des propriétés de haut niveau **présentées avec des** données sous forme logique, à l'aide de démonstrateurs de preuves et de systèmes d'inférences. Il existe deux grandes classes de logiques temporelles [231] : la logique temporelle linéaire (comme la logique temporelle des actions Temporal Logic of Actions (TLA+) [183,232]) et la logique temporelle arborescente (comme Computational Tree Logic (CTL)) [233,234]. Dans la logique linéaire, un instant n'admet qu'un instant successeur, tandis que, dans la logique temporelle arborescente, un instant admet plusieurs instants successeurs. La logique linéaire permet de spécifier des propriétés de programmes déterministes alors que la logique arborescente est particulièrement bien adaptée à la spécification de propriétés dynamiques de systèmes indéterminés.

1.4.5.4 La base fonctionnelle

Les modèles fonctionnels se basent sur la notion de fonction dans le sens mathématique du terme. Dans d'autres approches adoptant le point de vue impératif, le système est représenté par un ensemble d'états et d'actions qui modifient ces états. De façon : Le système est alors représenté par une série de définitions de fonctions. Chaque fonction $f \in E \rightarrow F$ est définie par son domaine E et son co-domaine F ; Les fonctions sont vues comme des objets et peuvent être arbitrairement manipulées selon la notion des arguments ou les résultats d'autres fonctions, ou bien elles peuvent être composées et la composition est une fonction où les compositions et les définitions récursives remplacent les séquences et les répétitions. La notion de variable est identique à la définition mathématique, c'est-à-dire que la variable correspond à un lien entre la valeur et le nom. La valeur

d'une expression ne dépend que de son contexte textuel et non de son historique de calcul.
La programmation fonctionnelle est un style de programmation qui met l'accent sur l'évaluation des expressions plutôt que sur l'exécution des commandes. Ainsi, une modélisation fonctionnelle peut directement, et naturellement, être implémentée dans ces langages fonctionnels, réduisant ainsi l'écart entre spécification et implémentation. Ces langages sont généralement considérés comme de bons outils pour la conception et le prototypage des programmes sûrs et corrects. Plus encore, les modèles fonctionnels, utilisés avec la théorie des types, représentent un cadre général de développement pour quelques ateliers dédiés au développement de preuves formelles comme PVS [235] et on présente Coq [236] et Isabelle [237].

(a) **Coq**
Coq est un assistant de preuve développé au sein de l'INRIA dans le projet *Logical* [236]. Il permet de proposer une spécification d'un système sous la forme d'assertions et de théorèmes. Le cadre de développement de Coq facilite la preuve mécanique de ces théorèmes. La preuve formelle ainsi établie assure que le système valide les propriétés exprimées sous la forme de théorèmes. Par la suite, de ces théorèmes et de leurs preuves complètes peut être extrait le programme correspondant au système modélisé. Le code extrait est du code fonctionnel. Coq est basé sur le lambda calcul [238] (formalisme de représentation des fonctions mathématiques avec des principes de logiques qui permet d'exprimer toutes les fonctions calculables) et la logique d'ordre supérieur (les quantificateurs portent sur les prédicats et les fonctions qui sont considérés comme des variables et pas des fonctions). Son utilisation reste l'apanage de spécialistes. En effet, il est assez difficile pour l'ingénieur de développement de se mettre à Coq directement. Le principal problème réside dans la difficile automatisation de la phase de preuve. L'écriture de tactiques facilite cette phase, mais elle reste complexe et nécessite une très bonne expérience dans ce domaine. Des travaux sont en cours pour simplifier cette phase en la rendant plus automatique. Cependant, Coq reste un assistant de preuve et souffre d'un manque de méthodologie permettant sa mise en œuvre.

(b) **Isabelle**
Isabelle est un prouveur de théorème développé entre l'université de Cambridge et l'Uni- versité de Munich [239]. Ce prouveur repose principalement sur les logiques du premier ordre et d'ordre supérieur ainsi que sur le calcul des séquents [240] et le lambda calcul [238]. Une syntaxe logique et les règles d'inférence sont spécifiées déclarativement, permettant la construction de la preuve des spécifications en une étape. Le prouveur de théorèmes Isabelle est très utilisé en Europe. Par rapport à Coq il propose une plus grande automatisation de la phase de preuve, ce qui en fait un assistant de preuve préféré. Néanmoins, il ne reste pas simple à appréhender par des ingénieurs de développement logiciel. De nombreux travaux sont en cours pour améliorer et fournir une méthodologie simple de mise en œuvre du prouveur et de sa formalisation. La modélisation à l'aide des modèles fonctionnels est très attirante grâce à la forme que prend les spécifications et à la force d'expressivité des logiques employées. En effet, le risque avec les techniques comme Z ou B est que la logique employée n'est pas suffisamment expressive pour parvenir à un modèle satisfaisant. Ce risque n'existe plus avec les modèles fonctionnels fondés sur la logique d'ordre supérieur. De plus, ces prouveurs de théorèmes permettent d'exprimer des propriétés de très haut niveau et d'en obtenir une preuve formelle. Ces assistants de preuve offrent des tactiques qui sont des commandes utilisées par l'utilisateur pour guider le système pour une compilation automatique des objets de preuves. Un des principaux avantages de ces systèmes est que l'utilisateur peut voir et manipuler la preuve, comme c'est le cas pour Coq.
Malheureusement, cela ne va pas sans quelques inconvénients. Le premier est la traduction de ces spécifications formelles et de ces modèles fonctionnels en du code exécutable. Cette traduction est difficile voire impossible dans certains cas et des problèmes d'efficacité apparaissent. Le second concerne la haute technicité et l'expérience à acquérir pour pouvoir manipuler ces outils.

1.4.5.5 Les machines d'états abstraits

Dans ce type de modèle (aussi connu par Abstract State Machine (ASM)), le système est représenté par son état et un ensemble d'opérations qui s'appliquent à cet état et le modifient. Chaque opération permet d'effectuer une transition entre les états que peut prendre le système. Plusieurs méthodes formelles utilisent ce formalisme. Parmi lesquelles on peut citer les ASMs [241], VDM [242], Z [243] et on s'intéresse par la méthode B et son extension l'Event-B.

(a) **La méthode Z**
Développée dans les années 70, en parallèle de VDM, avec notamment la participation de Jean-Raymond Abrial qui, quelques années plus tard, élabore la méthode B [244]. La méthode Z est basée sur la théorie des ensembles et le calcul des prédicats. La spécification finale est obtenue par raffinements successifs en partant de la forme la plus abstraite du système. Le schéma est la notion de base des spécifications Z. Les schémas sont utilisés pour décrire les états et les opérations du système. La spécification du système est décomposée en schémas qui permettent de décrire les aspects statiques (états et invariants) et dynamiques (opérations, relations entre les entrées et les sorties, changements d'états) du comportement du système.

(b) **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 représente les Logiciels par des données, caractérisées par leurs propriétés invariantes, et des services qui manipulent ces données [159, 177] Les caractéristiques de la méthode B sont les suivantes :

- Elle dispose d'un seul cadre formel pour décrire abstraitement et concrètement les logiciels, autrement dit, pour les spécifier et les concevoir ;
- 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.

Depuis quelques années, la méthode B a été utilisée avec succès dans le domaine du transport ferroviaire, notammment dans le cadre du métro sans conducteur METEOR [245] réalisé par Matra Transport International. Actuellement, ses applications industrielles touchent divers domaines, tels que les cartes à puce [246], le diagnostic automobile [247, 248] et les circuits électroniques [249].

1.5 La sélection de l'Event-B

Par la suite nous donnons des raisons pour lesquels nous choisissons l'Event-B, ces arguments sont établis en sorte des tableaux comparatives mais tout d'abord nous pouvons dire que l'Event-B est une extension de B [250] conçue pour modéliser des systèmes réactifs [244]. Dans cette extension, les concepts de machine abstraite et d'opérations sont respectivement remplacés par ceux de système abstrait et d'événements. Ces systèmes abstraits peuvent être vus comme des systèmes fermés qui modélisent le système et son environnement, et dont l'état peut évoluer par l'application des événements. Ceux-ci sont décrits en terme d'actions gardées, et ils sont susceptibles d'être déclenchés quand leur garde devient vraie. L'un des événements déclenchables peut alors être appliqué et l'état du système change en fonction de l'action associée à l'événement. Cette notion de système à événements est inspirée des actions de Back [251], ou des commandes gardées de Dijkstra [252]. En Event-B, les événements sont constitués d'un prédicat d'état appelé garde et d'une

substitution ayant pour but de modifier la valeur de variables. Dans un état donné, plusieursdes gardes du système peuvent être vraies. L'une d'elles est alors déclenchée et la substitution associée effectuée. Parmi les projets qui ont utilisé cette méthode Dependable Software Forum (DSF) [253] en Décembre 2009. Après une enquête initiale, Certaines grandes entreprises en informatique et technologies (NTT-Data, Fujitsu, Hitachi, NEC, Toshiba, et SCSK) se sont réunis pour former ce groupe de recherche conjoint, un ensemble des systèmes industrielles du projet ADVANCE réalisé par l'approche , comme l'amélioration du flux d'énergie du générateur au consommateur en utilisant des technologies de carbone [254], la preuve formelle qui répond aux normes ferroviaires européennes (CENELEC) : les opérateurs de transport Répond aux demandes (par exemple Paris, New York) [255].

1.5.1 La comparaison : Event-B, Isabelle / HOL, PVS et VDM

Dans cette section, un résumé des différences entre les Event-B, Isabelle/HOL, PVS et VDM sera délivrer. **Sachant qu'on parle de** la méthode Event-B avant l'amélioration en [36,37] la comparaison est fournie sous forme de tableau (voir le tableau I.2, I.3 et I.4), et est divisé en trois parties : la logique, le langage de spécification et le prouveur. Les points clés suivants résument les critères selon lesquels les trois formalismes seront comparés :

- **Logique :** les formalismes sont comparés par rapport à la logique utilisée, son expressivité et comment il gère partialité.
- **Langage de Spécification :** les trois formalismes sont contrastés en ce qui concerne la facilité d'utilisation, l'expressivité et l'extensibilité du langage de spécification.
- **Prouveur :** les formalismes sont comparés en termes d'efficacité de prouveur (à savoir, la puissance est le soutien des preuves automatiques), l'extensibilité et la solidité.

Notant que VDM repose sur démonstrateurs tiers pour exécuter des obligations de preuve, en tant que telle, VDM ne sont pas considérés pour la comparaison des prouveurs. Ainsi que dans les tableaux comparatifs ci-dessous :
N/A : fonctionnalité n'est pas prise en charge, ou que ne peut pas porter un jugement sur la base de la documentation disponible.
++/+ : Beaucoup plus utilisé.
++ : Plus utilisé.
+/- : Proche de rarement utilisé.
- : Rarement utilisé.
Dans cette section, l'aperçu des trois formalismes largement utilisés. Dans le contexte de la comparaison (Tableau I.2, Tableau I.3, Tableau I.4) les aspects suivants de l'Event-B sont celles existé avant l'amélioration dans [36, 37] utilisé dans le chapitre 4 :

	Event-B	Isabelle/HOL	PVS	VDM
La logique	Théorie des ensembles	Typage HOL	Typage HOL	LPF
Prédicats de sous-type	N/A	N/A	++	N/A
Types dépendant	N/A	N/A	++	N/A
Polymorphisme	N/A	++	-	+
Type Abstrait de Données (T.A.D)	N/A	++/+	++/+	N/A
Fonctions récursives	N/A	++/+	++/+	-

TABLEAU I.2 – Comparaison en terme de logique de preuve.

	Event-B	Isabelle/HOL	PVS	VDM
Syntaxe flexible	-	++	-	-
Système modulaire	-	+	++/+	+
Surcharge	N/A	-	++	-
Librairies	N/A	+	++/+	+

TABLEAU I.3 – Comparaison en terme de langage de spécification.

	Event-B	Isabelle/HOL	PVS
Automation	+	+	+
Gestion de preuve	++	+/-	++
Langage tactique	-	++	-
Arithmétique	-	+/-	++
Solidité	-	++	-

TABLEAU I.4 – Comparaison en terme de prouveur.

(a) ***Soutien pour le polymorphisme :*** Avant le travail [36, 37] Event-B ne supporte pas les utilisations des opérateurs polymorphes mais après, l'utilisateur peut contribuer opérateurs polymorphes dans une manière saine et utilisable. Dans le chapitre 4 que notre approche utilise la résolution **de** ce problème particulier ressemble à l'approche adoptée par Isabelle/HOL plutôt que celle prise par PVS.

(b) ***Types abstraits de données et les fonctions récursives :*** Parmi les contributions mineures de cette thèse (en chapitre 4) la fourniture d'un mécanisme pour spécifier les types de données inductifs et opérateurs récursifs. Cette contribution est fortement influencée par les types de données à Isabelle/HOL.

(c) ***Syntaxe flexibilité :*** La syntaxe peut être contribué à la langage mathématique Event-B sans compromettre la solidité du formalisme.

(d) ***Gestion de la preuve et la solidité :*** Comment la preuve en Event-B peut être augmentée avec de nouvelles règles de preuve en temps utile par la présence de règles de réécriture intégrées à Rodin.

(e) ***Système Modulaire :*** Dans chapitre 2, on présente comment structurer modèles Event-B d'une manière qui favorise la réutilisation même pour une génération automatique de code.

1.5.2 Logique de l'Event-B

Schmalz définit la logique de l'Event-B en utilisant un plongement peu profond Isabelle/HOL [35] : une intégration profonde nécessite la logique pour **que** l'Event-B doit être défini comme une logique d'objet Isabelle. Ce processus est très impliqué, et peut rendre la procédure de preuve utile d'Isabelle/HOL inutilisable.
Une intégration profonde d'une logique Isabelle exige :

(a) définissant la syntaxe de la logique de l'objet comme un type de données,

(b) fournir la sémantique de la logique de l'objet,

(c) la preuve que les axiomes qui régissent la syntaxe sont solides par rapport à la sémantique. Une incorporation peu profonde ne nécessite pas d'étapes (1) et (2) [256]. En conséquence, l'incorporation superficielle peut être considérée comme une traduction syntaxique.

Schmalz fournit une spécification complète de la logique de l'Event-B en un seul document [239]. Il donne la sémantique, conçoit **la** solidité en préservant les méthodes d'extension, se développe

un calcul similaire à la preuve [38], et prouve sa solidité. **Le travail dans** [239] présente un langage formel pour exprimer règles (y compris les conditions non-liberté **(non-free)** et de montrer comment raisonner dans l'Event-B sur la solidité des règles. La logique de l'Event-B dispose d'un système de type de style Hindley-Milner [35] semblable à Isabelle/HOL et ML [257]. Les opérateurs de type tels que $\times$ et $\leftrightarrow$ sont de nies par leurs Isabelle/HOL Type counter-parts(Z est considéré comme un opérateur de type avec un arité de zéro) **ces** substitutions sont essentielles à une logique qui prend en charge le polymorphisme, et sont également introduites. **En plus**, les modalités et **les** formules sont introduits **et** sont affectés **pour la** sémantique Isabelle/HOL au moyen d'un certain nombre d'ordre supérieur **de** constructions logiques. Notez que Schmalz considère **les** formules (**oubien** les prédicats) d'avoir un type booléen B. **les** moyens d'étendre la logique conservatrice en Event-B sont décrites dans [35] (voir le chapitre 4).

1.5.3 Avantages de la méthode Event-B

Un des avantages de la méthode Event-B est qu'elle permet plus de flexibilité et d'expressivité que les langages d'entrées des model-checkers tout en nécessitant moins d'interaction avec l'utilisateur que les prouveurs de théorèmes tel que PVS. Aussi, aucune supposition n'est faite quant à la taille du système à modéliser contrairement au model checking. Le raffinement, nous permet d'enrichir notre modèle graduellement, et de renforcer peu à peu notre invariant, alors qu'en général en model-checking on part directement d'un modèle très complexe du système à vérifier. La complexité du système est donc distribuée, et les preuves étapes par étapes sont plus faciles. Le raffinement permet d'ailleurs d'assurer un meilleur dialogue entre la personne concevant le modèle et son mandataire, et de réparer les erreurs en cours de développement. Ainsi on valide les modèles étape par étape, plutôt que sur un unique modèle final construit directement.

1.5.4 Limites de la méthode Event-B

La méthode Event-B permet d'établir des preuves de propriétés d'invariance, par conséquent, on peut vérifier la correction d'une spécification, mais aussi sa complétude. Ces propriétés peuvent être exprimées en faisant appel à la théorie des ensembles. En revanche, la méthode Event-B ne permet pas d'exprimer explicitement des propriétés temporelles (fatalité, équité, etc.).

Conclusion

Nous avons entamé dans ce chapitre à exprimer une brève présentation **qui explique** la communication sur une puce dans les systèmes de type NoC qui sont des réseaux proposés dans la littérature et peuvent être des systèmes auto-organisés reconfigurables à base de technologie FPGA, **le but est donc, essayer d'intégrer durant ce travail les aspects de versatilité dans ce type de réseau**.
L'adaptabilité et l'efficacité pour la conception de réseaux sur puce reconfigurables oblige alors de valider l'association de l'approche générale de communication sur puce reconfigurable avec les concepts architecturaux de mise en œuvre des propriétés d'auto-organisation pour la conception d'un système MPSoC reconfigurable auto-organisé.
Le développement de ces systèmes embarqués implique la conception d'un système dans lequel les ressources sont habituellement limitées, et lequel peut devoir s'exécuter sans intervention manuelle. En effet, ils sont soumis à des contraintes techniques strictes à la fois de performance fonctionnelles mais également temporelles, de consommation énergétique et de robustesse, qui pèsent sur eux du fait de leur taille et de leurs ressources limitées. Ces contraintes imposées par les besoins du système doivent être prises en compte dès les premières phases du cycle de développement.
Deux grandes approches pour la validation des systèmes embarqués de type NoC : la validation fonctionnelle et la validation formelle. La première qui réside dans la validation en utilisant des approches classiques où le système est construit de différents composants individuels et si leurs

tests indiquent que les performances du produit sont insuffisantes, on relance une étape se basant sur le prototypage et le débogage comme il fait recours à la simulation pour confirmer le partitionnement et le choix des composants. Une autre approche est utilisée : l'approche moderne basée sur la synthèse niveau système, qui peut être basée sur un langage de traitement pour supporter l'accès convenable aux caractéristiques particulières. **Il est aussi réalisé sur une plateforme pour mapper son comportement correctement**. Elle peut être basée sur des modèles représentent les comportements de chaque sous-systèmes, mais aussi elle peut être basée sur l'architecture pour définir les caractéristiques qui suivent l'architecture à réaliser. Malgré les nombreux projets définis pour la validation fonctionnelle des systèmes sur puce il reste toujours insuffisant en terme de clarté et **de** la précision de questionnement et raisonnement pour modéliser les différents critères de fonctionnement de système ceux qui nous amené à utiliser les méthodes formelles.
Des bases différentes se définissent les modèles des méthodes formelles (les automates, langages synchrones, logique temporelle et les machines abstraits) pour les classifier et **les** utiliser selon les besoins de validation, dans cette thèse **nous avons** présenté l'ultime bénéfice pendant l'utilisation de la méthode Event-B avec notre proposition qui améliore la validation générique et la génération du code en prenant les NoCs étant un exemple.
Dans le chapitre qui se suit on exprime l'environnement de validation formelle RODIN liée à la méthode Event-B et l'architecture des modèles **ainsi nous avons expliqués comment cette approche renforce le développement des système à base NoC dès qu'elle** nous aide à créer les opérateurs de raffinement inspirée de l'approche des opérateurs dans les différentes méthodes formelles détaillé dans le chapitre 2, notre approche d'opérateurs au sein de la méthode Event-B est largement expliquée dans le chapitre 3, le chapitre 4 est réservé pour discuter le dernier étape de validation basée sur les théories qui prend des grandes importances et parmi ces avantage la théorie qui modélise le comportement de système au niveau du l'environnement d'exécution interprété au code VHDL **et comment les opérateurs de raffinement est toujours présente pour rassurer les étapes de la validation du comportement de système à base NoC de façon convenable et non intuitive**.

Chapitre 2

RODIN : L'environnement extensible de la validation Formelle

« … it would be impossible to complete such a large project in the time given, were it necessary to do everything manually… human beings are simply not capable of handling such quantities of low level mathematical detail without making mistakes.…The European Project Rodin will create a new platform, implemented on Eclipse, for embedding the Event-B techniques described … »

JEAN-RAYMOND ABRIAL

Sommaire

Introduction

L'un des **formalismes** qui représentent une méthode formelle **est** celui qui est appelée **l'**Event-B. Ses caractéristiques font un outil fondamental, qui amène une certaine rigueur dans le processus de conception informatique. Cette rigueur, jointe à des concepts mathématiques de production (logique de prédicats de premier ordre et la théorie des ensembles) qui font leur succès dans l'industrie avec l'utilisation d'un grand atout **de l'**Event-B qui est la plate-forme RODIN. Il est à préciser que l'utilisation de cette méthode est imposée par les pouvoirs publics afin de vérifier automatiquement (prouveur interne) l'élaboration des systèmes informatiques en relation avec des êtres humains (prouveur interactif), en particulier pour la conception des logiciels critiques, logiciels dont tout dysfonctionnement entraînerait des conséquences inacceptables par les différentes étapes de vérification par preuve sous l'environnement RODIN et ses multiples types de plug-ins.
Ce chapitre est organisé comme suite : la première partie illustre l'utilisation des fameux concepts mathématiques de base pour la méthode Event-B pour vérifier les systèmes critiques, la deuxième partie détaille la méthode Event-B. La troisième partie présente la plateforme RODIN et ses plugins en expliquant les prouveurs (interne et externe) dans l'outil RODIN.

2.1 Les concepts mathématiques en Event-B

Le principe des méthodes formelles est d'utiliser des notions mathématiques pour représenter le comportement des programmes informatiques : c'est pourquoi on parle de **la** modélisation formelle. Les notions mathématiques sont donc les éléments fondamentaux dont **on** dispose l'utilisateur pour construire un modèle correspondant à ses besoins. Mieux il connaît ces notions, meilleure sera son utilisation du langage. Utiliser un langage formel permet d'exprimer des énoncés démontrables, et bien connaître ces notions mathématiques permet de conduire efficacement ces démonstrations.

2.1.1 Les symboles

L'écriture mathématique est très riche en symboles inhabituels en informatique. on utilise par exemple l'implication $\Rightarrow$, la surcharge $\lhd$, etc. Ces symboles nécessaires pour une écriture synthétique des formules ne sont pas disponibles sur un clavier d'ordinateur. Pour cette raison ils sont représentés par des combinaisons de caractères American Standard Code for Information Interchange (ASCII)(pour plus voir A) : par exemple $\Rightarrow$ est représenté par =>. Dans tous les documents il est préférable d'utiliser la notation symbolique qui facilite la lecture, plutôt que la notation ASCII.

2.1.2 Le raisonnement formel

Un raisonnement formel consiste à démontrer un énoncé sous un ensemble d'hypothèses à l'aide d'une collection de règles d'inférences. Par exemple, on se propose de démontrer $8 > 0$ sous l'hypothèse $8 > 5$. Notre énoncé est donc $8 > 0$ et l'ensemble des hypothèses se réduit à $8 > 5$. On suppose qu'on a "tout oublié", c'est-à-dire que nous voulons utiliser seulement les règles et les hypothèses qu'on présente explicitement. On supposera disposer des deux règles d'inférences :

$$Si\ 5 > 0,\ et\ si\ 8 > 5,\ alors\ 8 > 0 \tag{2.1}$$

$$5 > 0\ est\ toujours\ vrai \tag{2.2}$$

En appliquant la règle (2.1) pour démontrer $8 > 0$, comme nous supposons savoir $8 > 5$ (c'est notre hypothèse), il ne reste plus qu'à démontrer $5 > 0$. Le nouvel énoncé est donc $5 > 0$. On applique alors la règle (2.2) qui dit que $5 > 0$ est toujours vrai. L'application de cette règle ne produit pas de nouveau but, donc la preuve est finie. Ces notions de preuve sont très intuitives et naturelles. Il est néanmoins utile de bien les saisir à partir des éléments qu'on vient de voir, c'est-à-dire :

- Démonstration d'un énoncé sous certaines hypothèses,
- Collection des règles d'inférences autorisées.

Conventionnellement, on représentera l'ensemble des hypothèses par le mot **HYP**. Pour indiquer que nous ajoutons une hypothèse **H** à cet ensemble, on écrira**HYP,H**.
Que se passe-t-il si l'une des hypothèses qu'on suppose est toujours fausse ? Par exemple, doit-on considérer que $8 < 0$ est valide sous l'hypothèse fausse $5 < 0$? Intuitivement, cela revient à s'interroger sur un cas impossible. La réponse peut sembler une affaire de conventions, il n'en n'est rien. La cohérence globale de la théorie impose de considérer que **Tout énoncé est VRAI sous des hypothèses fausses**. On verra plus loin des exemples dans lesquelles cette nécessité apparait. Cette notion à première vue très abstraite est souvent employée dans la mise en œuvre du langage B et ses extensions **comme** l'Event-B. La génération des obligations de preuve d'un composant B dans certains cas peut et doit produire des obligations de preuve contradictoires (voir paragraphe 3.4). Ces dernières sont JUSTES et participent à la preuve du composant.

2.1.3 Le calcul propositionnel

Intuitivement, une proposition logique peut être définie comme une affirmation vraie ou fausse. Par exemple, "la maison est blanche" est une proposition logique, car la question "cette phrase est-elle vraie ou fausse" a un sens. Par contre "la maison" n'est pas une proposition logique. Une proposition logique est désignée par le terme de prédicat. Soient P et Q deux prédicats. On définît les notations suivantes :

- $P \wedge Q (P\ et\ Q)$
- $P \Rightarrow Q (P\ implique\ Q)$
- $\neg P$(Négation de P)

Ces notions sont utilisées en preuve formelle par les règles suivantes :

- Pour démontrer $P \wedge Q$ sous les hypothèses **HYP** il suffit de démontrer P sous **HYP**, puis de démontrer Q sous les mêmes hypothèses.
- Pour démontrer $P \Rightarrow Q$ sous les hypothèses **HYP** il suffit de démontrer Q sous les hypothèses **HYP** augmentées de l'hypothèse P, c'est-'a-dire d'après nos conventions : **HYP,P** . Ceci est connu sous le nom de règle de déduction. On dit aussi que P "monte" en hypothèse.
- Pour démontrer $\neg P$ sous les hypothèses **HYP**, nous disposons de la règle suivante : s'il existe un prédicat Q tel que sous les hypothèses **HYP,P** on puisse démontrer à la fois Q et $\neg Q$, alors $\neg P$ est démontré sous les hypothèses **HYP**. Intuitivement, en supposant P nous avons abouti à une contradiction.

Notant que si P est toujours faux, alors $P \Rightarrow Q$ est toujours vrai. Ceci découle de la règle de déduction et rejoint la remarque du paragraphe 1.2 sur les hypothèses fausses. Pour faciliter la manipulation des prédicats dont le statut **vrai** ou **faux** est connu, introduisons les notations suivantes :

- **btrue** est le prédicat toujours vrai ;
- **bfalse** est le prédicat toujours faux.

Il reste à introduire les deux dernières notations propositionnelles, qui se définissent à partir des précédentes :

- $P \vee Q$ (P ou Q) est défini comme $\neg P \Rightarrow Q$.
- $P \Leftrightarrow Q$ (P équivalent à Q) est défini comme $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

La définition du "ou" (disjonction) nécessite quelques commentaires. Intuitivement, elle indique la chose suivante : dire que P ou Q est vrai revient à dire que si P est **faux**, Q est forcément **vrai** (traduction de $\neg P \Rightarrow Q$). Cette définition n'est pas symétrique en P et Q, bien que l'on puisse démontrer que $\neg P \Rightarrow Q$ et $\neg Q \Rightarrow P$ soient équivalents, autrement dit que $P \vee Q$ est identique à

$Q \vee P$. D'autre part, la définition de $P \vee Q$ est un exemple justifiant notre assertion que tout but est vrai sous des hypothèses fausses (voir paragraphe 2.2). En effet, considérons la proposition $btrue \vee Q$. De façon à ce que la définition du ou corresponde à la notion naturelle, nous souhaitons que cette proposition soit toujours vraie. Autrement dit :

$$btrue \vee Q \Leftrightarrow btrue$$

D'après la définition du symbole $\vee$ cela s'écrit :

$$btrue \vee Q \Leftrightarrow \neg(btrue) \Rightarrow Q \Leftrightarrow bfalse \Rightarrow Q$$

Il est donc nécessaire de considérer que $bfalse \Rightarrow Q$ est toujours **vrai**.
On peut citer ici quelques propriétés(depuis le B-Book [15]) moins fondamentales, mais choisies pour leur importance lors de l'utilisation pendant la modélisation des systèmes :

— $(bfalse \Rightarrow P) \Leftrightarrow btrue$.

— $(btrue \Rightarrow P) \Leftrightarrow P$.

— $(P \Rightarrow btrue) \Leftrightarrow btrue$.

— $(P \Rightarrow bfalse) \Leftrightarrow \neg P$.

2.1.4 Les prédicats quantifiés

Afin d'exprimer les propriétés de nos composants écrits en Event-B, nous aurons besoin de nouvelles notions. Par exemple, on pourra avoir à démontrer une propriété sur un indice de boucle :

$$indice \in 1..10 \Rightarrow indice < MAXINT$$

Il manque encore beaucoup d'opérateurs pour cette écriture. Tout d'abord on a besoin de la notion de variable.

- **Variable :** tout identifiant non prédéfini, constitué avec certaines règles de lettres, chiffres et, est une variable.

Pour des raisons d'implantation de RODIN, les variables à une lettre ne sont pas autorisées (ce sont des *Jokers*,). La notion de variable nous permet d'introduire une notion essentielle, le prédicat universellement quantifié. Si v est une variable et P un prédicat, on a la construction suivante :

$$\forall v \cdot P(lire\ pour\ tout\ v, P.)$$

On dit que le prédicat P est quantifié par la quantification universelle $\forall v$. On dit aussi que la portée de la variable quantifiée v est le prédicat P. Donnons quelques exemples de prédicat quantifié :

$$\forall xx \cdot (xx \in N \wedge xx < 10 \Rightarrow xx < 100)$$

$$\forall var \cdot (var = 10 \Rightarrow var < 100)$$

Remarquant que pour des raisons de typage, on impose que **tout prédicat universellement quantifié soit mis sous la forme** $\forall v.(P \Rightarrow Q)$ (De plus, le contrôleur de types de l'Atelier B attend des prédicats quantifiés de la forme syntaxique $\forall v \cdot (P \Rightarrow Q)$ où P est un prédicat typant les variables introduites). Une autre remarque essentielle est que l**e nom de la variable quantifiée n'importe pas**. On dit que la variable quantifiée est une **variable muette**. Par exemple :

$$\forall xx \cdot (xx = 10 \Rightarrow xx < 100)$$

est équivalent à

$$\forall yy \cdot (yy = 10 \Rightarrow yy < 100)$$

La portée de la variable muette x dans $\forall x \cdot P$ est le prédicat P uniquement. En particulier une variable de même nom peut être utilisée dans d'autres prédicats, sans conflit. Par exemple :

$$xx = 2000 \wedge$$
$$\forall xx \cdot (xx = 10 \Rightarrow xx < 100)$$

Ce prédicat indique que la variable "externe" xx vaut 2000 et d'autre part que tout nombre égal à 10 est plus petit que 100. Il n'y a pas de confusion entre l'occurrence de la variable "externe" et celle de la variable muette. Une telle écriture bien que correcte prête toutefois à confusion, il faut l'éviter. Les règles d'inférence relatives aux prédicats universellement quantifiés sont légèrement plus complexes, car elles font appel à la notion de variable non libre dans une expression, notion que nous n'aborderons pas dans ce chapitre. La règle principale, restreinte aux prédicats de la forme $\forall x \cdot (P \Rightarrow Q)$, est la suivante :

— Pour démontrer $\forall x \cdot (P \Rightarrow Q)$ sous les hypothèses **HYP**, si la variable x n'est pas utilisée dans HYP,il suffit de démontrer Q sous les hypothèses **HYP,P**. Cette règle dite règle de généralisation signifie que pour démontrer que Q est vrai pour toute variable x vérifiant P, il suffit de se donner une variable x vérifiant P et de faire la preuve de Q sous ces hypothèses. Il y a évidemment un problème si la variable x est déjà utilisée avec un autre sens dans les hypothèses ; il faut alors réécrire $\forall x \cdot (P \Rightarrow Q)$ avec une autre variable. De telles réécritures de prédicats font intervenir la notion de substitution qui sera brièvement développer par la suite.

- ***Notation de la substitution*** En Event-B, la notion de substitution est fondamentale (C'est évidemment le cas de toutes les théories logiques et du lambda-calcul.). Elle sera la base de la spécification des opérations. Une substitution est notée :

  $$[x := E]P$$

  Elle indique le remplacement uniforme des occurrences libres de x par E dans P. On peut aussi avoir une substitution multiple avec une signification évidente :

  $$[x1, x2, \ldots := E1, E2, \ldots]P$$

  Sachant que :
  - Les identificateurs des spécifications Event-B doivent avoir au minimum deux lettres (ou une lettre et un chiffre).
  - Les sous-expressions d'une expression logique peuvent être parenthèses.
  - On ne redonne pas ici les axiomes qui s'appliquent aux connecteurs logiques (associativité, commutativité, distributivité, etc.).

Les substitutions primitives sont des substitutions généralisées à partir desquelles on peut construire toutes les autres substitutions. Elles ont une notation concise qui facilite les formules et les calculs (pour plus voir annexe.A.2). Dans les sections suivantes : x, y, z sont des variables ; E, F, V sont des expressions ; S, T, U, W sont des substitutions généralisées et I, J, P, Q, R sont des prédicats.

2.1.5 Notations de la théorie des ensembles

Parmi les notations dans cette théorie on peut citer :

2.1.5.1 Construction d'ensembles

Les ensembles peuvent être des ensembles d'éléments sans structure, ou bien des produits cartésiens d'ensembles, ou encore des parties d'un ensemble. Quelques notations de construction d'ensembles sont présentées dans le tableau suivants : Le non-terminal Ensemble désigne une expression construisant un ensemble. Le non-terminal $Expression\_liste$ représente une liste d'expressions quelconques séparées par des virgules. Les éléments d'un produit cartésien sont des paires d'éléments.

Expression	Signification
$x := E$	substitution simple.
$x, y := E, F$	substitution multiple simple.
$skip$	substitution sans effet.
$P \mid S$	substitution pré-conditionnée.
$P \Rightarrow S$	substitution gardée.
$S \llbracket P$	substitution de choix borné.
$@z \cdot S$	substitution de choix non borné.
$S;T$	séquencement de substitutions.
$W(P,S,J,V)$	substitution d'itération.

TABLEAU II.1 – Les type de substitutions.

Expression	Signification
$\emptyset$	Ensemble vide.
$Ensemble \times Ensemble$	Produit cartésien.
$\mathbb{P}(Ensemble)$	Ensemble des sous-ensembles.
$\mathbb{P}_1(Ensemble)$	Ensemble des sous-ensembles non vides.
$\mathbb{F}(Ensemble)$	Ensemble des sous-ensembles finis.
$\mathbb{F}_1(Ensemble)$	Ensemble des sous-ensembles finis non vides.
$\{Expression\_liste\}$	Ensembles définis en extension.
$\{Id\_liste \mid Prédicat\}$	Ensembles définis en compréhension.

TABLEAU II.2 – Les constructeurs d'ensembles

2.1.5.2 Prédicats sur les ensembles

On dispose des prédicats suivants relatifs aux ensembles :

Expression	Signification
$x \in s$	L'appartenance.
$x \notin s \overset{def}{=\!=} \neg(x \in s)$	Le non appartenance.
$s \subseteq t \overset{def}{=\!=} s \in \mathbb{P}(t)$	L'inclusion.
$s \not\subseteq t \overset{def}{=\!=} \neg(s \subseteq t)$	Le non inclusion.
$s \subset t \overset{def}{=\!=} (s \subseteq t \wedge s \neq t)$	L'inclusion stricte.
$s \not\subset t \overset{def}{=\!=} \neg(s \subset t)$	Le non inclusion stricte.

TABLEAU II.3 – Les prédicats sur les ensembles.

2.1.5.3 Expressions d'ensembles

Le premier jeu d'opérateurs concerne les opérations simples entre ensembles. On suppose, comme hypothèses $s_1 \subseteq t$ et $s_2 \subseteq t$; le deuxième jeu d'opérateurs traite d'union et d'intersection de familles d'ensembles. Ce sont présentés dans le tableau suivant : Dans les deux premières opérations, $Ensemble$ est un ensemble d'ensembles et l'union (resp. L'intersection) réalisé l'union (resp. L'intersection) de cette famille. Pour l'intersection, la famille doit être non vide.
Pour l'union ou l'intersection quantifiées, la partie $Ensemble$ est une expression d'ensembles qui dépend des variables Id_liste. Le Prédicat caractérisé le domaine de variation de ces variables, sur lequel se réalisé l'union ou l'intersection.

Expression	Signification
$S_1 \cup S_2 \overset{def}{=\!=} \{x \mid x \in t \wedge (x \in S_1 \vee x \in S_2)\}$	Union.
$S_1 \cap S_2 \overset{def}{=\!=} \{x \mid x \in t \wedge (x \in S_1 \wedge x \in S_2)\}$	Intersection.
$S_1 - S_2 \overset{def}{=\!=} \{x \mid x \in t \wedge (x \in S_1 \wedge x \notin S_2)\}$	Différence d'ensembles.
$union(Ensemble)$	Union généralisée.
$inter(Ensemble)$	Intersection généralisée.
$\bigcup Id\_liste \cdot (Prédicat \mid Ensemble)$	Union quantifiée.
$\bigcap Id\_liste \cdot (Prédicat \mid Ensemble)$	Intersection quantifiée.

TABLEAU II.4 – Les expression d'ensembles.

2.1.5.4 Relations

Les relations sont un cas particulier de construction d'ensembles. Elles sont très utilisées dans les spécifications (invariants, propriétés, etc.). Il s'agit simplement d'ensembles de couples d'éléments. La définition est :

$$E_1 \leftrightarrow E_2 \overset{def}{=\!=} \mathbb{P}(E_1 \times E_2)$$

On définit le domaine d'une relation comme les éléments du premier ensemble E_1 qui sont effectivement en relation avec des éléments du second ensemble E_2 Le codomaine (ou range en anglais) est l'ensemble des points du second ensemble E_2 qui sont en relation avec des éléments du premier ensemble E_1. L'image d'un ensemble par une relation, noté $r[F]$ est l'ensemble des éléments de E_2 qui sont en relation avec les éléments de F par la relation r.

Définition	Expression	Signification
$r \in E_1 \leftrightarrow E_2 \overset{def}{=\!=} \{x \mid x \in E_1 \wedge \exists y \cdot (y \in E_2 \wedge (x \mapsto y) \in r)\}$	$dom(r)$	Domaine.
$r \in E_1 \leftrightarrow E_2 \overset{def}{=\!=} \{y \mid y \in E_2 \wedge \exists y \cdot (x \in E_1 \wedge (x \mapsto y) \in r)\}$	$ran(r)$	Co-domaine.
$r \in E_1 \leftrightarrow E_2 \wedge F \subseteq E_1 \overset{def}{=\!=} \{y \mid y \in E_2 \wedge \exists y \cdot (x \in F \wedge (x \mapsto y) \in r)\}$	$r[F]$	Image.

TABLEAU II.5 – Les types de relations.

Plus formellement, il existe de nombreuses opérations sur les relations. Les opérations sur les ensembles s'appliquent évidemment aux relations (qui sont effectivement des ensembles de couples). En particulier, une relation vide est la même chose qu'un ensemble vide. Néanmoins, il y a quelques opérations spécifiques, dont la plupart sont très usuelles. On a la relation *identité* sur un ensemble, qui à chaque élément associe le même élément. La relation *inverse* d'une relation donnée. On distingue ensuite trois formes de *composition* de relations. Enfin, on a les opérations de *projection* qui construisent les relations entre les ensembles paramètres et les éléments projetés droite ou gauche.
Dans cette syntaxe, la méta-notion Relation est une expression d'ensemble de "type" relation, c'est-à-dire de la forme $\mathbb{P}(E_1 \times E_2)$. Itérations
Il existe des opérations pour itérer une relation, c'est-à-dire la composer séquentiellement avec elle-même un certain nombre de fois. Dans le tableau suivant, on a $r \in s \leftrightarrow s$. Les notations d'itération sont :

2.1.5.5 Restrictions

Enfin, les derniers operateurs sur les relations consistent à restreindre ces relations sur des sous-ensembles du domaine ou du Co-domaine. On a également l'opération de modification d'une rela-

Expression	Signification
$id(Ensemble)$	Relation identité.
$Relation^{-1}$	Inverse d'une relation.
$Relation; Relation$	Composition séquentielle.
$Relation \otimes Relation$	Produit direct.
$Relation \parallel Relation$	Produit parallèle.
$prj_1(Ensemble, Ensemble)$	Première projection.
$prj_2(Ensemble, Ensemble)$	Deuxième projection.

TABLEAU II.6 – Les différentes relations.

Expression	Signification
$r^n \overset{def}{=\!=} r; r^{n-1}\ si\ n > 0$	Itération n fois de r.
$r^0 \overset{def}{=\!=} id(s)$	
$r^+ \overset{def}{=\!=} \bigcup n \cdot (n \in \mathbb{N}_1 \mid r^n)$	Fermeture transitive.
$r^* \overset{def}{=\!=} \bigcap n \cdot (n \in \mathbb{N} \mid r^n)$	Fermeture réflexive transitive.

TABLEAU II.7 – Les itérations.

tion (on dit aussi surcharge d'une relation ou overriding en anglais). Dans le tableau des définitions, on a : $R \in s \leftrightarrow t$, $E \subseteq s$, $F \subseteq t$ *et* $Q \in s \leftrightarrow t$.

Expression	Signification
$E \lhd R \overset{def}{=\!=} \{x, y \mid (x \mapsto y) \in R \wedge x \in E\}$	Restriction de domaine.
$E - R \overset{def}{=\!=} \{x, y \mid (x \mapsto y) \in R \wedge x \notin E\}$	Soustraction de domaine.
$R \rhd F \overset{def}{=\!=} \{x, y \mid (x \mapsto y) \in R \wedge y \in F\}$	Restriction de Co-domaine.
$R \rhd\!\!\!\!- F \overset{def}{=\!=} \{x, y \mid (x \mapsto y) \in R \wedge y \notin F\}$	Soustraction de Co-domaine.
$R \leftarrow\!\!\!\!\lhd Q \overset{def}{=\!=} \{x, y \mid (x \mapsto y) \in s \times t \wedge (((x \mapsto y) \in R \wedge x \notin dom(Q)) \vee (x \mapsto y) \in Q)\}$	Modification ou surcharge.

TABLEAU II.8 – Les restrictions.

2.1.5.6 Fonctions

Les fonctions sont des relations dont chaque élément du domaine n'est associé qu'à un seul élément du Co-domaine. Les fonctions les plus générales sont les fonctions partielles, ensuite, on peut définir les fonctions totales, injectives, surjectives, etc. Il y a plusieurs manières de construire des fonctions autrement que par l'énumération des éléments. Ce sont la "lambda-notation", la notation de fonction constante et la transformée d'une relation en une fonction. A l'inverse, on peut construire une relation à partir d'une fonction. Enfin, on a l'évaluation d'une fonction en un point, comme d'habitude :

2.1.6 Types de données prédéfinis

Les constructeurs d'ensembles présentes au paragraphe précédent (2.1.5.1) sont "génériques" en ce sens qu'on ne fixe pas le type des éléments qu'ils contiennent (de même pour les relations ou les fonctions). Il faut donc pouvoir définir et utiliser des éléments concrets (ou des valeurs de types concrets), avec lesquels on pourra modéliser une application. En Event-B, on peut introduire des

Expression	Signification
$s \nrightarrow t \overset{def}{=\!=} \{r \mid r \in s \leftrightarrow t \wedge \forall x, y, z \cdot (x, y \in r \wedge x, z \in r \Rightarrow y = z)\}$	Fonctions partielles.
$s \rightarrow t \overset{def}{=\!=} \{f \mid f \in s \nrightarrow t \wedge dom(f) = s\}$	Fonctions totales.
$s \rightarrowtail t \overset{def}{=\!=} \{f \mid f \in s \nrightarrow t \wedge f^{-1} \in t \nrightarrow s\}$	Injectives partielles.
$s \rightarrowtail t \overset{def}{=\!=} s \rightarrowtail t \cap s \leftarrow t$	Injectives totales.
$s \twoheadrightarrow t \overset{def}{=\!=} \{f \mid f \in s \nrightarrow t \wedge ran(f) = s\}$	Surjectives partielles.
$s \twoheadrightarrow t \overset{def}{=\!=} s \twoheadrightarrow t \cap s \rightarrow t$	Surjectives totales.
$s \rightarrowtail\!\!\!\!\!\twoheadrightarrow t \overset{def}{=\!=} s \rightarrowtail t \cap s \twoheadrightarrow t$	Bijectives partielles.
$s \rightarrowtail\!\!\!\!\!\twoheadrightarrow t \overset{def}{=\!=} s \rightarrowtail t \cap s \twoheadrightarrow t$	Bijectives totales.

TABLEAU II.9 – Les fonction connues.

Expression	Signification
$\lambda Id\_liste \cdot (Prédicat \mid Expession)$	Lambda-expression.
$Expression \times \{Exprssion\}$	Fonction constante.
$fnc(Expression)$	Transformée en fonction.
$rel(Expression)$	Transformée en relation.
$Expression(Expression)$	Evaluation de fonction.

TABLEAU II.10 – Les expression connue sur les fonctions.

ensembles simplement par leur nom au moment de leur première déclaration. On ne précise pas leurs éléments, mais ils sont supposés non vides.
Ils sont appelés des "ensembles différés" ou ensembles abstraits. Leur véritable contenu ne sera connu qu'à la phase d'implémentation. D'autres ensembles peuvent être définis avec leurs éléments sous forme d'énumération. Ils sont appelés "ensembles définis". D'autre part, on peut construire des expressions d'ensembles avec des types effectifs qu'on peut brièvement citer :

- ***Les nombres entiers***
  qui sont les valeurs numériques habituelles avec laquelle on peut effectuer des calculs arithmétiques. Ils englobent les entiers relatifs (positifs et négatifs). Le type le plus général est noté Z et il représenté l'ensemble mathématique des entiers relatifs. On peut dénoter les valeurs d'entiers avec la syntaxe habituelle : $0, 1, \cdots, 342, \cdots, -4, \cdots$
  - — Sur toutes les valeurs entières on a le prédicat d'égalité et les prédicats de comparaison usuels. Les propriétés d'ordre total (réflexivité, antisymétrie, transitivité) sont vraies pour sur $<$ et $\leq$ (ordre croissant) et $>$ et $\geq$ (ordre décroissant).
  - — Les opérations arithmétiques usuelles sur les entiers $\mathbb{Z}$ ou des sous-ensembles et les opérateurs de somme et produit généralisés et les opérateurs à résultat entier sur des ensembles :
- En Event-B plusieurs type prédéfinis sont exprimés sous forme des ***théories polymorphes*** (cet élément sera bien détaillé en chapitre 5) par exemple :
  (a) **Les booléens** sont un cas particulier de type énuméré. Il est prédéfini par :

  $$BOOL = \{FALSE, TRUE\}$$

  Il ne faut pas confondre les valeurs de cet ensemble des booléens qui font partie de la catégorie des Expressions, avec les Prédicats, qui constituent une autre catégorie syntaxique (expliqué en. 1.5.1). Néanmoins, il est possible de convertir explicitement un prédicat en une valeur booléenne par l'opérateur prédéfini comme suit : $bool(Prédicat)$

Expression	Signification
$succ : \mathbb{Z} \rightarrow \mathbb{Z}$	Fonction successeur.
$pred : \mathbb{Z} \rightarrow \mathbb{Z}$	Fonction prédécesseur.
$- : \mathbb{Z} \rightarrow \mathbb{Z}$	Moins unaire.
$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$	Addition.
$- : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$	Soustraction ou différence.
$* : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$	Multiplication ou produit.
$/ : \mathbb{Z} \times (\mathbb{Z} - \{0\}) \rightarrow \mathbb{Z}$	Quotient de la division entière.
$mod : \mathbb{Z} \times \mathbb{N}_1 \rightarrow \mathbb{Z}$	Reste de la division entière.
$x^y : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Opération puissance entière.
$\sum Id\_liste \cdot (Prédicat \mid Expression)$	Somme d'expressions quantifiées.
$\prod Id\_liste \cdot (Prédicat \mid Expression)$	Produit d'expressions quantifiées
$card(Expression)$	Nombre d'éléments d'un ensemble quelconques.
$min(Expression)$	Minimum d'un ensemble fini non vide d'entiers.
$max(Expression)$	Maximum d'un ensemble fini non vide d'entiers.

TABLEAU II.11 – Les opérateurs sur les entiers.

(b) **Les séquences** (ou suites) qui sont un cas particulier de constructeurs de fonctions dont le domaine est un intervalle d'entiers et dont le Co-domaine est l'ensemble support des éléments des séquences considérées. De ce fait, les séquences héritent de toutes les opérations définies sur les relations et les fonctions. Le résultat de ces opérations est une séquence si les opérations préservent les conditions propres au domaine des séquences. Comme le domaine des séquences est fini, on caractérise uniquement les séquences finies d'éléments. les opérateurs sur les séquences définis récursivement par cas et d'autres

Expression	Signification
$seq(E) \overset{def}{=\!=} \bigcup n \cdot (n \in \mathbb{N} \mid 1..n \rightarrow E)$	Séquences finies d'éléments de E.
$seq_1(E) \overset{def}{=\!=} seq(E) - \emptyset$	Séquences non vides.
$iseq(E) \overset{def}{=\!=} seq(E) \cap (\mathbb{N}_1 \rightarrowtail E)$	Séquences injectives.
$iseq_1(E) \overset{def}{=\!=} iseq(E) - \emptyset$	Séquences injectives non vides.
$perm(E) \overset{def}{=\!=} seq(E) \cap (\mathbb{N}_1 \rightarrowtail\!\!\!\!\!\rightarrow E)$	Séquences bijectives ou permutations.

TABLEAU II.12 – Les séquences.

opérateurs permettent de sélectionner des sous-parties d'une séquence donnée et les opérateurs lorsque la séquence s en paramètre est non vide ($s \neq []$) comme suit :

Expression	Signification
$size([]) = 0$ $size(a \rightarrowtail s) = size(a) + 1$	longueur de séquence.
$[] \frown t = t$ $(a \rightarrowtail s) \frown t = a \rightarrowtail (s \frown t)$	Concaténation.
$[] \leftarrowtail b = [] \rightarrowtail b$ $(a \rightarrowtail s) \leftarrowtail b = a \rightarrowtail (s \leftarrowtail b)$	L'ajout à droite.
$rev([]) = []$ $rev(a \rightarrowtail s) = rev(a) \rightarrowtail s$	inversion de séquence.
$conc([]) = []$ $conc(u \rightarrowtail s) = u \frown conc(s)$	concaténation généralisée.
$s \uparrow n \stackrel{def}{=\!=} (1..n) \lhd s$ à condition : $n \in 0..size(s)$	la restriction à partir du bas.
$s \downarrow n \stackrel{def}{=\!=} \lambda i \cdot (i \in 1..(size(s) - n) \mid s(n + i))$ à condition : $n \in 0..size(s)$	la restriction à partir du haut.
$first(s) \stackrel{def}{=\!=} s(1)$	Premier élément.
$last(s) \stackrel{def}{=\!=} s(size(s))$	Premier élément.
$tail(s) \stackrel{def}{=\!=} s \downarrow 1$	Eléments sauf le premier.
$front(s) \stackrel{def}{=\!=} s \uparrow (size(s) - 1)$	Eléments sauf le dernier.

TABLEAU II.13 – Les opérateurs sur les séquences.

2.2 La méthode Event-B

La méthode Event-B [12] est une évolution de la méthode B, dont l'objectif est de modéliser des systèmes fermés. Un système fermé est un système modélisé avec l'ensemble de toutes les interactions avec son environnement. Il n'y a donc plus besoin de modéliser les entrées ou sorties pour communiquer avec l'environnement. Pour cela, les opérations B sont remplacées par des événements en Event-B.
Contrairement aux opérations B qui sont appelées par des composants, les événements Event-B se déclenchent spontanément si une condition (appelée garde) devienne vraie. Contrairement au B classique, Event-B offre également la possibilité d'exprimer certaines contraintes dynamiques telles que des contraintes de vivacité (liveness).
Ces points font que le B classique est mieux approprié pour le développement des logiciels et Event-B pour le développement des systèmes [258].

2.2.1 L'origine de l'Event-B

Comme il est cité par avant l'Event-B est considéré étant une extension de la méthode B classique et par la suite on donne une aperçue complète sur l'amélioration des méthodes formelles développées par Abrial jusqu'à l'apparence de la méthode Event-B.

2.2.1.1 De Z à B classique

En dehors de B, le langage de spécification Z joue actuellement l'un des rôles les plus importants dans les spécifications des logiciels. La structure principale de spécification utilisée dans Z est celle du schéma Z. Les schémas dans Z sont créés pour représenter les opérations ainsi que des données

abstraites. L'exemple dans la figure II.1 est donné pour illustrer la notation : La première partie du schéma est la partie déclarative, où on introduit toutes les variables. La deuxième partie contienne la spécification elle-même.
L'une des critiques qui pourraient être formulées à propos de Z, c'est que le cahier des charges des schémas Z(comme dans la Figure II.1) pourrait ne pas être très intuitif. En relation avec les caractéristiques de fonctionnement, les conditions utilisées dans un schéma Z doivent être considérés comme des invariants déclaratives. D'une part leur rôle n'est pas de raisonner sur les opérations, d'autre part à une spécification B les propriétés d'invariance de la machine sont utilisées comme hypothèses dans lesquelles la cohérence des opérations doit être affichée.
Un autre inconvénient des schémas Z, c'est que le rôle des préconditions n'est pas tout à fait clair. En général, une précondition est définie comme la partie d'un schéma impliquant exclusivement des variables de l'état d'avant. Cependant, en dehors de ces contraintes, le comportement du schéma peut aussi être interprété comme infaisable. Ces problèmes ont été résolus par Abrial [12] lors de la définition de la méthode B.

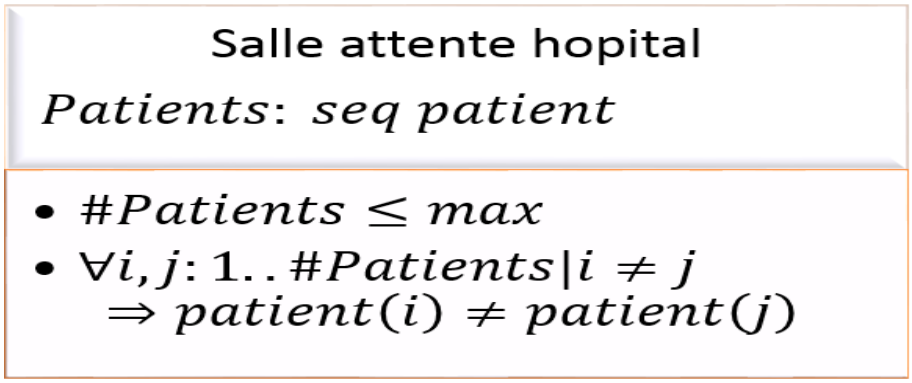


FIGURE II.1 – Exemple d'un schéma en Z.

La méthode B a sa propre histoire. Elle a été initiée par J.R. Abrial [12] au début des années 80, elle est née d'un besoin industriel et elle fait partie des réussites les plus importantes de l'application des méthodes formelles dans ce domaine, particulièrement la Mise en service de la ligne de métro 14 (METEOR) en 1998 [12]. Le logiciel critique embarqué a été modélisé, prouvé et généré à partir des spécifications formelles B, plusieurs métros en cours de rénovation font appel à la méthode B pour le développement des logiciels sécuritaires.
L'objectif d'un développement B est d'obtenir un modèle prouvé. Depuis le développement du processus de preuve, sa maîtrise est un enjeu crucial. Même si un outil de preuve est disponible, sa puissance efficace est limitée par les résultats classiques sur les théories logiques et nous devons distribuer la complexité des preuves sur les composantes du développement actuel, par exemple par raffinement. Le raffinement a le potentiel de réduire la complexité du processus de preuve, tout en permettant la traçabilité des exigences.

2.2.1.2 De la méthode B à l'Event-B

La méthode B est une méthode formelle de modélisation et de construction de logiciels proposée par Jean-Raymond ABRIAL [243]. Elle s'appuie sur les concepts mathématiques de la théorie des ensembles. Elle propose une démarche qui couvre toutes les étapes de développement d'un logiciel, depuis la spécification jusqu'à l'implémentation. Cette méthode est basée sur les notions de machine abstraite, de raffinement vérifiée par la preuve. Le cycle de développement commence par la construction d'une machine abstraite. Cette machine abstraite B est constituée, d'une part, de variables d'état, d'un invariant exprimant des propriétés sur les variables et d'autre part, d'opérations qui décrivent les transformations d'états correspondant à des changements de valeur des variables. Le développement d'un système complexe ne peut se faire en une seule étape [251, 259] et B pro-

pose une méthode de construction incrémentale par raffinements successifs.L'abstraction est une technique générale employée pour maîtriser la complexité des systèmes, Un développement en B consiste, alors, à construire un premier modèle abstrait, dans lequel certains détails ne sont pas pris en compte. Les étapes suivantes consistent à compléter le modèle abstrait pour obtenir un modèle concret. A l'issue de chaque étape, la vérification consiste à prouver des obligations de preuves engendrées par l'outil de développement, l'Atelier B [192].
Le raffinement en B(voir Figure II.2) est une technique de développement incrémental permettant de préciser progressivement les données et les opérations de la spécification abstraite de départ. L'objectif de cette technique est de passer progressivement d'une spécification non-déterministe abstraite à une spécification concrète déterministe, automatiquement traduisible dans un langage de programmation.

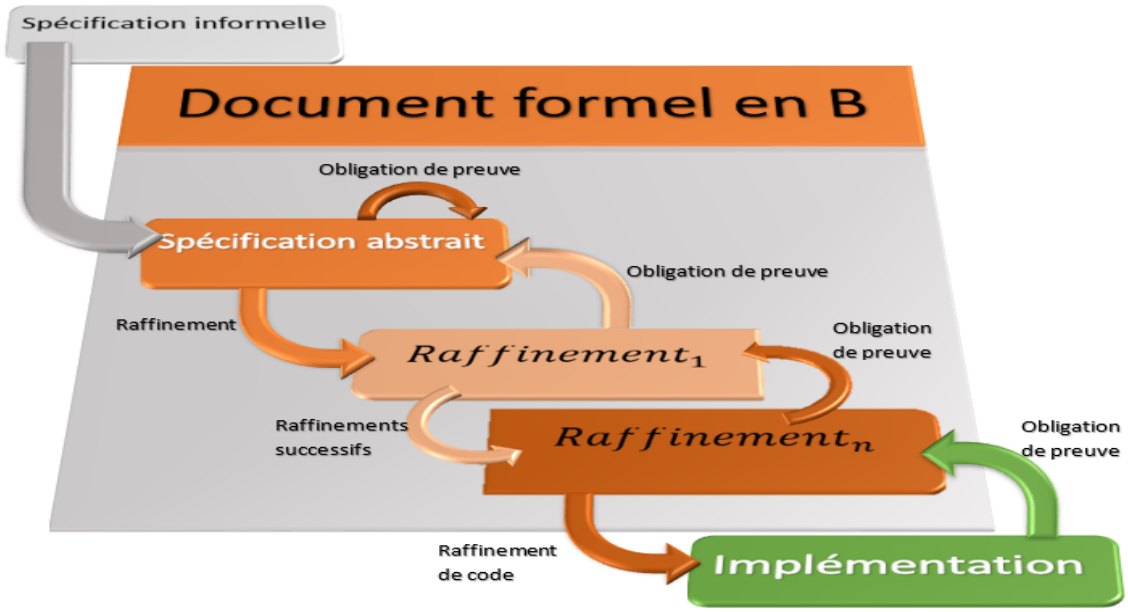


FIGURE II.2 – Le raffinement en B.

Les principales différences entre Event-B [12] et B classique sont les suivantes (voir Figure II.3) :

- La notion d'opération est remplacée par la notion d'événement. Un événement a une garde (sans précondition) et peut être déclenché si la garde est vraie.
- Le remplacement des notions de structuration des machines (USES, SEES, INCLUDES, IMPORTS ...) par la seule notion du contexte, qui regroupe les constantes et les ensembles de bases, ainsi que des axiomes associés.
- La disparition de certains types de données et leurs opérateurs associés, notamment les séquences (*seq*), les structures (*struct*) et les arbres (*btree*) dans les anciennes versions mais ils ont apparu sous formes des théories polymorphes.
- La disparition de certaines substitutions. Chaque événement en Event-B a en fait une forme très simple. La forme la plus compliquée peut-être exprimée par un seul ANY contenant des assignations (déterministes et non déterministes) parallèles. Les variables du ANY sont les « paramètres » de l'événement. Il existe deux formes simplifiées, une pour des événements sans paramètre et une pour des événements sans paramètre et sans garde.
- Une notion adaptée du raffinement, permettant l'introduction de nouveaux événements pour séparer un événement en plusieurs dans la machine raffinée et inversement de fusionner plusieurs événements.

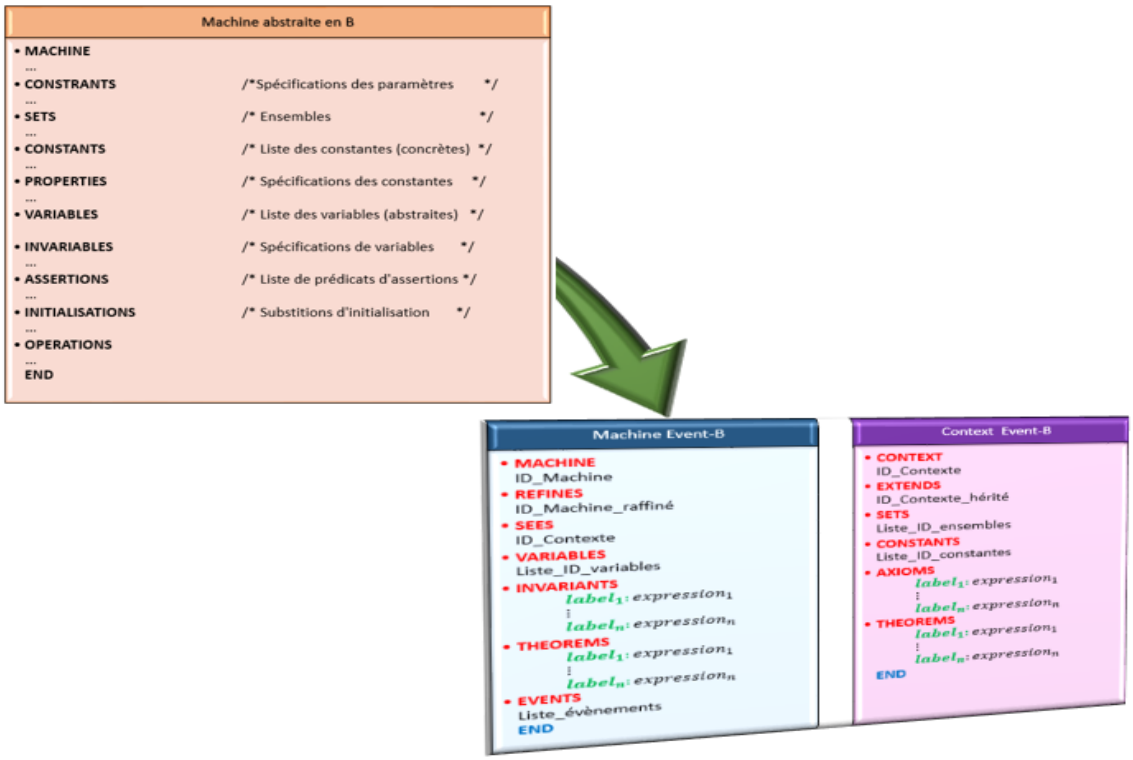


FIGURE II.3 – Le passage de la machine abstraite en B vers un modèle en Event-B.

2.2.2 La pragmatique de l'Event-B

La notation de modélisation Event-B a été conçue pour être « simple et facile à enseigner » [260]. Il est destiné à la modélisation de systèmes complexes **par** un support d'outil avec un nouvel aspect important d'utilisation. Dans ce qui suit, on a brièvement décrié certains des choix importants réalisés lors de la conception de la notation Event-B. Ces critères discutés par Hallerstede dans [260] :

2.2.2.1 La modélisation et la programmation

Des choix importants concernant la modélisation et la programmation ont été faites lors de la conception de la notation de l'Event-B. Hallerstede prétend que la modélisation et la programmation ne sont vues que des activités de natures différentes avec des objectifs différents. Un programme doit être exécuté, alors que l'exécution n'est pas nécessaire pour un modèle. Ainsi, de nombreux traitements à base des langages de programmation ont été omises afin de réduire la complexité de la notation et de mettre davantage l'accent sur le raisonnement. Toutefois, cela peut augmenter les efforts nécessaires pour préciser certains aspects de systèmes, y compris le séquencement :

(a) ***Composition séquentielle :*** la composition séquentielle peut compliquer les obligations de preuve et les rendre difficile à comprendre, de façon que la méthode Event-B ne les supporte pas.

(b) ***Instructions conditionnelles :*** Ce ne sont pas pris en charge dans l'Event-B. Les instructions conditionnelles posent un grand défi pendant la génération des obligations de preuve de raffinement, car il est difficile de **connaitre** sur ce qui branche **un raffinement concret** correspondent à quelles branches dans l'abstraction. Au lieu de cela, l'Event-B adopte une approche selon laquelle chaque branche correspond à un événement distinct.

2.2.2.2 Undefinedness

Les expressions conditionnellement définis sont fréquemment utilisés lors de l'élaboration des modèles. Cela pose un défi majeur lorsque la logique sous-jacente représente la logique du premier ordre à deux valeurs. Pour faire face à ce problème, l'Event-B estime que la bonne définition des expressions au niveau de typage. Le typage fonctionne en deux passes. La première passe si les expressions sont correctement tapées indépendamment de savoir si elles sont définies. Le second est pour crée des obligations de preuve de bonne définition qui doit être générés par la preuve [260]. Par exemple, l'expression $1 \div 0$ est correctement saisi, mais ne sont pas bien défini car il ne peut être démontré que $0 \neq 0$ **(racine non null**.

2.2.2.3 Paramétrage

Les modèles peuvent dépendre de nombreux paramètres, par exemple, le nombre de composants dans une structure. Les contextes en Event-B sont utilisés pour des machines comme de support à paramétrer à l'aide des ensembles et constantes. Ceux-ci peuvent être instanciés, et si elles satisfont aux axiomes du contexte, les théorèmes qui en découlent peuvent être facilement utilisés.

2.2.2.4 Transparence

La notation de modélisation en Event-B n'est pas finalisée, et devrait évoluer en fonction des différents besoins et domaines d'application. Le formalisme est ouvert à des extensions et modifications. Hallerstede souligne toutefois que des précautions doivent être prises pour éviter de compliquer la théorie existante, et les concepts doivent être interprétés d'une manière simple et sans ambiguïté [260].

2.2.3 Les modèles Event-B

Un modèle Event-B (voir Figure II.4) est décomposé en deux parties : le contexte qui contient la partie statique du modèle et la machine qui contient la partie dynamique du modèle. Cette séparation permet d'indiquer à une machine donnée les contextes qu'elle « voit ».
Un modèle Event-B peut contenir des contextes seulement, des machines seulement ou les deux. Dans le premier cas, le modèle représente une structure mathématique pure. Le deuxième cas représente quant à elle un modèle non paramétré. Le dernier cas représente un modèle paramétré par les contextes. Les deux sections suivantes présentent les deux composants avec leurs différentes clauses telles qu'elles sont déclarées dans la plateforme RODIN.

2.2.3.1 Le contexte

On peut considérer le contexte comme la partie statique du modèle contenant plusieurs clauses :

- La clause *CONTEXT* représente le nom du composant qui devrait être unique dans un modèle.
- La clause *EXTENDS* déclare la liste des contextes qu'étend le contexte décrit. Un contexte peut étendre un autre contexte en rajoutant de nouvelles constantes et de nouvelles propriétés.
- La clause *SETS* définit les ensembles porteurs du modèle. Ces ensembles non vides servent à typer le reste des entités du modèle.
- La clause *CONSTANTS* contient la liste des constantes utilisées par le modèle.
- La clause*AXIOMS* définit les propriétés liées aux constantes et notamment leurs types.
- La clause *THEOREMS* exprime des propriétés qui peuvent être déduites à partir des propriétés présentées dans la clause *AXIOMS*.

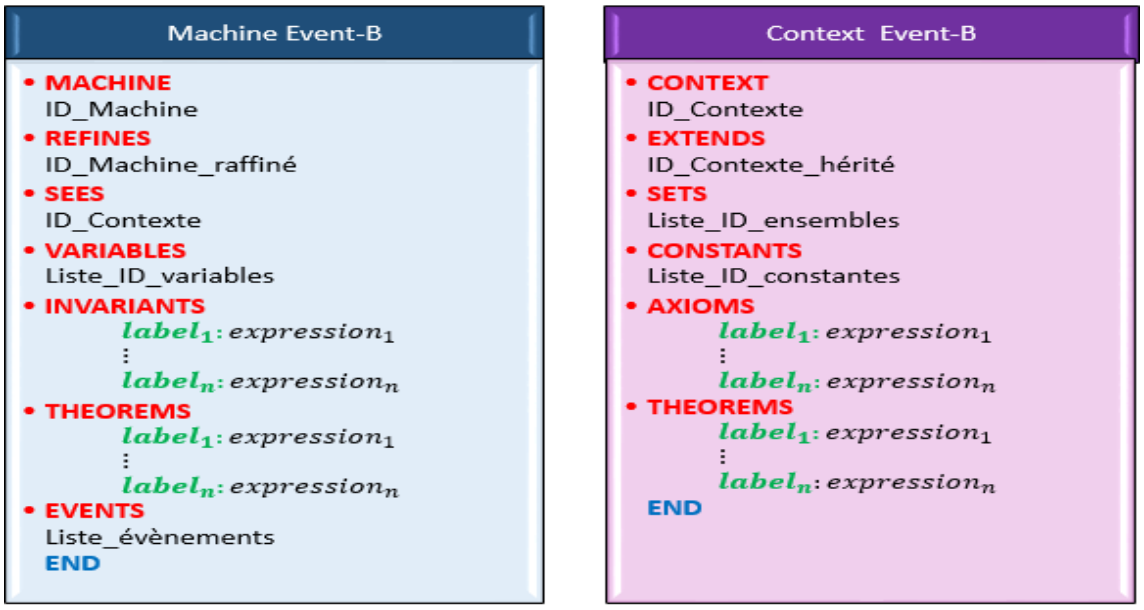


FIGURE II.4 – La structure d'un modèle en Event-B.

2.2.3.2 La machine

La machine est la partie dynamique du modèle et elle est constituée de plusieurs clauses :

- La clause *MACHINE* représente le nom du composant qui devrait être unique dans un modèle.
- La clause *REFINES* déclare le nom de la machine raffinée par la machine décrite.
- La clause *SEES* spécifie la liste des contextes « vus » par la machine. Dans ce cas, la machine peut utiliser les constantes et les propriétés figurant dans les contextes.
- La clause *VARIABLES* contient la liste des variables du modèle.
- La clause *INVARIANTS* définit les propriétés d'invariance du modèle telles que des informations sur les types des variables et des propriétés de sûreté.
- La clause *THEOREMS* exprime des propriétés qui peuvent être déduites des propriétés d'invariance de la machine et des propriétés présentes dans les clauses *AXIOMS* et *THEOREMS* du contexte vu. En outre, cette clause peut contenir des propriétés que l'on souhaite prouver afin de les employer dans la preuve des invariants du modèle.
- La clause *VARIANT* définit l'expression du variant du modèle.
- La clause *EVENTS* contient la liste des événements qui opèrent une ou plusieurs substitutions sur la valeur des variables.
- Parmi ces événements, l'événement *INITIALISATION* donne une valeur initiale aux variables.

Event-B propose des structures pour exprimer les systèmes réactifs comme un ensemble d'actions appelées événements et maintenir une liste d'affirmations appelés (inductive) des invariants. Ces invariants formulent des propriétés de sûreté de fonctionnement [172].

⁂ ***Les événements***
Un événement Event-B correspond à un changement d'état dénotant une transition dans le système modélisé. Il est essentiellement composé d'une garde (la clause *WHEN*) qui définit les conditions nécessaires au déclenchement de l'évènement et d'une action (la clause *THEN*) qui définit l'évolution des variables d'état. Notons que plusieurs gardes d'événements peuvent être vraies en même temps. Néanmoins, un seul événement peut se déclencher et le choix de cet événement est non déterministe. Un évènement peut posséder des paramètres (des variables locales) définis dans la clause *ANY* [258].

2.2.4 Notion du raffinement en Event-B

Le raffinement en Event-B consiste à développer le système de manière incrémentale en partant d'un modèle abstrait qui constitue une spécification du système. A chaque étape de raffinement, des détails du système sont rajoutés graduellement dans un modèle concret qui doit préserver la fonctionnalité et les propriétés des modèles plus abstraits. Ces détails rajoutés apparaissent dans l'état du système en ajoutant des variables et dans le comportement en détaillant les événements de l'abstraction ou en ajoutant de nouveaux événements. Notons que les nouveaux événements raffinent un événement particulier de l'abstraction qui est l'évènement vide (appelé "SKIP"). Des obligations de preuve sont générées à chaque étape de raffinement afin d'assurer la correction du raffinement [258].

2.2.5 Hiérarchie de modélisation en Event-B

Dans l'Event-B, les modèles sont exprimés dans la forme de machines et contextes. Les machines contiennent des variables, des invariants et des événements, alors que les contextes consistent en les ensembles, les constantes et les axiomes. Les machines peuvent être déclarées pour voir des contextes, **de façon** ils sont libres de faire usage des valeurs contenu dans le contexte. Les contextes peuvent étendre un autre contexte, alors que les machines peuvent raffiner l'un l'autre. Cela résulte en une chaîne de machines et contextes **où** les modèles sont développés dans une manière **incrémentale appelée** "stepwise". L'hiérarchie de la modélisation peut être illustrée comme dans Figure II.5, où les versions les plus concrètes sont montrées à droite([245]).

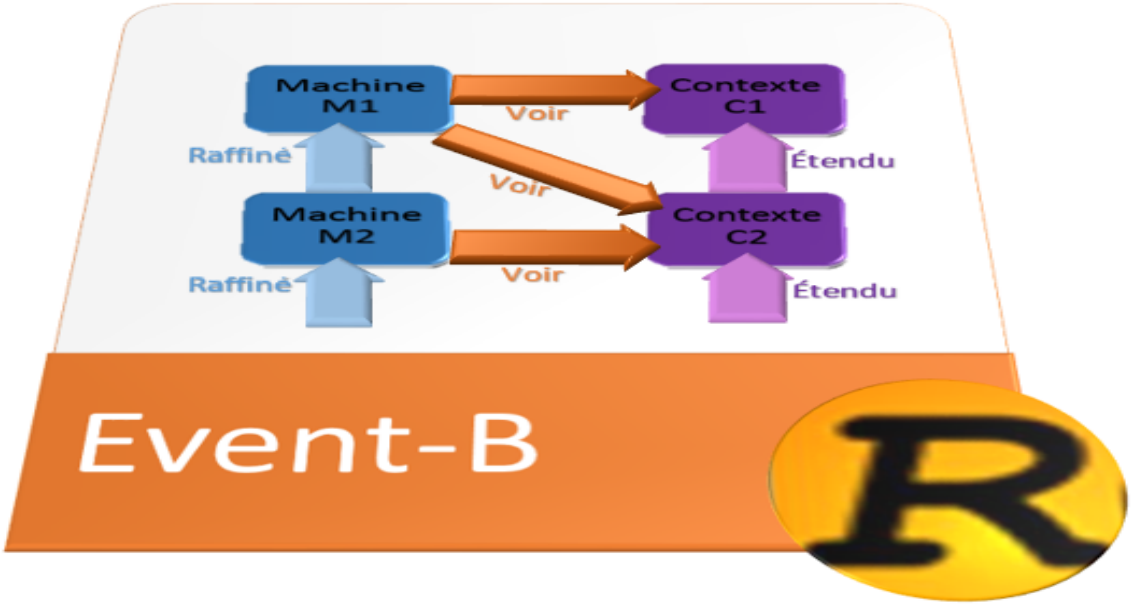


FIGURE II.5 – La relation entre le contexte et la machine de Event-B.

Voici quelques règles(selon [245]) de visibilité qui doivent être suivies par des machines et des contextes :

- Une machine peut voir explicitement plusieurs contextes (ou aucun contexte du tout).
- Un contexte peut étendre explicitement plusieurs contextes (ou aucun contexte du tout).
- La notion d'extension de contexte est transitive : un contexte C1 étendre explicitement un contexte C2, étend implicitement tous les contextes étendus en C1.
- Lorsqu'un contexte C1 s'étend un contexte C2, puis les ensembles et les constantes de C2 peuvent être utilisé en C1.

- Une machine voit implicitement tous les contextes, prolongés par un contexte explicitement vu.
- Lorsqu'une machine m voit un contexte C, cela signifie que les ensembles et les constantes de C peuvent servir dans M.
- Les relations **« raffine » et « voit »** mises en place qu'ils ne doivent pas entraîner de tout cycle.
- Une machine raffine uniquement au plus une autre machine.
- L'ensemble des contextes explicitement ou implicitement vus d'une machine doit être aussi importante que celle de l'abstraction de cette machine.

2.3 L'outil RODIN

RODIN [17, 261] est un environnement de modélisation intégrée pour **l'Event-B**. Il offre des installations et des outils pour développer et raisonner sur les modèles d'une manière réactive inspirés par des environnements modernes de développement intégré dans Eclipse(Interactive Development Environment (IDE)), tels que [262]. Lors de l'élaboration des programmes Java utilisant Eclipse, l'utilisateur n'a pas besoin d'ouvrir le processus de compilation. Au contraire, l'IDE réagit aux changements dans le code d'une manière transparente qui fournit une rétroaction efficace pour le développeur. De manière analogue, en RODIN, tout en développant un modèle d'un système complexe, la vérification statique, la preuve la génération des obligations et la gestion sont réalisées de façon transparente pour fournir une rétroaction immédiate pour le modélisateur. La combinaison de vérification statique et la production d'obligation de preuve dans DODIN peut être considéré comme un vérificateur statique prolongé [263] pour l'Event-B. Plus précisément, la plate-forme RODIN fournit les capacités à :

— Développer des modèles dans l'Event-B en précisant les contextes et les machines,

— Analyser les modèles au moyen de vérification statique qui comprend la syntaxe et la vérification de type,

— Analyser sémantiquement modèles au moyen d'obligations de preuve produites selon le cas,

— Effectuer preuve mathématique afin de vérifier la cohérence du modèle.

Afin de trouver un bon équilibre entre la facilité d'utilisation et l'efficacité, Rodin est conçu pour satisfaire aux exigences suivantes [17] :

- ***Le Feed-back en temps de modélisation "Design-Time Feedback" :*** l'outil répond rapidement aux changements et fournit un feed-back qui peut être facilement lié à des modèles.
- ***La génération des différentes Obligations de preuve pendant les phases de vérification "Distinct Proof Obligation Generation and Verification phases" :*** l'outil découple la modélisation et de prouver, tout en maintenant le lien entre les deux activités (c.à.d , traçabilité) dans le cas où des preuves automatiques sont échouées.

2.3.1 Architecture

La Figure II.6 montre une vue de haut niveau de l'architecture interne de RODIN. L'outil peut être divisé en quatre composants distincts qui sont décrits ci-dessous avec plus de détails nécessaire pour les prochaine chapitre :

2.3.1.1 Le cœur de RODIN

Il contient le référentiel RODIN et le constructeur de RODIN. Le référentiel gère la persistance entre les éléments de données (des objets Java, par exemple, les obligations de preuve) et leur

stockage dans les **fichiers** eXtensible Markup Language (XML) (par exemple, fichiers obligation de preuve). Le constructeur (analogue au constructeur Java dans l'IDE Eclipse Java) prévu des emplois en fonction des modifications apportées aux fichiers dans le référentiel.

2.3.1.2 La bibliothèque de paquetage en Event-B

La syntaxe du langage mathématique Event-B est spécifiée par une grammaire attribué et implémentée dans un module dédié pour l'arbre de syntaxe abstraite (Abstract Syntax Tree (AST)). Le module de preuve de séquence (SEQuent Prover (SEQP)) fournit l'infrastructure nécessaire pour bien charger les preuves.

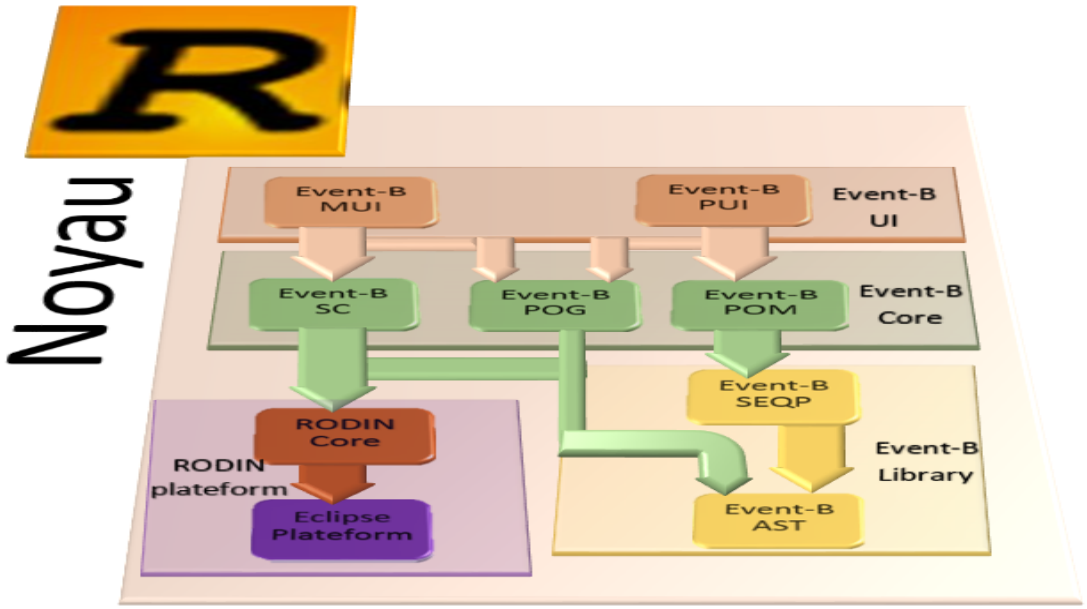


FIGURE II.6 – L'architecture de l'outil RODIN.

2.3.1.3 Le cœur Event-B

Il contient le vérificateur statique (Static Checker (SC)), le générateur de l'obligation de preuve (Proof Obliagtions Generator (POG)) et le gestionnaire d'obligations de preuve (Proofs Obligations Manager (POM)). Le vérificateur statique analyse de la bonne saisie des contextes et des machines en termes de syntaxe. Le générateur d'obligations de preuve génère obligations de preuve à partir d'éléments statiquement vérifiés du modèle, y compris axiomes, de théorèmes, invariants et l'ensemble des événements. Enfin, le gestionnaire de l'obligation de preuve conserve la trace des obligations de preuve et leurs preuves.

2.3.1.4 L'interface d'utilisateur de l'Event-B

Il contient le modèle d'interactivité graphique pour Event-B. Il offre deux perspectives distinctes : l'interface d'utilisateur de modélisation (Modelling User Interface (MUI)) et l'interface d'utilisateur de prouver (Proving User Interface (PUI)). La figure II.7 décrit la chaîne d'outils pour le développement de modèles en Event-B à l'aide de la plate-forme RODIN.

2.3.2 La Philosophie d'outillage de RODIN

La modélisation est une activité complexe, et est une étape extrêmement importante dans le développement de systèmes complexes et fiables. Le raisonnement peut améliorer considérablement la

compréhension d'un modèle particulier. Un support d'outil efficace devrait fournir un cadre pratique pour la création de modèles et de raisonner sur eux. Il convient également de faire la transition nécessaire entre les activités de modélisation et de raisonnement le plus transparent que possible.

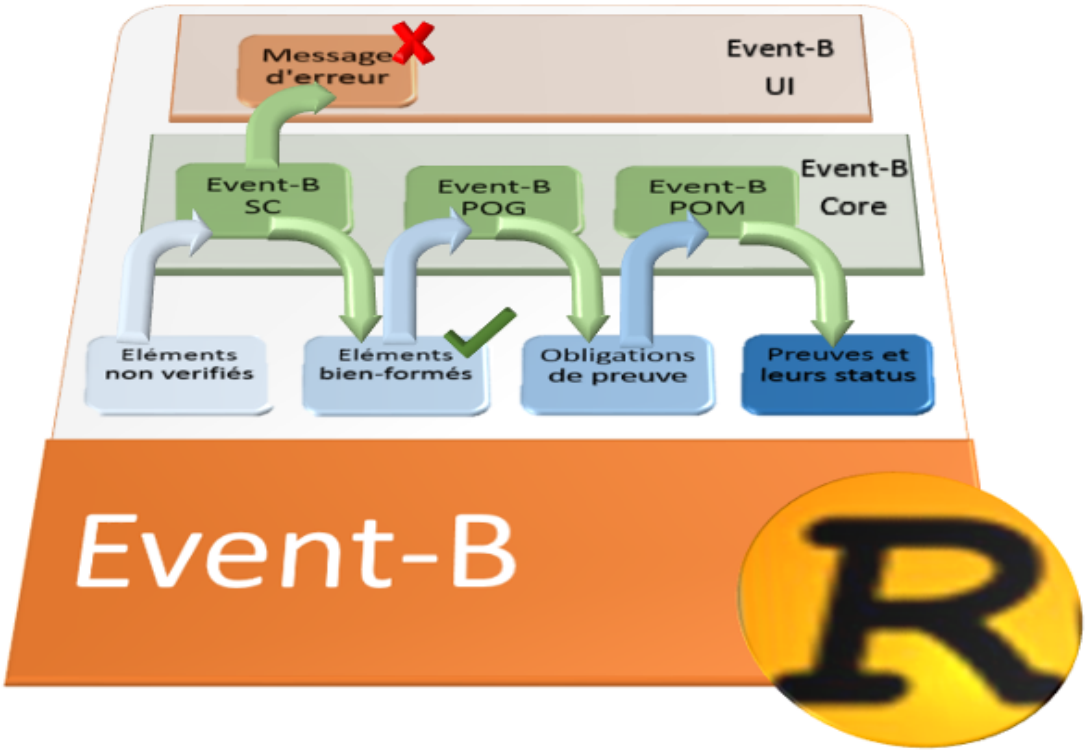


FIGURE II.7 – La chaine à outil dans RODIN pendant le développement En Event-B.

Hallerstede [260] donne un aperçu des différents choix et décisions faites lors de la conception de la notation et de l'environnement de modélisation pour l'Event-B. En outre, l'Event-B est**équippé par** un ensemble d'outils **"toolset"** (appelé RODIN Figure II.8) vise à satisfaire les deux conditions suivantes [261] : La première consiste que l'outil est très sensible et immédiatement fournit des informations liées au modèle Une autre exigence est que les évaluations devraient facilement se rapportent au modèle en question.Et la deuxième est important car elle permet à l'utilisateur de faire la distinction entre les activités de modélisation et la génération de preuve. Ceci est particulièrement important lorsque la génération des preuves est échouée, car il permet à l'origine de l'obligation de preuve à tracer plus facilement.

2.3.2.1 L'éditeurs

L'outil RODIN fournit aux éditeurs pour des contextes et des machines. Les éditeurs sont conçus pour refléter la structure de leur fichiers respectives. Depuis les fichiers de contexte et de la machine ont une structure XML, leurs éditeurs respectifs ont une vue sous forme de l'arbre, et sont à base de formulaires (Cette décision de conception particulière a été prise pour tenir compte des extensions possibles à la base de données RODIN). Il est un éditeur de texte pour RODIN appelé Camille [10]. Cependant, cet éditeur souffre de plusieurs bugs qui empêchent sa facilité d'utilisation.

2.3.2.2 L'outillage

Outillage se réfère à la collection d'outils qui fonctionnent par les fichier RODIN. Figure II.5 décrit les trois outils disponibles dans le répertoire RODIN. La chaîne d'outils RODIN se réfère aux différentes étapes de l'outillage :

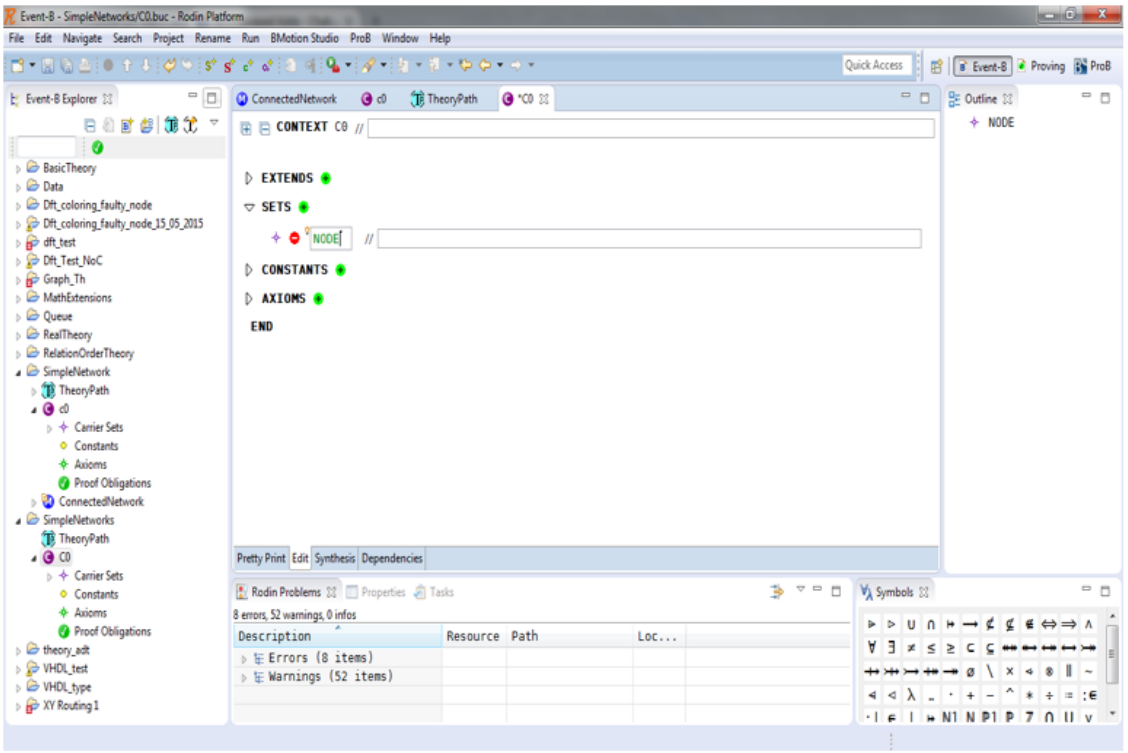


FIGURE II.8 – L'outil de RODIN.

(a) ***La vérification statique***
Les composants en Event-B (les contextes et les machines) sont statiquement vérifiées pour les erreurs de syntaxe et de la saisie. Chaque fichier en RODIN a deux versions :

i. *Une version non vérifiée :* Le fichier non contrôlé est la version qui peut être modifié par l'utilisateur. Les machines et des contextes non contrôlées ont le extensions '**.bum** '(la machine non vérifiée à base B) et' **.buc**' (le contexte non vérifié à base B), respectivement.

ii. *Une version statiquement vérifié :* les machines et contextes vérifiés ont les extensions '**bcm**' (à savoir, B vérifié machine) et '**bcc**' (à savoir, B vérifié contexte) respectivement. Le but de l'outil de la vérification statique(Static Checker ou SC) est de créer des fichiers statiquement vérifiés ('**.bcm** 'et' **.bcc**') de leurs homologues non vérifiées (' **.bum** 'et' **.buc**'), et l'élimination des éléments mal formés pendant ce processus. Le vérificateur statique passe par tous les sous-éléments de fichier non vérifié, et génère leurs homologues statiquement vérifié si toutes les conditions requises sont remplies par chaque élément. Les fichiers statiquement vérifiés sont donc l'objet d'outillage ultérieure.

(b) ***La génération des Obligations Preuve***
Ceci se rapporte à la génération des obligations de preuve des **en** éléments bien formés **pour** des contextes et des machines. La production d'Obligation fonctionne sur les contextes et les machines statiquement vérifiés. Les obligations de preuve produites dans RODIN sont présentées dans la section 2.3.4, et sont plus minutieusement justifiée dans [260].

(c) ***La gestion de preuve***
Ceci se rapporte à la gestion de la relation entre les obligations de preuve et leurs preuves. Une obligation de preuve peut être : dans l'attente, déchargée, a examiné (une obligation de preuve est examinée si elle a été inspecté par l'utilisateur et est marqué à évacuer plus tard). L'état d'une obligation de preuve est déterminé par l'état de sa preuve (complète ou incomplète). Le prouveur RODIN modifie l'état d'une preuve par l'application des règles de preuve, ou en invoquant **des** proveurs tiers (ML, PP [264] et, plus récemment, le prouveur Isabelle en Event-B [265]).

2.3.2.3 L'élaboration réactive

La plate-forme RODIN propose un environnement réactif de modélisation [33, 34] similaire à des environnements de développement intégrés modernes (de l'IDE), d'où la décision de mettre en œuvre de RODIN sur le dessus de l'IDE Eclipse. L'utilisateur travaille sur un modèle est constamment mis à jour sur l'état de ses preuves. Pour ce faire, les outils de répertoire RODIN fonctionnent de manière réactive par :

(a) La vérification des modèles pour les erreurs de syntaxe et de saisie,

(b) La génération des obligations de preuve, **et dans** le cas échéant,

(c) Mise à jour le statut de ses preuves en appelant prouveurs automatiques ou réutilisation de vieilles tentatives de preuve [34].

La nature réactive de RODIN pose de nombreux défis en matière de preuves. Mehta [34] décrit les différentes questions et son approche de traiter avec eux (réutilisation preuve et de restructuration).

2.3.2.4 Les obligations de preuve

Les obligations de preuve sont au cœur de la modélisation en Event-B. La désignation des obligations de preuve et leur structure est crucial **pour** faciliter l'activité de **la** modélisation [260]. Les obligations de preuve sont facilement traçables à leur élément correspondant dans des contextes et des machines, pour créer la transition entre la modélisation et la preuve plus facilement.

2.3.3 Event-B le langage mathématique

La syntaxe utilisée pour écrire des modèles en Event-B peut être décomposée en deux niveaux qui seront détaillé par la suite :

2.3.3.1 Syntaxe externe

Ce niveau de la syntaxe correspond aux parties non fixés de la définition du contexte. Elle est utilisée pour spécifier les éléments de contextes et des machines individuelles., d'autre part, elle est définie par une base de données d'éléments dont les relations sont définies par un graphe. Grace à la base de données RODIN [260,261], la syntaxe externe est facilement extensible. Cela a facilité le développement de plusieurs plug-ins utiles, par exemple, le plug-in modularisation [266] et Records plug-in [267].

2.3.3.2 Syntaxe interne

Ce niveau de la syntaxe correspond aux parties fixé par le contexte. Cette syntaxe est utilisée pour spécifier les formules mathématiques correspondant aux axiomes, invariants, des gardes et des actions. La syntaxe interne de l'Event-B est spécifié à l'aide d'une grammaire attribuée, et définie dans le sous-module (AST) de RODIN, La syntaxe interne, avant RODIN version 2.0, a été liée avec le sous-module(AST), et ne pouvait être prolongée aussi facilement que la syntaxe externe. Cependant, RODIN 2.0 fourni un analyseur dynamique pour la syntaxe interne qui peut être facilement augmentée avec une nouvelle syntaxe [268].

2.3.4 Obligations de preuve

Afin de garantir la correction de notre modèle, il est indispensable de le prouver. Pour y parvenir, le générateur d'obligations de preuve génère automatiquement des obligations de preuve(voir Figure II.9). Une obligation de preuve définit ce que doit être prouvé pour un modèle. Les obligations de

preuve sont de la forme $H => G$: le but G est à démontrer en partant de l'ensemble H des hypothèses.
Les règles des obligations de preuve sont au nombre de onze, et pour chacune le générateur d'obligations de preuve génère une forme spécifique [12].
Etant donné : un événement *evt*, un axiome *axm*, un théorème *thm*, un invariant *inv*, une garde *grd*, une action *act*, un variant ou une witness *x*. Les règles d'obligation de preuve pouvant être générées pour ces éléments sont illustrés dans le tableau II.15.

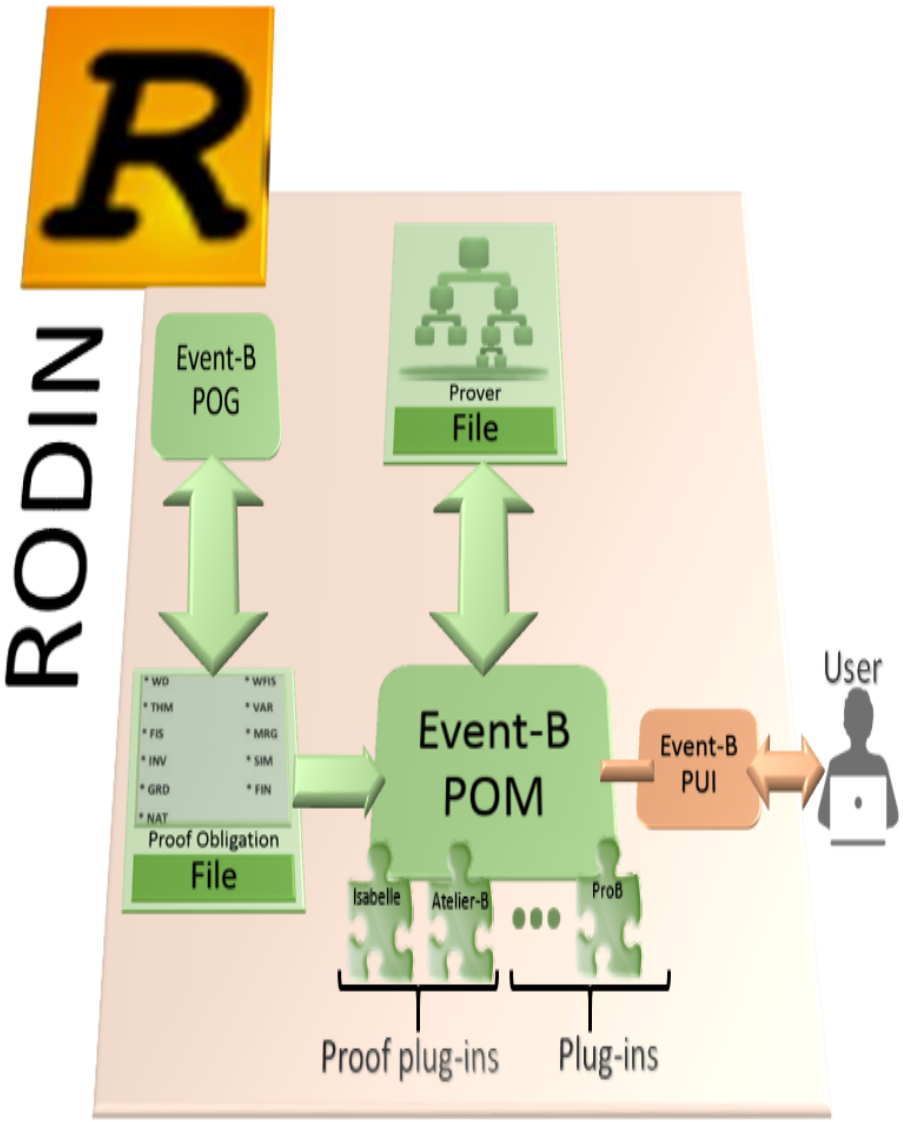


FIGURE II.9 – Architecture de la preuve à l'intérieur de RODIN.

2.3.4.1 L'infrastructure de Preuve

Le gestionnaire d'obligation de preuve (POM), gère la relation entre les obligations de preuve et leurs preuves. Le gestionnaire de preuve (Proof Manager (PM)) est chargé pour gérer et de maintenir des preuves, et fournit des services importants pour le POM. Pour chaque obligation de preuve, il construit un arbre de preuve dont la racine est le séquent de l'obligation elle-même. Le gestionnaire de preuve travaille à la fois automatiquement (sans intervention de l'utilisateur) et de manière interactive (avec intervention de l'utilisateur et, éventuellement, avec entrée). Une description détaillée de l'architecture de prouveur de RODIN est décrit par Mehta dans sa thèse [34]. Résumant

Règle	Description	Forme
Préservation de l'invariant INV	Assure que chaque invariant dans une machine donnée est préservé par tous les événements	«evt/inv/INV »
Faisabilité FIS	se rassurer qu'une action non déterministe est faisable	« evt/act/FIS »
Renforcement des gardes	Les gardes des événements concrets sont plus fortes que celles des abstractions	« evt/grd/GRD »
La fusion du garde MRG	Assure que la garde d'un événement concret fusionner deux événements abstraites est plus forte que la disjonction des gardes des événements abstraits.	« evt / MRG »
La simulation SIM	chaque action dans un événement abstrait est correctement simulée dans le raffinement correspondant. Autrement dit, l'exécution d'un événement concret n'est pas en contradiction avec son abstraction.	« evt/act/SIM »
Variant numérique NAT	sous une condition que les gardes d'un événement convergent ou anticipé sont vérifiées, le variant numérique proposé est un entier naturel.	« evt/NAT »
La règle finis de variant set FIN	dans une condition où les gardes d'un événement convergent ou anticipé sont vérifiées, l'ensemble variant proposé est un ensemble fini.	« evt/FIN »
Variant VAR	chaque événement convergent diminue le variant numérique proposé ou l'ensemble variant proposé. De plus, chaque événement anticipé n'augmente pas le variant numérique proposé ou l'ensemble variant proposé.	« evt/VAR »
Témoin non déterministe WFIS	chaque témoin (witness) proposé dans la clause WITH (si elle existe) d'un événement concret existe vraiment.	« evt/x/WFIS »
Théorème THM	Un théorème écrit dans une machine ou dans un contexte est vraiment prouvable.	« thm/THM »
well-definedness WD	Règle de bonne définition des axiomes, théorèmes, invariants, gardes, actions, variant et witness. Selon la nature de l'élément.	«axm/WD », « thm/WD », «inv/WD », « grd/WD », « act/WD», «VWD», «evt/x/WWD»

TABLEAU II.14 – Les règles d'Obligation de preuve.

les éléments clés de cette architecture :

- **Les arbres de preuve** qui sont des structures récursives basées sur les nœuds de preuve. Un nœud dans l'arbre de preuve représente par un nœud unique ainsi que l'arbre de preuve (ou sous-arbre) ancrée à ce nœud, voir Figure II.10. Chaque nœud de l'arbre de preuve a un séquent. Il peut également avoir une règle de preuve justificative et une liste de nœuds enfants. Un nœud de l'arbre de preuve peut être soit dans l'attente, si sa règle de preuve est nulle, par conséquent, la liste des nœuds enfants est nulle, ou, soit dans le non-attente, s'il a une règle de preuve non nulle, et les nœuds enfants correspondent au résultat de l'application de la règle de preuve à son séquent.

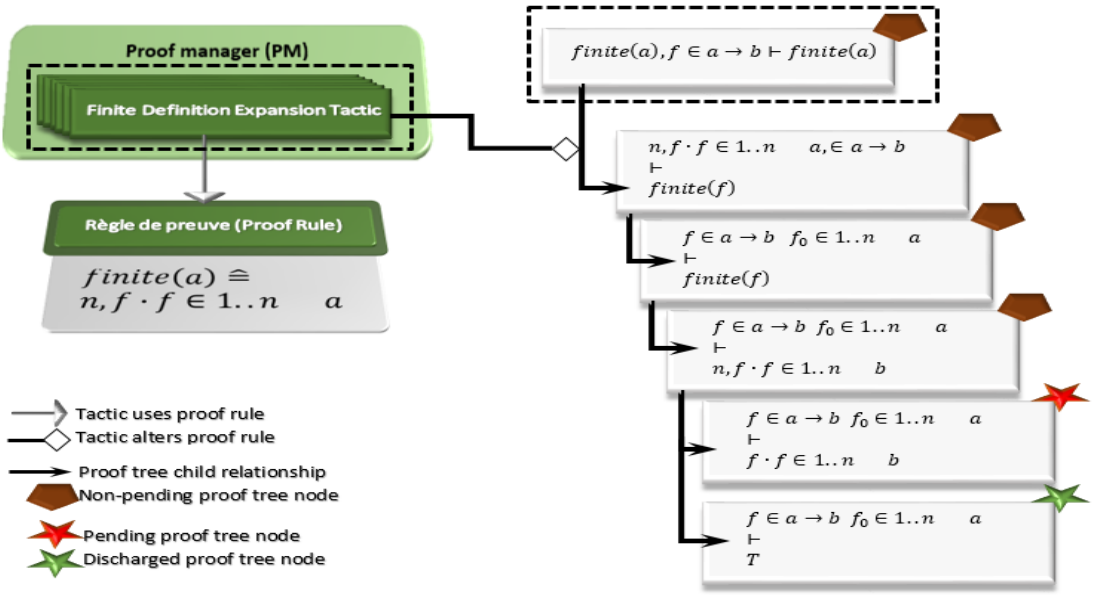


FIGURE II.10 – Le gestionnaire de preuve (proof manager).

- **Des tactiques** qui ont été introduits par Robin Milner au début des années 1970 pour la Logic of Computable Functions (LCF) démonstrateur [269]. Ils fournissent un mécanisme uniforme pour manipuler arbres de preuve. Une tactique pourrait être une couvre autour d'une règle de preuve dans ce cas, **Elle est appelée** *une tactique de base*. **D'autre part,** ***des tactiques secondaires (subtils)***sont plus structurés et peuvent être utilisés pour définir une stratégie de preuve [34]. **Prenant l'exemple de la tactique qui applique une boucle récursive d'**une autre tactique jusqu'à l'échoue.
- **Raisonneurs** qui sont des générateurs concrets de règles de preuve. Un exemple **de la règle de** preuve est la règle d'introduction-conjonction suivante qui est bien documentée comme suit : Des règles de preuve concrets peuvent être générés par l'instanciation de manière appropriée la méta-variables H ; P et Q (Notez que H représente un ensemble de formules ou précisément l'ensemble des hypothèses). Utilisant un langage de type Java simple, 2.3.5 décrit les raisonneurs d'interface générale obéissent [34] :
  - La méthode appliquée vérifie si la règle de preuve est applicable au séquent donné avec l'entrée fournie (entrée pourrait être par exemple un terme à instancier une formule universellement quantifiée.), **si elle est vraie, alors elle** génère une règle de preuve concrète qui sera le justification de l'étape la preuve, **Sinon** pas de changement se produit dans l'arbre de preuve.
  - Le gestionnaire preuve peut être étendue avec de nouvelles raisonneurs et tactiques. Il y a un protocole bien défini pour les deux extensions. **Les** raisonneurs sont également utilisés pour intégrer **prouveur** externe. L'idée consiste à encapsuler un appel à un prouveur

externe comme une application raisonneuse. L'appel est réussi si **le prouveur** externe déclenche le séquent, **c.à.d** si elle constate une preuve complète pour le séquent. Une limitation est que l'information sur la façon dont le prouveur externe allait de la preuve (par exemple, des hypothèses retenues **qui ne sont pas toujours à la disposition du gestionnaire preuve)**.

2.3.4.2 Les Limites de preuve

En dépit d'être optimisé pour la réutilisation preuve [34], l'architecture actuelle avant [36,37] a des limites suivantes :

- Afin d'ajouter une nouvelle règle de preuve, il a été nécessaire de mettre en œuvre un raisonneur et une tactique d'emballage. Par conséquent, un certain niveau de compétence avec le langage de programmation Java ainsi que la connaissance de l'architecture RODIN étaient nécessaires ;
- Après une nouvelle règle est ajoutée, **la solidité de preuve fournie par le prouveur est en augmentation** avec la nouvelle règle qui doit être établie. On ne sait pas comment cela peut être réalisé au niveau du code Java. L'utilisation d'outils de vérification Java, par exemple, JML [270] n'a pas été adoptée par RODIN à compter du moment de la rédaction de cette thèse.

2.3.5 Les prouveurs internes

Le résultat d'une preuve est visualisé aux utilisateurs par un jeton qui s'appelle le smiley (voir Figure II.10). Le smiley peut avoir trois états différentes interprètent le statut de la preuve. Les statuts de la preuve suivent le couleur de smiley et son état sachant que :

(a) Si le smiley mécontent et coloré en rouge, cela vaut dire que la preuve contient un ou plus qu'un nœud de l'arbre de preuve est non chargé (vérifie par une règle de preuve correspondante à l'obligation de preuve Proof Obliagtions (PO)) par le prouveur de RODIN.

(b) Si le smiley est en bleu , cela vaut dire que toute les nœuds de l'arbre de preuve non chargés restent forcés à l'état revoir par un utilisateur selon ces besoins de preuve.

(c) Si le smiley est content et coloré en vert , cela vaut dire que toute les nœuds de l'arbre de preuve sont chargés (vérifie par une règle de preuve correspondante à l'obligation de preuve PO).

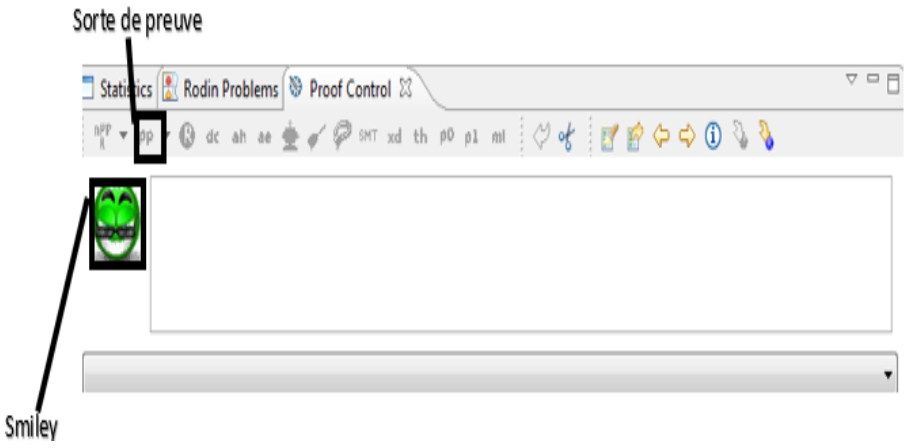


FIGURE II.11 – Le barre de contrôle de preuve Sous RODIN

Par défaut, il existe deux sortes de prouveurs internes qui ont été intégrées dans l'environnement RODIN sont les suivants :

2.3.5.1 Le prouveur de prédicats (Predicate Prouver)

Ce démonstrateur (aussi appellé en anglais PP voir Figure II.11) est construit autour d'une hiérarchie des prouveurs. Il contient une procédure de décision pour la logique propositionnelle et une procédure semi-décisionnelle pour la logique du premier ordre. Un autre élément important est le traducteur de la théorie des ensembles au logique de premier ordre. Il est construit en conformité avec la construction de la théorie des ensembles décrits dans le travail : [15].

2.3.5.2 Le Prouveur de Mono-Lemme (Mono-Lemma prouver)

C'est un démonstrateur (en anglais ML voir Figure II.11) basé sur des règles utilisées dans le Solver Logic qui est le compilateur interprète utilisé pour B. PP a été initialement développé pour valider les nombreuses règles de preuve de ML. ML et le PP font partie de l'Atelier-B [264], qui fournit l'infrastructure d'essai pour B.

2.3.6 Le prouveur externe Atelier B

Cette partie est destinée aux utilisateurs de l'Event-B et de **l'Atelier-B** connaissant les principes de la preuve, mais n'ayant pas encore utilisé le prouveur de l'Atelier B. Il s'agit d'une "visite guidée" qui permet de savoir où sont les fonctionnalités de preuve de l'Atelier B, et où se trouve leur documentation.

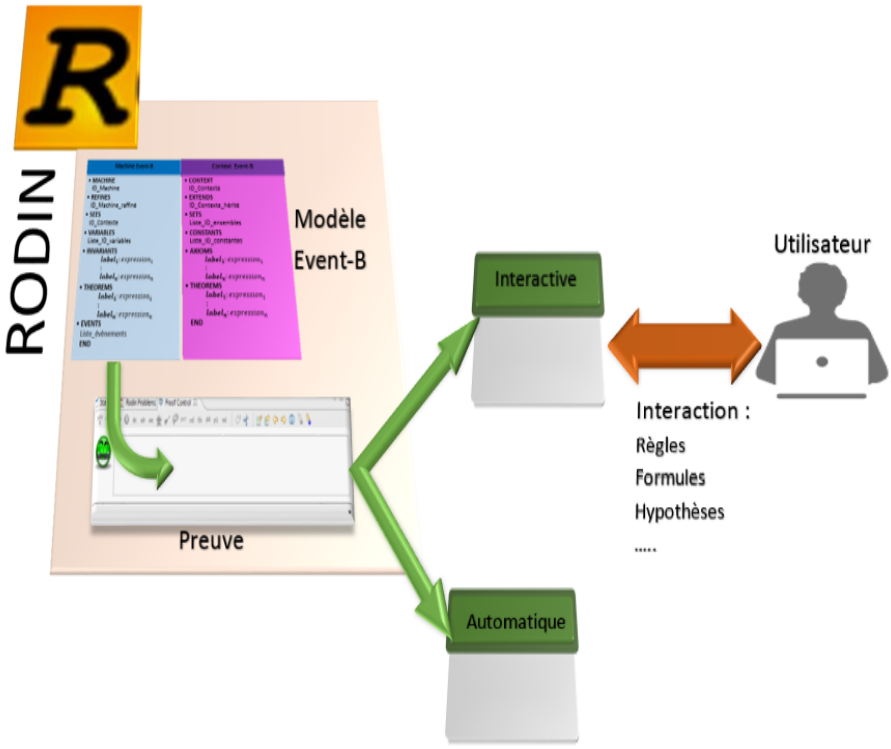


FIGURE II.12 – Les genres de preuve en Atelier-B plug-in.

2.3.6.1 Les type de preuves

La preuve sert à trouver des erreurs et ce n'est pas de programmer, elle peut être partiellement automatique. Le prouveur ne sait pas démontrer qu'une obligation est fausse par les mécanismes de preuve qui choisissent les règles à utiliser à partir de la base de règles représentant l'ensemble des connaissances mathématiques du prouveur(voir Figure II.12) . Le cœur de preuve désigne la base et les mécanismes les forces Rapide,0,1,2,3 regroupent les mécanismes en niveaux pour valider une obligation de preuve et par conséquence possèdera un état : Proved,Unproved. Les commandes de preuve contrôlent le prouveur, en automatique comme en interactif alors qu'une PO possède un niveau de démonstration automatique ou interactive. Le schéma général de l'outil : un cœur de preuve, des commandes, un pilote automatique ou interactif, le prouveur interactif communique avec son interface en mode ligne pour orienter la preuve sans ajout de connaissance non validé et les règles de prouve manuelles sont des règles non validées alors qu'une démonstration manuelle peut n'employer que des règles validées. Par la suite on donne les principes du prouveur automatique et manuel (interactive).

(a) ***La preuve automatique***
Ce que l'on désigne par prouveur automatique, c'est en fait le mode de pilotage du cœur de preuve est essayée sur chaque obligation de preuve. La démonstration interactive au contraire, permet à l'opérateur de décider lui-même quelles commandes de preuve sont appliquées par exemple le lancement du cœur de preuve en force 0, c'est-'a-dire avec les mécanismes de la force 0, sur chaque obligation de preuve. Les démonstrations réussies se résument à l'application de règles issues de la base de règles, choisies par les mécanismes de la force 0. Pour le mode automatique, il suffit de mémoriser la force maximale tentée pour chaque obligation, c'est le niveau de preuve automatique de l'obligation.

(b) ***La preuve interactive***
Le cœur de preuve est toujours disponible en mode interactif, simplement ce n'est plus la seule commande possible. Il y a d'autres commandes de preuve possibles, qui permettent d'appliquer spécifiquement une règle, de faire de la preuve par cas. Ce sont les commandes qui pilotent la preuve. Elles peuvent être soit des appels au cœur de preuve, soit des actions directes de preuve (par exemple une déduction). Les commandes de preuve applicables à chaque obligation de preuve sont mémorisées par l'outil, la séquence des commandes qu'il a choisies pour démontrer une obligation est mémorisée avec l'état de preuve.

2.3.6.2 Le développement et la preuve

Dans cette partie on entame à connaitre quelles sont les activités de preuve dans le développement d'un projet informatique utilisant la méthode B et l'Atelier B en appliquant une étude sur un exemple d'un projet constitué d'une spécification et son implantation (le programme concret) probablement réalisé de la manière classique (notre contribution permet aussi de valider le projet de façon que le programme sera intégré étant des modèles de l'environnement de l'exécution) suivante (voir Figure II.13) :

(a) Ecrire le modèle (contexte et machine) abstrait en fonction du cahier des charges ;

(b) Contrôler la formalisation correcte du besoin ;

(c) Lancer le prouveur automatique sur ce modèle abstraite ;

(d) S'il reste des obligations de preuve non automatiquement démontrées, contrôler rapidement qu'elles soient justes. Si certaines sont fausses, le modèle abstrait est incohérent, il faut le corriger ;

(e) Ecrire l'implantation ;

(f) Relire cette implantation par rapport au modèle abstrait ;

(g) Lancer le prouveur automatique sur l'implantation ;

(h) S'il reste des obligations de preuve non démontrées, contrôler qu'elles soient justes. Si certaines sont fausses l'implantation n'est pas correcte, il faut le corriger ;

(i) Faire la démonstration formelle des obligations de preuve restantes dans le modèle abstrait et dans l'implantation à l'aide du prouveur interactif.

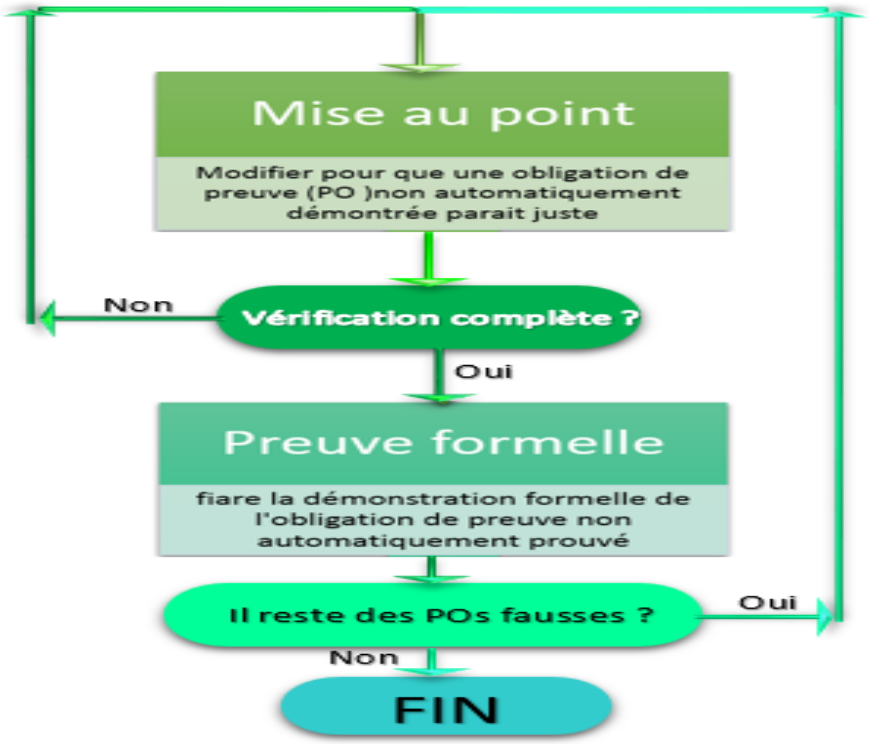


FIGURE II.13 – La méthode générale de preuve dans RODIN.

Dans le processus de développement ci-dessus, les étapes 3, 4, 7, 8 et 9 sont les étapes de preuve. On voit que la preuve formelle complète est faite à la fin : il faut éviter les démonstrations longues tant que les composants risquent de devoir être modifiés. C'est pourquoi il y a deux phases bien distinctes dans l'activité de preuve en Event-B : la mise au point des composants par vérification des obligations de preuve et la preuve formelle finale. Cette distinction se retrouve toujours quelle que soit la méthode de développement utilisée. Notant que dans les étapes 3 et 7, il faut utiliser le prouveur automatique configuré pour être assez rapide (force 0, voir paragraphe suivant) et attendre qu'il termine pour passer à l'étape suivante. Savoir si on se place en phase de mise au point ou en phase de preuve finale est essentiel. Cette méthode par phase peut se représenter par le schéma présneté précédement.

(a) ***La mise en points de preuve***
La phase mise au point désigne implicitement la mise au point du point de vue preuve : les méthodes générales pour écrire et contrôler des projets en Event-B. Faut-il avoir fini la phase de mise au point de tous les composants du projet avant de passer en phase de preuve formelle ? Faut-il finir complétement la mise au point d'un composant avant d'écrire le composant suivant ? La réponse volontairement restera imprécis sur ce sujet, qui dépend de la taille et de la structure du projet. Tout au plus peut-on dire qu'il ne faut pas attendre d'avoir écrit tous les composants du projet avant d'aborder les problèmes de preuve, et qu'il ne faut pas entreprendre trop tôt la preuve formelle d'un composant. Durant la phase de preuve formelle, on suppose ne plus avoir à retoucher les composants, sauf si une obligation de preuve supposée juste est en fait fausse. Dans ce cas l'impact des modifications sur les d´démonstrations déjà faites peut provoquer des pertes de temps. C'est pourquoi la phase de mise au point est

très importante. Les changements de phase de preuve sont des étapes délicates. Lors de ces changements, attention aux pièges suivants :

— ***S'assurer que les composants ont bien leur forme définitive avant la phase de preuve formelle***. Il est en effet courant d'écrire les composants dans une version réduite ou incomplète pour une mise au point rapide, en prévoyant une étape de finition. Cette finition doit être faite avant la preuve formelle.

— ***En phase de preuve formelle***, s'assurer que toutes les obligations de preuve peuvent être présumées justes. En effet, si une obligation de preuve fausse est découverte durant la phase de preuve formelle, l'opérateur est tenté de poursuivre cette phase après avoir modifié un composant, alors qu'il est impératif de refaire une phase de mise au point.

(b) ***Les règles de force de preuve***
L'opérateur ne s'attend jamais à ce qu'un ordinateur conçoive et réalise les programmes à sa place, parce que l'ordinateur ne peut pas deviner ce qu'il faut obtenir. Dans le domaine de la preuve au contraire, ce qu'il faut obtenir est clair : nous voulons des démonstrations des énoncés à prouver à partir d'un ensemble de règles connues. Il n'existe malheureusement pas d'algorithme qui produise la démonstration de tout énoncé correct, les démonstrateurs automatiques et en particulier celui de l'Atelier B appliquent donc un ensemble de tactiques plus ou moins heuristiques qui peuvent échouer ou aboutir. Si une démonstration est obtenue elle est correcte mais l'échec d'une tactique ne prouve pas que l'énoncée est faux. Une différence importante entre la preuve et d'autres tâches plus classiques comme par exemple la conception de programmes est donc la possibilité d'aboutir par le travail automatique d'un ordinateur. Pour cette raison il est toujours souhaitable de faire travailler les prouveurs automatiques sur les projets à démontrer quel que soit le temps de calcul nécessaire, parallèlement au travail de preuve manuel. Les tactiques employées en preuve automatique sont généralement d'autant plus coûteuses en temps de calcul qu'elles sont capables de trouver des démonstrations complexes. De plus les tactiques les plus complètes peuvent souvent provoquer des boucles infinies dans les démonstrations. C'est pourquoi les différentes tactiques du prouveur de l'atelier B ont été regroupées en forces. Les différentes forces sont les suivantes : Les

Force	Temps indicatifs par lemme	Taux de performance
0	**toujours moins de 10 secondes**	**70%**
1	de quelques secondes à 2 ou 3 minutes	+1%
2	de quelques minutes à quelques dizaines minutes	+3%
3	de quelques dizainesminutes à plusieurs heures	+1%
Rapide	moins de trois seconds	+1%

TABLEAU II.15 – Les Forces de preuve dans Atelier-B.

temps ci-dessus sont indicatifs, ils concernent surtout les premières obligations de preuve de chaque opération. En effet les obligations de preuve suivantes ont beaucoup d'hypothèses en commun avec les premières et le traitement de ces hypothèses est factorisé. Les performances sont très indicatives ; elles sont indiquées en pourcentage d'obligations de preuve démontrées sur un projet "standard" entièrement juste. Les performances des forces 1, 2 et 3 sont indiquées en gain par rapport à la force précédente parce que les forces 1, 2 et 3 s'emploient toujours en séquence à partir de la force 0. Ainsi les forces les plus élevées ne peuvent traiter des lemmes démontrés dans une force inférieure, ce qui économise le temps de calcul et limite le risque de déclencher des boucles infinies. La force "Rapide" s'emploie seule. La force 0 est considérée comme l'optimum entre l'efficacité et le temps de calcul. C'est cette force qui doit être utilisée pour tenter de démontrer les obligations de preuve avant même de les lire, afin de limiter leur nombre. Elles sont en effet très nombreuses, on compte en moyenne une obligation de preuve par ligne de code exécutable produite. La force "Rapide" n'a pas des

performances suffisantes pour cet emploi. Les forces 1, 2 et 3 s'emploient plutôt en parallèle durant les phases de mise au point et de preuve formelle, en espérant que certaines obligations de preuve seront démontrées automatiquement avant d'avoir été traitées manuellement. Les principes d'utilisation des forces du prouveur de l'Atelier B sont les suivants :

— ***Employer la force 0 :*** ne jamais examiner une obligation de preuve avant d'avoir tenté de la démontrer avec le prouveur automatique en force 0.

— ***"Occuper" les ordinateurs :*** si vous disposez d'ordinateurs inemployés sur lesquels l'Atelier B est installé, il est toujours utile de lancer le prouveur automatique en force 1, 2 ou 3 sur ceux-ci pour démontrer des obligations de preuve justes de votre projet.

— ***Ne pas attendre :*** n'attendez pas que le prouveur automatique en force 1, et ou 3 termine le traitement de votre projet pour commencer les phases de mise au point ou de preuve formelle. Le prouveur automatique est aussi employé en preuve interactive. Ceci peut sembler paradoxal, mais ce que nous appelons preuve interactive est en fait une preuve semi-automatique dans laquelle les actions de l'opérateur s'intercalent entre des appels au cœur de preuve. Il faut donc choisir la force utilisée également en preuve interactive, elle conditionne toutes les interventions du cœur de preuve dans les démonstrations manuelles. Dans la majorité des cas, il est conseillé d'employer la force 0 ; la force 1 est parfois utilisée aussi.

(c) ***L'examen d'une obligation de preuve***

Avant de procéder à l'examen d'une obligation de preuve, le choix de cette obligation doit être de telle manière à commencer par celles qui ont le plus de chances de détecter des erreurs. Dans la phase de mise au point, il faut obtenir le plus rapidement possible la démonstration intuitive de chaque obligation de preuve. Commençant par présenter les principes de l'examen d'une obligation de preuve ; puis la manière de parcourir et visualiser ces obligations de preuve.

La méthode d'examen d'une obligation de preuve est la suivante :

i. ***Interprétation du but :*** examiner les différentes variables présentes dans le but et retrouver le sens de chacune d'entre elles dans son interprétation physique.

ii. ***Justification intuitive :*** déterminer pour quelles raisons ce but doit être vrai dans le contexte du composant.

iii. ***Sélection des hypothèses :*** isoler dans l'obligation de preuve les hypothèses correspondantes à ces raisons.

iv. ***Démonstration intuitive :*** faire une démonstration intuitive de l'obligation de preuve réduite à ces hypothèses.

v. ***Notes et essais :*** la démonstration intuitive faite peut donner des idées pour la démonstration formelle, sur les causes de l'échec de la démonstration automatique, etc. Dans cette dernière étape, on cherche à profiter de ces idées. Eventuellement, une démonstration formelle rapide sera recherchée et généralisée à d'autres obligations, permettant ainsi de réduire le nombre d'obligations à lire.

Cette liste décrit la méthode préconisée pour la phase de mise au point et les étapes à suivre pour chaque obligation de preuve. L'idée maîtresse de cette méthode en cinq étapes consiste à interpréter l'obligation de preuve dans le contexte du composant à prouver. On bénéficie ainsi de toute la démarche intellectuelle qui a été faite pour comprendre ou construire le composant, et qui deviendra à terme un ensemble de démonstrations rigoureuses.

(d) ***La phase de preuve formelle***

La phase de preuve formelle consiste à démontrer avec le prouveur interactif les obligations de preuve qui restent après la phase de mise au point. Pour être efficace dans ces preuves interactives, il faut tirer le meilleur parti possible des tactiques automatiques de preuve ; c'est-à-dire **utiliser autant que possible les appels au prouveur**. Ces appels se font par lancer le

cœur de preuve dans la force courante ; dans certains cas on essaie également la commande *Predicate Prover (pp)* qui peut aussi décharger automatiquement le but. Le cœur de preuve est prévu pour obtenir les meilleurs résultats possibles en preuve automatique, c'est pourquoi il tente des tactiques exploratoires avant de conclure à l'échec. Ces tactiques sont nuisibles en preuve interactive quand elles échouent, car elles conditionnent le but en échec. Le plus souvent, il s'agit de preuves par cas.
La méthode générale pour faire les démonstrations formelles et le schéma de figure II.14 explique comment "avancer" dans la preuve en traitant chaque nouveau but qui se présente. Quand il ne reste plus de but à traiter, la preuve est finie. Détaillant les étapes :

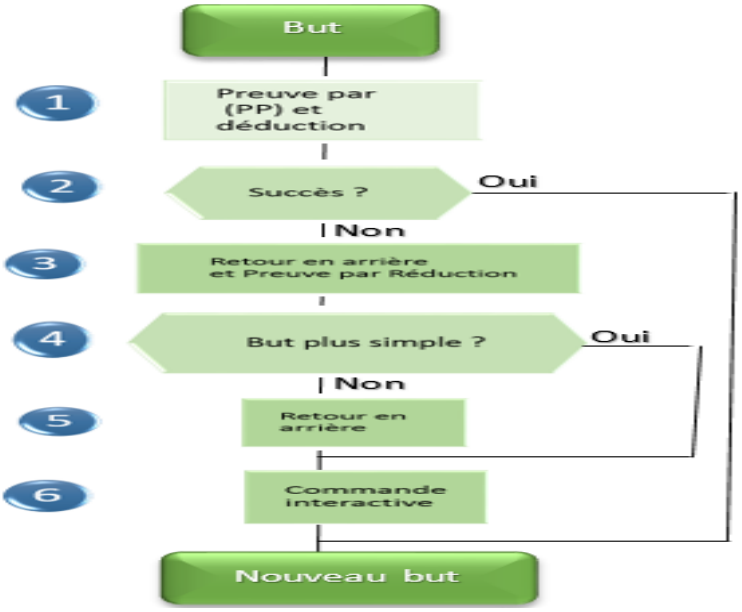


FIGURE II.14 – Les étapes de la phase de la preuve formelles dans Atelier-B.

- ***Etape 1*** Pour chaque nouveau but, il faut lancer le prouveur pour ne jamais perdre de temps s'il est automatiquement démontré. Il ne faut pas faire cette étape lorsqu'il est certain que le prouveur ne peut pas aboutir : c'est le cas en particulier du but de départ. Sa preuve échoue forcement puisque l'obligation de preuve n'a pas été démontrée automatiquement. Penser aussi à essayer la preuve (*pp*) à cette étape, éventuellement précédée de déduction pour faire monter des hypothèses (Sachant que le prouveur de prédicat réagit généralement mieux si toutes les hypothèses sont montées).
- ***Etape 2*** Si la dernière branche de preuve est déchargée, on a obtenu un nouveau but. Sinon on passe à l'étape 3.
- ***Etape 3*** L'étape précédente peut avoir tenté des preuves par cas exploratoires, il faut donc revenir en arrière et utiliser une preuve réduire pour progresser dans la preuve sans risque.
- ***Etape 4*** La précédente étape utilise les tactiques et les règles de la preuve automatique : il est possible que la direction prise ne soit pas celle souhaitée par l'opérateur. On doit donc juger à cette étape s'il faut continuer à partir du but simplifié par la preuve réduite ou revenir en arrière. En pratique il suffit de juger la simplicité de ce nouveau but.
- ***Etape 5*** Retour en arrière si le but simplifié n'est pas jugé bénéfique.
- ***Etape 6*** C'est bien sûr l'étape la plus délicate : il s'agit de trouver la bonne commande interactive pour faire progresser la preuve vers son aboutissement. La suite de cette section traite principalement ce problème.

(e) ***Les obligations de preuve qui semblent fausses*** Lorsque le composant examiné comporte des erreurs, certaines obligations de preuve sont fausses. C'est ainsi que l'utilisation de l'Event-B signale les erreurs telles que le non-respect de la spécification, ou des propriétés invariantes. En cas d'obligation de preuve fausse :

- ***S'assurer que l'obligation de preuve est bien fausse :*** quand une obligation de preuve semble fausse, il faut en avoir une assurance suffisante avant de commencer les modifications dans le composant. Les conséquences d'une modification faite à tort peuvent être très coûteuses.
- ***Trouver la faute dans le composant :*** il suffit de reporter dans le contexte du composant les raisons qui font que l'obligation de preuve est fausse. En particulier, si on dispose d'un contre-exemple, reporter les valeurs trouvées dans le composant.
- ***Corriger :*** l'impact d'une PO fausse varie du simple cas oublié à la remise en cause du modèle mathématique employé. C'est pourquoi la détection des obligations de preuve fausses doit être faite en continu ; il n'est pas raisonnable de supposer ne plus avoir à modifier les composants avant les phases de preuve.

Les corrections suite à une PO fausse relèvent de la méthode Event-B, mais on peut borner aux conseils suivants :

— Se demander si, dans le sens physique attribué aux variables du modèle, les valeurs qui rendent l'obligation de preuve fausse sont possibles. Si ce n'est pas le cas, l'invariant est trop faible.

— Si on est amené à renforcer un invariant trop faible, s'assurer que la nouvelle propriété est toujours vraie, pour toutes les opérations. En effet il arrive souvent qu'une propriété physiquement vraie, oubliée dans l'invariant, ne soit pas conservée par des opérations dont le codage est biaisé par l'invariant trop faible.

Pour éviter des modifications nuisibles dans le composant, il faut vérifier que l'obligation de preuve qui semble fausse l'est vraiment. Pour cela, il faut considérer les points suivants :

- ***Vérifier que l'obligation de preuve n'est pas rendue vraie par la présence d'hypothèses contradictoires :*** il est normal que de telles POs apparaissent, elles traduisent l'application rigoureuse de la théorie. Les buts de ces obligations de preuve n'ont pas à être vrais, ils peuvent même être dépourvus de sens. Dans tous les cas, la contradiction dans laquelle on se place correspond à des branches du composant et de son niveau supérieur : c'est en se guidant sur ces branches qu'il faut chercher la contradiction.
- ***Chercher un contre-exemple :*** chercher des valeurs particulières des variables qui vérifient les hypothèses, mais pas le but. On fera attention aux points suivants :

— Chercher à utiliser les valeurs 0 et $\emptyset$, elles permettent souvent de fabriquer des contre-exemples simples.

— S'il y a beaucoup d'hypothèses, chercher le contre-exemple uniquement sur les variables qui interviennent dans le but. Ne pas chercher à contrôler si l'obligation de preuve est vraie par hypothèses contradictoire en cherchant le contre-exemple, c'est trop compliqué.

- ***Obligations de preuve dépourvues de sens :*** il est parfaitement possible qu'une obligation de preuve soit dépourvue de sens. De telles obligations signalent bien une erreur dans le composant, mais elles peuvent surprendre en particulier parce qu'il n'y a pas clairement un contre-exemple. En particulier le contre-exemple parfois ne serait pas accepté par l'Atelier B à cause d'un problème de typage, il est néanmoins mathématiquement valide. Ceci est bien une obligation de preuve trahissant une erreur dans le composant, qu'il faut absolument modifier avant de poursuivre.
  Dans le cas d'obligations de preuve fausses par absence d'information sur une variable, penser à :

— Un invariant de boucle trop faible : dans un invariant de boucle, il faut au moins typer toutes les variables utilisées dans le corps de la boucle ;

— Un invariant de liaison manquant entre une variable abstraite et la ou les variables importées qui la réalisent. Les obligations de preuve apparemment dépourvues de sens peuvent néanmoins être vraies par hypothèses contradictoires.

2.3.7 Les plug-ins de RODIN

La possibilité d'étendre RODIN à l'aide de « plug-ins » est toujours présent sachant que cette plateforme est basée sur Eclipse, les plug-ins couvrent plusieurs axes de développement (modélisation, animation, preuves et théories, génération de code et expérimentales) et seront présentés [271] comme suit :

2.3.7.1 La modélisation

Les plug-ins dans cette partie touche la modélisation soit en Event-B (la décomposition des contextes et les machines à base des évènements communs « decomposition plug-in » et la composition des machine selon les évènements « parallel composition plug-in ») soit d'autres méthodes qui peuvent intégrés en Event-B comme flow , Qualitative probability, Mode\FTon peut détailler :

- ***Le plug-in UML-B***
  UML-B fournit une interface graphique semble à UML « UML-like » pour Event-B. Il fournit diverses notations et les éditeurs de modélisation schématiques pour créer des modèles qui sont ensuite traduits en Event-B pour la vérification. Deux versions d'UML-B sont disponibles. L'UML-B d'origine où un projet complet est modélisée dans un projet schématique. Un projet Event-B séparé est ensuite généré pour la vérification. La version appelée iUML-B (i pour intégré) intègre des diagrammes dans les modèles Event-B. Cela permet au modélisateur de modéliser dans l'Event-B normal, mais aussi contribuer certains aspects du modèle via des diagrammes. Ceux qui sont plus familiers avec Event-B peut préférer cette version.

2.3.7.2 L'animation

Les plug-ins dans cette partie anime la modélisation soit en Event-B (« ProB plug-in » qui sera détaillé par la suite) soit d'autres méthodes qui peuvent **être** intégrés en Event-B basé sur ProB « UML-B-Statemachine Animation plug-in » pour animer les modele de iUML-B plug-in.

- ***Le plug-in ProB***
  L'extension de ProB appelé aussi prouver permet l'animation et le model-checking des spécifications B [272], Event B, Z et même TLA+ [183]. Le ProB est entièrement automatique qui permet de visualiser le comportement dynamique d'une machine abstraite et on peut systématiquement explorer tous les états accessibles d'une machine Event-B pour vérifier des propriétés temporelles. ProB contient deux sous-systèmes importants : un noyau qui traite les types de données de B (comme les ensembles et les relations), et qui implémente les opérations sur ces données (par exemple, le calcul du domaine d'une relation), et un interpréteur pour les substitutions, prédicats et expressions d'une machine B.

2.3.7.3 Preuves et théories

Les plug-ins dans cette partie enrichissent la preuve de modélisation en Event-B (« Atelier-B prover plug-in » voir 2.3.6) soit étendent les modèles par des nouveau concepts mathématiques (« theory plug-in » qui sera détaillé par la suite et encore utilisé par notre thèse dans le chapitre 4) soit d'autres méthodes qui peuvent intégrés en Event-B (« Isabelle for RODIN plug-in » pour transformer les modèles Event-B et leur preuve en théories dans la méthode Isabelle).

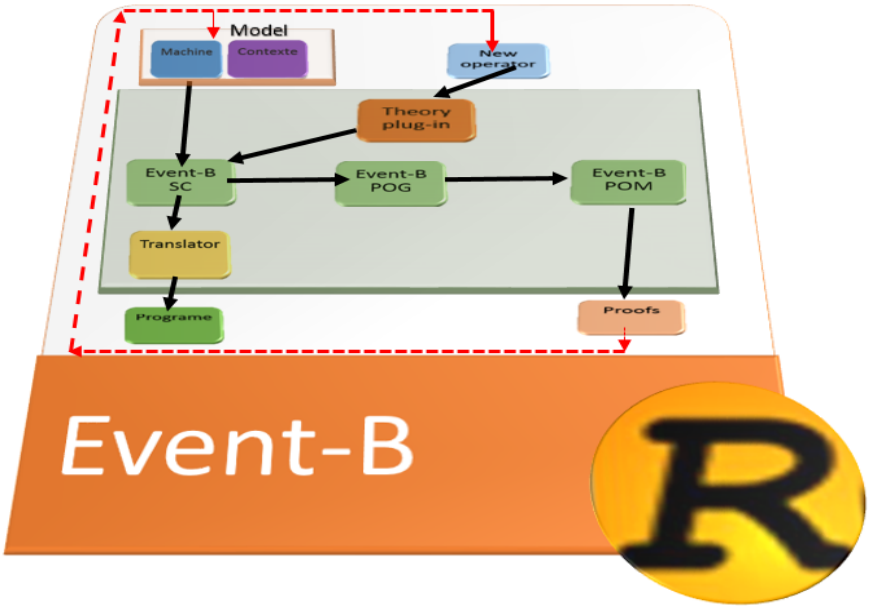


FIGURE II.15 – La plateforme RODIN avec le Theory plug-in.

- ***Le plug-in de la théorie***
  Le plug-in de la théorie « theory plug-in » est une contribution à la plate-forme RODIN qui facilite la spécification, la validation, le déploiement et l'utilisation **l'extension du langage** et de la preuve de la méthodologie Event-B. Les extensions de langage sont des ajouts au langage mathématique Event-B sous la forme de types de données, opérateurs et de notions axiomatiques toujours restent prouvable par l'outil RODIN(Figure II.15) .

2.3.7.4 Génération de code

Les plug-ins dans cette partie enrichissent la phase de modélisation en Event-B par la génération de code vers des langages différents comme : JAVA, Dafny, C, SQL et VHDL.

- ***Le plug-in EHDL***
  Le plug-in EHDL (voir Figure II.16) est une extension liée à la plate-forme RODIN. qui

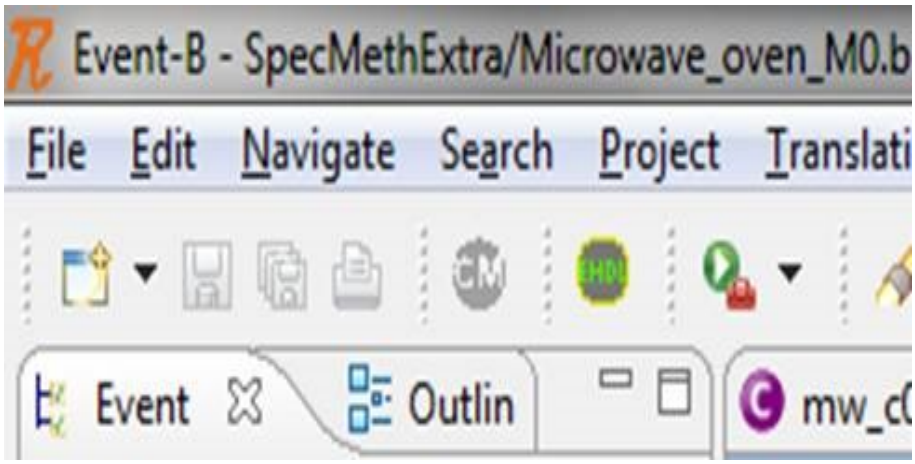


FIGURE II.16 – Le plug-in EHDL installé dans l'outil RODIN.

génère un code VHDL à partir des modèles spécifiés par la méthodologie Event-B. La génération dans cette extension dépend toujours des modèles et purement syntaxique parce qu'elle reste indépendante de comportement du system à spécifier. Les limites de cette extension nous

à guider pour faire la translation sémantiques l'un des objectives de notre contribution fortement détaillé dans le chapitre 4.

2.3.7.5 Expérimentales

Les plug-ins dans cette partie enrichissent la modélisation en Event-B pour l'opérateur de raffinement (« Group Refinement plug-in » détaillé par la suite) et le formaliser des nouveau concepts (« Feature Composition plug-in »pour composer les modèles, «Model Critic plug-in » pour analyser les modèles des système complexe par le langage de script Epsilon ou 'Epsilon scripting language') soit d'autres fonctionnalités intégrés en Event-B (« Project Diagram plug-in » pour visualiser les modèles Event-B sous formes des diagrammes «Pattern plug-in » pour la réutilisation des modèles et leurs preuves dans des projets différents).

- ***Le plug-in de raffinement de groupe***
  Le plugin de raffinement de groupe modifie l'ensemble des lois de raffinement dans un cadre d'une étape de raffinement unique pour éliminer la nécessité d'un événement d'entretien. Elle simplifie également la spécification de comportement **de base** en permettant à un utilisateur de renoncer les variables et les invariants nécessaires pour relier le comportement abstrait et concret. L'intention est de ne pas remplacer ou supprimer l'une des lois de raffinement existantes, mais plutôt les augmenter avec une nouvelle série de lois pour rendre l'ensemble du processus de raffinement plus souple.

Conclusion

Dans ce chapitre, on a présenté un aperçu des différents concepts (théorie des ensembles, logique de premier ordre) qui entrent en jeu dans les chapitres suivants et **focalisent** sur l'Event-B et la plateforme RODIN **pour offrir** un cadre pratique pour cette thèse. L'infrastructure preuve de RODIN a été présenté et ses lacunes identifiées.**Ainsi que nous avons decrit** le système de preuve utilisé dans l'Event-B **et nous avons effectué** une brève comparaison entre Event-B et trois **d'**autres formalismes. **Ensuite nous avons** présenté un bref aperçu de la logique de l'Event-B comme décrit dans [35]. Le but de cette section est de fournir la mise en pratique des contributions de cette thèse concernant la possibilité d'étendre la modélisation par Event-B en terme d'opérateurs de raffinement et la bonne définition de code VHDL sous forme de la théorie. **Nous avons** consacré la dernière section pour détailler le mécanisme interne de génération des obligations de preuve au sein de la plateforme RODIN, qui garantit la consistance du modèle, la correction du raffinement, la préservation de l'invariant, la décrémentation de variant, **et de** vérifier le respect de la sémantique Event-B.
L'outil RODIN fournit un ensemble d'outils extensible pour le développement et le raisonnement sur les modèles Event-B. RODIN comprend un ensemble d'outils « plug-ins » qui sont nécessaires pour un environnement de développement réactif. Dans ce chapitre, nous **posons** un peu de lumière sur quelques-unes des caractéristiques importantes de RODIN. La majorité des plug-ins dans RODIN se focalisent sur l'opérateur de raffinement et comment le mettre encore plus adéquate et robuste pour un modélisateur peu importe son expérience de la formalisation pendant le développement des projets et cela pourra prendre comme une motivation.
Cette partie joue un rôle très important pour comprendre le bénéfice de notre contribution basée sur l'amélioration de raffinement en Event-B en se basant sur des opérateurs détaillés dans le prochain chapitre et le dernier chapitre sera réservé pour l'utilisation des théories polymorphes pour vérifier les comportements liés aux environnements d'exécution des réseaux sur puce auto-organisé(SONoC) sous forme des modèles contiennent des routines en VHDL spécifiés en Event-B.

Chapitre 3

Le raffinement en Event-B à base d'opérateurs

« (1) … We typically construct an ordered sequence of embedded models, where each is a refinement of its predecessor…
(2) Algebraic specification aims to provide a formal basis to support the systematic development of correct programs…a central piece of the puzzle is how best to formalize concepts like ***specification, program and refinement step****…*
(3) … So we must build our theories up from small intelligible pieces…. A many-sorted algebraic theory is given by naming a set of sorts, a set of operators over those sorts and a set of laws which those operators must satisfy… »

JEAN-RAYMOND ABRIAL (1), DONALD SANNELLA(2) AND ROD BURSTALL (3)

Sommaire

Introduction

Actuellement, les méthodes utilisant le raffinement se classent en deux catégories. La première nous oblige à prouver, après chaque étape du raffinement de chaque système, la préservation de propriétés avec le niveau d'abstraction précédent (le raffinement est vérifié a posteriori). La seconde nous permet de construire la nouvelle spécification concrète à partir de la spécification abstraite (le raffinement s'inscrit dans un développement génératif). Cette dernière catégorie est la plus intéressante, puisqu'en garantissant que chaque étape de raffinement conserve les propriétés de la description architecturale de départ, indépendamment **des descriptions logicielles** en cours de développement, nous pouvons nous abstenir d'une phase de vérification a posteriori. Le langage de spécification formelle est celui de la méthode Event-B : elle a en effet le grand avantage de disposer d'outils pour aider au développement formel et à la génération du code.
Nous proposons une approche qui s'appuiera, en amont d'un développement formel classique, sur une description architecturale de haut niveau que nous raffinerons dans le langage formel cible d'une méthode formelle classique. Notre approche s'effectue naturellement en traitant le raffinement par des opérateurs en raison de mettre le développement à base de raffinement très explicite. Dans ce chapitre on fait une étude pour le raffinement pour l'ensembles des méthodes formelles et les critères **d'utilisation**, puis **nous prenons** la méthode Event-B comme cible **pour faire une explication objective** de l'avantage de spécifier de façon constructif en se basant sur des opérateurs de raffinement **qui** aident à systématiser le développement des systèmes.

3.1 La notion de raffinement

Le raffinement est lié au développement de logiciels, tout au long du cycle de vie, il est possible de voir des approches de raffinement plus ou moins explicites dans de nombreuses méthodologies. Les langages de description d'architectures permettent habituellement de décomposer des architectures logicielles, et peu d'entre eux comportent des mécanismes appropriés pour un raffinement d'architecture couvrant plusieurs niveaux d'abstraction. Par conséquent, on ne considère pas seulement les méthodes de raffinement d'architectures logicielles.
Tout d'abord, des techniques de développement complètement informelles, tel l'établissement d'un cahier des charges en langage naturel, n'a pas une sorte d'intéressante, vu qu'elles ne permettent ni d'établir une description complète du système, ni de vérifier la présence de propriétés spécifiques. Cependant, certaines autres méthodes du génie logiciel, bien qu'elles ne manipulent pas vraiment les mêmes concepts que les langages de description d'architectures logicielles, utilisent le raffinement pour assurer le lien entre les spécifications de systèmes à différentes étapes du développement. Elles incluent à la fois des approches formelles et semi-formelles.
Le raffinement peut être abordé de façons tout à fait différentes, selon les caractéristiques et les buts de la méthode de développement de logiciels considérée **(voir [273–275] ou [276])**.
D'une part, certains systèmes ne font intervenir que pour une unique étape de raffinement, au moment de la phase de compilation. Mais cette étape se prête mal à la vérification de conservation des propriétés. La notion de raffinement est alors associée à la phase de conception et peut ainsi être employée plusieurs fois de suite. D'autre part, une partie des méthodes fournit des mécanismes propres à construire la description concrète pour raffiner une spécification, à partir de l'originale, tandis que les autres se contentent d'effectuer des vérifications sur des spécifications déjà existantes. Enfin, la relation de raffinement elle-même peut s'avérer être un critère discriminant entre les différentes approches, étant donné qu'elle ne repose pas toujours sur les mêmes concepts, voire sur les mêmes objectifs.
Les différentes méthodologies offrent autant de relations de raffinement. Trois critères fondamentaux suffisent à distinguer les principales catégories de relations de raffinement :

- **la place du raffinement dans le cycle de vie**, chacune de ces approches peut soit introduire une étape de raffinement durant la phase de conception de façon à changer une spécifica-

tion abstraite, architecturale ou pas, en une autre spécification plus concrète, soit conduire directement à une implémentation, se cantonnant donc à la phase d'implémentation ou de programmation

- **le type de processus de raffinement**, le raffinement est quelquefois construit à partir de la spécification abstraite, en se basant sur des constructions spécifiques, et parfois il n'est vérifié qu'après **une telle** propriété particulière du développement.
- **la relation de raffinement** cette critère elle-même peut vouloir que le comportement concret soit un comportement possible du comportement abstrait (menant ainsi à une spécification plus détaillée du même problème), ou qu'il soit un comportement plus simple correspondant à un système plus général qui saurait gérer le problème abstrait spécifique, ou encore que le comportement reste inchangé.

En plus de ces principaux critères de discrimination, plusieurs autres peuvent servir à préciser de façon plus précise les différents aspects de la relation de raffinement. Par exemple, il peut y avoir un raffinement horizontal (décomposition) ou vertical (avec changement de niveau d'abstraction) et éventuellement une distinction claire entre les deux. Il est également possible qu'il y ait des constructions spécifiques pour un raffinement de données, un raffinement fonctionnel ou un raffinement comportemental, mais aussi que le raffinement soit compositionnel. En outre, toutes les méthodes ne conduisent pas d'une spécification abstraite jusqu'à la génération du code, tout comme elles pourraient ne pas permettre la distinction entre différents niveaux d'abstraction.
En dépit de ces aspects liés à l'action de raffiner, quelques critères supplémentaires concernent la réutilisation possible de parties du processus de raffinement et la préservation de propriétés, qu'elles soient inhérentes à la description du système ou définies par l'utilisateur. Finalement, de multiples langages ou vocabulaires peuvent être utilisés pour identifier les différents niveaux d'abstraction ou étapes de raffinement.Par conséquent, ces critères peuvent servir à organiser les différentes approches dans des catégories en fonction de leurs relations de raffinement.
Dans la suite, on décrit plusieurs méthodes du génie logiciel, représentant les différentes relations de raffinement vertical. Nous les présentons dans un premier temps par "familles", afin d'être capable de comparer, par la suite, leurs approches respectives du raffinement.
On peut distinguer des approches n'utilisent le raffinement qu'à la fin du processus de développement du système logiciel (**pendant la phase d'implémentation**), d'autres méthodologies fournissent un meilleur support pour le raffinement d'architecture **pendant la phase de conception**. Néanmoins, elles ne procèdent pas toutes de la même manière. La troisième catégorie de systèmes de raffinement concerne les approches mettant à disposition des constructions spécifiques pour construire une spécification concrète à partir de la spécification abstraite (**pendant la phase de conception de façon constructif**) : le raffinement n'a pas à être justifié a posteriori, il constitue en lui-même une phase de développement.

3.1.1 Le raffinement pendant la phase d'implémentation

Parmi les méthodes qui appliquent ce type de raffinement on cite les suivantes :

3.1.1.1 Darwin, MetaH, UniCon et Weaves

Ces langages [20,24,277,278] ne montrent pas de véritable étape intermédiaire distincte de raffinement : les spécifications sont déjà contraintes par des détails liés à l'implémentation. Dans ce cas, la relation de raffinement se réduit à une sorte de compilation, souvent assurée par un compilateur spécifique. Au lieu d'être effectué durant la phase de conception, le raffinement se prépare dans la spécification et est opéré de façon effective pendant la phase d'implémentation. Il pourrait alors être assimilé à un genre de raffinement vertical limité, qui préserverait des propriétés inhérentes des spécifications (orientées implémentation). Puisqu'aucun raffinement horizontal ne peut avoir lieu

en même temps, il n'est pas possible d'affirmer qu'il existe une distinction claire entre raffinement horizontal et vertical dans ces approches.

- ***Propriétés***
  Ces méthodes sont détaillées en plus dans le tableau III.1.

Nom	Darwin, MetaH, UniCon et Weaves
Description de l'objectif	-Ce sont des langages de description d'architecture logicielle ; -leur support au raffinement est limité à la génération automatique du code de l'application à partir de la spécification architecturale et de commentaires appropriés ; -il n'y a pas d'étape intermédiaire de raffinement ; -les spécifications sont déjà contraintes par des détails d'implémentation ; -seules des propriétés intrinsèques ("incorporées") des spécifications (orientées implémentation) peuvent être préservées.

TABLEAU III.1 – Description des méthodes Darwin, MetaH, UniCon et Weaves.

- ***Critères principaux***
  D'autre part, des langages comme Darwin, MetaH, UniCon et Weaves, présentés dans les tableaux III.2 et III.3, ne permettent qu'une forme limitée de raffinement, restreinte à une implémentation possible de la description du système qu'ils auront servi à spécifier. Cette étape de raffinement est de ce fait analogue à une compilation de la spécification.

Place du raffinement	Type de processus de raffinement	Relation de raffinement
Implémentation	compilateur (**à l'exception pour Weaves**)	Compilation

TABLEAU III.2 – Les critères principaux de raffinement dans Darwin, MetaH, UniCon et Weaves.

- ***Critères auxiliaires***
  Par conséquent, cette unique étape de raffinement, souvent basée sur des commentaires de la spécification, peut être vue comme une étape de raffinement vertical. Cependant, plusieurs critères auxiliaires (tableau III.3) qui concernent le processus de raffinement n'ont pas vraiment de sens dans le contexte d'une implémentation (par exemple, la présence de constructions pour un raffinement horizontal, de données, fonctionnel ou comportemental, ou encore de langages de description différents).

3.1.2 Le raffinement pendant la phase de conception

Dans ce type de raffinement on peut citer :

3.1.2.1 La notation Z

La notation Z (dans le tableau III.4) est également une approche orientée modèle mais elle utilise une relation de raffinement différente de la précédente. En réalité, en plus de ce que fait VDM, Z effectue une distinction entre les raffinements d'opérations, de données et de données fonctionnelles.

Distinction RH \ RV	NON
Raffinement Horizontal (RH)	-
Raffinement Vertical (RV)	OUI
Raffinement de Données	-
Raffinement Fonctionnel	-
Raffinement Comportemental	-
Raffinement Compositionnel	NON
Génération de code	OUI
Multiples niveaux d'abstraction	NON
Réutilisation	NON
Préservation des propriétés inhérentes	OUI
Préservations des propriétés définies par l'utilisateur	NON
Multi-langages	-

TABLEAU III.3 – Les critères auxiliaires de raffinement dans Darwin, MetaH, UniCon et Weaves.

Par conséquent, le raffinement recouvre un peu plus de choses, mais en fait ce raffinement est apparemment contraint par le cadre formel : comme dans la méthode B (qui serait développée ultérieurement à partir de l'expérience de Z), le raffinement vertical est plutôt limité à la modification des corps d'opérations, en exprimant plus précisément comment le résultat est obtenu. L'établissement de la relation de raffinement repose sur la vérification de conditions mathématiques (logiques) ; il s'agit à nouveau d'un test a posteriori, mais les obligations de preuve constituent indéniablement un moyen propice à la préservation de propriétés. Malgré tout, la relation de raffinement de Z, contrairement à VDM ou la réduction, suit la même idée que dans la méthode B classique : effectivement, elle s'exprime comme l'affaiblissement des préconditions de l'opération abstraite et le renforcement de ses postconditions (l'opération qui raffine l'autre est plus déterministe), à chaque fois que l'abstraction est en situation de terminer. Par conséquent, le raffinement fait montre d'un comportement possible de l'abstraction. Enfin, tout comme dans le cas de VDM, le raffinement de la notation Z est compositionnel et ne conduit pas non plus à la génération de code exécutable, sans distinction claire entre les niveaux d'abstraction.

- ***Propriétés***
  On peut décrir la méthode Z comme il est détaillé dans le tableau III.4.
- **Critères principaux**
  La notation Z [279] est également une approche orientée modèle. Sa relation de raffinement, décrite dans les tableaux III.5 et III.6, repose, comme pour la méthode B, sur l'affaiblissement des préconditions et le renforcement des post-conditions (les opérations issues du raffinement doivent être plus déterministes, lorsque l'abstraction se termine). Le raffinement en Z devrait être établi sur la base de la preuve que les propriétés reliant les variables abstraites sont encore vérifiées au niveau concret.
- **Critères auxiliaires** Comme pour VDM, il n'est pas vraiment possible de dire que Z supporte le raffinement horizontal (cf. tableau III.6) car elle traite le système dans son ensemble, et pas au niveau des composants et connecteurs. Néanmoins, le raffinement est principalement établi sur la base de corps d'opérations, arrivant ainsi à une sorte de raffinement vertical. En réalité, il affecte plusieurs aspects de la spécification, comme les structures de données ou, à travers la modification des corps d'opérations, les fonctions et, de façon plus limitée, les comportements. Les étapes de raffinement peuvent être composées, mais les niveaux d'abstraction ne sont pas clairement distincts : le formalisme ne permet pas de les différencier. Par conséquent, même si le développement en Z cherche à produire une implémentation, il n'y aura pas de génération directe du code. Enfin, puisque le raffinement est établi à partir de la preuve que les propriétés d'une spécification donnée sont préservées dans une autre, la

Nom	Z
Description de l'objectif	- La notation de spécification formelle Z est basée sur la théorie des ensembles de Zermelo- Fraenkel et sur la logique des prédicats du premier ordre ; - Z supporte des raffinements d'opérations, de données et de données fonctionnelles, dans le même esprit que le raffinement en B : les préconditions sont affaiblies et les opérations deviennent plus déterministes ; - un schéma Z est une signature, plus la propriétaire liant les variables à la signature ; - des conditions (invariants) doivent être démontrées pour s'assurer de la terminaison et de la compatibilité des résultats ou des états initiaux (elles sont simplifiées lorsqu'il y a une fonction totale entre les états abstraits et concrets) ; - le développement d'un programme depuis une spécification obéit à deux types de décisions de conception : les opérations décrites par des prédicats dans la spécification doivent être implémentées par des algorithmes exprimés dans un langage de programmation ; -les données décrites par des types de données mathématiques dans la spécification doivent être implémentées par des structures de données du langage de programmation ; -les règles pour un raffinement d'opération simple nécessitent de montrer qu'une opération est une implémentation correcte d'une autre opération avec le même espace d'états, lorsque les deux opérations sont spécifiées par des schémas ; ce genre de raffinement d'opération peut être étendu dans deux directions de façon à le rendre utilisable généralement, en introduisant des constructions du langage de programmation et par raffinement de données (introduction de structures de données orientées machine) ; - une opération concrète Cop, issue du raffinement d'opération d'une opération abstraite Aop, peut en différer de deux manières : la précondition de Cop peut être plus libérale que celle de Aop (Cop est garantie de terminer dans plus de cas que Aop), Cop peut être plus déterministe que Aop (pour certains états avant l'opération, il y aura moins d'états suivants possibles) ; -Cop doit être assurée de terminer lorsque Aop l'est ; si Aop est garantie de terminer, alors chaque état que Cop pourrait produire doit être un état que Aop pourrait produire (c'est-à –dire que si la précondition de Aop est satisfaite, alors tout résultat que Cop pourrait produire doit être un résultat possible de Aop) ; -le raffinement de données étend le raffinement d'opération en permettant à l'espace d'états des opérations concrètes d'être différent de l'espace d'états des opérations abstraites ;

	- il permet aux types de données mathématiques d'une spécification d'être remplacés par des types de données plus orientés machine dans la conception (une étape de raffinement de données relie un type abstrait de données, la spécification, à un type de données concret, la conception) ; le type de données concret peut alors être considéré comme un autre type de données abstrait (constitué d'un espace d'états et d'opérations décrites par des schémas) ; -Afin de prouver que le type de données concret implémente correctement le type de données abstrait, il faut expliquer quels états concrets représentent quels états abstraits ; -un raffinement de données peut être correct seulement si deux conditions sont satisfaites pour chaque opération du type données abstrait (de façon analogue aux deux conditions du raffinement d'opération) : celles-ci doivent permettre la terminaison de l'opération concrète à chaque fois que l'opération abstraite est garantie de terminer et faire que l'état à la fin de l'opération concrète représente un des états abstraits dans lesquels l'opération abstraite peut terminer ; -une autre condition relie les états initiaux des types abstraits et concrets (chaque état initial possible du type concret doit représenter un état initial possible du type abstrait) ; -le schéma d'abstraction Abs entre les états abstraits et concrets définit une fonction totale des états concrets vers les états abstraits ; le raffinement de données fonctionnelles a un ensemble plus simple de conditions si le schéma d'abstraction est une fonction totale : la première condition est identique, la deuxième condition et la condition sur les états initiaux sont simplifiées (élimination du quantificateur existentiel) ; l'avantage est que la preuve que Abs est fonctionnel n'a besoin d'être faite qu'une seule fois par type (ce travail n'a pas besoin d'être répété pour chaque opération) ; -la relation de raffinement est similaire à celle de B ; -le raffinement est vérifié a posteriori, sur la base de preuves de conditions mathématiques (logiques) ; -il n'y a pas de construction de raffinement explicite ; -les différentes notions de raffinement ont quelques différences les unes avec les autres : le raffinement d'opération nécessite que les objets abstraits et concrets aient les mêmes espaces d'états et le raffinement de données a besoin d'espaces d'états différents.

TABLEAU III.4 – Description de Z.

Place du raffinement	Type de processus de raffinement	Relation de raffinement
Conception	vérification a posteriori de conditions mathématiques (logiques)	affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme), lorsque l'abstraction se termine

TABLEAU III.5 – Les critères principaux de raffinement en Z.

réutilisation n'est pas envisageable.

Distinction RH \ RV	-
Raffinement Horizontal (RH)	-
Raffinement Vertical (RV)	OUI
Raffinement de Données	OUI
Raffinement Fonctionnel	OUI
Raffinement Comportemental	Limité
Raffinement Compositionnel	OUI
Génération de code	NON
Multiples niveaux d'abstraction	Pas très distincts
Réutilisation	NON
Préservation des propriétés inhérentes	OUI (obligations de preuves)
Préservations des propriétés définies par l'utilisateur	NON
Multi-langages	NON

TABLEAU III.6 – Les critères auxiliaires de raffinement en Z.

3.1.3 Le raffinement pendant la phase de conception

Event-B est une méthode utilisée pendant la phase de conception et donc on peut la détailler comme suit :

3.1.3.1 Event-B

Mis en place dans l'idée de faciliter l'utilisation de la méthode B pour la conception d'applications complexes, le Event-B (tableau III.7) a été conçu comme une extension au B classique avec des propriétés basées sur des événements. Le langage original de B est étendu avec de nouvelles caractéristiques facilement traduisibles en B standard, dont essentiellement une syntaxe basée événements. Le développement de modèle permet maintenant deux types d'observations : les états (amenant aux propriétés de sûreté) et les événements (pour ce que l'on peut observer, ou propriétés de vivacité). De plus, les raffinements (comme en B classique) peuvent être enchaînés avec des décompositions. Par conséquent, on peut voir dans ces dernières une espèce de raffinement horizontal explicite, même si elles ne s'intéressent pas à des composants et connecteurs architecturaux. Néanmoins, la relation de raffinement admet quelques différences respectivement à la méthode B : les obligations de preuves peuvent être séparées en deux ensembles, selon qu'elles concernent des états ou des événements. Fondamentalement, la spécification qui raffine l'autre génère un état plus précis et davantage d'événements.

- ***Propriétés***
  On peut décrir la méthode Z comme il est détaillé dans le tableau 3.4.
- **Critères principaux**
  Par ailleurs, la méthode B a été le sujet de nombreux travaux différents ([280–282] ⋯). L'Event-B [283, 284], est une extension du langage B classique avec un sucre syntaxique supplémentaire, utilisant des postconditions, des modalités, une syntaxe basée événement et des constructions de raffinement explicites. Néanmoins, cela ne change pas de faç on drastique la relation de raffinement, comme illustré dans les tableaux III.8 et XX.2 : le formalisme dispose d'un pouvoir expressif amélioré pour les spécifications, mais les opérations après raffinement doivent encore contenir des préconditions plus faibles et des postconditions plus fortes que dans leurs versions abstraites.

Nom	Event-B
Description de l'objectif	L'approche du Event-B a été conçue pour la modélisation de systèmes complexes, comme des extensions au B classique, en utilisant un développement suivant un modèle (ou aspect) ; -**L'Event-B est appliquée successivement** à la spécification du logiciel et son développement, mais une nouvelle approche est nécessaire pour concevoir des systèmes complexes (il faut au moins le même langage, avec un point de vue différent) ; -**L'Event-B est basée** sur du raffinement et des preuves, a été adaptée à la conception de systèmes, non pas en changeant le langage sous-jacent, mais en modifiant la façon de modéliser (une approche basée sur **des événements observés pour un système**) ; -**Certaines extensions de langage sont nécessaires pour pouvoir exprimer des propriétés basées sur les événements ; afin de conserver la compatibilité avec l'Atelier B, le langage B est étendu avec de nouvelles caractéristiques facilement traduisibles en Event-B, en utilisant des post-conditions, des modalités, une syntaxe basée événements et un raffinement explicite ;** -La construction de modèle reflète ce qui peut être observé ; il y a deux types d'observations : les états (pour les propriétés de sûreté) et les événements (pour les propriétés de vivacité) ; des preuves sont requises pour valider ces propriétés ; -Le paradigme du "parachute" **(progressive)** repose sur le fait que d'autres choses intéressantes peuvent être observées à un niveau inférieur (un état plus précis et plus d'événements) ; -Chaque étape de la construction globale doit assurer les preuves de ce que les parties de contrôle et de communication en construction préservent des lois élaborées uniquement(**obligations des preuves**) à partir de l'état physique aux premières étapes du développement ; -Lorsqu'il devient difficile de développer le modèle d'avantage, celui-ci est décomposé en plusieurs sous-modèles, basés sur du logiciel, du matériel ou en représentant un canal de communication ; pour les modèles basés logiciels, un ordonnanceur devrait être ajouté avant d'arriver au code, par des moyens traditionnels ou en achevant un développement de logiciel en B ; -Un autre outil B a été développé pour transformer un ensemble d'événements en un fragment de code en assemblant les événements ensemble ; il est donc capable de générer les algorithmes équivalents : les règles d'assemblage sont définies et appliquées sur un ensemble d'événements, que ce soit de façon automatique ou interactive ; -L'Event-B offre la possibilité d'enchaîner raffinements et décompositions.

TABLEAU III.7 – Description de l'Event-B.

Place du raffinement	Type de processus de raffinement	Relation de raffinement
Conception	différentiel (extension du B classique)	affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme)

TABLEAU III.8 – Les critères principaux de raffinement en Event-B.

- **Critères auxiliaires**
  Les propriétés ne sont plus établies que sur des états seulement, mais elles peuvent être maintenant établies sur des événements également. Un changement majeur repose sur le fait que les événements d'un module abstrait peuvent être décomposés dans un module plus concret. Cela constitue, d'une certaine manière, un genre de raffinement horizontal, bien que cela ne concerne pas des éléments architecturaux (cf. **tableau XX.2**). D'autre part, cette version de la méthode de développement permet d'enchaîner des étapes de raffinement et de décomposition. **Sachant que les modèles durant ces étapes peuvent être traduit en B classique, nous pouvons énoncer que les différentes étapes de raffinement peuvent toujours être réalisés de façon compositionnelle**.

Distinction RH \ RV	NON
Raffinement Horizontal (RH)	OUI (mais pas sous forme de composant et connecteur)
Raffinement Vertical (RV)	OUI (mais limité surtout aux corps d'opérations)
Raffinement de Données	limité
Raffinement Fonctionnel	OUI Modification des corps d'opérations
Raffinement Comportemental	Limité
Raffinement Compositionnel	OUI (excepté les implémentations) :Possibilité d'enchaîner raffinements et décompositions
Génération de code	OUI
Multiples niveaux d'abstraction	OUI
Réutilisation	NON
Préservation des propriétés inhérentes	OUI (obligations de preuve sur des états et des événements)
Préservations des propriétés définies par l'utilisateur	NON (pas automatisée)
Multi-langages	Mots clés réservés en fonction du niveau d'abstraction

TABLEAU III.9 – Les critères auxiliaires de raffinement en Event-B.

3.1.4 Discussion et commentaire

La conclusion tirée de toute l'étude précédente de raffinement est le fruit de notre approche **est que** le raffinement sur **l'Event-B suit plus de critères pour établir une vérification comportementale des propriétés des systèmes complexes**.
Les méthodes formelles **sont constituées par des algèbres pour modéliser des interactions entre les modèles produits par le processus de raffinement**. Elles permettent **de construire un modèle**

mathématique qui reprend certaines descriptions du modèle d'analyse de l'application (celles relatives aux interactions), de garantir la cohérence du modèle et **sa conformité avec le programme cible**. L'approche que nous proposons **est faite** pour traiter le raffinement vertical d'une description d'architecture et pour obtenir une spécification dans une méthode de développement formel, qui pourra par la suite mener à la génération de code.
Les opérateurs de raffinement nous permettent à la fois de traduire les différentes constructions d'un langage de description d'architecture pour se rapprocher d'un niveau d'abstraction d'implémentation, tout en garantissant la conformité des raffinements, et de faire bénéficier une méthode formelle de développement d'une étape de conception plus "ergonomique". Les opérateurs pour construire le processus de raffinement sont :

- opérateur de **création**, noté *'create'*,
- opérateur d'**enrichissement**, noté *'Enrich'*,
- opérateur de **restriction**, noté *'Restrict'*,
- opérateur de **renommage**, noté *'Rename'*.

L'idée principale de cette approche **rend** le développement de raffinement très explicite en ajoutant **un nouvel opérateur qui développe le modèle en Event-B et essentiellement les événements**, nous introduisons quatre opérateurs : *créer*, *enrichir*, *restreindre* et renommer et **nous faisons une étude comparative pour prendre une vue très abstraite entre cette nouvelle méthode de raffinement et les approches qui existent déjà : ADL et** le concept de l'extension de l'événement offert par l'outil RODIN.

3.2 La formalisation par l'approche d'opérateurs

Historiquement, les spécifications des systèmes devraient être étudiées dans leur propre droit et **les chercheurs ont proposé par conséquence** une langue Clear, pour les écrire d'une manière bien structurée, ce n'est pas un langage de programmation ; la dénotation d'un texte **par** Algol par exemple est un programme, mais la dénotation d'un texte **par** Clear est une théorie beaucoup plus structurée. La présentation habituelle d'une telle théorie est**faite** par une collection d'équations, mais **pour l'intérêt des théories établies pour la validation en informatique, ils ont imposer une structure sur la présentation Elle est de même type général que dans la structure modulaire post-Algol des langages de programmation (Simula, Alphard, Clu, Iota) ont imposé aux programmes**. Le travail sur ce problème tout d'abord était de point de vue sémantique, tenu en compte des théories décrites comme des co-limites de diagrammes dans la catégorie des théories, liées à des idées antérieures en théories générales des systèmes. Ensuite, des formes syntaxiques ont conçu pour ces idées, empruntant autant que possible de la technologie standard des langages de programmation (expressions, variables, procédures). Une description informelle de Clear **fait seulement une brève référence qui a été présenté en travaux de Burstall et Goguen [285] pour exprimer sa sémantique**. **Durant la periode entre** l'automne 1977 et le printemps de 1978, plus de **travaux** sur les propriétés de base des théories qui ont été présupposées par les idées sémantiques de [286]. L'accent a été mis sur le travail avec une notion abstraite et générale de la théorie jusqu'à la notion de Lawvere ou la généralisation des théories de Lawvere de multiples sortes.
Cette approche se fait par des théories, plutôt que des algèbres. Une théorie (par exemple la théorie des groupes) contient beaucoup d'algèbres pour ses modèles (par exemple le-groupe 4). Souvent, nous sommes seulement intéressés par le modèle initial, mais ce n'est pas toujours le cas, comme lorsque nous décrivons les arguments possibles aux procédures théoriques ou faiblement spécifions la solution à un problème que par des conditions (par exemple un plus court chemin dans un graphe donné). La manipulation mathématique des théories est souvent plus pratique que celle des algèbres, principalement parce que les théories morphismes permettent le changement de signature. Nous considérons tous les modèles de la théorie désignée par la spécification Clear comme une solution acceptable au problème spécifié.

Les idées catégoriques dans une sémantique à savoir sont les notions de la catégorie, les fonctions, les co-limites et les Somme amalgamée **(nous avons préfèré d'utiliser le terme anglaise "pushouts")**. Nous ne tentons pas de les expliquer ici ; une exposition élémentaire peut être trouvée dans le travail de Arbib et Manes [287] et ils sont définis dans un texte sur la théorie des catégories, telles que celui présenté par Herrlich et Strecker [288].

3.2.1 Les fondations de l'approche

Une grande quantité de travail pertinente est élaborée dans la littérature sur les types de données abstraites et des travaux sur les langages de spécification. on peut citer les points suivant :

— Des travaux réservés pour les types de données, nous renvoyons le lecteur à Liskov et Zilles [289], Guttag et Horning [290], Goguen, Thatcher et Wagner [291] et Lehman et Smyth [292] ;

— Des recherches posées sur la question particulière des types de données paramétriques nous pouvons citer les travaux de : Thatcher, Wagner et Wright [293] et Ehrig et al [294] ;

— Des travaux qui focalisent sur des algèbres,on peut citer : Ehrich et Lohberger [295] ;

— Des recherches travaillent avec des théories comme nous le faisons. Ce dernier est étroitement liée à notre approche, car il utilise des "pushouts" pour l'application de la procédure (notre approche est inspiré de l'article de [285] qui a suggéré d'utiliser colimites, dont les "pushouts" sont un cas particulier, mais nous ne donne pas les détails ;

— Des travaux basé sur des équations sémantiques en utilisant des "pushouts" sont apparus dans un document qui a donné à une réunion de Varsovie, [286]). Honda et Nakajima [296] considèrent ces aspects comme des démonstrations des théorèmes.

— Certains des travaux qui ont des base théoriques sur les méthodes formelles sont cités par la suite :

  * Les langages de spécification ont été développés par SRI [297] dont ce langage spéciale utilise l'abstraction de données a été essayé sur une spécification de système d'exploitation, aussi par Abrial, Schuman et Meyer [298], qui ont une langue la plus élégante sur la base de la théorie des ensembles axiomatique avec celle du saveur Bourbaki.

  * Genrich [299] a conçu un langage basé sur les réseaux de pétri (Petri net) qui préfigure beaucoup de Clear, mais sans procédures.

  * Caplain [300] propose un langage de spécification orientée vers l'expressivité linguistique.

  * D'autres langages se sont concentrés pas sur la structure globale comme celle que nous le faisons, mais la question importante est posé sur les primitives bien choisies, notamment Dahl [301], en utilisant des séquences, et les travaux avec des vecteurs par Reynolds [302]. Pour une étude des travaux sur les spécifications voir Liskov et Berzins [303].

  * Enfin, nous mentionnons le travail sur la spécification des langages de programmation comme Bjørner a montré, la sémantique dénotationnelle peut être utilisé pour spécifier des problèmes autres que ceux concernés par les langages ou la compilation , et il soulève également des questions de modularité, comme Mosses a souligné [304].

3.2.2 Les bases mathématiques de l'approche d'opérateurs

Dans cette section, nous définissons tout d'abord les notions bien connues de la signature Multisortes),et de l'algèbre, les équations et les théories qui font les bases attribué aux opérateurs. Nous développons alors la notion d'une théorie comme un moyen pour spécifier toute une classe d'algèbres ayant tous la même signature et obéissant à certaines lois équationnelles telles que l'associativité ou la commutativité. Une présentation de la théorie consiste à une signature avec un

ensemble d'équations. Nous expliquons les équations sous forme de paires de « termes », c'est-à-dire des éléments de l'algèbre initiale pour une signature, et cela signifie une algèbre doit satisfaire une équation, ce qui conduit à la notion d'un ensemble fermé d'équations, et cela nous permet de définir une théorie *comme une présentation de la théorie dont l'ensemble d'équation est fermée*.
Nous passons ensuite à l'examen de la catégorie des théories, définissant les théories morphismes comme morphismes de signature qui « préservent les équations ». La sémantique des opérations de construction de la théorie de Clear sont donnés en termes de diagrammes de « pushouts» et dans la catégorie des théories : un "pushout" est essentiellement l'union de deux théories qui ont une certaine sous-théorie commune. Depuis que les "pushouts" sont un type particulier de la co-limite, nous montrons que la catégorie des théories a des co-limites.
Le développement ci-dessus (**des définitions présentées par le travail de Burstall [42]**) est mise en oeuvre dans le cas particulier de nombreux sortes d'algèbres, mais celui-ci bien de passer par un réglage beaucoup plus général. Une façon de généraliser est de définir une langage comme une catégorie de signatures avec des catégories associées de phrases (par exemple des équations) et des modèles (par exemple des algèbres) avec certaines interrelations. Cela nous donne une notion abstraite de la théorie et de la même preuve est applicable pour montrer qu'il a des co-limites.
D'autres approches ont conçu à utiliser la notion axiomatique de la signature et de la théorie et à utiliser la notion assez générale de la théorie monadique où La généralité est recherchée afin que nous puissions traiter des aspects tels que les erreurs ou les structures de données infinies.

3.2.2.1 Les signatures

Une signature se compose de toutes sortes (**les** noms de types de données) et les opérateurs (**les** noms des opérations sur les éléments de ces types de données), chaque opérateur étant donné un $n-uplet$ de toutes sortes d'entrée et une sortie de tri ° représente un morphisme entre les signatures des cartes "maps" de sortes, et les opérateurs , en préservant les sortes d'entrée et de sortie.
La signature, Σ, est une paire $< S, \Sigma >$, où S est un ensemble (des sortes) et Σ est une famille d'ensembles (des opérateurs) indexé par $S^* \times S$.
Soit σ signature morphisme d'une signature $< S, \Sigma >$ à une signature $< S', \Sigma' >$ est une paire $< f, g >$ constitué d'une "map" $f : S \to S'$et une famille de "map" $g_{us} : \Sigma_{us} \to \Sigma'_{f*(u)f(s)}$où $f^* : S^* \to S'^*$ est l'extension de f.
Il est commode de $\sigma(s)$ pour écrire $f(s)$, $\sigma(u)$ pour$f^*(u)$ et $f(\omega)$ pour $g_{us}(\omega)$. La catégorie des signatures, Sig, a comme des objets : les signatures et comme morphismes : les morphismes de signature. Le morphisme d'identité est la "map" d'identité et la composition des morphismes est la composition de leurs composants correspondants sous forme de "maps". (Ceci est clairement une catégorie.).

3.2.2.2 Les algèbres

Σ-algèbre associe un ensemble à chaque sorte de Σ et une fonction à chaque opérateur de Σ. Soit Σ une signature. Une Σ-algèbre A est une famille d'ensembles $A\,S$-indexé, appelée le transporteur de Σ, avec un $S^* \times S$-indexé famille de "map" $\alpha_{us} : \Sigma_{us} \to (A_u \to A_s)$ où $u \in S^*, s \in S$et $A_{u_1,\cdots,u_n} = A_{u_1} \times \cdots, \times A_{u_1}$Nous écrivons $|A|$ pour A.
Si $\omega \in \Sigma_{us}$et $< a_i, \cdots, a_n > \in A_u$, nous écrivons $\omega(a_1, \cdots, a_n)$ pour $\sigma_{us}(\omega)(a_1, \cdots, a_n)$ où il n'y a pas d'ambiguïté.
Si X et Y sont I-indexé ensembles nous parlons d'une "map" $g : X \to Y$, ce qui signifie qu'une famille I-indexé de "maps" $g_i : X_i \to Y_i, i \in I$ est la composition évidente $(f \cdot g)_i = f_i \cdot g_i$ et cela donne une catégorie, Set_i, pour des ensembles I-indexés.
Si $\underline{\Sigma} =< S, \Sigma >$ puis un $\underline{\Sigma}$-homomorphisme de Σ d'un $\underline{\Sigma}$-algèbre de$\Sigma < A, \alpha >$ à un Σ-algèbre $< A', \alpha' >$ est une "map" $f : A \to A'$ de telle sorte que pour chaque $\omega \in \Sigma$ et chaque

$$a_1 \in A_{s_1}, \cdots, a_n \in A_{s_n}, f_s(\omega(a_1, \cdots, a_n)) = \omega(f_{s_1}(a_1), \cdots, f_{s_n}(a_n))$$

La catégorie de $\underline{\sigma}$-algèbres, $AIg_{\underline{\sigma}}$, a $\underline{\Sigma}$-algèbres comme des objets et Σ-homomorphisme comme morphismes ; la composition et l'identité sont représentées en tant que "maps". (Ceci est clairement une catégorie.)
Il y a une fonction oublieux "forgetfull functor" $\upsilon : Alg_{\underline{\Sigma}} \to Set_S$, prenant chaque algèbre à son support et chaque morphisme à sa carte sous-jacente.
Il y a une algèbre libre "free algebra" (également appelé un 'mot' algèbre ou 'word' algebra) $\underline{W_{\Sigma}}(X)$ sur chaque S-indexé ensemble X, avec une $\eta_X : X \to |\underline{W_{\Sigma}}(X)|$, C'est-à-dire pour toute $\underline{B}$ de Σ-algèbre
Aucune "map" $f : X \to |\underline{B}|$ étend de manière unique en un homomorphisme $f^{\#} : \underline{W_{\Sigma}}(X) \to \underline{B}$, de telle sorte que $\eta_X \cdot |f^{\#}| = f \cdot \underline{W_{\Sigma}}(X)$ est l'algèbre des $\underline{\Sigma}$-termes sur X. Au-delà, nous allons souvent omet un soulignement "underbars" sur les signatures et algèbres, dans la conviction que cela ne causera pas de grave confusion.

3.2.2.3 Les équations

Nous définissons d'abord les équations pour une signature donnée et la notion d'une algèbre satisfait une équation. Une Σ-équation, e, est un triple $< X, \tau_1, \tau_2 >$ où X est un ensemble de S-indexées (des variables) et $\tau_1, \tau_2 \in |\underline{W_{\Sigma}}(X)|$ sont des termes sur X de la même sorte. (L'équation devrait normalement être écrit : 'pour tous les X, $\tau_1 = \tau_2$')
Le Σ-algèbre A satisfait une Σ-équation $< X, \tau_1, \tau_2 >$ si pour toutes les "maps" $f : x \to |A|, f^{\#}(\tau_1) = f^{\#}(\tau_2)$ Nous écrivons $A \models e$ A satisfait e .
Nous définissons à titre d'exemple deux formes de "functors" de la catégorie des signatures Sig. La première, $Alg : Sig \to Set^{op}$ (Set^{op} est une ensemble avec des morphismes inversés, la catégorie opposée), associe à chaque Σ une liste des ensemble de tous les Σ-algèbres ; (En fait, il représente la classe de tous les Σ-algèbres, mais la distinction : ensemble \ classe est techniquement n'a pas d'importance ici). La deuxième fonction équation : $Eqn : ALg \to Set$ associe à chaque Σ une liste des ensemble des Σ-équations.

3.2.2.4 Les théories et les théories morphismes

Nous définissons d'abord une présentation de la théorie et sa signature par certaines d'équations comme des « axiomes ». Ensuite, nous définissons la fermeture d'un ensemble d'équations comme un ensemble de toutes les équations qui ont les mêmes modèles que ceux d'origine (c.à.d que les mêmes algèbres les satisfaire).
Nous pouvons dire qu'une théorie est une signature avec un ensemble fermé d'équations. La présentation Σ-théorie est une paire $< \Sigma, E >$ où Σ est une signature et E est un ensemble de Σ-équations.
La Σ-algèbre A satisfait à une présentation de la théorie $< \Sigma, E >$ si A satisfait chaque équation dans E. Si E est un ensemble de Σ-équations, soit E^* l'ensemble de tous les Σ-algèbres qui satisfont chaque équation dans E.
Si M est un ensemble de Σ-algèbres, soit M^* l'ensemble de tous les Σ-équations qui sont satisfaites par chaque algèbre M.
Par la fermeture d'un ensemble E de Σ-équations, nous entendons l'ensemble E^{**}, on écrit $\overline{E}$ et nous disons l'ensemble E est fermé si $E = \overline{E}$.
La Σ-théorie T est une présentation de la théorie $< \Sigma, E >$ tel que E est fermé.
La Σ-théorie présentée par la présentation : $< \Sigma, E >$ est $< \Sigma, \overline{E} >$.
La Σ-théorie, ou tout simplement la « théorie », est un cas particulier de ce que nous avons d'ailleurs appellent une « théorie signée » [286]. Contrairement à Lawvere [305] la notion de la théorie algébrique comprend une signature. Il est adapté à notre objectif et plus simple à définir qu'une théorie algébrique comme une « signature indépendante ».
Que nous avons donné une définition de modèle-théorique de la fermeture. La notion de preuve

théorique correspondant est facile à définir en utilisant les règles d'inférence qui incarnent les propriétés d'équivalence de "=" et la substitution des termes équivalents en leurs termes équivalents.
Si T et T' sont des théories, dites $< \Sigma, E >$ et $< \Sigma', E' >$, par un morphisme de T théorie à T', nous entendons une signature morphisme $\sigma : \Sigma \to \Sigma'$ tel que $\sigma(e) \in E'$ pour chaque $e \in E$. (Nous écrivons $\sigma : T \to T'$). Nous allons souvent utiliser F pour désigner des théories morphismes pour les distinguer comme des morphismes de signature.
La catégorie des théories a des théories comme des objets et théorie morphismes comme des morphismes, leurs composition et leurs identités sont des morphismes de signature. (Il est facile de les voir comme une catégorie).
Il y a le "forgetfull functor" suivant $\upsilon : Th \to Sig$ prends $< \Sigma, E >$ pour Σ et $\sigma-$ à lui-même. (Notez que cette catégorie de théories a certains morphismes de signature comme des morphismes, une autre catégorie avec les mêmes objets aurait pour ses morphismes de dérivations "derivors morphisms" qui fait le "map" d'opérateurs Σ aux opérateurs Σ' dérivés. C'est à dire en termes de Σ' dérivés crées avec une séquence de liaison pour leurs variables ; mais nous aurons pas besoin de cette catégorie plus compliquée pour notre apporoche derivé de Gognen et Burstall [286]).

3.2.2.5 Les théories de données

Pour spécifier des tâches de programmation, nous voulons qu'une théorie peut inclure non seulement les équations mais également des contraintes de liberté sur les modèles. Cela signifie que certaines sortes doivent être interprétées comme des certaines structures de données, bien que d'autres peuvent être laissés comme des paramétriques sensibles à une variété d'interprétation. Par exemple, les nombres naturels, les piles et les chaînes obtiennent une interprétation unique (à l'intérieur de l'isomorphisme) sous forme de données, tandis qu'une sorte avec un ordre partiel peut être paramétrique. Les sortes paramétriques dans Clear se produiront dans les théories 'de metasort'. Nous ferons un appelle de ce type de théories : les théories de données et pour les décrire, nous commençons par la notion de contrainte.
Si Σ est une signature puis une Σ-contrainte, c, est une paire $< F : T \to T', \sigma : Signature(T') \to \Sigma >$où F est un morphisme de théorie, σ est un morphisme de signature.
Maintenant, ces contraintes peuvent être traités comme des équations, même si elles ont une structure interne différente. Juste les équations qui les font ils les imposent comme une restriction à l'ensemble des Σ-algèbres.
la Σ-algèbre A satisfait la contrainte c, nous disons $< F : T \to T', \sigma : Signature(T') \to \Sigma >$, $\underline{ssi}$ $\sigma(A)$ satisfait T' et F-free. Nous écrivons $A \models c$ dans ce cas.
Tout comme un morphisme de signature de Σ à Σ' détermine une traduction de Σ-équations à Σ'-équations, il faut aussi déterminer une traduction de Σ-contraintes à Σ'-contraintes.
Soit $\sigma : \Sigma \to \Sigma'$ un morphisme de signature et c un Σ'-contrainte, nous disons $< F, \sigma >$ alors la σ'-translation de $< F, \sigma' >$est le σ'-contrainte $< F, \sigma \cdot \sigma' >$; nous écrivons ceci comme $\sigma'(c)$.
Lemme (Constraint Satisfaction) Si $\sigma : \Sigma \to \Sigma'$ est un morphisme de signature, c est un Σ-contrainte et A' est un Σ'-algèbre alors $A' \models \sigma(c)$ $\underline{ssi}$ $\sigma(A') \models c$.
Notant que **la preuve de cette lemme** est immédiate à partir de la définition.
Maintenant, nous pouvons étendre la fonction $Eqn : Sig \to Set$ pour donner non seulement les Σ-équations, mais aussi les Σ-contraintes.
La fonction $Eqn : Sig \to Set$ prend chaque Σ-signature à l'ensemble des Σ-équations et Σ-contraintes.
Si e est une équation $Eqn(\sigma)(e) = \sigma(e)$ (comme indiqué précédemment) et si c est une équation de contrainte $Eqn(\sigma)(c)$ et elle est la σ-translation de c. Comme précédemment, nous écrivons $\sigma(e)$ et $\sigma(c)$ au lieu de $Eqn(\sigma)(e)$ ou $Eqn(\sigma)(e)$.
Depuis, nous avons étendu les définitions des $\models$ et l'équation et la Lemme de satisfaction détient pour ces définitions étendues, l'ensemble du développement de la section des théories passe par inchangée lorsque nous incluons les contraintes ainsi que les équations. Nous obtenons des notions de présentation, de la fermeture et de la théorie, appelée théorie des données pour le distinguer d'une théorie équationnelle simple.

3.2.3 La généralisation de l'approche

L'approche de définir ce que nous entendons par une théorie est de prendre la notion Lawvere de la théorie [305] généralisé à plusieurs sortes [306] ou même à des théories continues avec des termes infinis [291]. Une théorie présentée dans une logique ou mathématique manuel consiste en une signature et quelques phrases. Lawvere résumé de deux manières afin de supprimer la distinction entre les théories qui sont essentiellement les mêmes :
(1) L'abstraction d'axiome : il n'a pas d'importance notamment l'ensemble d'axiomes que nous prenons à condition qu'ils aient les mêmes conséquences. Alors, nous prenons la théorie comme une fermeture sémantique des axiomes.
(2) l'abstraction de l'opérateur : il n'a pas d'importance le jeu particulier des opérateurs primitifs que nous prenons tant que nous pouvons définir une collection similaire de termes (algèbre de Boole est essentiellement la même théorie que nous utilisons 'et', 'ou' et 'non' ou nous utilisons la course de Sheffer). Alors, nous prenons tous les opérateurs «dérivés» qui sont fabriqués à partir des primitives par '$\lambda x_1 \cdots x_n.terme$'.
Maintenant, pour notre abstraction langage de spécification axiome semble souhaitable, mais pas d'abstraction de l'opérateur, qui «perd» la signature. Donc, si nous utilisons l'idée Lawvere, ou quelque chose qui ressemble à elle, nous devons rattacher la signature. Nous pouvons le faire de la façon similaire à celle utilisée par [286]. Prisant en compte d'une catégorie de théories et d'une catégorie de signatures avec une fonction oublieux "forgetfull functor" entre eux, nous supposons qu'il y ait un adjoint gauche, donnant une théorie libre sur chaque signature dite T_Σ. Maintenant, si T' est une autre théorie et $t : T_\Sigma \to T'$, un morphisme de théorie, nous pouvons considérer $< \Sigma, t >$ comme une théorie équipée par une signature. pour de plusieurs-sortes de théories de Lawvere cette «théorie équipée de signature» est essentiellement la notion de théorie qui nous avons mis au point dans cette section. Il est appelé aussi une «théorie signé» dans le travail de Goguen et Burstall [286].
Maintenant, Goguen et Burstall ont construit des théories signées comme cela en utilisant des notions dune théorie différente de celle développée par Lawvere. Goguen et Burstall [286] prend l'approche la plus générale : les théories et les signatures sont caractérisées par des axiomes juste suffisantes pour permettre de les mettre ensemble et donc donner une base sémantique pour Clear. L'axiome implique une contiguïté entre les théories et les signatures, les co-limites, certaines propriétés de factorisation (nécessaires à la dérive) et la fermeture inductive. Ceci est la signature pour les opérateur de Clear : d'enrichir, dériver, appliquer, etc.
Une approche moins générale, plus liée à des travaux antérieurs est suivi des travaux de [307] ; il utilise la notion de monade de la théorie [308–310].Plusieurs notions abstraites de la théorie et la théorie signé sont dignes d'attention, comme la logique du premier ordre qui introduisent des quantificateurs et de traiter soigneusement avec des erreurs en utilisant des algèbres qui sont tous des basses possibles pour l'approche des opérateurs algébriques.

3.2.4 Les opérateurs de l'approche

Dans cette section, nous définissons les opérations sémantiques qui seront utilisés plus tard pendant la formalisation en Event-B. Ils utilisent les idées algébriques développés à l'article [32] ; les opérateurs originaux de Brustall [285,286] sont :*se combiner*, *enrichir* et *dériver*, correspondent aux opérations principales de Clear, et *appliquer* qui est utilisé pour définir l'application d'une procédure dans Clear correspondante à une théorie. Dans la définition de ces opérateurs sémantiques, nous donnons l'essence mathématique de la sémantique de Clear ; les équations sémantiques servent à attacher la syntaxe et d'expliquer comment les environnements sont manipulés.

3.2.4.1 Extend-signature-morphism

Étant donné un morphisme de signature et deux théories ce convertit en un morphisme entre les théories de la théorie, il (le morphisme) devient indéfini "undefined" si elles (les théories) ne pré-

servent pas les équations.

$$extend\_signature\_morphism : theory \times signature\_morphism \times theory \rightarrow theory\_morphism$$

Supposons $< \Sigma, E >$ et$< \Sigma', E' >$ sont les théories et $\sigma : \Sigma \rightarrow \Sigma'$ alors étendre-signature-morphisme est noté par $(< \Sigma, E >, \sigma, < \Sigma', E' >)$, l'extension de Σ à un morphisme de la théorie, est $\sigma :< \Sigma, E > \rightarrow < \Sigma', E' >$ si $\sigma(E) \subseteq E'$ et elle est considéré comme non-défini autrement.

3.2.4.2 Combiner

Ceci implémente l'opération '+' de Clear, en tenant compte des sous-théories

$$combine : based\_theory \times based\_theory \rightarrow based\_theory$$

combine (T, T') est le co-produit de T et T' dans la catégorie des théories fondées.

3.2.4.3 Enrichir

Un enrichissement en Clear se compose de quelques nouvelles sortes, les opérateurs et les équations. La dénotation d'un tel enrichissement est considéré comme un morphisme de la théorie pour la théorie à enrichir vers la théorie enrichie. Nous aurons besoin de quelques concepts auxiliaires, travaillant d'abord sur la nouvelle signature, puis la nouvelle théorie et enfin définir le morphisme de la théorie.

$$enriched\_signature : signature \times sort\_set \times operator\_set \rightarrow signature$$

$$enrichment : signature \times sort\_set \times operator\_set \times equation\_set \rightarrow theory\_morphism$$

3.2.4.4 L'enrichissement des données

Lorsque nous enrichissons une théorie on peut vouloir ajouter la contrainte que cet enrichissement doit être interprété librement. Pour cela nous avons besoin de l'opération :

$$data\_constrain : simple\_theory\_morphism \times data\_theory\_morphism.$$

$$add\_equality : sort\_set \times theory \rightarrow theory\_morphism$$

$$data\_constrain\_with\_equality : simple\_theory\_morphism \rightarrow data\_theory\_morphism$$

3.2.4.5 Dériver

L'opération « dériver ou derive » en Clear prend une théorie «ancienne» à base, une «nouvelle» théorie fondée, et un morphisme de la nouvelle à l'ancienne ; il ajoute aux nouvelles équations de la théorie qui détiennent pour les opérateurs correspondants dans l'ancien.

$$derive : based\_theory \times signature\_morphism \times based\_theory \rightarrow based\_theory$$

3.2.4.6 Appliquer

Appliquer est utilisé pour définir l'application d'une procédure de Clear à ses arguments. Considérons par exemple une procédure P avec le paramètre X de la "Poset" de «méta-sorte». Nous pouvons appliquer P à une autre théorie, mais quelle est la dénotation de la procédure P? Son corps est évaluée à une théorie basée sur la méta-sorte "Poset" et nous pouvons convertir cette théorie basée sur un morphisme de la théorie de la "Poset" à sa pointe .Ceci est la dénotation de P.

En général, une procédure désigne un morphisme théorique dont la source est le co-produit de son paramètre de méta-types.
Supposons que F est la dénotation du corps de la procédure, T le co-produit des paramètres méta-sorts et $F_1, \cdots, F_n$ les morphismes qui relient ces méta-types aux théories de paramètres réels. Ensuite, le résultat de l'application de la procédure de paramètres réels est donnée par une "pushout".

$$apply : based\_theory\_morphism \times based\_theory\_morphism^* \rightarrow based\_theory.$$

3.3 Le raffinement en Event-B à base operateurs

Notre approche de raffinement est basé de modéliser de façon constructif [311] et de concrétiser la manière de raffiner un modèle abstrait vers un autre concret sachant que cette approche manipule les opérateurs de raffinements entre les théories et les type abstraits de données Abstract Data Type (ADT)(expliqué dans 3.2.3) et par la suite on exploite les bases théoriques et les de donnée récemment introduites en Event-B qui sera une plateforme solide pour notre approche et rassure que notre approche valide les système dans toutes les phases de cycle de vie de la production d'un système à base NoC.

3.3.1 Les bases mathématiques en Event-B

Dans le langage mathématique Event-B (la syntaxe intérieure Event-B) [312], les termes (expressions) et des formules (prédicats) sont des catégories syntaxiques distinctes. Les termes sont définis à l'aide des constantes (par exemple, 1), les variables et les opérateurs (par exemple, $\bigcup$). Les opérateurs à terme peuvent avoir des termes comme arguments. Ils peuvent aussi avoir des formules comme arguments par exemple, $(\lambda x \cdots P(x) | E(x))$où $P(x)$ est une formule et $E(x)$ est un terme. Les Formules, d'autre part, sont construites à partir de formules de base, par exemple, $x \in S$, conjonctions et quantificateurs logiques. Les formules de base prendre termes comme arguments par exemple, $x \in S$ prend x et S comme des arguments. Les termes ont un type qui peut être l'un des éléments suivants :

- Un ensemble de base tels que $\mathbb{Z}$ ou un ensemble de support fourni par le modélisateur dans des contextes ;
- Une ensemble de partie d'un autre type ;
- Un produit cartésien de deux types.

Les opérateurs à terme ont des règles de typage de la forme :

$$\frac{type(x_1) = \alpha_1 \cdots type(x_n) = \alpha_n}{type(op(x_1, \cdots, x_n)) = \alpha}$$

Les arguments d'une formule de base doivent satisfaire sa règle de typage, par exemple, la règle de typage pour la de formule de base $finite(R)$ est :

$$\mathbf{type(R)} = \mathbb{P}(\alpha)$$

En outre les règles de typage, les opérateurs de terme ont des formules de bonne définition. $D(E)$ est utilisé pour désigner la formule bonne définition du terme E. Les obligations de preuve sont générées (si nécessaire) pour établir la bonne définition des termes apparaissant dans les modèles. Pour illustrer, nous considérons le terme $card(E)$ pour lesquelles nous disposons :

$$D(card(E)) \Leftrightarrow D(E) \wedge finite(E)$$

3.3.2 Les théorèmes polymorphes

Dans un contexte Event-B, un modélisateur peut spécifier certaines propriétés statiques du système en question au moyen des ensembles, des constantes et des axiomes. Afin de veiller à ce que ces propriétés statiques capturent la compréhension prévue du système, des théorèmes peuvent être définies dans des contextes. De même, lors de la spécification des aspects dynamiques d'un système dans une machine, certains invariants peuvent être étiquetés comme des théorèmes pour vérifier que les invariants précédemment ajoutés limitent suffisamment le système. Les théorèmes définis de telle manière sont spécifiques au modèle et surtout ne sont pas polymorphes. On présente l'ajout de théorèmes polymorphes à la composante de la théorie à atteindre les deux objectifs suivants :

- Un paquetage de propriétés importantes et réutilisables des opérateurs prédéfinis dans un bref et d'une manière vérifiable,
- vérifier que les définitions de toutes les définitions de l'opérateur nouvellement introduites capturent la compréhension prévue du modélisateur.

3.3.2.1 Définition des théorèmes polymorphes

Les théorèmes polymorphes sont des formules spéciales en Event-B où toutes les variables sauf les variables de type (**c.à.d** les paramètres de type) sont liées. Intuitivement, nous envisageons théorèmes polymorphes être utilisés dans les preuves de la façon suivante :

- Le modélisateur choisit le théorème d'incorporer dans sa preuve d'une collection de théorèmes,
- Le modélisateur fournit de type instanciations appropriées pour le séquent courant à prouver, et le théorème s'instancié avec lesdits instanciations de type et ajouté à l'ensemble des hypothèses du séquent. La définition suivante décrit les propriétés syntaxiques satisfaites par les théorèmes polymorphes.

Soit $\alpha_1, \cdots,$ *et* α_n être des paramètres de type. Une formule $P(\alpha_1, \cdots, \alpha_n)$ est une théorème polymorphe en Event-B si

$$Var(P(\alpha_1, \cdots, \alpha_n)) = \{\alpha_1, \cdots, \alpha_n\}$$

Dans ce cas, nous disons que le théorème $P(\alpha_1, \cdots, \alpha_n)$ est polymorphique sur chacun des paramètres de type $\alpha_1, \cdots,$ *et* α_n. En d'autres termes, une formule Event-B est un théorème polymorphique si ses variables libres sont tous des paramètres de type.

3.3.2.2 La validation des théorèmes polymorphes

La définition précédente décrit les propriétés syntaxiques des théorèmes polymorphes. La définition suivante présente la notion de **la solidité de théorèmes polymorphes dans un contexte**.Une théorème polymorphes $P(\alpha_1, \cdots, \alpha_n)$ en Event-B est dit **bien-défini** si les séquents suivants sont démontrables(prouvables) :

- $\vdash_D D(P(\alpha_1, \cdots, \alpha_n))$
- $\vdash_D P(\alpha_1, \cdots, \alpha_n)$

Cette définition assure que les théorèmes polymorphes sont bien définies et valides. Notez la similitude entre les séquents dans cette définition et les obligations de preuve relatifs à des théorèmes dans des contextes en Event-B [12].

3.3.2.3 L'utilisation de théorèmes polymorphes

Les règles d'inférence utilisées dans les preuves Event-B et la règle de coupe en particulier,

$$\frac{H \vdash_D D(P)\ H \vdash_D P\ H, P \vdash_D Q}{H \vdash_D Q} cut_D$$

Cette règle peut être extrêmement utile lors de la conduite des preuves qu'il imite l'approche générale adoptée lorsque vous faites des preuves en mathématiques (**c.à.d** en utilisant lemmes intermédiaires pour guider les preuves). Dans ce qui suit, nous montrons comment la règle **d'inférence** peut fournir une plate-forme solide pour l'utilisation des théorèmes polymorphes dans les preuves Event-B. Tout d'abord, nous introduisons des substitutions de type qui sont la **base** pour l'utilisation de théorèmes polymorphes.
Une substitution de type σ_t est constitué d'un type de séquence de variables (paramètres) mis en correspondance avec une séquence (de même longueur) de types. Le domaine de σ_t est l'ensemble des variables de type mappé par la substitution de type

$$P' \widehat{=} \sigma_t(P(\alpha_1, \cdots, \alpha_n))$$

Une formule P' est dit être une instance du théorème polymorphes $P(\alpha_1, \cdots, \alpha_n)$ s'il existe une substitution de type σ_t tel que :

$$\frac{\boxed{H \vdash_D D(\sigma_t(P(\alpha_1, \cdots, \alpha_n)))} \quad \boxed{H \vdash_D \sigma_t(P(\alpha_1, \cdots, \alpha_n))} \quad H, \sigma_t(P(\alpha_1, \cdots, \alpha_n)) \vdash_D Q}{H \vdash_D Q} cut_D$$

où σ_t fournit une substitution pour tous les paramètres de type apparaissant dans $P(\alpha_1, \cdots, \alpha_n)$. Une instance d'une théorème polymorphes peut être ajouté comme une hypothèse dans un séquent comme suit :

$$\frac{H, \sigma_t(P(\alpha_1, \cdots, \alpha_n)) \vdash_D Q}{H \vdash_D Q} thm_D$$

3.3.2.4 Les opérateurs polymorphes

Un nouvel opérateur polymorphes Event-B peut être définie dans une théorie en fournissant les informations suivantes :

- ***Information de parseur « Parser Information » :*** cela inclut la syntaxe, la notation (infixe ou un préfixe), et la classe syntaxique (terme ou formule). La priorité de l'opérateur ne sont pas fournis par l'utilisateur.
- ***Information de contrôle de type « Type Checker Information » :*** cela inclut les types des arguments, et le type qui en résulte si l'opérateur est un opérateur de terme.
- ***Information de prouveur « Prover Information » :*** cela inclut la bonne définition de l'opérateur ainsi que sa définition qui peut être utilisé pour raisonner à ce sujet. La Figure 4.7 décrit la structure générale d'une nouvelle définition de l'opérateur, où :

— **« syntax » :** définit la syntaxe du nouvel opérateur. Il peut être distinct de syntaxes opérateur précédemment utilisés en tant que notre approche ne permet pas la surcharge des opérateurs.

— **« prefix » ou «infix » :** définit le type de la notation qui sera utilisée pour cet opérateur soit infix (par exemple, $a\,op\,b$) ou un préfixe (par exemple, $op(a, b)$).

— **« commutative » :** indique si l'opérateur est commutative. Cette propriété particulière **pour des opérateurs déclenche** la génération d'une obligation de preuve.

— **« associative » :** indique si l'opérateur est associatif. Cette propriété particulière **pour des opérateurs déclenche** la génération d'une obligation de preuve.

— **« args » :** définit les arguments de l'opérateur. Chaque argument doit avoir un nom et un type. Les noms des arguments sont deux à deux distincts.

— **« condition » :** fournit l'état de bonne définition à générer pour cet opérateur. Nous montrerons plus loin comment les conditions de bonne définition sont correctement générés à partir de la définition ci-dessus.

— **« Definition » :** donne la définition directe de l'opérateur en termes de langage mathématique existant. La classe syntaxique de l'opérateur est déduite de la classe syntaxique de $Q(x_1, \cdots, x_n)$. Si $Q(x_1, \cdots, x_n)$ est un terme, le type résultant de l'opérateur est le type de $Q(x_1, \cdots, x_n)$.

3.3.3 Types de données abstrait

Les types de données **représentent** un élément très important dans les spécifications et les langages de programmation. En résumé **les** types de données jouent un rôle majeur dans le langage de programmation tels que Java, C ++ et le langage fonctionnel ML. Les types de données algébriques décrivent la théorie derrière la création de types et les opérations qui manipulent et produisent des éléments de ces types. A noter que les types de données abstraits peuvent être modélisées en utilisant des contextes et des machines, comme cela est effectué dans [313]. Cependant, ces modèles de types de données ne font pas le type de données spécifié disponible en tant que type pour les spécifications suivantes. Ceci est ce que nous contrastons avec le langage de spécification Maude [314] et OBJ3 [315], où la plupart des types sont construits algébriquement, et mis à disposition pour les réutiliser comme des types. L'objectif de types de données dans les théories est de fournir d'autres types (à côté du haut-types et des ensembles de support) qui peuvent être utilisés dans la modélisation, par exemple, pour le raffinement de données.
Les types de données peuvent être définis en Isabelle/HOL [239] et PVS [316] ; à la fois dans le formalisme, une théorie est facilement disponible pour raisonner sur le type de données créé. En particulier, [170, 317] **elles** donnent un aperçu de la construction de types de données à . Et comme l'a souligné Schmalz [35], la construction de types de données pourrait suivre une voie similaire dans la logique de l'Event-B. Notez que cette construction était absent dans la pratique précède la théorie. Néanmoins, il offre un point de départ pour de nouvelles recherches sur la logique de l'Event-B et de ses extensions possibles.
Comme on a **présenté** par avant et un peut détaillé dans la section **précédente**(voir 3.3.2) les types de données ou Data-types sont des ingrédients importants de nombreux formalismes et langages de programmation [317, 318] . La composition de la théorie qui peut être utilisée pour définir de nouveaux types de données sera cité par l'exemple des théories NoC,des graphe, du langage VHDL.
Dans notre discussion, nous ne fournissons pas un traitement rigoureux du sujet, et nous ne prétendons que le développement a atteint un stade de maturité. Cependant, comme il est mentionné dans [35], les types de données peuvent être ajoutés sur le dessus de la logique de l'Event-B tel que défini par Schmalz. Les restrictions syntaxiques imposées aux types de données ressemblent à ceux placés sur data-types tel que **les types** développés dans [170].
Dans ce bref traitement, nous serons concernés par les types de données qui sont générées à partir d'un certain nombre de constructeurs. Chaque élément du type peut être écrit comme un terme constructeur. De plus, les types de données sont librement générés qui exige que les constructeurs d'être distinct et injective. Cela garantit que tous les éléments du type de données nouvellement défini est désigné par un terme constructeur unique, et par conséquent, un théorème d'induction structurelle tient pour un tel type de données. Le théorème d'induction structurelle permet la définition des opérateurs par récursion primitive [170, 317]. Un nouveau type de données est introduit en fournissant les éléments suivants :

- Un opérateur type de constructeur,
- Un certain nombre de constructeurs d'éléments dont l'un doit être un constructeur de base,
- Les axiomes d'extensibilité pour assurer construit des éléments sont déterminés uniquement par leurs électeurs,
- Les axiomes Disjonctifs interarmées veillant à ce que les constructeurs distincts donnent des éléments distincts,
- Un axiome d'induction. D'une manière générale, une spécification de type de données dans la construction de la théorie a la forme suivante :

$$t(\alpha_1, \cdots, \alpha_n) ::= C_1(\pi_1^1, \cdots, \pi_1^m) | \cdots | C_k(\pi_k^1, \cdots, \pi_k^l)$$

  Où $\alpha_1, \cdots, \alpha_n$ sont des paramètres de type, $C_1, \cdots, C_k$ sont les constructeurs du nouveau type de données, et chacune des π_i^j est un type qui ne peut se référer aux paramètres de type de

type de données. Les noms de constructeur doivent être distincts. Les Types dans Event-B sont supposés d'être non vide, et cela doit tenir pour les types de données. Ainsi, chaque type de données nouvellement défini doit avoir un constructeur de base, par exemple, un constructeur qui ne se réfère pas au type de données déjà défini. En outre, le contrôle de recevabilité discuté dans [170] doit être appliquée pour éviter un problème majeur avec l'imbrication des définitions de types de données. Si le contrôle de recevabilité est supprimé, le type de données ne peut pas être construit [170]. Dans le cadre de l'Event-B, les règles de contrôle de la recevabilité de la définition de type de données suivant :

$$t(\alpha) ::= C_1 | C_2(\mathbb{P}(t))$$

car il n'y a pas de fonction injective de Type $\mathbb{P}(t) \rightarrow t$ par le théorème de Cantor.

3.3.4 Formalisation des modèles

Le raffinement est un processus de passage d'un modèle abstrait AbsM vers un modèle concret ConcM en utilisant des approches différentes , selon R. Burstall tout un langage peut être vu clairement par un type abstrait de données ou en anglais ADT model [42, 319–321] , donc les modèles en Event-B ont les structures ADT suivantes :

$Model : Context \times Machine$ `//Static and dynamic part of the system`
$Machine : Variant \times Invariants \times Events$
$Invariant : variable \times inv\_predicate$
$Events : Guards \times Actions$
$Context : sets \times Constants \times Axioms \times Theorem$

Nous pouvons aussi représenter les modèle de l'Event-B en UML comme suit (voir III.1) :

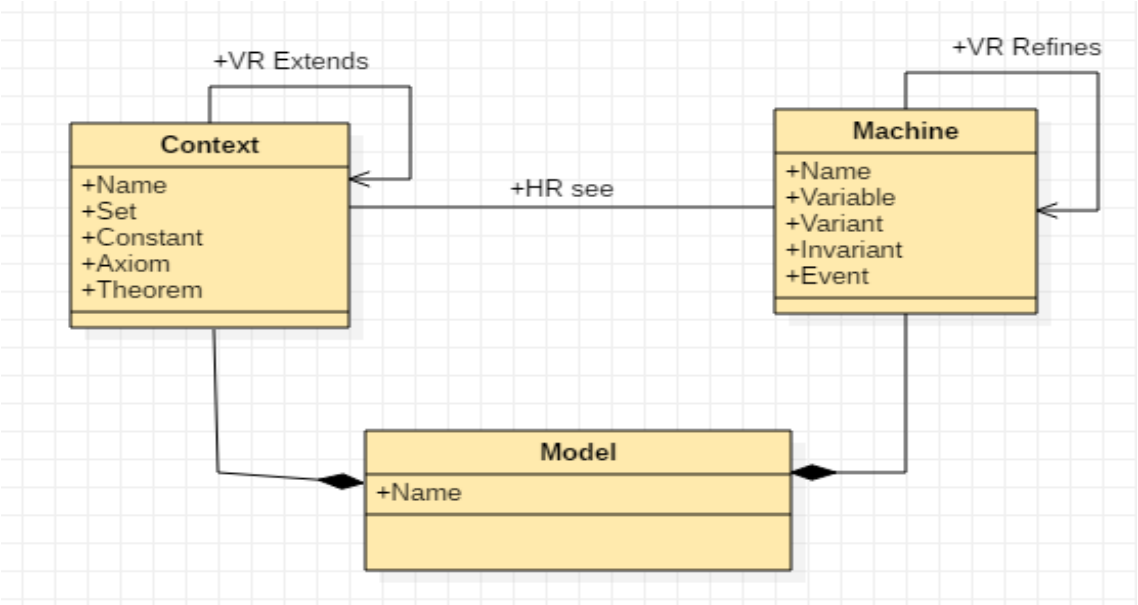


FIGURE III.1 – Diagramme de classe pour le raffinement en Event-B en générale.

- **Un modèle est vu comme une classe composée de deux classes qui expriment les contextes et les machines.**
- **Le raffinement est l'extension des classes contextes("extends") et des machines ("refines") appelés les raffinement verticaux (VR pour "Veritical refinements").**

- **Pour chaque classe modèle les déclarations sont vues comme des raffinements horizontaux(HR pour "Horizontal Refinement").**

Dans un modèle en Event-B une machine est vu d'un contexte comme suit :

$$See : (M : Model, Mach : Machine) \rightarrow Cont : Context$$
$$\forall M, Mach, Cont \cdot Mach \in M \wedge Cont \in M \Rightarrow See(M, Mach) := Cont$$

Alors on peut représenter un raffinement comme une opération entre les modèles concret et leur version abstrait :

$$Refine : (AbsM : Model) \rightarrow ConcM : Model$$
$$\forall AbsM, ConcM \cdot AbsM \neq ConcM \Rightarrow Refine(AbsM) := ConcM$$

En UML on peut présenter les cas possibles(voir III.2) d'un raffinement comme suit :

- **L'utilisateur de RODIN peut créer des classes : contextes et machines dans l'état plus abstrait du système par des déclarations expriment des propriétés statiques (sets, constants ···) et dynamiques(invariants,events, ···) .**
- **Le raffinement est l'extension des classes contextes("extends") et des machines ("refines") appelés les raffinement verticaux (VR pour "Veritical refinements").**
- **Pour chaque classe modèle les déclarations sont vues comme des raffinements horizontaux(HR pour "Horizontal Refinement").**
- **Les raffinements qui produisent le niveau N sont des actions d'extension qui touchent les parties déclarés et les parties raffinés du niveau $N-1$.**

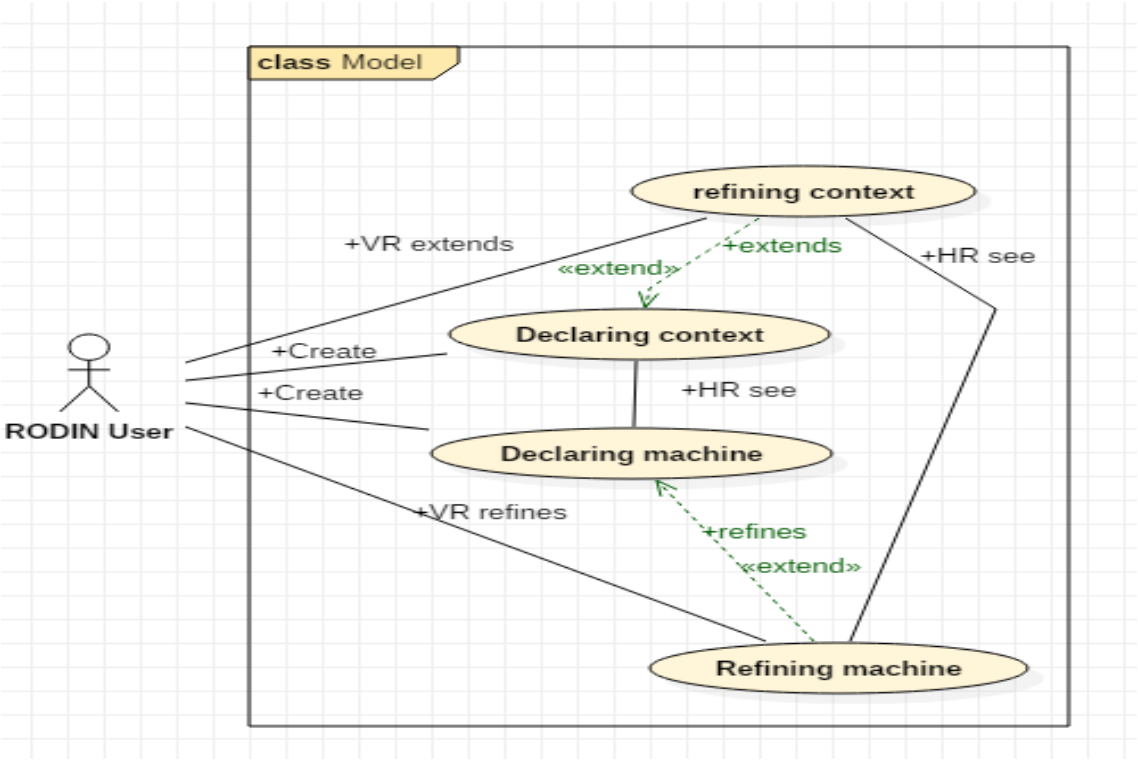


FIGURE III.2 – Diagramme de cas d'utilisation pour le raffinement en Event-B en générale.

3.3.5 Les méthodes de raffinement en Event-B

Le processus de raffinement en Event-B est mécanisé selon des approches différentes pour faciliter la tâche de concrétisation des systèmes formalisé, dans notre approche on essaye de mettre des

règles et **des conditions** pour avoir un raffinement respecte la spécification et la vérification correcte et générique en utilisant des opérateurs, et pour comprendre l'importance de notre approche on a représenté toutes les approches utilisé pour **le** raffinement par des structures ADT.

3.3.5.1 La décomposition atomique

La décomposition atomique ou en anglais simplement on dit Atomicity Decomposition (AD) est une notation graphique qui structure le raffinement basé sur l'attribution d'ordre des événements des modèles développés sous forme d'un arbre,cette approche est basé sur les diagrammes de structure de Jackson ou Jackson Structure Diagram (JSD) (leurs principe pour les flots des composants : les feuilles et les fils), AD est proposée par M. Butler [40,322] basé sur un algorithme et des règles de translation pour générer automatiquement par l'utilisation d'un plug-in spécial pour AD dans l'outil de RODIN(voir Figure III.3). Le language Atomicity Decomposition Language (ADL) représenté comme dans cette figure suivante où la ligne pointillée représente un nouvel événement et la ligne continué représente l'événement abstrait ou nouvelle introduite.
Avant ADL Toutes les combinaisons sont possibles pour passer d'un modèle AbsM à un autre ConcM, nous pouvons ajouter à des événements ou seulement quelques gardes dans les événements, mais en respectant la validation de l'obligation de preuve pour l'ensemble de l'outil de RODIN, donc le processus de **raffinement** est intuitive, mais il pourrait être une autre façon pour briser cette faiblesse, de sorte que nous adressons à l'approche de l'ADL qui introduisent une nouvelle opération pour affecter un ordonnancement entre les relations entre les événements. **Avant**

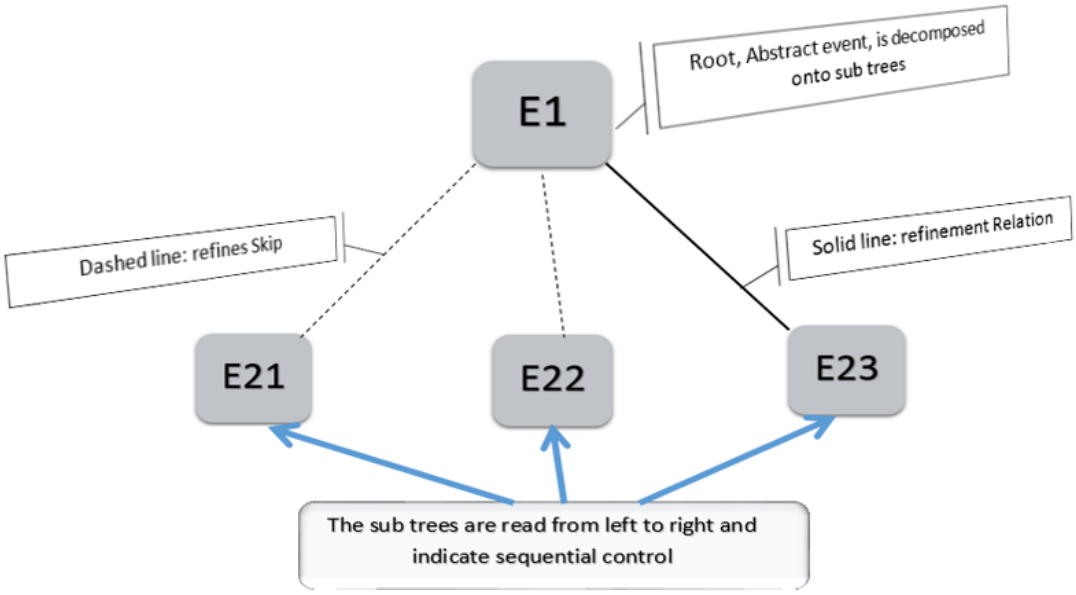


FIGURE III.3 – La décomposition atomique (The Atomicity decomposition).

tout nous pouvons présenter par UML la structure des modèles (voir III.4) en utilisent ADL comme suit :

- L'évènement est la classe de base pour un modèle sachant que le raffinement est basé sur les changements appliquée sur l'évènement.
- **Les autres classes et leurs relations sont toujours les mêmes que celle présentés dans la figure III.1**.

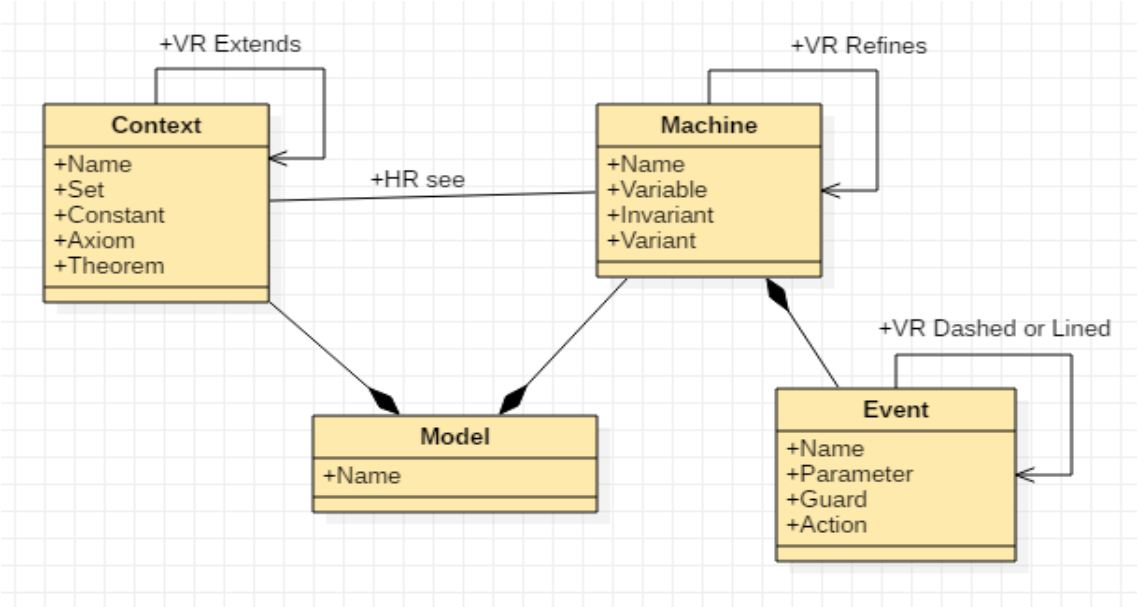


FIGURE III.4 – Le diagramme de classe pour la structure des modèle en Event-B pour ADL.

$$Dashedline : (AbsE : Events \times AbsM : Model) \rightarrow ConcE : Events \times ConcM : Model$$
$$\forall AbsE, AbsM, ConcE, ConcM \cdot AbsE \in AbsM \wedge ConcE \in ConcM \wedge AbsE \neq ConcE \wedge AbsE \neq \emptyset \wedge Refine(AbsM) := ConcM \Rightarrow Dashedline(AbsE \mapsto AbsM) := ConcE \mapsto ConcM$$

$$Solidline : (AbsM : Model \times AbsE : Events \times Prm : Parameter) \rightarrow ConcM : Model \times ConcE : Events$$
$$\forall AbsE, AbsM, Prm, ConcE, ConcM \cdot AbsE \in AbsM \wedge Prm \notin AbsE \wedge Prm \in ConcE \wedge ConcE \in ConcM \wedge Refine(AbsM) := ConcM \Rightarrow Solidline(AbsE, AbsM, Prm) := (ConcE \mapsto ConcM) \wedge (ConcE := AbsE \cup Prm)$$

En UML on peut présenter les cas possibles(voir III.2) d'un raffinement avec ADLcomme suit :

- **L'utilisateur de RODIN peut créer des classes : contextes et machines dans l'état plus abstrait du système par des déclarations expriment des propriétés statiques (sets, constants ···) et dynamiques(invariants,events, ···) .**
- **Le raffinement de création est l'extension de classe "Event"("Dashed line")et les éléments liés d'un évènement qui se trouvent dans des contextes et des machines ("Dashed line") appelés les raffinement verticaux (VR pour "Veritical refinements").**
- **Le raffinement est l'extension des classes évènements, contextes("Lined") et des machines ("Lined") appelés les raffinement verticaux (VR pour "Veritical refinements").**
- **Pour chaque classe modèle les déclarations sont vues comme des raffinements horizontaux(HR pour "Horizontal Refinement").**
- **Les raffinements des évènements qui produisent le niveau N sont des actions d'extension**

ou d'inclusion (selon les besoins) qui touchent les parties déclarés et les parties raffinés du niveau $N-1$.

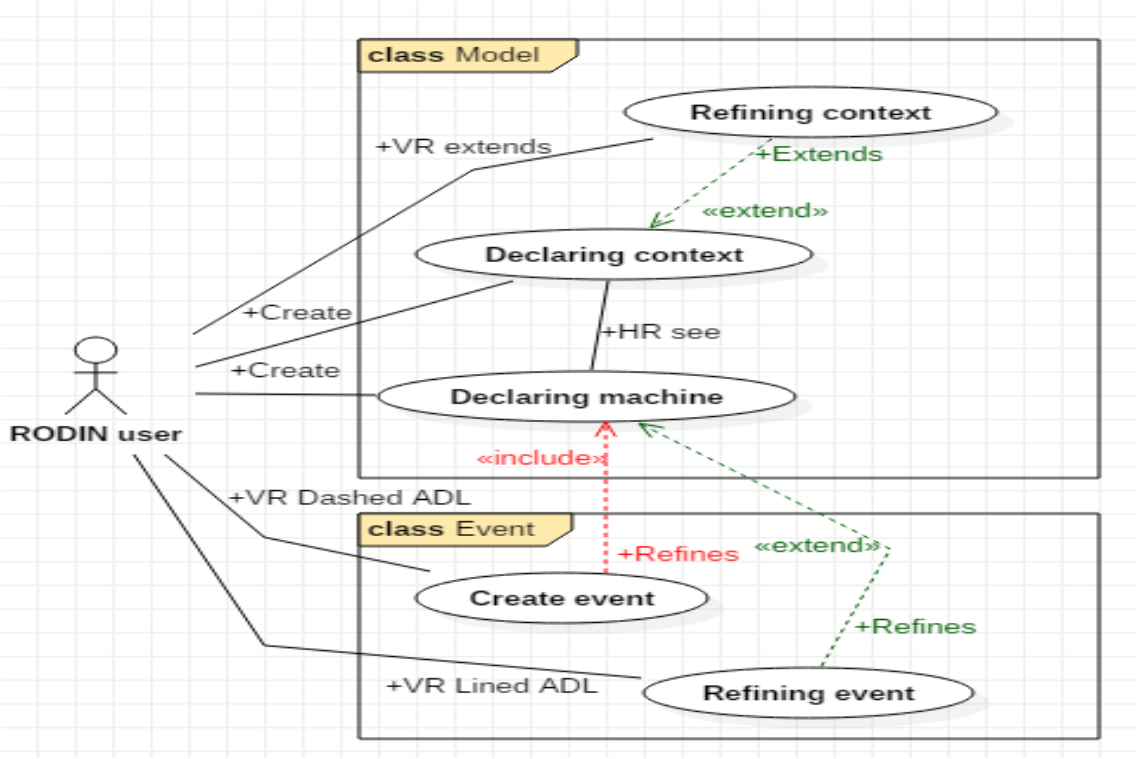


FIGURE III.5 – Diagramme de cas d'utilisation pour le raffinement en Event-B par ADL.

Et il y a tellement de règles pour savoir intuitivement dans quelle partie est le paramètre *Prm* introduit et bien sûr le plug-in de décomposition atomique dans l'outil RODIN avec l'obligation de preuve permettent de faire le raffinement, mais toujours ce raffinement n'est pas explicite parce qu'il manipule avec l'opération par un concept qui est toujours borné de contexte, machine.

3.3.5.2 L'extension de l'évènement

Dans l'outil RODIN on peut obtenir plus d'informations sur ce qui est raffiné en recherchant sur les changements ajoutés dans chaque événement ou tout simplement quelle est la nouvelle valeur introduite qui nous permettent d'étendre un événement et cela est ce mécanisme est appelé l' Extension de l'Événement ou en anglais "Event Extension".

$$
\begin{aligned}
&EventExt : (AbsE : Events \times ConcE : Events \times AbsM : Model \\
&\quad \times ConcM : Models) \rightarrow Changes : Parameter \\
&\quad \forall AbsE, AbsM, Changes, ConcE, ConcM \cdot AbsE \in AbsM \wedge ConcE \in ConcM \\
&\quad \wedge ((Changes \notin AbsE \wedge Changes \in ConcE \wedge ConcE \setminus AbsE = Changes) \\
&\quad \vee (Changes \notin AbsE \wedge Changes \in ConcE \wedge ConcM \setminus AbsM = Changes)) \\
&\quad \Rightarrow \\
&EventExt(AbsE, ConcE, AbsM, ConcM) := Changes
\end{aligned}
$$

3.3.6 Le raffinement à base d'operateurs

Notre opération de *OP_refine* prendre en considération : Invariant, événements, contexte et se soucie de l'état de ces trois avant et après le processus de raffinement, sachant que chaque événement

doit être préservé au moins par un invariants, les types d'un événement et son(ses) invariant (s) est (sont) déclarée du contexte afin : **Dans cette partie nous pouvons présenter par UML la**

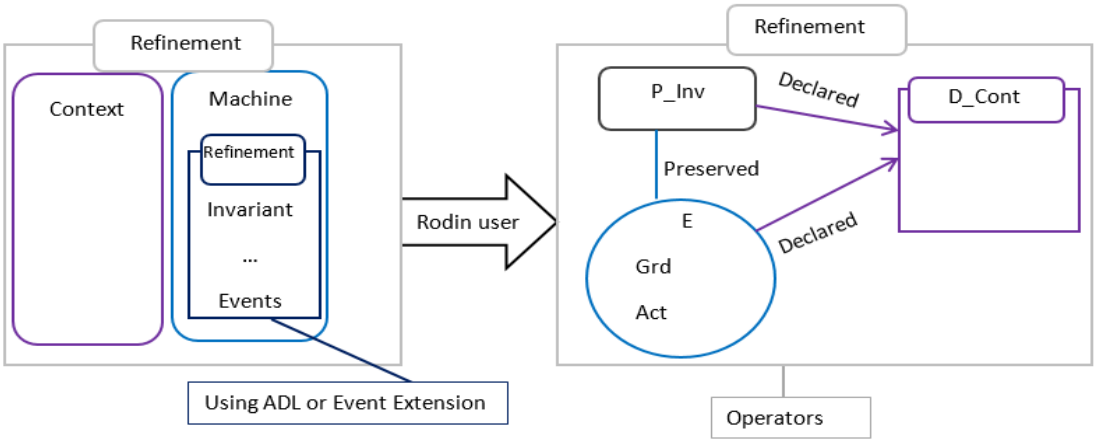


FIGURE III.6 – Notre approche de raffinement à base opérateur en Event-B.

structure des modèles (voir III.4) en utilisent "Event extension" de l'outil RODIN comme suit :

- L'évènement est la classe de base pour un modèle sachant que le raffinement est basé sur les changements appliquée sur l'évènement supervisé par un état" extended" ou "not extended".
- **Les autres classes et leurs relations sont toujours les mêmes que celle présentés dans la figure III.2.**

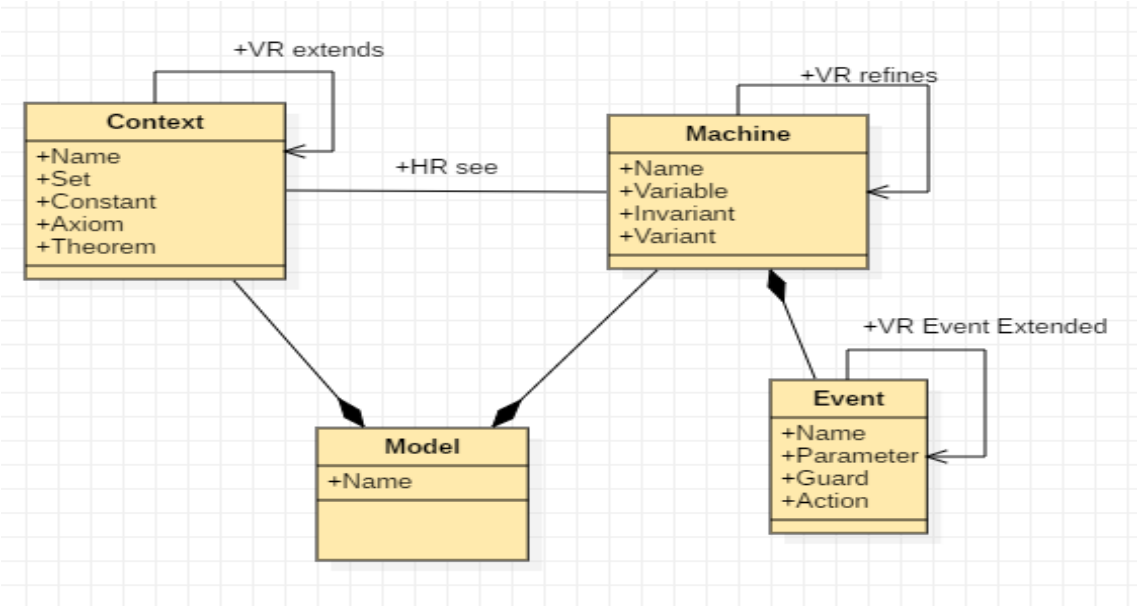


FIGURE III.7 – Le diagramme de classe pour la structure des modèle en Event-B pour l"Event Extension".

$$Preserved : (Evt : Events \times M : Models) \rightarrow P_{Invs} : Invariants$$
$$\forall Evt, n, Invs, M \cdot Evt \in M \wedge Invs \in M \wedge n \in 0 \cdots card(Invs)$$
$$\wedge ((Invs(n) \subset Guards) \vee (Invs(n) \subset Actions))$$
$$\Rightarrow$$
$$(Preseved(Evt) := P_{Invs}) \wedge (P_{Invs}(n) := P_{Invs}(n) \cup Invs(n))$$

$$
\begin{aligned}
&Declared : (Evt : Events \times P_{Invs} : Invariants \times Cont : Context) \rightarrow \\
&D_{Cont} : Context \\
&\quad \forall Evt, n, P_{Invs}, D_{Cont}, l, M \cdot Evt \in M \wedge Preserved(Evt) := P_{Invs} \\
&\quad \wedge l \in 0 \cdots card(P_{Invs}) M \wedge n \in 0 \cdots card(Cont) \\
&\quad \wedge ((P_{Invs}(l) \subset Cont) \vee (P_{Invs}(n) \subset Cont)) \\
&\quad \Rightarrow \\
&\quad (Declared(Evt, P_{Invs}, Cont) := D_{Cont}) \\
&\quad \wedge (D_{Cont}(n) := D_{Cont}(n) \cup Cont(n))
\end{aligned}
$$

RODIN utilise la notion d''Event Extension'' et en UML nous pouvons présenter les cas possibles(voir III.8) d'un raffinement avec ADLcomme suit :

- **L'utilisateur de RODIN peut créer des classes : évènements, contextes et machines dans l'état plus abstrait du système par des déclarations expriment des propriétés statiques (sets, constants ···) et dynamiques(invariants,events, ···) .**
- **Le raffinement de création est l'extension de classe ''Event''(''not extended'')et les éléments liés d'un évènement qui se trouvent dans des contextes et des machines (''not extended'') appelés les raffinement verticaux (VR pour ''Veritical refinements'').**
- **Le raffinement est l'extension des classes évènements, contextes(''extended'') et des machines (''extended'') appelés les raffinement verticaux (VR pour ''VR Event Extension'').**
- **Pour chaque classe modèle les déclarations sont vues comme des raffinements horizontaux(HR pour ''Horizontal Refinement'').**
- **Les raffinements des évènements qui produisent le niveau N sont des actions d'extension ou d'inclusion (selon les besoins) qui touchent les parties déclarés et les parties raffinés du niveau $N-1$.**

Le raffinement(voir Figure III.6) peut être fait en utilisant :

- *Rename operator*. l'**operateur de renommage**.
- *Enrich operator*. l'**operateur d'enrichissement**.
- *Restrict operator*. l**'operateur de restriction**.
- *Create operator*.l'**operateur de création**.

Si nous voulons expliquer notre approche en utilisant UML la structure des modèles (voir III.9) en utilisent ''OBR une abréviation de Operator-Based Refinement'' pour améliorer l'outil RODIN est présenté comme suit :

- L'évènement est declaré (''declared'') comme la classe (composé de deux classes : action et guard) de base pour un modèle sachant que le raffinement est basé sur les changements appliquée sur l'évènement supervisé par un état comprend quatre valeur :'' created'' , ''enriched'', ''renamed'' ou ''restricted''.
- **Les autres classes sont toujours les mêmes que celle présentés dans la figure III.2**.
- Le raffinement est fait par quatre relations : ''OBR create'' , ''OBR enrich'', ''OBR rename'' ou '' OBR restrict''.

En UML nous pouvons présenter les cas possibles(voir III.10) d'un raffinement généré par notre proposition ''OBR'' comme suit :

- **L'utilisateur de RODIN peut créer des classes en utilisant l'opérateur ''CREATE'' : évènements (guard et action sont étendu) , contextes et machines dans l'état plus abstrait du système par des déclarations expriment des propriétés statiques (sets, constants ···) et dynamiques(invariants,events, ···) .**

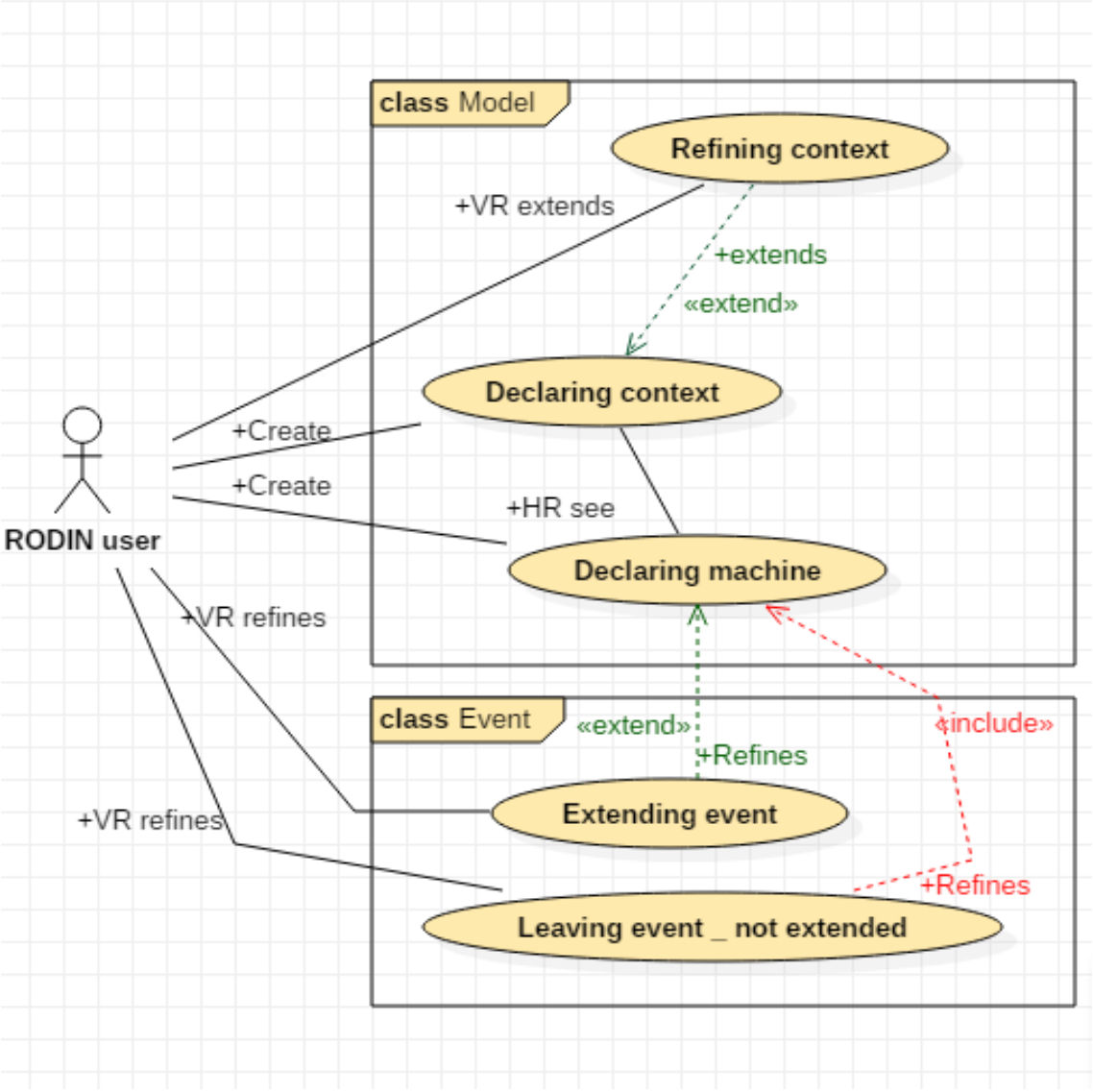


FIGURE III.8 – Diagramme de cas d'utilisation pour le raffinement en Event-B établi par "Event Extension".

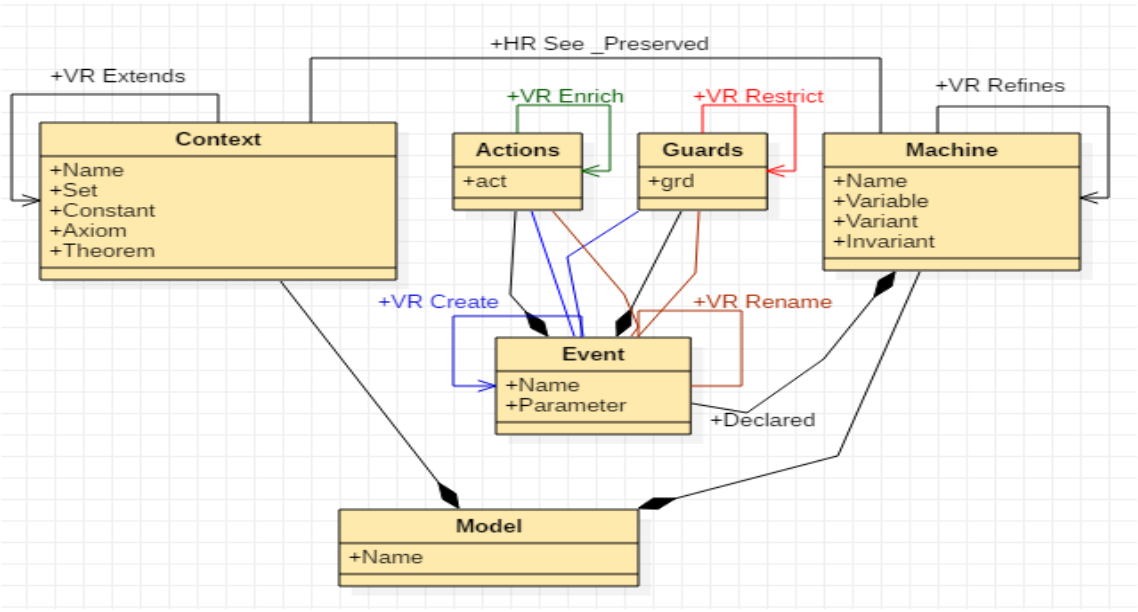


FIGURE III.9 – Le diagramme de classe pour la structure des modèles en Event-B pour notre proposition "OBR".

- **Le raffinement de renommage "RENAME" est l'extension de classe "Event" et les éléments liés d'un évènement (gaurd et action reste inchangés) qui se trouvent dans des contextes et des machines par l'état ("renamed") appelés les raffinement verticaux (VR pour "OBR Rename").**

- **Le raffinement de restriction est l'extension des classes évènements et guard (changements de guard), contextes et des machines. Ils sont appelés des raffinement verticaux (VR pour "OBR Restrict").**

- **Le raffinement d'enrichissement est l'extension des classes évènements et guard et action (changements de guard et d'action), contextes et des machines. Ils sont appelés des raffinement verticaux (VR pour "OBR Enrich").**

- **Pour chaque classe modèle les déclarations sont vues comme des raffinements horizontaux(HR pour "Horizontal Refinement").**

- **Les raffinements des évènements qui produisent le niveau N sont des actions d'extension ou d'inclusion (selon les besoins) qui touchent les parties déclarés et les parties raffinés du niveau $N-1$.**

3.3.6.1 Le renommage

On peut dire que cet opérateur (voir Figure III.11) est similaire à la ligne en pointillé un dans l'approche de ADL il est basé de prendre l'événement abstrait avec aucun changement de sorte

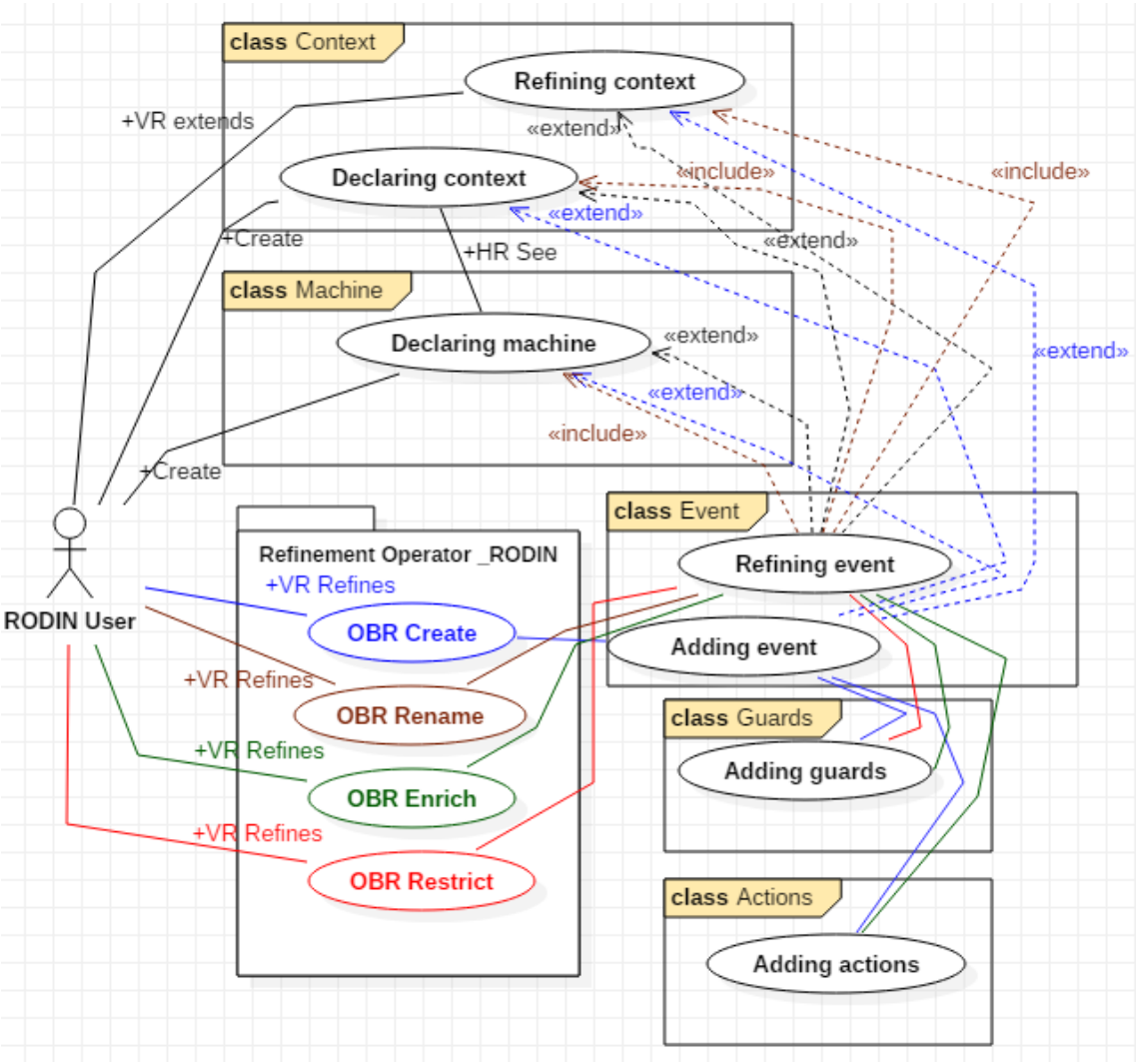


FIGURE III.10 – Diagramme de cas d'utilisation pour le raffinement en Event-B à base d'opérateur.

qu'on peut le formuler comme suit :

$$
\begin{aligned}
&Rename : (AbsE : Events \times AbsP_{Invs} : Invariants \times AbsD_{Cont} : Context) \rightarrow \\
&(ConcE : Events \times ConcP_{Invs} : Invariants \times ConcD_{Cont} : Context) \\
&\forall AbsE, ConcE, AbsC, AbsM, ConcM, AbsP_{Invs}, ConcP_{Invs}, \\
&AbsD_{Cont}, ConcD_{Invs} \cdot AbsE \in ConcM \wedge ConcE \in ConcM \\
&\wedge Preserved(AbsE, AbsM) = P_{Invs} \\
&\wedge Decalred(AbsE, AbsP_{Invs}, AbsC) = AbsD_{Cont} \\
&\Rightarrow \\
&Rename(AbsE, AbsP_{Invs}, AbsD_{Cont}) := (ConcE, ConcP_{Invs}, ConcD_{Cont}) \\
&\wedge ((ConcE := AbsE) \wedge (AbsP_{Invs} := ConcP_{Invs}) \wedge (ConcD_{Cont} := AbsD_{Cont}))
\end{aligned}
$$

Lorsque le modèle est la continuation d'un problème existe dans un modèle $Mod0$ donc le raf-

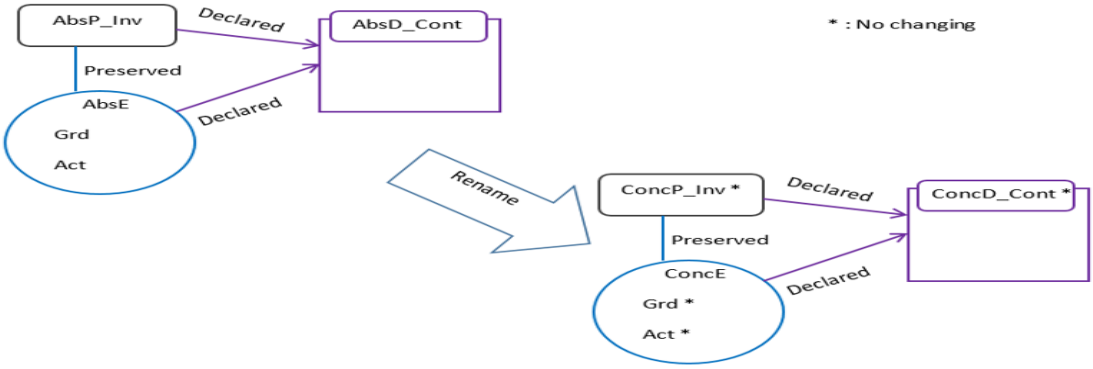


FIGURE III.11 – L'opérateur de Renommage (Rename).

finement est liée à renommer ce problème comme RMod0 nous pouvons dire qu'il est une réutilisation pour le problème de $Mod0$ et généralement nous avons besoin de changer le nom des variables et assurer la preuve de raffinement entre les variables abstraites Var_Mod0 et variables concrètes Var_RMod0, c'est-à-dire pour vérifier la validation de chacun des éléments remplacés, par exemple, remplacer un type par type et constantes par des constantes, et remplacer les variables, les constantes et les événements ne créera pas de conflit avec les noms dans la raison d'assurer la preuve obligations réutiliser, mais en remplaçant différente ensembles avec le même ensemble renommé dans le modèle est possible, lors du remplacement des ensembles qui sont définis par l'utilisateur, par exemple de ne pas remplacer ces types $\mathbb{Z}$ *et* $BOOL$.
Renommer un modèle est de prendre l'évènement préservé par des invariants et les types dans le contexte sans aucun changement de l'évènement du niveau abstrait ($AbsE$, $AbsP_{Invs}$, $AbsD_{Cont}$) vers le niveau concret($ConcE$, $ConcP_{Invs}$, $ConcD_{Cont}$).

3.3.6.2 L'enrichissement

$$
\begin{aligned}
&Enrich : (AbsE : Events \times AbsP_{Invs} : Invariants \times AbsD_{Cont} : Context \\
&\quad \times NewAct : Actions \times NewInv : Invariants \times NewCont : Context) \rightarrow \\
&(ConcE : Events \times ConcP_{Invs} : Invariants \times ConcD_{Cont} : Context) \\
&\quad \forall AbsE, ConcE, AbsC, AbsM, ConcM, AbsP_{Invs}, ConcP_{Invs}, \\
&AbsD_{Cont}, ConcD_{Invs} \cdot AbsE \in ConcM \wedge ConcE \in ConcM \\
&\quad \wedge Preserved(AbsE, AbsM) = P_{Invs} \\
&\quad \wedge Decalred(AbsE, AbsP_{Invs}, AbsC) = AbsD_{Cont} \\
&\quad \Rightarrow \\
&Enrich(AbsE, AbsP_{Invs}, AbsD_{Cont}) := (ConcE, ConcP_{Invs}, ConcD_{Cont}) \\
&\quad \wedge ((ConcE := AbsE \cup NewAct) \wedge ((ConcP_{Invs} := AbsP_{Invs} \cup NewInv) \\
&\quad \vee (ConcD_{Cont} := AbsD_{Cont} \cup newCont))
\end{aligned}
$$

Le contexte du modèle $EC0$ étendue du contexte abstrait $C0$ contient des propriétés de remplacement des ensembles ou des expressions, même pour la machine $EM0$ étendue de la $M0$ de la machine qui contiennent des variables et des événements avec de nouveaux éléments à ajouter dans les actions ; généralement l'opérateur enrichir représente une amélioration de raffinement avec les obligations de preuve, qui comprend un modèle enrichi $EDMod0$ qui doit être vu à partir d'un contexte $EC0$ lorsque les ensembles et les constantes remplacent leurs semblables dans le contexte abstrait $C0$ du modèle $MOD0$, les variables et les événements de l'$EM0$ machine ne pas créer de conflit avec leurs semblables dans la machine abstraite $M0$. La machine $EM0$ du modèle

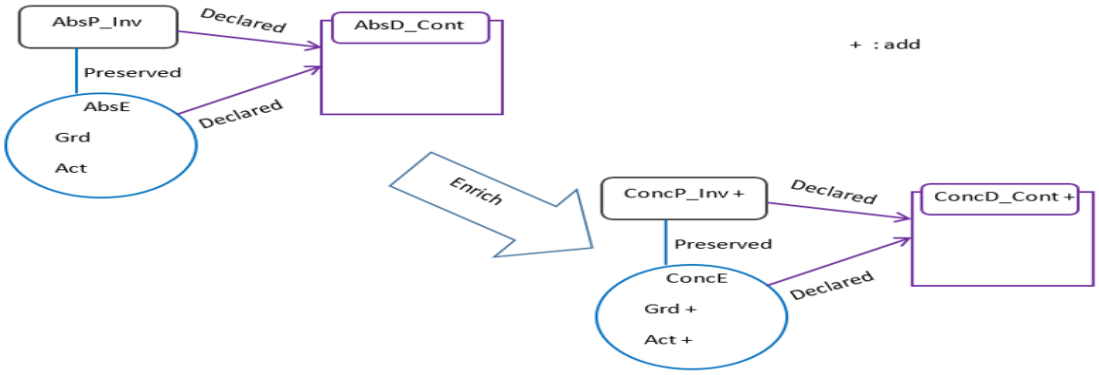


FIGURE III.12 – L'opérateur d'enrechissement (Enrich).

$EMod0$ est un raffinement de la machine $M0$ la partie dynamique du modèle $Mod0$ c'est-à-dire $M0 \subseteq EM0$ lorsqu'on enrichit un modèle (pour obtenir l'évènement $ConcE$) en ajoutant dans les actions $newAct$ de l'évènement abstrait $AbsE$ préservé par des invariants $ConcP_{Inv}$ declarés dans le contexte $ConcD_{Inv}$(voir Figure III.12).

3.3.6.3 La restriction

$$
\begin{aligned}
&Restrict : (AbsE : Events \times AbsP_{Invs} : Invariants \times AbsD_{Cont} : Context \\
&\quad \times NewGrd : Guards \times NewInv : Invariants \times NewCont : Context) \rightarrow \\
&(ConcE : Events \times ConcP_{Invs} : Invariants \times ConcD_{Cont} : Context) \\
&\quad \forall AbsE, ConcE, AbsC, AbsM, ConcM, AbsP_{Invs}, ConcP_{Invs}, \\
&AbsD_{Cont}, ConcD_{Invs} \cdot AbsE \in ConcM \wedge ConcE \in ConcM \\
&\quad \wedge Preserved(AbsE, AbsM) = P_{Invs} \\
&\quad \wedge Decalred(AbsE, AbsP_{Invs}, AbsC) = AbsD_{Cont} \\
&\quad \Rightarrow \\
&Restrict(AbsE, AbsP_{Invs}, AbsD_{Cont}) := (ConcE, ConcP_{Invs}, ConcD_{Cont}) \\
&\quad \wedge ((ConcE := AbsE \cup NewGrd) \wedge ((ConcP_{Invs} := AbsP_{Invs} \cup NewInv) \\
&\quad \vee (ConcD_{Cont} := AbsD_{Cont} \cup newCont))
\end{aligned}
$$

Dans certains cas, le modèle CMod0 est la réutilisation du modèle $Mod0$ avec une certaine par-

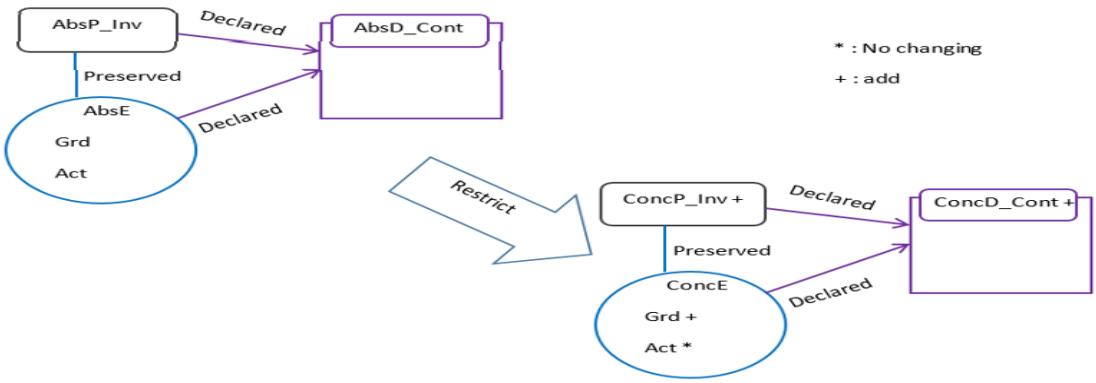


FIGURE III.13 – L'opérateur de restriction (Restrict).

ticularité dans les contraintes (gardes, invariants ou axiomes) représentés par Grd_CMod0 ou Inv_CMod0 ou même Axm_CMod0. Donc, nous pouvons dire que le raffinement avait introduit de nouvelles contraintes que $newGrd$ ou $NewCont$ à plus de clarifier un invariant ($ConcP_{inv}$ ou créé un nouveau $newInv$) conservées par l'événement $ConcE$ (voir Figure III.13).

3.3.6.4 La création

Pendant le raffinement nous avons trouvé un nouvel événement créé qu'il n'a pas eu de relation avec les autres introduites avant qu'il ne soit si juste le raffinement de vide Skip (voir Figure III.14) pour le modèle concret par un nouveau événement avec ses nouveaux invariants conservés qui a

été déclaré dans le contexte (ou re-déclaré).

$$
\begin{aligned}
&Create : (AbsE : Events \times AbsP_{Invs} : Invariants \times AbsD_{Cont} : Context \\
&\quad \times NewEvents : Events \times NewInv : Invariants \times NewCont : Context) \rightarrow \\
&(ConcE : Events \times ConcP_{Invs} : Invariants \times ConcD_{Cont} : Context) \\
&\quad \forall AbsE, ConcE, AbsC, AbsM, ConcM, AbsP_{Invs}, ConcP_{Invs}, \\
&AbsD_{Cont}, AbsCConcD_{Invs} \cdot AbsE \in ConcM \wedge ConcE \in ConcM \\
&\quad \wedge Preserved(NewEvents, AbsM) = \emptyset \\
&\quad \wedge ((Decalred(NewEvents, Preserved(NewEvents, AbsM), \\
&AbsC) = AbsD_{Cont}) \vee (Decalred(NewEvents, \\
&Preserved(NewEvents, AbsM), AbsC) = \emptyset)) \\
&\quad \Rightarrow \\
&Restrict(AbsE, AbsP_{Invs}, AbsD_{Cont}) := (ConcE, ConcP_{Invs}, ConcD_{Cont}) \\
&\quad \wedge ((ConcE := NewEvents) \wedge ((ConcP_{Invs} := AbsP_{Invs} \cup NewInv) \\
&\quad \vee (ConcD_{Cont} := AbsD_{Cont} \cup newCont))
\end{aligned}
$$

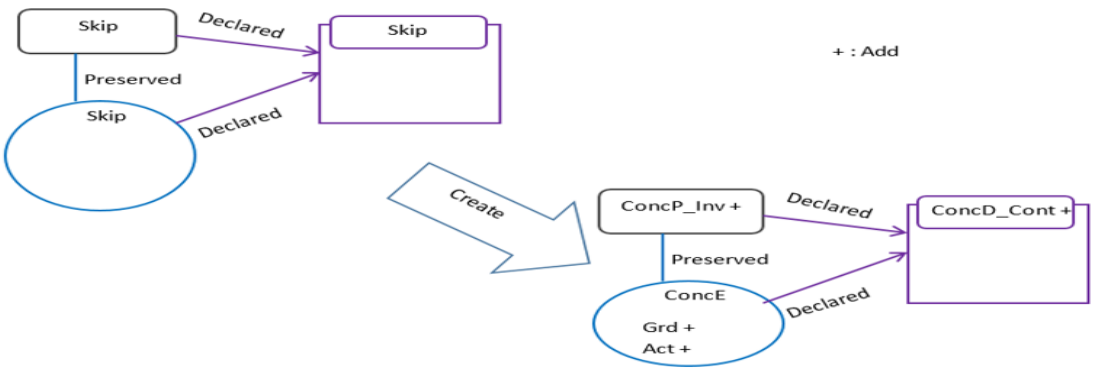


FIGURE III.14 – L'opérateur de création (Create).

3.3.7 Processus de raffinement via les opérateurs

Le raffinement en utilisant cette approche(voir Figure III.15) sera très spécifique en particulier **que** la façon de passer d'un modèle abstrait à un concret **on précise** avec cette **proposition** quelle partie est ajouté **et le composant dans lequel nous voulons** faire la spécification de contrôle **d'une manière très facilement** à manipuler et à vérifier en cas de mise à jour modèles.

3.3.8 L'intérêt de l'approche

Notre approche a eu **l'avantage d'obtenir** surtout les informations sur la transaction du raffinement intuitivement à partir d'un modèle abstrait au concret. On peut la comparer avec des approches existant lors de **la** phase de raffinement en Event-B, soit par des regles formules dans le langage ADL ou bien pour les informations tirés sur les éléments ajoutés pendant le developpement des modèles au sein de RODIN (notament dit l'extension de l'évènement « Event exytension »).

3.3.8.1 Avec ADL

ADL avait une des règles et de l'algorithme spécifiques pour contrôler le passage d'un événement à l'autre. **Lorsqu'on crée un nouveau on le représente par la ligne pointillé alors si on intro-**

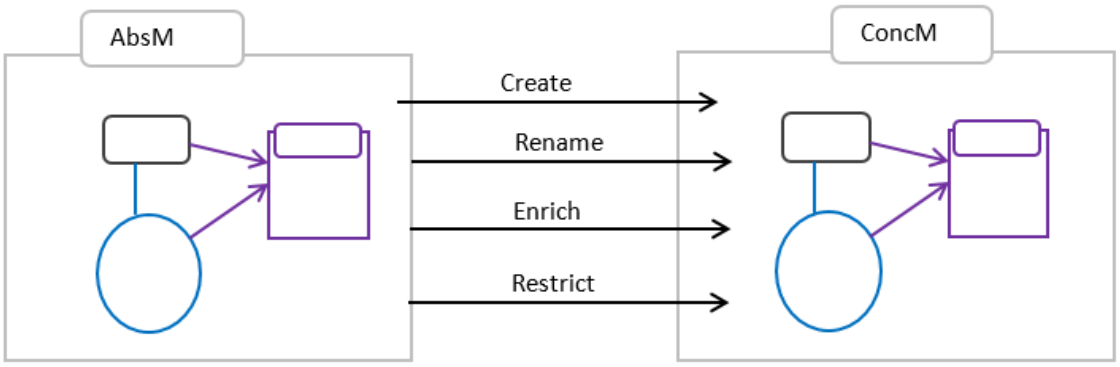


FIGURE III.15 – Le raffinement à base d'opérateurs en Event-B.

duit certains changements dans l'événement abstrait, ces nouveaux changements on peut les déduire par l'observationn de l'arbre ADL et par la consultation des règles de la langage de l'AD, mais le volume de changements possibles met le raffinement difficile à suivre en utilisant des règles. Cette difficulté de traçabilité est résolue dans notre approche(voir Figure III.16) pour le raison de mettre le développement dans RODIN plus facile à gérer.

FIGURE III.16 – L'approche de l'opérateur de raffinement vs. ADL.

3.3.8.2 Event Extension Dans l'outil RODIN

L'approche Extension de l'événement(voir figure III.17) est **utilisée pour générer** l'événement raffiné **et** pour obtenir un modèle concret et les changements **apparus peuvent être observé par l'état du drapeau (Extension flag)** dans chaque événement de façon beaucoup plus facile de l'outil RODIN, mais comment on peut savoir dans quelle partie les changements sont introduits cela n'est pas offert par ce drapeau, alors notre approche sert d'ajouter un drapeau qui font ce contrôle de raffinement par des valeurs : Enrichi, renommé, restreint, créé pour faire savoir ce qui est le changement et que fait savoir dans quelle partie **est appliqué et même** savoir s'il s'agit de prendre la même partie abstraite ou de créer un nouveau évènement.

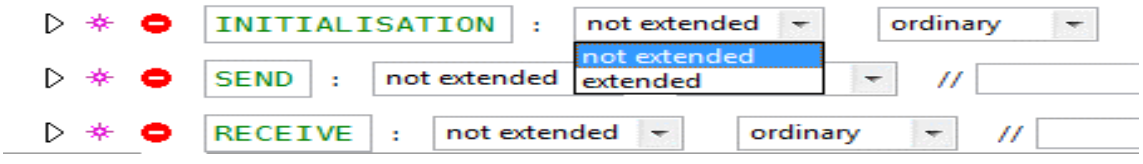


FIGURE III.17 – « Event Extesnion » en Event-B dans l'outil RODIN.

Conclusion

Le génie logiciel ne comporte qu'un nombre limité de relations de raffinement possibles. Certaines spécifications peuvent être compilées pour obtenir du code exécutable, mais elles sont, dès le départ, très liées à l'implémentation. Une deuxième catégorie de méthodologies, rassemble à la fois des approches formelles et semi-formelles, qui permettent de comparer des descriptions abstraites et concrètes entre elles au cours de la phase de conception pour vérifier leur conformité. Leur principal défaut réside dans le fait qu'elles doivent s'écarter d'un niveau de structure architecturale afin de se focaliser sur un grain plus fin de représentation. Enfin, une troisième famille d'approches, plus intéressante dans le cadre de nos travaux, est capable d'élaborer la spécification qui raffine, à partir de la spécification abstraite à raffiner, en se conformant à des règles explicites de construction correcte. Ces règles sont d'autant plus importantes qu'elles régissent la notion de conformité du "comportement raffinant" vis-à -vis du "comportement abstrait" : elles décident si le raffinement doit permettre de faire une action plus déterministe (dans éventuellement plus de cas possibles, comme par exemple dans le cas de la méthode B), une fonctionnalité plus complexe (mais qui couvre au moins ce que l'abstraction spécifie, comme par exemple pour la réduction dans la méthode Event-B).
Cette approche facilitera le flux de contrôle de spécification et fera le raffinement de manière explicite lorsque nous ne devons réflexe intuitivement comment nous avons **raffiné** le modèle abstrait nous appliquons simplement les opérateurs qui introduiront ce processus en raison de leur comportement pour les événements qui ont conservé par invariants et déclarée avec un contexte.

Chapitre 4

La validation des NoCs en Event-B par la notion "théorie" et par des opérateurs

« (1) Our approach is via theories, rather than algebras… Mathematical manipulation of theories is often more convenient than that of algebras, mainly because theory morphisms permit change of signatur…
(2) we propose an approach to support user-defined extension of the mathematical language and theory of Event-B….
(3) … Rodin has in-built support for a rich set of operators and proof rules, for some application areas there may be a need to extend the set of operators and proof rules supported by the tool… »

ROD BURSTALL (1), JEAN-RAYMOND ABRIAL (2) AND MICHAEL BUTLER(3)

Sommaire

Introduction

Il est largement admis que les méthodes formelles sont de plus en plus essentiel au développement de logiciels [17, 323, 324]. Il existe plusieurs cas qui montrent l'applicabilité et l'utilité des techniques formelles en génie logiciel [173]. Un élément important de toute méthodologie formelle réussie est le support d'outil (ou bien l'extensibilité). Ce facteur efficace facilite l'intégration des méthodes formelles dans le processus de développement des systèmes informatiques [14]. Il peut être même fait valoir que l'extensibilité **représente** le facteur le plus important pour déterminer le succès ou l'échec d'une méthode formelle.A cet égard l'extensibilité peut être présenté dans :Isabelle, PVS et surtout pour RODIN.
Isabelle [325] dispose d'un setup efficace qui combine solidité et la facilité d'utilisation. Il fournit également un mécanisme puissant pour intégrer des logiques. L'un des attributs les plus attrayantes qui ont contribué à la réussite de Isabelle est l'architecture de LCF comme il sera discuté par la suite. PVS [191] fournit un démonstrateur constitué par une variété de procédures d'inférence primitives. PVS emploie sous GNU ou X−Emacs pour fournir un environnement intégré pour **son langage et pour le prouveur**. **L'**un des nombreux points forts de PVS est la possibilité d'intégrer des procédures de décision externes (par exemple, pour l'arithmétique). **L'outil** RODIN fournit un ensemble d'outils extensible pour le développement et le raisonnement sur les modèles Event-B. RODIN comprend un ensemble d'outils qui sont nécessaires pour un environnement de développement réactif. Dans ce chapitre, nous **posons** la lumière sur quelques-unes des caractéristiques importantes de RODIN.
Ce chapitre est structuré de la façon suivante **: une description sur la** notion d'extensibité en générale et son apport dans les methodes formelles et L'outil RODIN. **Puis l**e volet théorique est introduit avec l'outillage approprié. **Nous présentons comment** le déploiement de la théorie est décrite en termes pratiques. Nous concluons **ce chapitre** en décrivant la façon dont **laquelle** les différentes extensions mathématiques et **de** prouveur peuvent être utilisées dans des modèles et **aussi dans** des preuves. Notre objectif dans ce chapitre est de montrer comment les différentes idées présentées dans le chapitre **3** ont été mises en œuvre dans le but de régler les problèmes d'extensibilité décrites dans 1.5.1. Nous avons entrepris le développement du "Theory plug-in" dans le cadre de notre recherche ; **il est pris comme une preuve de notre concept, et construit une base à évoluer** vers une plate-forme solide pour raisonner sur les extensions Event-B.

4.1 Le concept d'extensibilité

La possibilité d'intégrer des outils est un facteur primordial pour la flexibilité de développement en utilisant les méthodes formelles, l'extensibilité réside dans la phase de spécification en terme des langages extensibles, ainsi pour la vérification surtout pour les démonstrateurs. Par la suite nous tiendrons d'entamer **à** détailler l'extensibilité pour quelques méthodes formelles avant de **mettre** notre attention vers l'extension par théorie qui nous a permet de valider l'étape de raffinement pour la génération de code VHDL et généraliser l'approche de validation des systèmes comme les NoCs reconfigurables en des théories.

4.1.1 La théorie dans les méthodes formelles

La modularité est une préoccupation importante dans les spécifications et les langages de programmation. Les langages de programmation modernes comme Java et C ++ intégrer différence constructions de fournir une approche modulaire de développement de logiciels, par exemple, des classes et l'héritage. Maude est un langage et un système de réflexion prenant en charge les spécifications logique équationnelle et la réécriture et la programmation pour une large gamme d'applications [166, 314]. La Réécriture logique est une logique de changement simultané qui peut

naturellement faire face à les états et aux calculs simultanés [326]. Maude fournit un système modulaire permettant de spécifier les théories de réécriture. Chaque module fournit sortes, types et les opérateurs, et peut avoir des équations, des adhésions et des règles [314]. La construction de la théorie est similaire à un module dans Maude étant donné les facilités prévues pour spécifier les opérateurs, les types et les règles de réécriture.
Cependant, le développement de la théorie est secondaire à l'élaboration des modèles (des contextes et des machines), dans l'Event-B. Les théories ne doivent pas être considérés comme des éléments de modélisation à une spécification. Au contraire, leur rôle reste comme un **atout (les développeur de RODIN ont appelée son rôle "modelling vehicles")** de méta-raisonnement pour la logique de l'Event-B plutôt que le langage de spécification d'Event-B, autrement dit, la syntaxe externe. Une comparaison similaire peut être faite entre les théories et les modules **par le fait de choisir l'**Event-B et OBJ3 [315].
Extended ML est un cadre pour la spécification et le développement formel des programmes en Standard ML (SML). L'élaboration d'un programme en Extended ML signifie d'écrire une spécification d'un module de SML générique, puis raffiner cette spécification de manière descendante au moyen d'un certain nombre d'étapes de raffinement jusqu'à ce qu'un programme de SML est obtenue [327, 328]. La contrepartie d'un module de l'Extended ML est le fait d'une machine. Cependant, le notions parallèles peuvent être établis entre un module et une théorie. Une théorie peut être utilisée pour spécifier les opérateurs, les types et les règles de preuve de façon modulaire. Il existe des hiérarchies de théories pour spécifier un ensemble de structures mathématiques connexes. Cependant, une différence essentielle entre les théories et les modules Extended ML est que la génération de code n'est pas une exigence pour les théories. En fait, la génération de code est plus pressante dans le cas des contextes et des machines. En tant que tel, nous concluons que plus de calculs parallèles peuvent être établis entre les modèles Event-B et des modules d'Extended ML qu'entre les théories Event-B et des modules en Extended ML.
Les théories dans Isabelle [239] et PVS [316] sont similaires aux théories en Event-B, mais sont de plus grande portée. *Les théories* en Isabelle et PVS peuvent être utilisés pour effectuer des activités de modélisation et de raisonnement importants. Nous soutenons que la combinaison de la modélisation et le développement de la théorie de l'Event-B fournit un niveau comparable **en terme de sophistication aux théories de l'Isabelle et PVS. La modélisation en** Event-B utilise la théorie des ensembles qui peuvent fournir puissante force expressive qui est proche de la logique d'ordre supérieur [329]. L'ajout de la composante de la théorie assure que le polymorphisme peut être exploitée pour améliorer la puissance expressive du langage mathématique Event-B.

4.1.2 L'extensibilité de prouveur

L'architecture des outils de preuve continue de susciter beaucoup de débats chauffées. L'un des principaux points de discussion est de savoir comment trouver un équilibre raisonnable entre trois attributs importants de prouveurs : **l'**efficacité, **l'**extensibilité et de **la** solidité. Dans [330], Harrison décrit trois options pour atteindre **un** prouveur extensibilité :

(a) Si une nouvelle règle est considérée comme utile, **il faut** étendre simplement les primitives de base du prouveur qui l'inclure.

(b) Utiliser un langage de programmation complet pour spécifier de nouvelles règles en utilisant les primitives de base. Les nouvelles règles en fin de compte se décomposent à ces primitives.

(c) Intégrer le principe de réflexion, de sorte que l'utilisateur peut ajouter et vérifier de nouvelles règles au sein de l'infrastructure existante.

Beaucoup de démonstrateurs dont **lesquels** Isabelle [239] et HOL [331] emploient l'approche LCF. Le langage fonctionnel ML [257] est utilisé pour mettre en œuvre ces systèmes, tandis qu'il sert de méta-langage. L'approche adoptée par de tels systèmes est d'utiliser ML à De types de données ne correspondent **pas** à des entités logiques telles que les termes et **les** théorèmes. Un certain nombre de fonctions ML sont fournis **pour pouvoir** générer des théorèmes ; ces fonctions mettent en œuvre

les règles **basiques** d'inférence de la logique. Le système de type ML assure que les théorèmes ne sont **pas** construits par les fonctions mentionnées ci-dessus. Par conséquent, l'approche de la LCF offre à la fois la fiabilité et la contrôlabilité d'un vérificateur **de** preuve de bas niveau combinée avec la puissance et la flexibilité d'un **prouveur** sophistiqué en terme de démonstration [330]. **C**ependant, un inconvénient majeur de cette approche est **d'un autre côté,** que chaque procédure **de** preuve nouvellement développé doit se décomposer dans les règles d'inférence de base. Dans certains cas, cela peut ne pas être possible **sans proposer** une solution efficace par exemple, la méthode de table de vérité de la logique propositionnelle [332].
Le système PVS [316] suit une approche similaire à LCF avec un soutien plus libérale pour ajouter démonstrateurs externes. Cette libéralité arrive à un risque de rencontrer des bugs de solidité. Cependant Il présente à l'utilisateur plusieurs choix de démonstrateurs automatisés qui peuvent faciliter l'expérience de **preuve par un "prouver"**. Une comparaison entre et PVS, du point de vue de l'utilisateur est présenté dans [333]. **On mentionne ce qui Fait intéressant que la solidité des bugs ne sont presque jamais présenter sans les explorer** pendant la preuve, et que plupart des erreurs dans un système à vérifier sont détectés dans le processus de fabrication d'une spécification formelle. Une expérience similaire est rapporté lors de l'utilisation de la plate-forme RODIN [34].
Le système de développement formelle MURAL [334] se compose d'un outil d'aide à **la base du langage de la méthode VDM** et un assistant de preuve. Cependant, en substance, il fournit un support pour de nombreux triés calculs sous-jacentes qui sont exprimables dans le style de la déduction naturelle. Le système MURAL permet d'ajouter des règles **internement** prouvées qui découlent directement des règles existantes. Cela se traduit par l'exclusion d'une grande classe de règles qui pourraient être prouvé en utilisant un méta-raisonnement plus sophistiqué. Ajouter de nouvelles règles en MURAL est peut être réalisé par l'extension des théories existantes en fournissant un système vérifiable et «open».
Les outils de programmation tels que JML [270], ESC \Java [335], Boogie [336] (Spec # [337] programme vérificateur) et VCC [338] fournissent des capacités afin de vérifier les programmes d'ordinateur. Les conditions de vérification sont générées et transmises à des démonstrateurs externes, par exemple des solveurs SMT. Depuis la démonstration de théorèmes n'est pas un composant intégré dans ces outils, l'extensibilité de prouveur n'est pas une préoccupation immédiate. Cependant, le choix des outils hautement configurables et personnalisables est facilement disponible, par exemple, Isabelle et **le solveurs (ou raisonneur "solver") SMT**. Notant qu'une approche similaire est adoptée par VDM [205].
Le KIV [339] démonstrateur est un outil pour le développement et la vérification formelle interactive. KIV fournit **une** assistance de contrôle pour tous les éléments du langage de spécification basée sur le calcul des séquents, la réécriture et l'exécution symbolique de programmes. Ce démonstrateur suit une approche fondée sur la tactique à la preuve, et fournit un certain nombre d'heuristiques **de** preuve qui ne peuvent être modifiés ou augmentée par le développeur du système. Ses facilités ne sont pas fournis pour spécifier de nouvelles procédures de preuve par les utilisateurs du système. Cette limitation particulière de KIV est similaire aux limites était existé dans le « Tool-set » **RODIN** de l'Event-B avant **les travaux** [36, 37].
Le système de KEY [340, 341] est un « tool-set » de développement formelle de systèmes qui propose l'intégration de la conception, la mise en œuvre, spécification formelle, et la vérification formelle de logiciel orienté objet aussi transparente que possible. Taclets [342] fournissent un mécanisme par lequel les règles de preuve peuvent être définis pour le système de KEY. Par exemple, un Taclet très simple peut être écrite comme suit :

$$find(b->c ==>)\ if(b ==>)\ replacewith(c ==>)\ heuristics(simplify)$$

Ce Taclet indique qu'une implication $b \Rightarrow c$ doit être remplacée par la formule c si b se trouve dans le côté gauche de séquent (étant des hypothèses). Le Taclet précitée fait partie de l'heuristique de preuve appelées « simplify». Même si, nous ne soutenons pas explicitement directives de preuve (comme «find »), il y a un support implicite limité pour de telles constructions comme **il est** illustré

dans la demande de règle de réécriture et d'inférence des règles conditionnelles simples.
La nouvelle architecture de démonstrateur ressemble à celle de PVS. Il permet encore la libéralité de l'intégration des procédures de décision externes (par exemple, pour l'arithmétique), tout en offrant un ensemble de règles saines. D'autre part, de vérifier le bien-fondé des règles ajoutés à l'aide obligations de preuve permet méta-raisonnement au sein de la même plate-forme. Cela peut être considéré comme une intégration limitée du principe de réflexion au sein de RODIN. Les limites de cette approche, cependant, sont similaires aux limitations de l'architecture de MURAL, **dès que la méta-raisonnement sophistiquée n'est pas** possible pour le moment. (Les variables jacentes ont été ajoutés à la langue mathématique depuis RODIN 2.0. Cependant, **la méta-raisonnement** dans RODIN peut être sensiblement améliorée par des plug-ins).

4.1.3 L'extensibilité du Langage formel

L'extensibilité de la langage est une préoccupation majeure dans les méthodes formelles. Isabelle/HOL **permet d'atteindre** un bon niveau d'extensibilité grâce à la notion polymorphisme. Il a également béné de la disponibilité d'une méta-logique qui peut être utilisé pour spécifier les opérateurs avec un bon contrôle sur les représentations syntaxiques [343]. Le caractère générique de **la méthode** Isabelle permet la spécification de plusieurs logiques, et, de manière appropriée, il y a une tentative de coder Event-B dans l'Isabelle. L'extensibilité de la langue est une préoccupation réelle dans PVS comme discuté dans [344]. PVS permet l'utilisation des théories paramétrés qui offre quelques-uns des avantages de l'extensibilité de la langue telles que la réutilisation. Les deux**méthodes** PVS et Isabelle/HOL fournissent **que** des facilitations d'utilisation et les définitions récursives de types de données. Dans les deux formalismes, quand un nouveau type de données est défini, une théorie simple contenant au moins un principe d'induction est fourni.

4.2 L'extensibilité par théorie dans RODIN

Nous commençons **cette partie** en rappelant les limites de l'infrastructure existante qui a déclenché la nécessité **de proposer** notre travail. Ensuite, nous présentons la construction de la théorie qui est **l'atout (ou " modeling vehicle" comme les développeurs de RODIN aiment de l'appeler)** que nous utilisons pour spécifier **en se basant** sur les extensions.Nous **présentons dans cette partie** les travaux connexes qui ont influencé notre approche dans le traitement des prouveur et l'extensibilité de la langage. Les travaux présentés dans ce chapitre est une continuation de l'effort décrit dans [37].

4.2.1 Limites de la modélisation à base modèle

L'infrastructure de **la** modélisation en Event-B était basé sur les points suivants :

4.2.1.1 L'infrastructure existante

La plate-forme RODIN offre une infrastructure **de** preuve qui est hautement optimisé pour l'ingénierie **de** preuve et **de** la réutilisation. Mehta fournit une description **explicite** de ces infrastructures dans sa thèse [34]. Toutefois, avant le travail [36, 37], l'architecture avait des limites (2.3.4.2) pour les prouveurs externes qui peuvent être branchés sur l'infrastructure de preuve. **Nous explorons des** exemples de tels ajouts comprennent ML et PP [18]. D'autres efforts récents comprennent un solveur SMT [345] et une Traducteur/prouveur : [265]. ML et PP ne fournissent pas suffisamment d'informations sur la façon dont **laquelle** la preuve d'un séquent a été atteint. **Les preuves par ML et PP sont** exécutées dans des processus externes à **l'outil** RODIN, et seulement revient un succès ou un état d'échec sans fournir une trace de preuve à RODIN. En outre, des informations telles que l'ensemble des hypothèses nécessaires est important pour la réutilisation et la relecture

de preuve [34]. Ces propriétés de preuves sont cruciales pour un fonctionnement efficace d'un environnement de modélisation réactive.

4.2.1.2 Les constructions existantes

La modélisation en Event-B est effectuée au moyen **des** contextes et des machines (comme indiqué dans 2.2.3) de façon que : Les contextes sont utilisés pour spécifier les propriétés statiques du

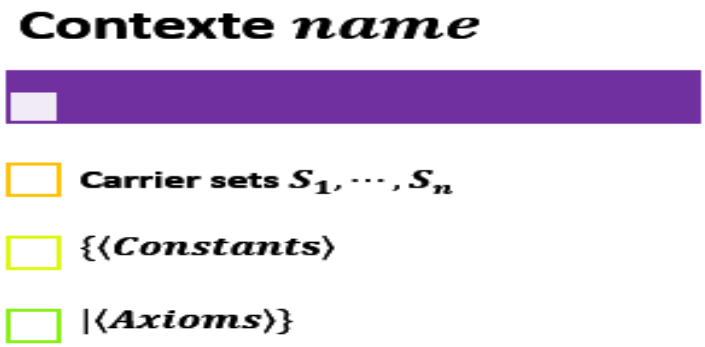


FIGURE IV.1 – La structure d'un contexte en Event-B.

système à modéliser.**Les** contextes ont la disposition générale à la figure IV.1 que les modélisateurs peuvent spécifier **des** théorèmes dans le cadre de contextes afin de garantir que les axiomes capturent leurs intentions. Les obligations de preuve appropriés sont générés pour assurer **que les** théorèmes sont bien définies et valides. Les Machines, d'autre part, sont utilisés pour spécifier les propriétés dynamiques du système. Les machines ont la disposition générale représentée sur la figure IV.2. Nous soutenons que les contextes et les machines ne sont pas adaptés pour définir les prouveurs et les langages extensibles pour les deux raisons suivantes :

(a) Les contextes et machines sont considéré comme des **atouts** de la modélisation **(ou bien «modelling vehicles »)** . Ils sont destinés pour spécifier et raisonner sur des modèles de systèmes complexes. En tant que tel, ils ne devraient pas être surchargés **pour** spécifier et en terme de méta-raisonnement sur les extensions des langages mathématiques et des prouveurs.

(b) Les contextes ont été utilisées pour définir des structures utiles axiomatique (voir l'exemple [313]), et à faciliter la preuve en soutenant des théorèmes. Toutefois, leur utilisation prévue était de paramétriser **les** machines [260]. En **effet, l'objectif** de notre travail est de prendre ce qui boldredsimplifie l'utilisation des contextes en fournissant une troisième construction indépendante des contextes et des machines afin de séparer les préoccupations,**la manipulation et la méta-raisonnement** sur les extensions. La nouvelle construction est appelé une théorie. Grâce à cette approche, contextes agissent comme un mécanisme de paramétrage pour les machines, et les théories agissent comme un espace réservé pour les extensions.

4.2.2 La modélisation à base théorie

Apres les travaux de [36, 37] la modélisation en Event-B **est améliorée** par le suivant :

4.2.2.1 La construction de la théorie

Les théories [37] sont des constructions Event-B qui sont semblables dans leur morphologie aux contextes et aux machines. Le nom de la construction est basée sur un concept similaire dans le démonstrateur Isabelle [346]. Les Théories en Event-B, cependant, diffèrent en effet des théories Isabelle. Car les théories en Isabelle peuvent être utilisés pour spécifier les théories mathématiques

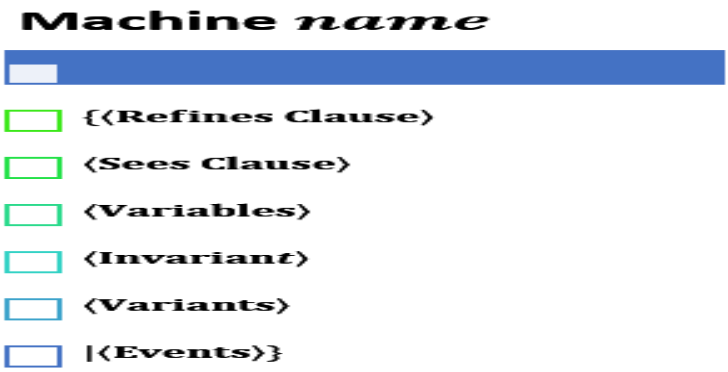


FIGURE IV.2 – La structure d'une machine en Event-B.

ainsi que des logiques entières telles que la logique d'ordre supérieur. Les notions de syntaxe interne et externe [346] renvoient à la logique de l'objet et de la méta-logique, respectivement. Une théorie de l'Event-B, au contraire, est utilisé uniquement pour la méta-raisonnement sur le langage mathématique Event-B. Une théorie agit comme un **espace réservé ou " place-holder"** pour les extensions mathématiques et **de** prouveur. La liste suivante décrit la structure globale des théories **en** Event-B. Une théorie (voir Figure IV.3) de l'Event-B a un nom qui l'identifie au sein de l'espace

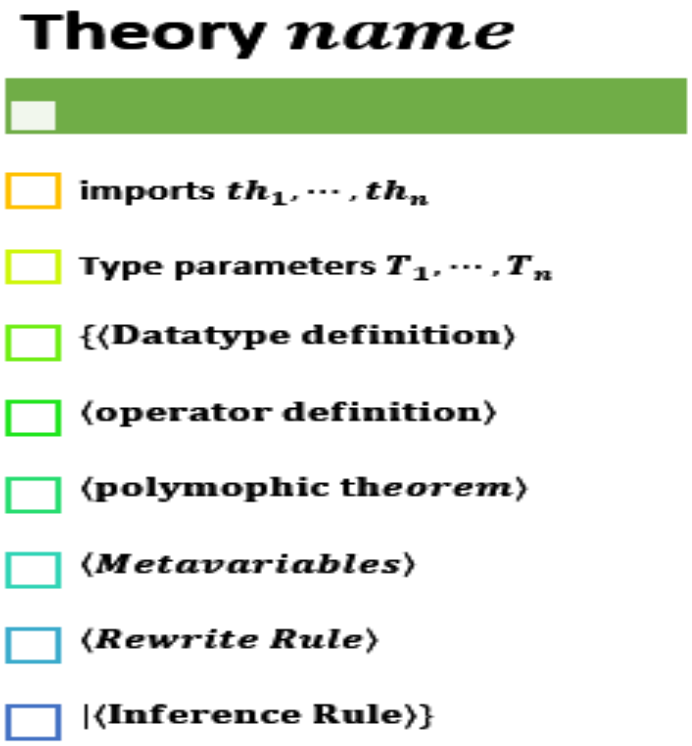


FIGURE IV.3 – La construction d'une théorie en Event-B.

de travail. Hiérarchies de théories peuvent être créés au moyen de la directive d'importation. **Une théorie** *A* importe la théorie *B* indique que toutes les définitions et les règles de la théorie *B* peuvent être utilisés dans la théorie *A*. Une théorie peut avoir un nombre arbitraire de paramètres de type qui sont des ensembles qui sont supposées **d'être** non-vides et **paire par paire (ou "pairwise" en anglais)** distincts dans ce cas, la théorie **est considérée d'être** ***polymorphe*** sur ses paramètres de type. Une théorie peut également contenir un nombre arbitraire de définitions et de règles. Dans les sections suivantes de ce chapitre, nous décrivons les règles de preuve et les théorèmes polymorphes et **nous** montrons comment ils peuvent être précisées et validées par la construction de la théorie.

La Figure IV.4 résume la nouvelle anatomie de modèles Event-B (par opposition à l'ancien anatomie **montré par** la figure IV.5) à la suite de l'introduction de la nouvelle composante de la théorie.

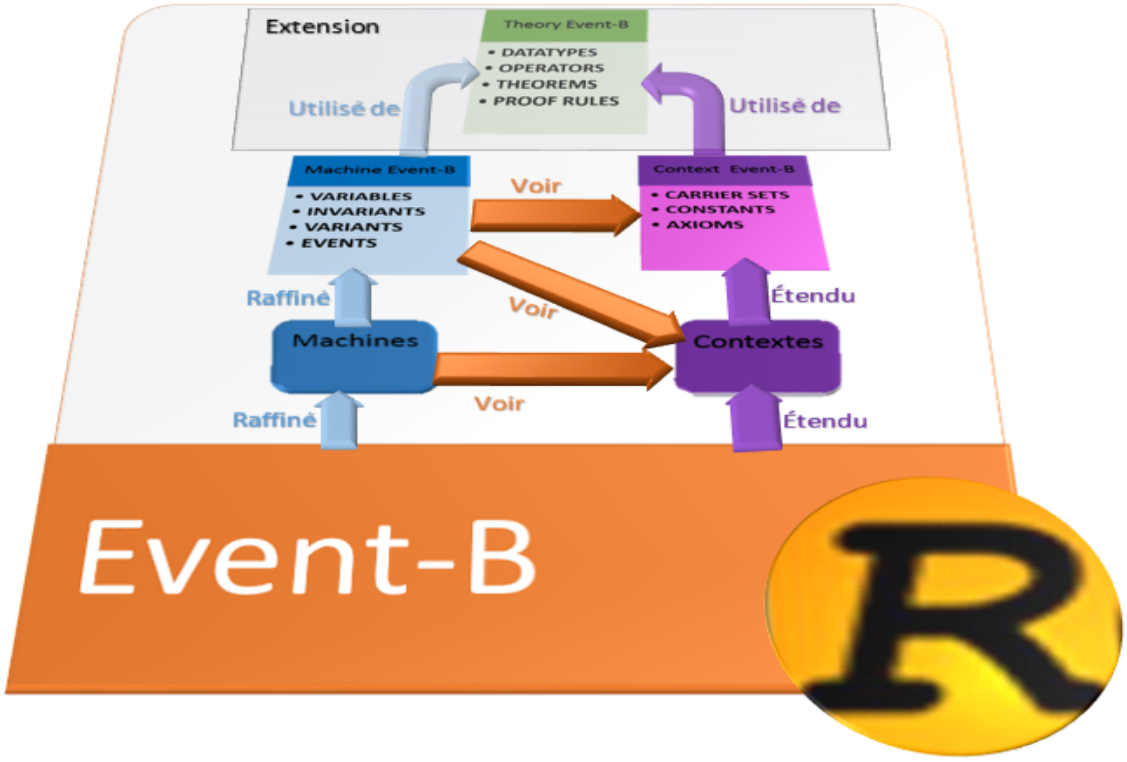


FIGURE IV.4 – La nouvelle anatomie de la modélisation en Event-B.

4.2.2.2 Préservation de solidité

Dans le processus de définition de nouvelles extensions (par exemple, nouvel opérateur ou d'une nouvelle règle de réécriture), il est possible d'introduire **la** non solidité de prouveur. **par conséquence**, il est impératif que la facilité d'utilisation de la composante de la théorie est complétée par une mesure efficace de découvrir et d'éliminer toutes les extensions de solidité en danger. En outre, nous pensons que cette mesure ne devrait pas entraver l'utilisation de tout support de l'outil fourni.
L'utilisation d'obligations de preuve est très répandue dans de nombreuses techniques formelles et **seulement** dans l'Event-B. Dans le cas de la modélisation en Event-B, les obligations de preuve fournissent **des** sémantique simples par lesquels il est possible de comprendre le système modélisé [16]. Nous soutenons que l'utilisation des obligations de preuve pour vérifier les extensions définies par l'utilisateur veillera à ce que les extensions potentiellement non solide sont portées à l'attention de l'utilisateur. En outre, étant donné que les modélisateurs ont l'habitude avec l'utilisation des obligations de preuve durant la vérification des contextes et des machines, cette approche permet d'obtenir un bon équilibre entre l'efficacité et la facilité d'utilisation. **À cet effet**, la surcharge des preuves dans des théories peut être similaire à celle dans les modèles. Cependant, la nature polymorphique des théories permet la réutilisation des preuves, par exemple, de définir et de prouver un théorème polymorphique une fois dans une théorie, puis l'utiliser plusieurs fois dans différents modèles sans avoir besoin de la re-prouver. Dans le reste de ce travail, chaque fois qu'une nouvelle extension est introduite, toutes les obligations de preuve requis sont isolés et leur adéquation est justifiée.

4.2.2.3 Déploiement de la théorie

On peut distinguer deux activités distinctes mais intrinsèquement liées dans le cadre des théories de l'Event-B : Le développement de la théorie se réfère à l'activité de la définition et à la validation des théories. A ce stade, les extensions sont définies et **les** obligations de preuve sont générés automatiquement pour chaque extension selon les besoins. Cette activité peut suivre un motif répétitif depuis inspecter tentatives de preuve automatique échoué peut révéler des informations importantes sur **la solidité des** extensions. L'exécution de preuves interactives fournit une rétroaction et **un guide pour que** le modélisateur de changer les définitions pour le cas échéant. Par conséquent, le développement de la théorie bénéficie grandement de la nature réactive de la plateforme RODIN [33, 261]. Le déploiement de la théorie se réfère à l'activité de fabrication des théories développées **et tout le temps disponibles pour les utiliser durant la phase de la modélisation**. Une théorie peut être utilisé par de nombreux modèles **aux** même temps, ce qui favorise la facilité d'utilisation. le déploiement de la théorie en sorte que les obligations de preuve sont au moins contrôlés par l'utilisateur, et une fois **elle est** déployée, toutes les extensions mathématiques et **les** règles de preuve peuvent être utilisés pour spécifier les contextes Event-B et des machines. À titre d'exemple, envisager une théorie des opérateurs pour un réseau sur puce NoC_th. L'utilisateur peut spécifier les opérateurs habituels (par exemple, send , receive ,forward, disable , relink), définir certains inférence et réécrire les règles, et de tenter **de décharger(c.à.d prouver et le terme en anglais est "discharge a proof obligation") les** obligations de preuve produites. Une fois que l'utilisateur décharge toutes les obligations de preuve produites, la théorie peut être déployé et utilisé dans un modèle qui spécifie un réseau sans file à base NoC. L'utilisation de la théorie défini des règles de preuve et les théorèmes polymorphes permet à l'utilisateur à **modéliser le système par** des extensions mathématiques sans **être lié** par l'Event-B et son langage mathématique existant au moyen de tactiques de preuve construits à cet effet.

4.2.3 Le Plug-in de la théorie

Le plug-in de la théorie facilite beaucoup des idées présentées dans cette thèse. Il **représente** notre solution aux différents problèmes d'extensibilité pendant la validation des systèmes (décrites dans 4.1.1) Ce plug-in bénéficie de la nature hautement configurable et extensible de la plateforme RODIN dans les aspects suivants :

- ***Base de données de RODIN :*** Le plug-in de la théorie contribue le volet théorique sous forme de fichier RODIN.
- ***RODIN outillage :*** Le plug-in de la théorie fournit un correcteur statique et générateur d'obligations de preuve pour les fichiers de la théorie.
- ***L'AST Dynamique :*** Le plug-in de la théorie fournit un front-end à l'analyseur dynamique RODIN pour le langage mathématique.
- ***Les raisonneurs et les tactiques :*** Le plug-in de Théorie crée dynamiquement des raisonneurs et des tactiques pour encapsuler des règles de preuve spécifiés par l'utilisateur.

Le plug-in de la théorie suit la philosophie **dans l'outil** RODIN par l'adoption de l'approche habituelle de développement réactif et l'utilisation des obligations de preuve pour assurer la préservation de la solidité.

4.2.3.1 La construction de la théorie

La construction de la théorie (composant) est **établie par un fichier qui peut agir dans l'outil RODIN** comme un espace réservé pour les extensions mathématiques et la preuve. La construction de la théorie peut être utilisée pour spécifier :

- ***Des extensions mathématiques de langage***, y compris les types de données et les opérateurs avec des définitions directes ou primitives récursive.

- ***Des extensions de preuve dont des théorèmes polymorphes***, la réécriture et les règles d'inférence. Les théories ont la structure décrite à la figure IV.4 Une théorie est paramétrée au moyen d'un certain nombre de paramètres de type. Toutes les extensions sont polymorphes sur les paramètres de type auxquels ils se réfèrent.

Une nouvelle théorie peut être créé en spécifiant son nom et son projet d'accueil conformément à la figure IV.5. Les théories en Event-B peuvent comprendre un certain nombre des éléments suivants :

(a) ***Les importations de la théorie :*** Ceci indique une relation dirigée entre la théorie de parent (l'importateur) et la théorie de référence (l'importée). La théorie de l'importateur peut se référer et utiliser l'une des extensions définies dans la théorie importée. La relation à l'importation permet la théorie de l'importation d'utiliser toutes les extensions mathématiques et preuve de ni dans la théorie importée. La directive import permet la création de hiérarchies de la théorie. Par exemple, deux théories distinctes peuvent être créés pour les séquences et les listes inductives, et une troisième théorie importer les deux théories précédentes peuvent être créés pour spécifier un isomorphisme entre les séquences et les listes inductives. En effet, la directive d'importation établit un ordre partiel sur la collecte de théories au sein d'un projet. Les théories importées ne doivent pas être instanciés avec des paramètres de type **similaire à ce qui existe pour la méthode** PVS [191].

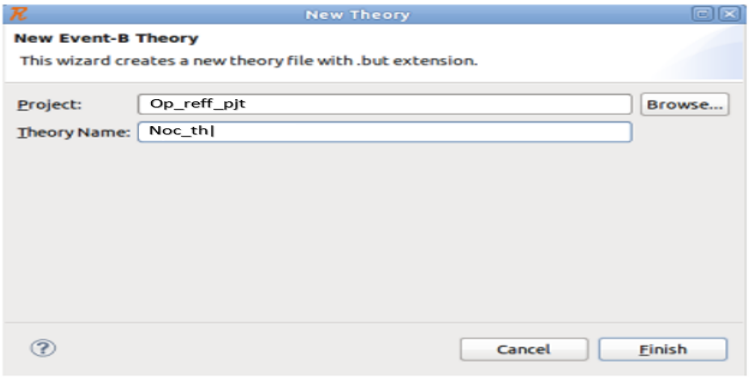


FIGURE IV.5 – La création d'une théorie dans RODIN.

(b) ***Les types des paramètres :*** Ceci définit les types sur lesquels les extensions de théorie peuvent être polymorphe. Les Types de paramètres sont similaires à des ensembles de support dans des contextes ; la seule hypothèse concernant les types de paramètres est qu'ils ne peuvent pas être vide.

(c) ***Les types de données :*** Les types de données (**comme il montre** l'exemple de La figure IV.6) sont définis en fournissant les informations suivantes :

- La syntaxe des expressions de type par exemple, *pio* (signifie les port d'entrée sortie),
- Les paramètres de type de type de données, par exemple, un seul type de paramètre T pour la Pio,
- Un certain nombre de constructeurs d'éléments **par exemple : *in*, *out* et *inout* pour exprimer les différentes sortes de port**. Chaque constructeur peut avoir un certain nombre de destructeurs (accesseurs), par exemple, la tête et les accessoires de la queue par contre.

(d) ***Les opérateurs :*** Les opérateurs sont définis en fournissant les informations suivantes :

(A) Le signal de la syntaxe de l'opérateur,

(B) La catégorie syntaxique (prédicat ou l'expression),

(C) La notation (Préfix et infixe sont actuellement pris en charge),

(D) La liste des arguments et leurs types,

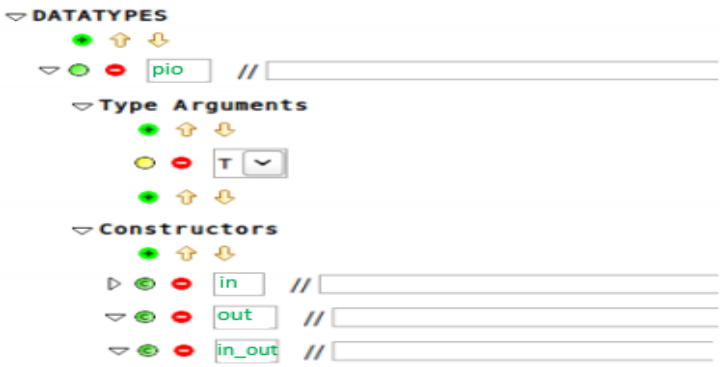


FIGURE IV.6 – La definition de type de donnée pio.

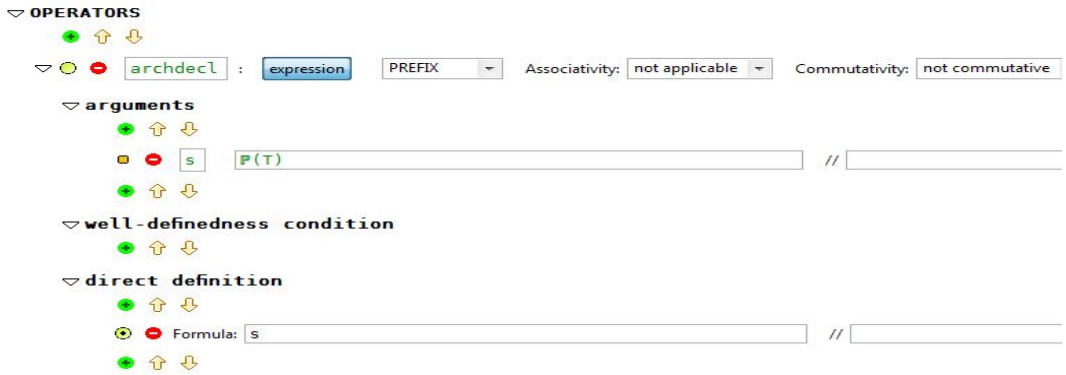


FIGURE IV.7 – La définition direct de l'opérateur archdecl.

(E) Les conditions dans lesquelles l'opérateur doit être utilisé,

(F) De définition qui peut être *directe* ou *récursive primitive*.

La Figure IV.7 illustre une définition directe pour l'opérateur de déclaration des type en code VHDL archdecl. La Figure IV.8, d'autre part, illustre une définition récursive primitive pour l'opérateur de taille de la liste.

(e) ***Les théorèmes :*** Un théorème polymorphe peut être ajouté en spécifiant son nom (par exemple, son identifiant) et sa formule. La figure IV.9 montre un théorème simple, à propos de la finitude de coulours *colors* dans la théorie des graphes colorés par quatre couleurs au maximum .

(f) ***Les règles de preuve :*** Deux types de règles de preuve peuvent être définis : *la réécriture* et *l'inférence*. Les méta-variables sont utilisées comme des modèles variables dans les règles pour faciliter la recherche de motifs, et l'inférence de type et la vérification. Une méta-variable (comme le montre la figure IV.10) a un nom et un type.

(A) **Les règles de réécriture :** Une règle de réécriture peut être défini en fournissant les informations suivantes (voir figure IV.11) :

(i) Le côté gauche pour être réécrite,

(ii) L'applicabilité de la règle (automatique, interactive ou les deux),

(iii) La description de la règle selon les besoins de l'interface utilisateur,

(iv) Les côtés de la main droite à laquelle le côté gauche peut être réécrite ; de chaque côté de la main droite est gardée par un état.

(B) **Les règles d'inférence :** Une règle d'inférence (Figure IV.12) peut être définie en fournissant les informations suivantes :

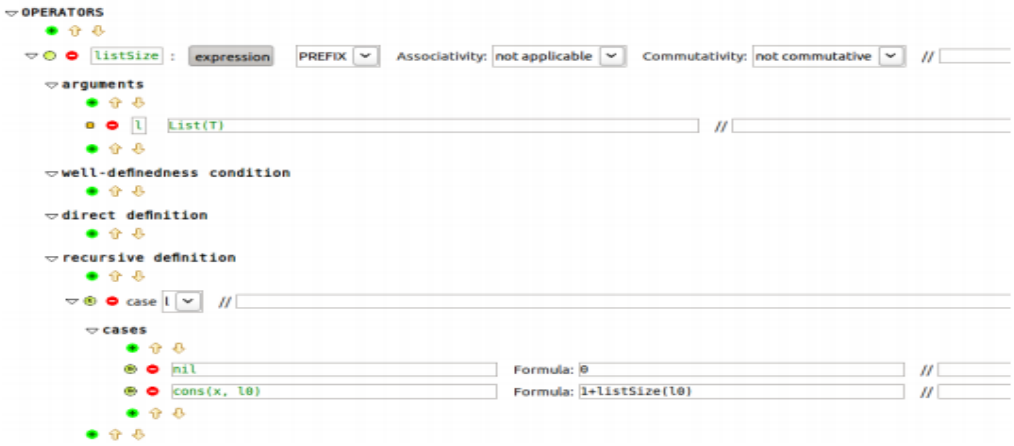


FIGURE IV.8 – La définition récursive de l'opérateur sizelist.

FIGURE IV.9 – Le théorème polymorphique.

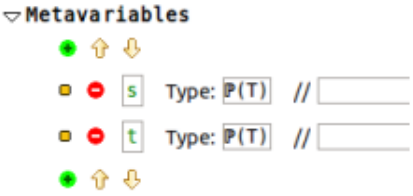


FIGURE IV.10 – Les méta-variables.

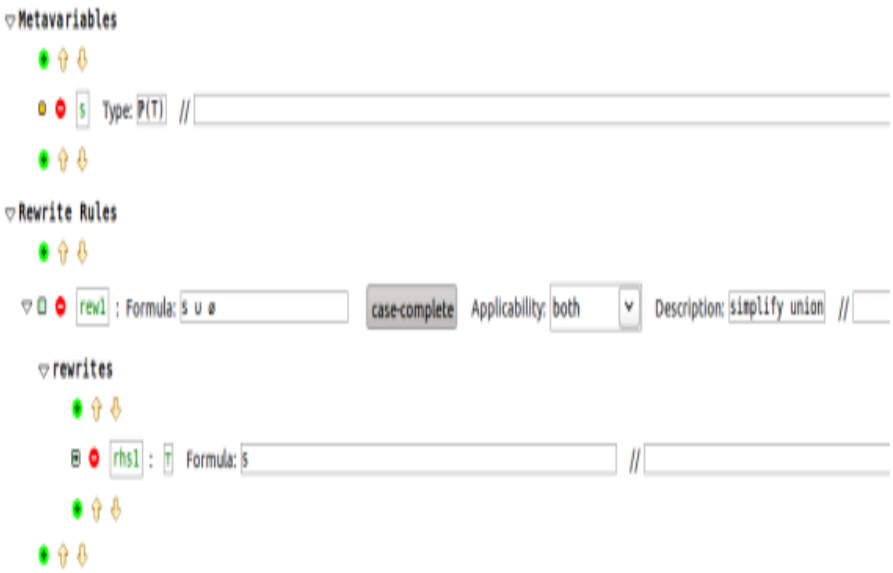


FIGURE IV.11 – La règle de réécriture.

(i) L'applicabilité de la règle (automatique, interactive ou les deux),

(ii) La description de la règle selon les besoins de l'interface utilisateur,

(iii) Les clauses données de la règle d'inférence,

(iv) La clause de la règle d'inférence de déduire.

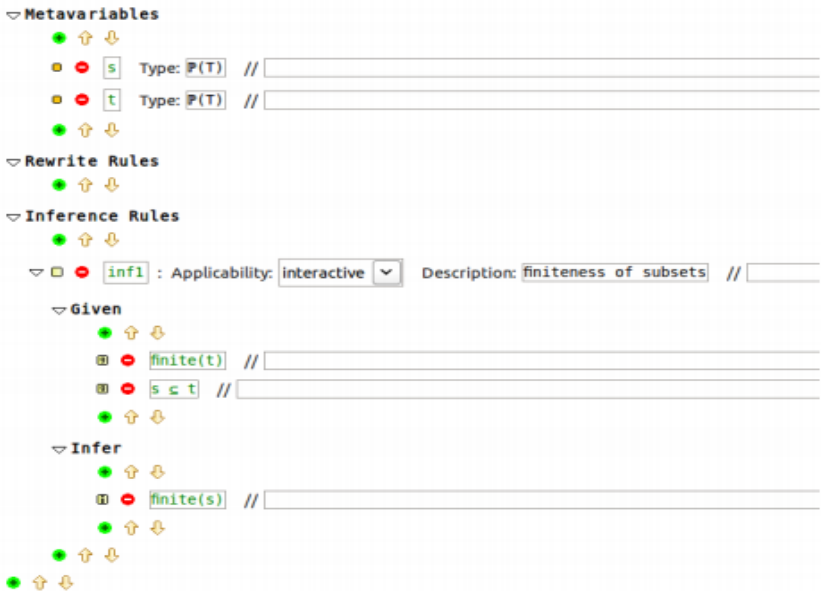


FIGURE IV.12 – La règle d'inférence.

4.2.4 La vérification statique de la théorie

Les théories en Event-B sont soumis à un contrôle statique. (theorie static checker) inspecte une théorie non vérifiée avec le extension '**.tuf** ', et la théorie vérifié avec l'extension '**.tcf** '. La liste non exhaustive suivante énumère les contrôles mis en œuvre dans la théorie :

4.2.4.1 Les contrôles à l'importation

Il est nécessaire de vérifier pendant l'importation des théories :

- Non-circularité de la relation à l'importation,
- Redondance de la relation à l'importation : en cas d'une théorie est importé plusieurs fois ; ce qui est particulièrement utile si une théorie est importée directement (à l'aide d'une importation directive) et indirectement (en vertu de la transitivité de l'importation directive).

4.2.4.2 Les contrôles de type de données

Pendant l'introduction d'un type de données le contrôle est concerné :

- L'incompatibilité de syntaxe des symboles : pour l'expression de type ainsi que les constructeurs et les accesseurs,
- Présence d'un constructeur de base : chaque définition de type de données doit inclure un constructeur de base,
- La vérification de l'admissibilité : voir les type de données (cité par avant) pour plus sur ce contrôle.

4.2.4.3 Le contrôle de l'opérateur

Un opérateur dans une théorie en Event-B statiquement vérifié pour :

- L'incompatibilité de syntaxe des symboles : pour les symboles de la syntaxe de l'opérateur,
- L'analyse et le typage des arguments de l'opérateur,
- L'analyse et la vérification du type de conditions de bonne définition WD,
- L'analyse et la vérification du type de définitions directes,
- Unicité de définition : une seule définition est autorisée pour chaque opérateur,
- La couverture de constructeur : pour les définitions récursives primitives,
- L'analyse de propriétés d'opérateur : par exemple, un opérateur avec un seul argument ne peut pas être considéré associative ou commutative.

4.2.4.4 Les contrôles de Théorème

Lorsqu'une théorie contient des théorèmes le contrôle statique est lancé pour :

- L'analyse et la vérification du type de la formule,
- L'analyse de variables : ce qui garantit que seules les variables libres du théorème sont des paramètres de type.

4.2.4.5 Le contrôle de règle de réécriture

Pendant la preuve des éléments composants d'une théorie il est possible d'avoir des règles de réécriture alors le contrôle statique couvre surtout pour les deux côtés de la règle :

- L'analyse et la vérification du type du côté de la gauche de la règle,
- Le côté gauche est pas une variable de contrôle,
- Présence au moins d'un côté du côté droite,
- L'analyse de variables : ce qui garantit que le côté droite **de la formule** se réfère uniquement aux variables se présente dans le côté gauche,
- L'analyse de classe syntaxique : ce qui garantit que les côtés de la main droite sont de la même catégorie syntaxique du côté gauche.
- L'analyse de type des côtés : Cela garantit que les deux côtés de la règle ont le même type en Event-B.

4.2.4.6 Le contrôle de règle d'inférence

La preuve d'une théorie en Event-B des fois besoin des règles d'inférence donc le contrôle est pour :

- La présence de la clause de déduction (*infer*) : chaque règle d'inférence doit avoir une déduire qui est syntaxiquement différent de $\perp$
- L'analyse et la vérification du type de clauses,
- Le contrôle de la possibilité d'application : faire en sorte que la règle d'inférence est applicable au moins dans une direction.

4.2.5 Génération d'obligation de preuve de la théorie

Les théories en Event-B sont soumis à la génération d'obligations de preuve. Le générateur de l'obligation de preuve pour une théorie inspecte le vérificateur statique (SC) de la théorie (avec le extension '**.tcf** '), et génère l'obligation de preuve appropriée pour chaque élément inspecté. Les obligations de preuve générés sont décrits au chapitre 2.3 En résumé, l'outillage réservé pour les théories suit la même approche de RODIN (voir Figure IV.13) comme il est décrit à la Figure II.7.

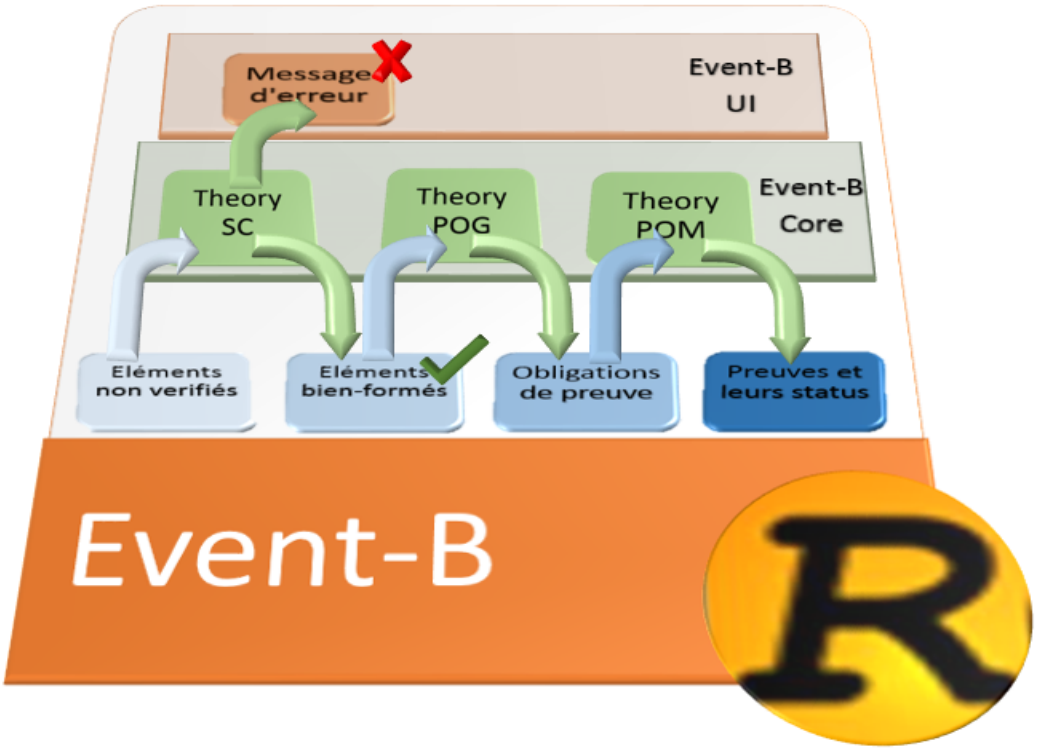


FIGURE IV.13 – L'enchainement des outils pour les théorie en Event-B.

4.2.6 Le déploiement de la théorie

Le développement de la théorie est effectué séparément de la modélisation. Cela se justifie par la nature différente de la modélisation et de la méta-raisonnement. Dans cette sorte de développement, les théories sont créées et organisées en hiérarchies.
Idéalement, chaque théorie doit definir une structure majeure mathématique, par exemple, la séquence, et les opérateurs de soutien et les règles de preuve. Si une théorie contre-structure est nécessaire, une théorie différente peut être créé dans ce raison, et l'importation directive permet une telle théorie de se référer à des théories nécessaires. Les obligations de preuve produits par les

théories doivent être déchargés (ou "discharged") par l'utilisateur pour assurer la préservation de la solidité,cependant, **elles** ne sont pas appliquées par l'outil. **À un niveau antérieur de la flexibilité de ce plug-in de l'outil RODIN durant cycle de vie , l'application de cette exigence peut avoir entravé plusieurs outils flexibles** en vue de la mesure que l'utilisateur est concerné. Les versions de futures pour le plug-in de la théorie peuvent appliquer ces bonnes pratiques notamment. Le

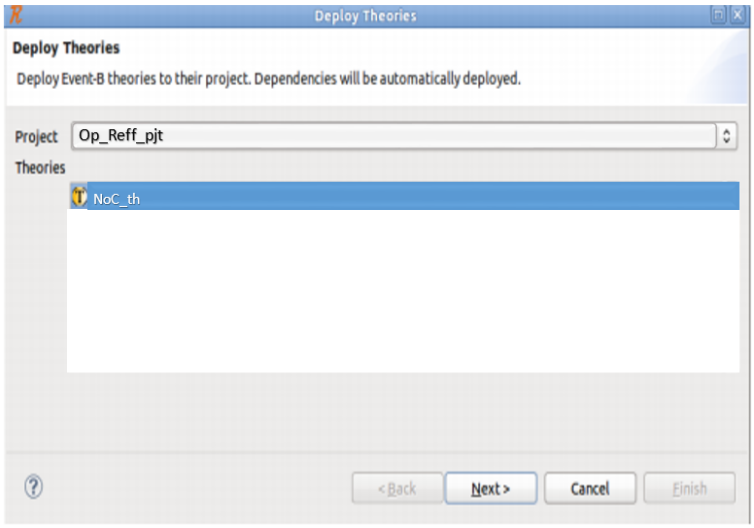


FIGURE IV.14 – La fenêtre de déploiement de la théorie.

déploiement de la théorie (voir Figure IV.14) est le processus par lequel les théories deviennent disponibles pour la modélisation. Nous voulons dire par « la disponibilité pour la modélisation », que les extensions mathématiques et de preuve peuvent être utilisés et liées lors de l'élaboration des modèles et effectuer des preuves. C'est un processus continu ; pas de mesures supplémentaires **qui** sont requises par l'utilisateur final. Parlant techniquement, le déploiement de la théorie crée la théorie déployé le (avec le extension "**.dtf** "), qui est une copie exacte du fichier statiquement vérifié de la théorie (ce qu'il lecteur peut se demander à propos de la nécessité d'un autre fichier si elle est juste une copie exacte). La motivation derrière cette décision de conception est de garder la modélisation et **la** méta-raisonnement comme des activités séparées. Le fichier statiquement vérifié de la théorie est utilisé pour la méta-raisonnement, et le fichier de la théorie déployé est utilisé pour la modélisation.
On rajoute avec la vérification statique et la génération d'obligation de preuve, le déploiement de la théorie est un processus initié par l'utilisateur. Les dépendances entre les théories (au moyen de l'importation directive) sont observées automatiquement par le processus de déploiement. Le processus de déploiement d'une théorie assure que ses théories importées sont également déployées, créant ainsi une hiérarchie des théories déployées qui reflète la hiérarchie de la théorie statiquement vérifiée. Le déploiement de la théorie permet d'atteindre les deux objectifs suivants :

- Permet à l'utilisateur final d'inspecter théories en terme de **la** solidité en observant l'état des obligations de preuve,
- Découple **la modélisation et la** méta-raisonnement. Les théories déployées sont les seules théories disponibles pour une utilisation dans les modèles.

4.2.7 Le téléchargement des Extensions

Les extensions mathématiques et de preuve sont chargés à partir des théories déployées. Les théories peuvent avoir l'un des deux champs globale ou bien locale comme il sera discuté par la suite. Charger des extensions représente un processus initié par l'outil. Cependant, l'utilisateur peut exercer un contrôle sur ce qui est chargé par l'édition « modification des théories ». La raison derrière

les champs de théories est le suivant **: c**ertaines théories sont suffisamment pour être fournie dans le cadre d'une bibliothèque par exemple, la séquence, listes et l'ordre général. Ces théories doivent avoir une portée globale. D'autres théories peuvent être pour un projet spécifique et à ce titre elles doivent avoir une portée locale.

4.2.7.1 Extensions à portée globale

Aussi connu comme « portée d'espace de travail ». Cela fait référence à des théories qui font partie d'un projet global désigné. (Dans la version du plug-in Theory (1.3.1), le projet global est appelé « MathExtensions ». Toutefois, cela pourrait changer dans les prochaines versions.) les extensions mathématiques et de la preuve dans les théories globales sont disponibles pour tous les projets.

4.2.7.2 Extensions à portée du projet

Aussi connu sous le nom « portée locale ». Cela fait référence à des théories qui font partie des autres projets que le projet global. Les extensions mathématiques et de la preuve dans les théories locales ne sont disponibles que pour les modèles dans leur projet correspondant.

4.2.8 Support de preuve

Le plug-in de la théorie fournit un mécanisme pour l'application des règles et l'utilisation de théorèmes polymorphes. Le Prouveur à base des Règles : PbR [37] (**en anglais est connue sous le nom "Rules-based Prover" ou "RbP" dans l'outil RODIN**) est une contribution à l'infrastructure **de** preuve de RODIN (Voir figure IV.15), **elle** fournit un certain nombre de raisonneurs et tactiques. **L'**élément important **du** prouveur à base des règles est le moteur correspondant à motif. **L'**aspect particulièrement intéressant de ce moteur est la routine de recherche associative et commutative et associative (AC) qui est inspiré par les œuvres en correspondant **aux travaux** [347–349]. (Une procédure complète d'adaptation d'AC est mise en œuvre dans le cadre de la PbR).

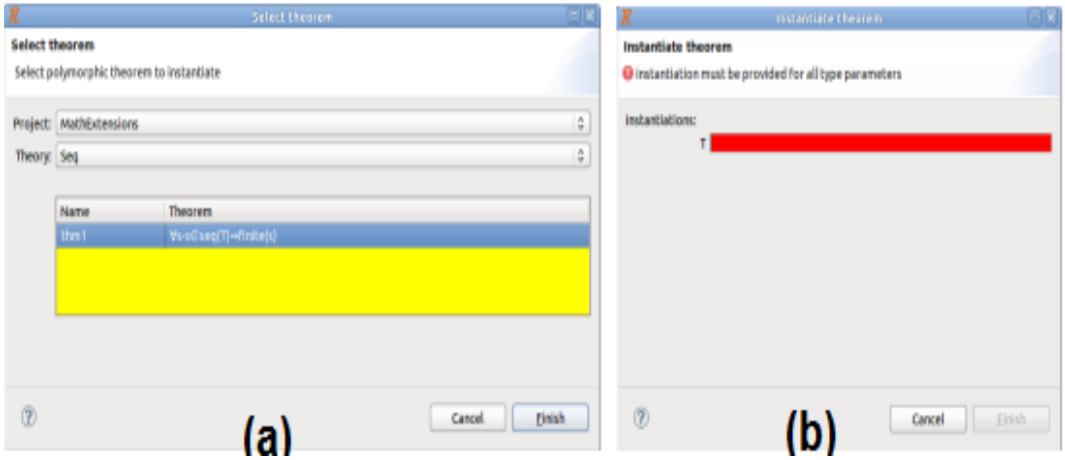


FIGURE IV.15 – L'utilisation des théorèmes polymorphiques : (a) selection d'une théorème.(b) instantiation d'une théorème.

4.2.8.1 La réécriture et les règles d'inférence

La réécriture et les règles d'inférence spécifiées dans les théories sont utilisables de la même manière que réécriture et d'inférence des règles existantes. (Ceci est habituellement réalisé par un hyperlien ou un menu déroulant à côté de l'objectif ou de l'hypothèse sous-jacente).

4.2.8.2 Les théorèmes polymorphes

Pour utiliser un théorème polymorphe, une instanciation de type approprié est requise. Par « appropriée », nous voulons dire que l'instanciation de type ne devrait **pas** se référer qu'aux types reconnus dans le séquent à prouver (ensembles porteurs reconnus ou tout du haut-types $BOOL$ et $\mathbb{Z}$). **La** Figure IV.11 montre l'assistant **qui** permet de sélectionner et d'instancier un théorème polymorphe. Le théorème sélectionné et instancié devient une hypothèse visible dans le séquent actuel.

4.2.8.3 Autres tactiques utiles

Il est dans certains cas, utile d'élargir les définitions (réécriture de définition) de tous les opérateurs utilisés dans un séquent. Une tactique est fournie à cet effet. Il tente de réécrire autant que possible tous les opérateurs de la théorie à l'exception des opérateurs de manière récursive définies et **des expressions liés aux types** de données (par exemple, des constructeurs).

4.3 La validation à base théorie du réseau SONoC

Dans le but de valoriser le bénéfice **d'utiliser notre approche est de montrer comment des théories améliorent** la sémantique réservée pour valider formellement la stratégie de reconfiguration des nœuds pour les réseaux sur puces, **nous avons** pris comme exemple l'architecture des réseaux sans fil à base NoC (Wireless Network on Chips (WNoC) voir Figure IV.16) pour créer un système auto-organisé [44] **ou simplement appellé** SONoC, cependant cette stratégie a besoin plus qu'une approche pendant l'étape de formalisation. Pour cela **nous avons** utilisé une théorie de graphe pour représenter le système distribué WNoC comme un graphe et la stratégie d'adaptation pour la tolérance aux fautes contre les pannes en utilisant les propriétés de la théorie de coloration par la suite et après la validation de cette stratégie fonctionnellement, **nous utilisons** la théorie VHDL pour finaliser les étapes de validation pendant la réalisation de ce genre de systèmes distribués.

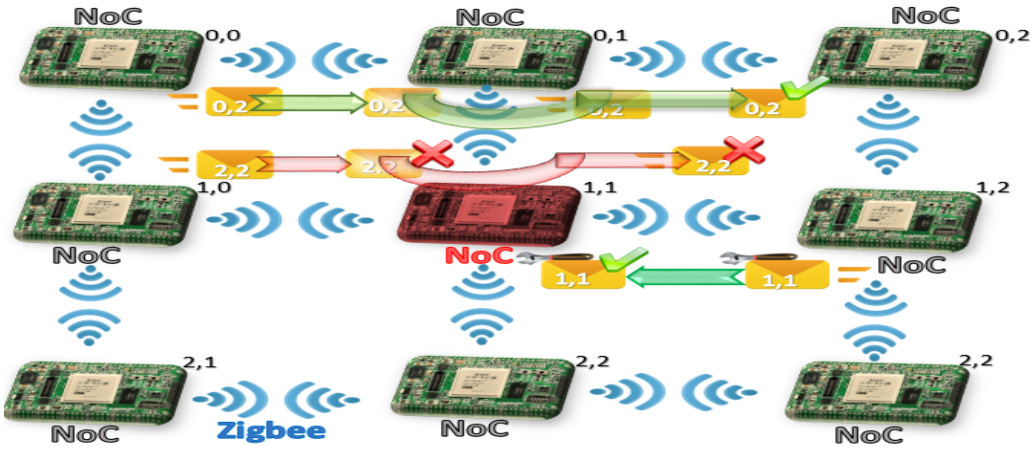


FIGURE IV.16 – Visualisation de système SONoC et la strategie proposés.

4.3.1 La nature des système distribué SONoCs

Un système distribué est considéré comme un ensemble d'événements composés d'événements locaux liés aux agents et à leurs calculs locaux et les événements mondiaux tels que l'utilisation du canal de communication, les messages d'envoi / réception ··· et **ce système** est souvent vu comme un graphe [350], où les sommets sont les nœuds et les arêtes **sont** les liens de communication directs entre eux.

Le travail adapté pour vérifier un système d'auto-reconfigurable **et** multi-nœud, ce réseau est composé d'un ensemble de nœuds sans fil auto-organisés qui communiquent via le protocole réseau Zigbee [351] (présenté sur la Figure IV.16). Chaque nœud est indépendant. Ceci permet l'entretien et la fiabilité de fonctionnement du système en cas de panne.

4.3.2 Les théories développées

Ce travail se propose de valider le réseau de capteurs sans fil (IV.16). Étant donné que ce réseau est composé d'un ensemble de commutateurs(qu'on aime les appeler des noeuds ou aussi des *switches*) à base de NoC, nous introduisons la théorie NoC liées à *la théorie des graphes* et nous proposons une nouvelle stratégie pour la récupération de l'ensemble des noeuds défectueux sous forme de la *théorie WNoC* et pour traiter et de gérer la complexité de la multi-échec des capteurs utilisant *la théorie des graphes colorés*, la dernière étape consiste à veiller à ce que toutes les propriétés de ce système embarqué sera appliqué correctement dans l'environnement d'application en combinant ces théories précédentes avec *la théorie VHDL*.
Dans tous les niveaux au cours de ces multiples propriétés**, le raffinement de l'ensemble des événements doit** suivre un mécanisme ; **et plus tard il pourrait** être présenté par les opérateurs pour faire le passage d'un niveau à **un autre très systématique, en fin ce passage sera expliqué** dans les sections suivantes.

4.3.2.1 La théorie des NoCs

Au cours de la modélisation pour ce réseau particulier du NoC, nous devons présenter la nouvelle stratégie de reconfiguration pour tout nœud défectueux, nous ajoutons une nouvelle extension de *la théorie de NoC*, mais avant tous il faut présenter *la théorie NoC*(expliqué **par** la figure IV.17) qui se caracterise par :

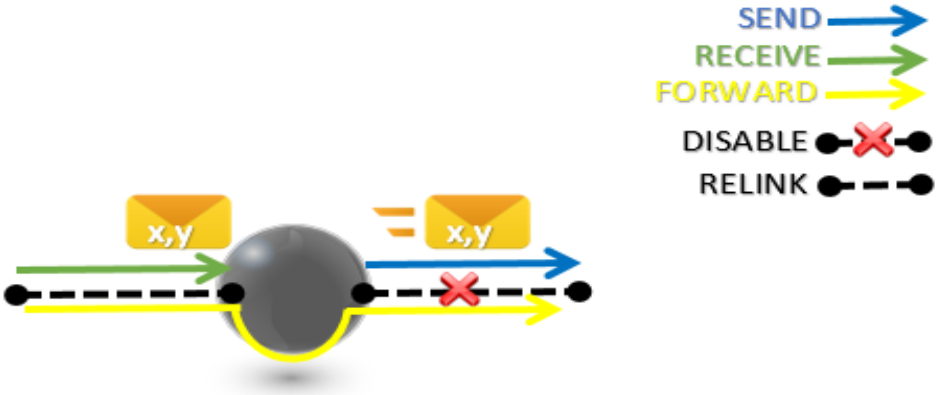


FIGURE IV.17 – L' illustration de la théorie de NoC.

- Le réseau NoC est un ensemble de noeuds qui pourraient avoir tant de rôles comme étant une src source de transition de paquets ou **la** destination rcv ou un dst intermédiaire lors de la transition de phase, même les nœuds pourrait également avoir aucun rôle *nop*.
- Le NoC est présenté comme un réseau (un graphe) entre les sources et les destinations de paquets. Ce graphe est non vide, non-transitive et symétrique. Ce graph aussi présente comme une matrice sa taille est représenté par $netsize$.
- Chaque nœud dans le réseau sans fil à base de NoC pourrait **déclencher** l'un des événements comme :

— L'émission de donnée (data sending) vue lorsqu'une source envoie un paquet m, le paquet est placé dans le réseau d'un port d'entrée IP du noeud s.

— La reception de donnée (data receiving) où un paquet est reçu par son destinataire, si le paquet a atteint sa destination.

— Dans le réseau, un paquet m passe par un noeud x vers un autre noeud y, jusqu'à ce qu'il atteigne sa destination d. Ce passage est de la logique de sortie op de passer à un canal $Switchcontrol$.

4.3.2.2 La théorie des graphes

Pour notre cas on a utilisé une théorie (*closure*) pour représenter le réseau à base NoC (voir Figure IV.18 et Figure IV.19) et représenter par la suite toute les propriétés de graphe délivrées par une standard crée par JR. ABRIAL [352, 353] et le groupe de **développement de l'outil** RODIN.

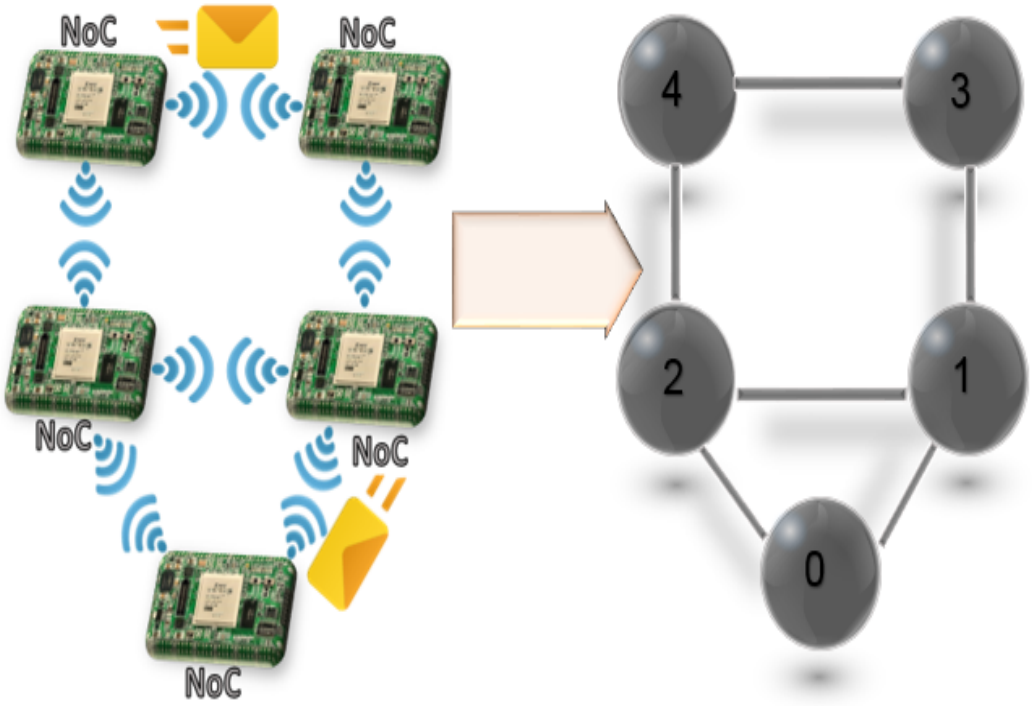


FIGURE IV.18 – La vision du système SONoC par théorie de la fermeture.

- La théorie de *closure* est une extension de la théorie *FixPoint* cette dernière nous permet de faire une projection en utilisant l'opérateur fix pour un ensemble de partie S pour avoir une couple des éléments (points), la closure est la $i^{\text{ème}}$ composition de ses couple de points.
- La théorie closure détermine les propriétés de graphe et parmi ces propriétés on peut expliquer :

$$Thm1 : \forall r, x, y \cdot r \in \mathbb{P}(S \times S) \wedge x \mapsto y \in cls(r) \Rightarrow x \in dom(r)$$

Ce théorème présente la closure comme un couple des points.

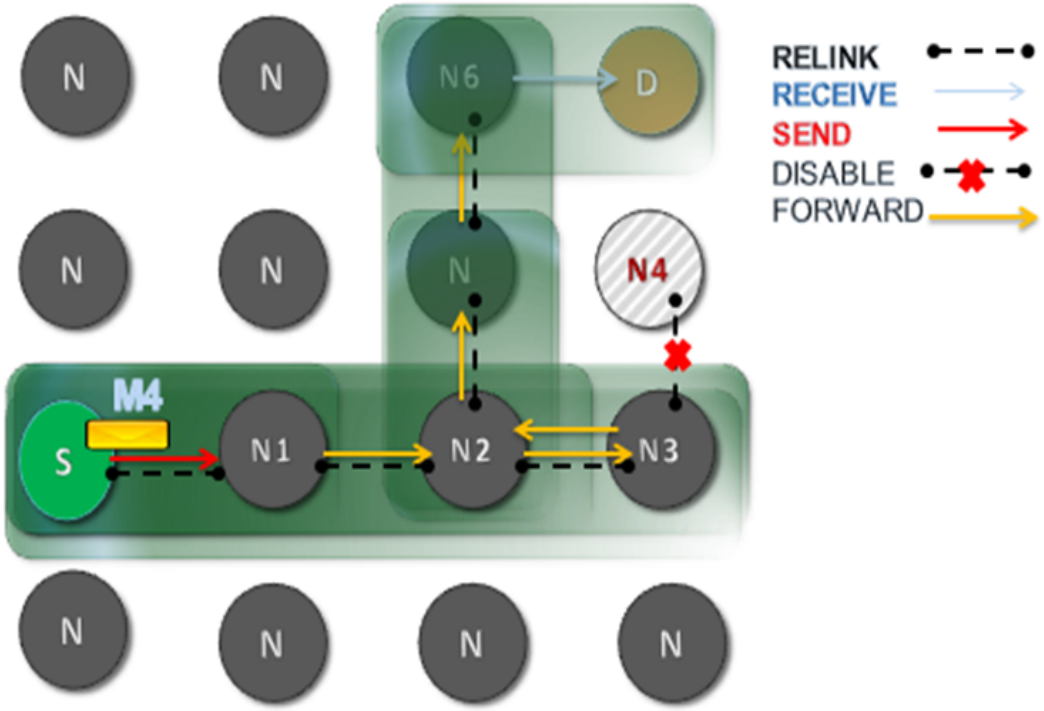
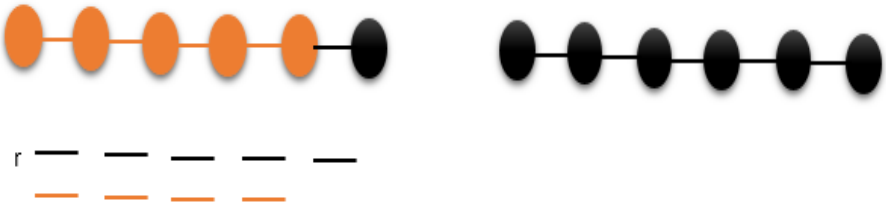


FIGURE IV.19 – L'illustration de la théorie graphe utilisé dans le système SONoC.

$Thm2 : \forall r, s \cdot r \subseteq S \times S \wedge s \subseteq r \Rightarrow cls(s) \subseteq cls(r)$



Ce théorème montre si deux closures qui contient des liens en communs sont inclut l'une dans l'autre.

$Thm3 : \forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r) = r \cup (cls(r); r)$
$Thm4 : \forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow r \subseteq cls(r)$

Ces deux théorèmes présentent que la closure est définit comme l'union par composition d'un couple des points.

$Thm5 : \forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r); cls(r) \subseteq cls(r)$

Ce théorème montre la closure est transitive pour la composer avec plus des points.

$$Thm6 : \forall(p,q,v \cdot p \in S \leftrightarrow S \wedge q \in T \leftrightarrow T \wedge v \in S \leftrightarrow T \wedge v;q \subseteq p;v) \Rightarrow v;cls(q) \subseteq cls(p);v$$

Ce théorème montre la closure est transitive pour la composer avec plus des closures.

$$Thm7 : \forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r\sim) = cls(r)\sim$$

Ce théorème montre la closure est inversive.
La théorie qui a des règles d'inférences **est invoquée** pour forcer la preuve pendant leur utilisation dans les modéles Event-B **nous prenons** comme exemple :

METAVARIABLES
$r \in \mathbb{P}(S \times S)$
$x \in \mathbb{P}(S)$
REWRITE RULES

- $rew1 : cls(r)$
- $\cdot\ rhs1 : \top \rhd r \cup (r;cls(r))$
- $rew2 : cls(r)$
- $\cdot\ rhs1 : \top \rhd r \cup (cls(r);r)$
- $rew3 : cls(\emptyset : S \leftrightarrow S)$
- $\cdot\ rhs1 : \top \rhd \emptyset : S \leftrightarrow S$

INFERENCE RULES

- $inf1 : \cdot r[x] \subseteq x$

$cls(r)[x] \subseteq x$: Cette règle d'inférence nous permet de forcer la preuve pendant la substitution de la $cls(r)$ ou r peut être closure $cls(r)$, un point x, ou un l'ensemble vide$\emptyset$.

4.3.2.3 La théorie WNoC

Au cours de la modélisation de ce réseau particulier (Figure IV.20) basé sur les NoCs, on doit présenter le mode de reconfiguration pour tout nœud défectueux et pour cette nouvelle contrainte, on ajoute une nouvelle extension de *la théorie NoC* **appelée** *théorie Wireless-NoC*, **elle** contient les nouvelles **types de données** «état» **qui** est utilisé pour représenter l'état de chaque nœud. Cette utilisation de *la théorie Wireless-NoC* qui fait **l'appelle** à *la théorie NoC* a toujours les mêmes propriétés citées pour *la théorie NoC* en revanche que :

- Lorsqu'une source $Srcnod$ envoie un paquet msg, avec l'avantage que le paquet doit avoir une valeur de 0 pour l'argument flg et les états des noeuds $Srcnod$, $Desnod$ **doivent** être changé avec la valeur $ocpy$ (cas 1),
- Si un paquet est reçu par sa destination, le nœud destinataire change sa valeur de champ de rôle à rcv **(voir l'annexe "D" le cas "b" dans la partie des prédicats)** .Ce paquet doit avoir une valeur de 0 pour l'argument flg et les états des noeuds $Srcnod$, $Desnod$ **doivent** être changé avec la valeur $free$,

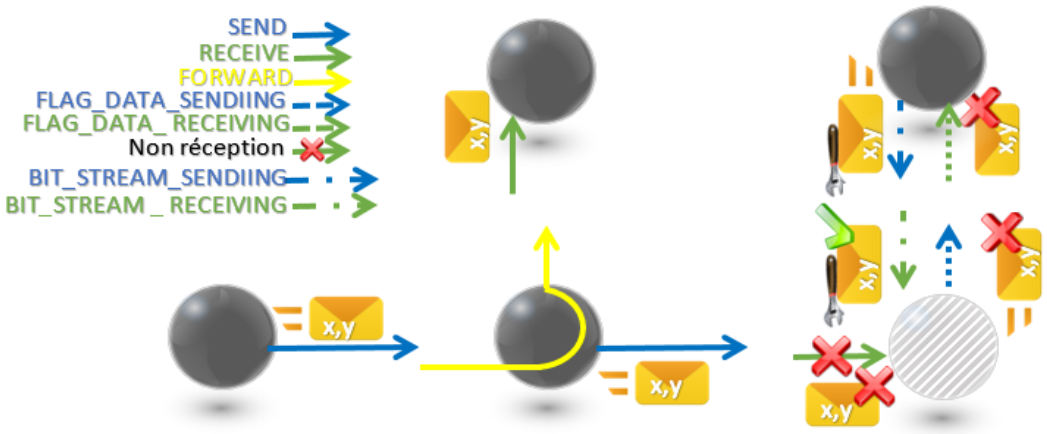


FIGURE IV.20 – La théorie WNoC utilisé pendant la modélisation de système SONoC.

- La dernière chose sur le réseau, un paquet msg passe à partir d'un nœud $Srcnod$ vers un autre nœud $Desnod$ (noeud avec un rôle égal à dst **voir l'annexe "D" le cas "b" dans la partie des prédicats**) ce paquet doit avoir une valeur de 0 pour l'argument flg et l'état de l '$Srcnod$ de noeud doit être changé avec la valeur $free$ lorsque le nœud Desnod aura la valeur $ocpy$
- Dans le cas optimal ces derniers événements pourraient représenter la communication entre les différents nœuds sinon certains nœuds pourraient être en état d'échec de sorte qu'ils doivent informer les autres nœuds en envoyant un $flagData$ avec $flg = 1$, alors que dans le cas d'un nœud $Srcnod$ peuvent vérifier les règles de la capacité de reconfiguration (par la suite), il envoie des données de configuration $Bitstream$ au noeud défaillant $Fnod$, après que, quand un paquet $FlagData$ est reçu par le noeud $Srcnod$, si elle **est envoyé** à partir d'un noeud défaillant $Fnod$.
- Le choix du nœud test est spécifié d'une façon simple afin de briser la complexité du rôle de ce nœud, il suffit alors de respecter les règles du choix du nœud test et le reste est considéré comme procédure à appliquer après la sélection du nœud test, ces règles sont :
  - ***La règle 1 :*** **un nœud ne peut** pas être un nœud défectueux. Ce qui est **définit par** $grd9 : Testnode \neq faultynode$
  - ***La règle 2 :*** **un nœud dispose** des ressources matérielles nécessaires pour mettre en œuvre l'IP testeur.
  - ***La règle 3 :*** **un nœud dispose du "bitsream"** de configuration de l'IP testeur. **Les gardes** $grd10 : src(bitstreampacket) = Testnode$ **et** $grd11 : dst(bitstreampacket) = Testnode$ **expriment** cette règle parce que si le nœud ne dispose pas du fichier de reconfiguration il peut le recevoir à partir d'un notre nœud.
  - ***La règle 4 :*** **un nœud ne peut pas être** occupé par une tâche prioritaire. **Les gardes** $grd7 : Testnode \mapsto p \notin sent$ **et** $grd8 : Testnode \mapsto p \notin rcvd$ **valent dire** que ce nœud n'est pas occupé par l'envoie ou la réception des paquets.

4.3.2.4 La théorie de coloration appliqué sur les WNoCs

Dans le cadre de notre projet, nous sommes basées sur les règles suivantes(voir figure IV.21) :

- Les nœuds du graphe à l'état initial ne sont pas colorés.
- Le nœud défaillant **est ce qui ne peut pas** ni envoyer ni recevoir des paquets ou l'un des deux, il doit être coloré par une des 4 couleurs ($rouge$, $jaune$, $bleu$, $vert$). Si on a **plusieurs** nœuds défaillants on les mit les en attente pour être colorés.
- Le nœud source du paquet ne se colore pas et le nœud destinataire aussi.

- Le nœud de reconfiguration un nœud choisi pour corriger le nœud défaillant, et il ne doit pas être coloré.

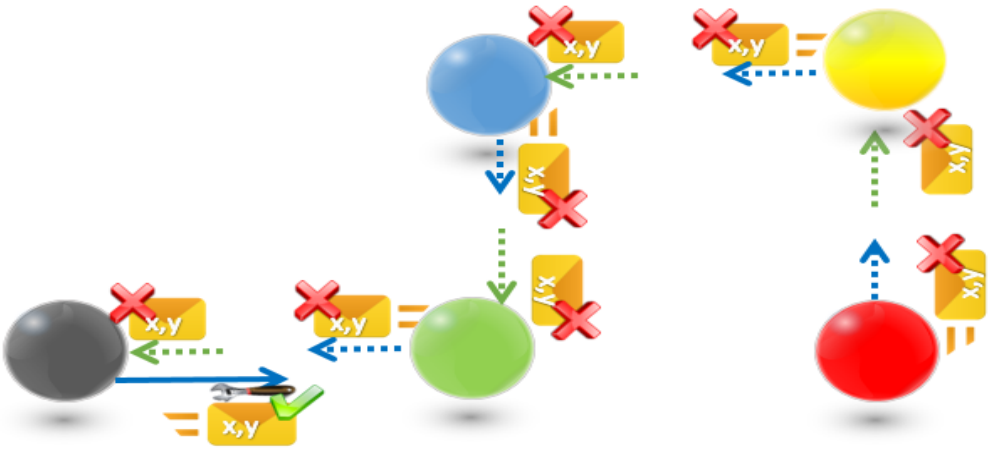


FIGURE IV.21 – Le système SONoC vue avec la théorie de graphe colorès.

4.3.2.5 La théorie VHDL

Elle inspiré du code VHDL existant qui décrit notre architecture

, l'utilité de cette théorie (Figure IV.22) est de représenter en terme des invariants les fonctionnalités validés pour notre architecture et cela nous aide à renforcer la validation de système dans son environnement d'exécution et sera même **une phase de** pré-traitement de la génération du code VHDL.

$THEORY\ VHDL\_th$
$DATATYPES$
$pio \hat{=} in, out, inout$
$OPERATORS$
$\bullet arch\_decl :$
$arch\_decl(s : \mathbb{P}(T))$
direct definition
$arch\_decl(s : \mathbb{P}(T)) \hat{=} s$
$\bullet port\_decl : port\_decl(p : pio, s : \mathbb{P}(T))$
direct definition
$port\_decl(p : pio, s : \mathbb{P}(T)) \hat{=} s$
$\bullet std\_logic : std\_logic$
direct definition
$std\_logicl \hat{=} 0..1$

$\bullet vector :$
$vector(s : \mathbb{P}(T))$
direct definition
$vector(s : \mathbb{P}(T)) \hat{=} \{n, f \cdot n \in \mathbb{N}$
$\wedge f \in 0..(n-1) \rightarrow s | f\}$
$\bullet std\_logic\_vector :$
$vector(length : \mathbb{N}, s : \mathbb{P}(T))$
well-definedness condition
$length \in \mathbb{N}$
$finite(s)$
direct definition
$std\_logic\_vector(length : \mathbb{N}, s : \mathbb{P}(T)) \hat{=}$
$\{v | v \in vector(s) \wedge card(s) = length\}$

La théorie VHDL est utilisée pendant la modélisation de l'auto-organisation(voir Figure IV.22) , cette dernière est composée d'une entité contient les ports '*in*' ,'*out* 'ou '*in_out*'et une architecture qui contient un ensemble de variables ,d'où on a créé deux opérateurs '*arch_decl* et *ports_decl*' cependant on a aussi créé un data type paramètre '*pio*' (voir Figure IV.6 et paragraphe 4.2.3.1),revenant au code VHDL dont on a utilisé deux types de signals '*std_logic* et '*std_logic_vector*'. La variable VHDL doit avoir un type et une définition par exemple : le type entier en VHDL est représenté comme un **opérateur** et assuré qu'il ne doit dépasser $int\_max = 2^{32}$ et chaque opération qui utilise ce type de variables ne doit pas dépasser cette valeur.

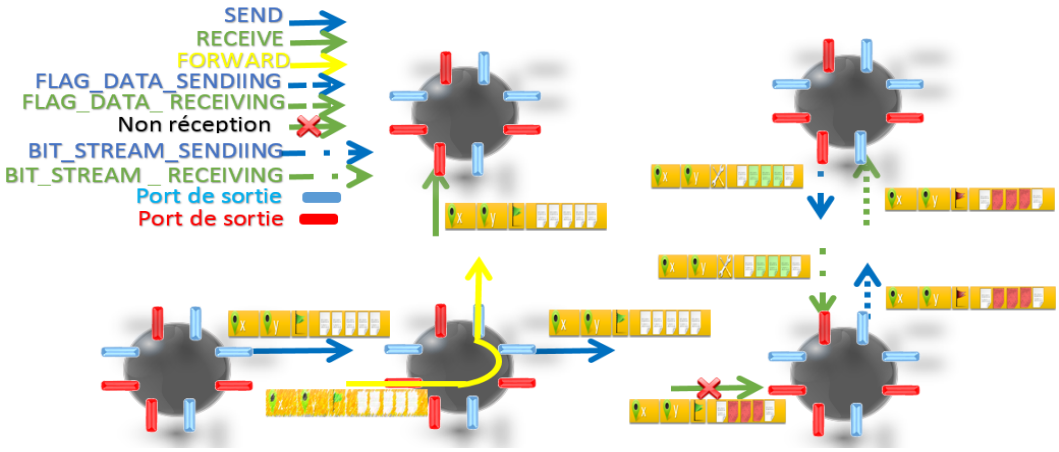


FIGURE IV.22 – Le système SONoC vue par la théorie VHDL.

- Dans le cas ordinaire de données de la $datatoroute$ qui doit être envoyé à partir du file d'entrée $buffer\_in$ au bus de port de sortie $outportsbus$ ces données doivent être transmises à partir d'un nœud à un autre jusqu'à ce qu'il atteigne sa destination et stockées ces données reçues à partir des ports d'entrée de bus $inportbus$ dans le file d'entre de la destination $buffer\_in$.
- **Les** nœuds pourraient avoir une défaillance au cours de ce processus d'échange de données **nous attribuons** un état d'échec dans ce réseau sans fil de système NoC et **nous avons géré** cette affaire en faisant une sorte de protocole de reconfiguration de départ en ajoutant un bit d'état dans la structure de chaque données d'échange (voir figure IV.23).

FIGURE IV.23 – Le format de donnée proposé dans le système SONoC.

- Si ce bit est égal à 1, les données considérées comme un $Fail\_Data$ après que le nœud reçoit ces données et considère ses voisins comme des nœuds défectueux, aussi ce nœud défectueux devrait changer la valeur de la variable occ_rqst_vect par '1111' pour ne reçoivent aucune autre donnée sauf les données $bitstream$**, ce paquet est** envoyé par un nœud de reconfiguration juste après cette étape pour faire le nœud défectueux re-modifiera la valeur de occ_rqst_vect à '0000'.

4.3.3 La modélisation à base théorie

Commençant **de citer que la spécification passe toujours par le modèle** le plus abstrait raffinant jusqu'à le plus concret possible. Dans le niveau abstrait le nœud peut envoyer ,recevoir et acheminer des paquets, ensuite **un raffinement est fait où le nœud peut être défaillant alors on peut le colorer par les 4 couleurs disponibles puis on passe vers le prochain raffinement qui spécifie l'envoi du** paquet de reconfiguration par l'évènement $reconfig\_node$ et le fait l'action de recevoir cet paquet par $Receiving\_bitstream$,le troisiéme raffinement contient l'évènement ou le nœud été reconfiguré et devient capable d'envoyer ,de recevoir et d'acheminer des paquets donc on revient à l'état **initial du système (pas de couleur attribué) ou tous les nœuds sont corrects (non colorés)** , il nous reste de colorer les nœuds défaillants qui sont en atttente cela est exprimé par l'évènement $Colore\_w\_faulty\_nodes$ raffiné de l'évènement $Colore\_faulty\_node$ ajoutant de nouveaux conditions(qu'il soit en attente et n'est pas encore coloré).

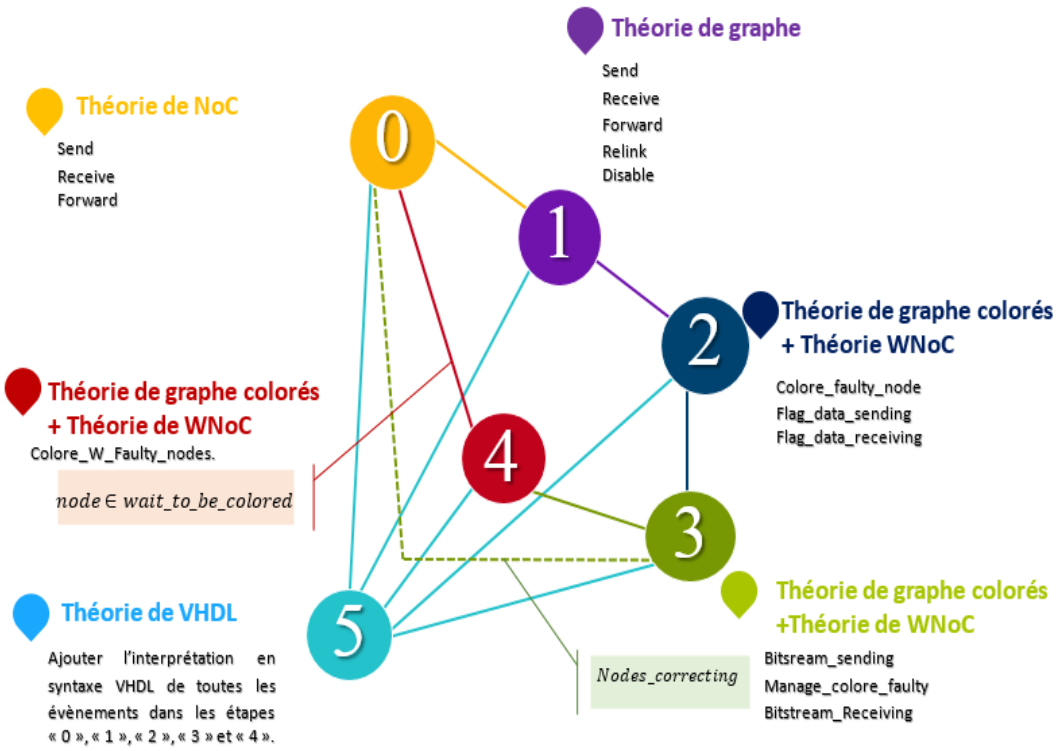


FIGURE IV.24 – Illustration des évènements pendant la modélisation de système à base NoC SONoC.

4.3.3.1 Model abstrait

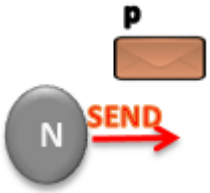
Dans le 1^{er} Contexte on définit le rôle du réseau **présenté par l'ensemble des nœuds qui envoient et reçoivent** des paquets. Donc deux ensembles seront définis dans ce niveau ; les nœuds existant $NODES$ et **les** paquets $PACKETS$ **ces paquets sont envoyés par des nœuds source** $src \in PACKETS \rightarrow NODES$ **et sont reçus par leurs destinations** $dst \in PACKETS \rightarrow NODES$. Les sources sont différents des destinations. Ensuite nous définissons que le graphe est donné **comme un produit cartésien de l'ensemble** $NODES$.

$axm1 : NODES \neq \emptyset$ // l'ensemble des noeuds ne soit pas vide
$axm2 : PACKETS \neq \emptyset$ // l'ensemble des paquets ne soit pas vide
$axm3 : src \in PACKETS \rightarrow NODES$ // chaque paquet est associe à une seule source
$axm4 : dst \in PACKETS \rightarrow NODES$ // chaque paquet est associe à une seule destination
$axm5 : \forall p \cdot p \in PACKETS \Rightarrow src(p) \neq dst(p)$ // la dst d'un p est différente de sa src
$axm6 : g \subseteq NODES \times NODES$
$axm7 : g \neq \emptyset$

$1^{\div re}$ Machine : On définit les variables $sent$, $rcvd$, $switchcontrol$, str et gr qui nous permet d'effectuer les actions $SEND$, $RECEIVE$ et $FORWARD$. Les invariants suivant sont décrits dans la machine du niveau abstrait comme suit :

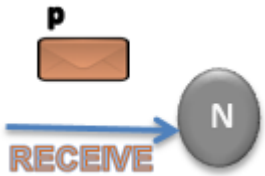
$INITIALISATION \widehat{=}$
$act : str := \emptyset, act : switchcontrol := \emptyset, act : rcvd := \emptyset, act : sent := \emptyset, act : gr := g$

- Les valeurs initiales des variables sont vides et on a un graph initial.
- L'événement $SEND$ fait les deux actions suivantes :



$act1 : sent := sent \cup \{s \mapsto p\}$
$act2 : str := str \cup \{s \mapsto p\}$// le p est mis dans le réseau

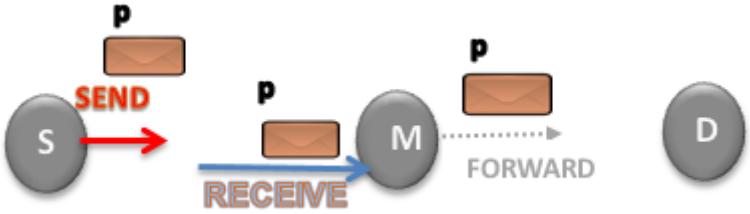
- RECIEVE fait les actions suivantes :



$act1 : rcvd := rcvd \cup \{d \mapsto p\}$
$act2 : str := str \backslash \{d \mapsto p\}$// le p est enlevé du réseau

- L'évènement FORWARD fait l'action suivante :

$act1 : switchcontrol := (switchcontrol \backslash \{n \mapsto p\}) \cup \{m \mapsto p\}$// le p n'est plus dans n car il est transmis en m



4.3.3.2 Premier raffinement

Nous définissons un ensemble de couleurs (red, $green$, $blue$, $yellow$), **ces couleurs sont** choisies par les nœuds lors de l'exécution de l'algorithme de coloration.

$CONSTANTS\ Red, green, Blue, yellow$
$inv6 : \forall n1, n2, c1, c2 \cdot c1 \in COLORS \wedge c2 \in COLORS \wedge n1 = faulty\_node \wedge n2 = faulty\_node \wedge c1 \in colored[g[\{n1\}]] \wedge c2 \in colored[g[\{n2\}]] \wedge n1 \mapsto n2 \in cls(gr) \Rightarrow c1 \neq c2$
$inv8 : switchcontrol \in NODES \leftrightarrow PACKETS$// contenu de noeud
$inv9 : str \in NODES \leftrightarrow PACKETS$// store l'acheminement d'un paquet dans le graphe
$inv10 : gr \subseteq NODES \times NODES$// le graphe courant

- Dans la machine de ce niveau, on introduit la coloration des nœuds défaillants par quatre couleurs **au maximum**, pour cela on a ajouté quelque variables nécessaires ($faulty\_node$, $colored$, $has\_colored$, $count\_colored$) qui nous permettent d'effectuer l'évènement COLORE

_FAULTY_NODES, on cite le plus important des invariant **est celui qui assure que la règle de base de coloration est que :** *deux nœuds adjacents ne prennent pas la même couleur.*

- L'évènement $COLORE\_FAULTY\_NODES$ effectue plusieurs actions, sachant que notre stratégie nous permet de colorer jusqu'à quatre nœuds à la fois, puis on incrémente le nombre de ces nœuds grâce à la variable $count\_colored$, les actions seront décrites par suite :

— Coloration des nœuds défaillants :

$act1 : colored(n) := clr$ // il prend l'une des quatre couleurs disponibles

— L'ajout des nœuds colorés dans l'ensemble des nœuds colorés 'has_colored' :

$act2 : has\_colored := has\_colored \cup \{n\}$ // l'ajout de nœud coloré dans l'ensemble des nœuds colorés

— Et compter le nombre des nœuds colorés

$act3 : count\_colored := count\_colored + 1$ // incrémente le nbr des nœuds défaillants

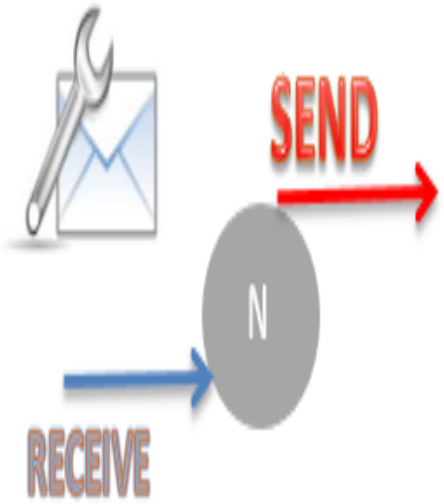
4.3.3.3 Deuxième raffinement

Afin d'avoir une stratégie complète qui détectent tous les nœuds défaillants et les colorent et respecte au même temps les règles de coloration de graphe, nous avons ajouté les évènements MANAGE _ COLORE _ FAULTY _ NODES, $RECONFIG\_NODE$ et RECIEVING_BITSTREAM.

- **L**'évènement '$MANAGE\_COLORE\_FAULTY\_NODES$' : pour effectuer cet évènement on a ajouté la variable '$wait\_to\_be\_colored$'.

$MANAGE\_COLORE\_FAULTY\_NODES \widehat{=}$
$\vdots$
$WHERE$
$grd1 : count\_colored \geq 4$
$grd2 : n = faulty\_node$
$grd3 : n \notin has\_colored$
$grd4 : n2 = faulty\_node \wedge n2 \in has\_colored$
$grd5 : n1 \mapsto n2 \in cls(g)$
$THEN$
$act1$: wait_to_be_colored := wait_to_be_-coloredU$\{n\}$
$act2$:
$count\_colored := count\_colored + 1$

- **L'**évènement '$RECONFIG\_NODE$' est choisie suivant 4 règles explique dans *la théorie des WNoCs* :



$RECONFIG\_NODE \triangleq$
⋮
$WHERE$
$grd1 : reconfig\_node \neq faulty\_node$
$grd2$: src(BitstreamPacket)=reconfig_node
$grd3$: dst(BitstreamPacket)=reconfig_node
$grd4 : n \in has\_colored$
$grd5 : reconfig\_node \mapsto p \notin sent$
$grd6 : reconfig\_node \mapsto p \notin rcvd$

$THEN$
$act1 : sent := sent \cup \{n \mapsto BitstreamPacket\}$

- Un évènement '$RECEIVING\_BITSTREAM$' pour assurer la reconfiguration des nœuds défaillants, par l'action suivante :

$act1 : rcvd := rcvd \cup \{n \mapsto BitstreamPacket\}$

4.3.3.4 Troisième raffinement

Les nœuds qui ont été reconfigurés, il faut les relier à nouveau avec les autres nœuds du graphe par l'évènement '$CORRECTED\_NODES$', on ajout la variable 'was_faulty', définit par :

$inv1 : was\_faulty \in NODES$

— Et les actions suivantes seront effectuées :

$act1 : has\_colored := has\_colored \setminus \{n\}$
$act2 : count\_colored := count\_colored - 1$

Aussi dans ce niveau, on a essayé de détaillé le font de l'acheminement de paquet dans notre réseau, par la création de deux évènements :

- '$COLORE\_WAITED\_TO\_BE\_COLORED$' : colorer les noeuds qui appartiennent à l'ensemble '$wait\_to\_be\_colored$'après décrémentation du compteur.

$WHERE$
$grd1 : n \in wait\_to\_be\_colored$
$THEN$
$act1 : colored(n) := clr$
$act2 : count\_colored := count\_colored + 1$

4.3.3.5 Quatrième raffinement

Après avoir prouvé les modèles **représentants les trois raffinements précédents**, on introduit les variables VHDL dans les évènements des systèmes et les typés comme suit :

- Dès que la théorie sera déployée et importée, on peut utiliser les variables VHDL dans notre modèle selon notre besoin.

INVARIANTS
$inv1 : OccNode \in port\_decl(out, std\_logic)$// soit 0 ou 1
$inv2 : data\_bus\_out \in$ port_decl(out,std_logic_vector(max_buff,std_logic)) // on définit le data_bus_in à partir l'opérateurcde la theorie
$inv3 : max\_buff \in vhdl\_int$// la taille du buffer appartient à vhdl_int
$inv4 : bitstream \in arch\_decl(vector(std\_logic))$// the bitstream est un vecteur d'architecture
$inv5 : data\_bus\_in \in port\_decl(in, std\_logic\_vector(max\_buff, std\_logic))$// data_bus_in est un vecteurr de type std_logic_vector deja defini
$inv6 : Datatoroute \in archdecl(std\_logic\_vector(max\_buff))$//contenu à routé vecteur d'architecture de type std_logic_vector pour la donnée à route
$inv7 : buffering \in archdecl(std\_logic\_vector(max\_buff))$// contenu de buffer(input buffer) est de type std_logic_vecotor a une taille max_buff

- Par exemple, dans l'évènement '$COLORE\_FAULTY\_NODES$', on exprime la condition d'occupation par la variable $\}Occ\_node' = 1$, sachant que n'importe quel nœud **peut devenir défaillant, alors il peut envoyer** un "bitsream" à ces voisins directs pour indique sa défaillance. Ce bistream soit mis dans un '$data\_bus\_out$ '.

$COLORE\_FAULTY\_NODES \widehat{=}$
$WHERE$
$grd1 : OccNode \in port_dec(out, std\_logic)$
$grd2 : data\_bus\_in \in port\_decl(in, std\_logic\_vector)$
$grd3 : birstream \in arch\_decl(std\_logic\_port)$
$grd4 : OccNode = 1$
$THEN$
$act1 : bitstream := data\_bus\_in$

— D'où l'évènement '$RECEIVING\_BITSTREAM$' peut être raffiné et la deuxième action sera :

$act2 : Buffer\_in := BitstreamPacket$

4.3.4 La modélisation à base des opérateurs

Le système que nous devons introduire est un réseau à Base NoC où chaque nœud est capable de se rétablir après une panne de logiciel de façon automatique, une expertise massive de formalisation pour valider un tel type de système, en particulier dans la spécification Event-B qui doivent couvrir tant de niveau de concrétisation par le processus de raffinement. Dans l'approche classique de raffinement intuitive le point de départ par les propriétés générales de l'échange de données dans le

nœud à base de NoC alors que le réseau dans ce type de système est construit par un ensemble de nœuds interconnectés, donc le raffinement doit se soucier des propriétés des graphes sans oublier que ce système est un réseau sans fil qui signifie que le raffinement prend également en considération la stratégie de récupération de nœuds qui sont dans l'état d'échec, ce dernier point a une énorme complexité pour gérer le nombre de nœuds défectueux et de maîtriser la manière de signaler l'état de l'échec pour l'ensembles des nœuds, c'est pourquoi nous avons besoin d'un nouveau raffinement suivre les propriétés de graphe colorés pour marquer par quatre couleurs différentes tous les nœuds défectueux et les décolorer après la récupération de leur état correct. Le prochain niveau de raffinement est le final qui est en train de la interpréter toute les propriétés précédentes avec le code VHDL pour assurer la bonne définition du système dans son propre environnement d'exécution.
Notre approche des opérateurs de raffinement(voir figure IV.25) pourrait être illustré dans cet exemple concernant le développement des événements sachant que notre approche toujours dépends de trois éléments (événements, invariants, contextes), mais nous essayons de simplifier l'explication de la manière suivante :

(a) Dans le premier niveau de la modélisation nous introduisons les propriétés NoC en créant de nouveaux événements et ce genre d'introduction pourrait être réalisé par l'opérateur CREATE par exemple l'envoi de données « $send\_data$ » et la reception des données « $receive\_data$ », l'activation des liens entre les nœuds « RELINK » ou couper ces liens « DISABLE ».

(b) Le nouveau raffinement dans le niveau suivant est celui où des propriétés du NoC basé sur l'échange de données à l'aide de l'opérateur RENAME pour certains événements comme « $send\_data$ » et « $receive\_data$ », et certains événements sont limités par des conditions pour les vérifier comme « RELINK » qui a besoin de convenir avec les propriétés du réseau sans fil, à cause de cette condition le raffinement est fait par l'opérateur RESTRICT, nouvellement il y a aussi quelques événements à introduire en utilisant l'opérateur CREATE pour la stratégie de reconfiguration en envoyant des données lors de la phase de signalisation d'échec « $sending\_flagdata$ » et « $receiving\_flagdata$ ».

(c) Dans ce niveau des propriétés de graphe sont introduites, ce fait faire quelques événements se soucient de l'ajout de nouvelles actions à l'aide de l'opérateur ENRICH aux événements liées ou caractéristiques de NoC « $send\_data$ » et « $receive\_data$ » et de la stratégie sans fil de récupération de nœuds « $sending\_flagdata$ » et « $receiving\_flagdata$ » , ajouter d'autres critères de propriétés des graphes pour l'événement « RELINK »en utilisant l'opérateur RESTRICT ou l'ajout d'un nouvel événement par l'opérateur CREATE pour exprimer le cas d'une donnée qui est dans l'état de passage d'un nœud intermédiaire à sa destination « Forward ».

(d) Pendant ce nouveau niveau de raffinement des propriétés des graphes colorés sont introduits, de sorte que les événements de l'échange de données au sein d'un réseau à base de NoC comme « $send\_data$ » et « $receive\_data$ » n'avaient pas besoin d'être changé, ce qui signifie qu'ils sont raffinés par l'opérateur RENAME. Dans la raison que le principe de coloration est **attribuée** pour les nœuds en état d'échec donc nous ajoutons une (ou plus d') action(s) par l'opérateur ENRICH pour les événements « $sending\_flagdata$ » et « $receiving\_flagdata$ », alors que certains événements après l'application de l'opérateur RESTRICT ont besoin de vérifier plusieurs conditions sur l'état de la couleur des nœuds comme les événements « Forward ».

(e) Dans le dernier niveau le raffinement applique l'opérateur ENRICH pour tous les événements cités précédemment en ajoutant des actions qui interprètent le comportement de VHDL pour le système.

4.3.5 La combinaison durant le processus de raffinement

Nous pouvons structurer le processus de raffinement avec les opérateurs pour spécifier les modèles qui utilisent les théories précédentes comme suit (voir figure IV.26) :

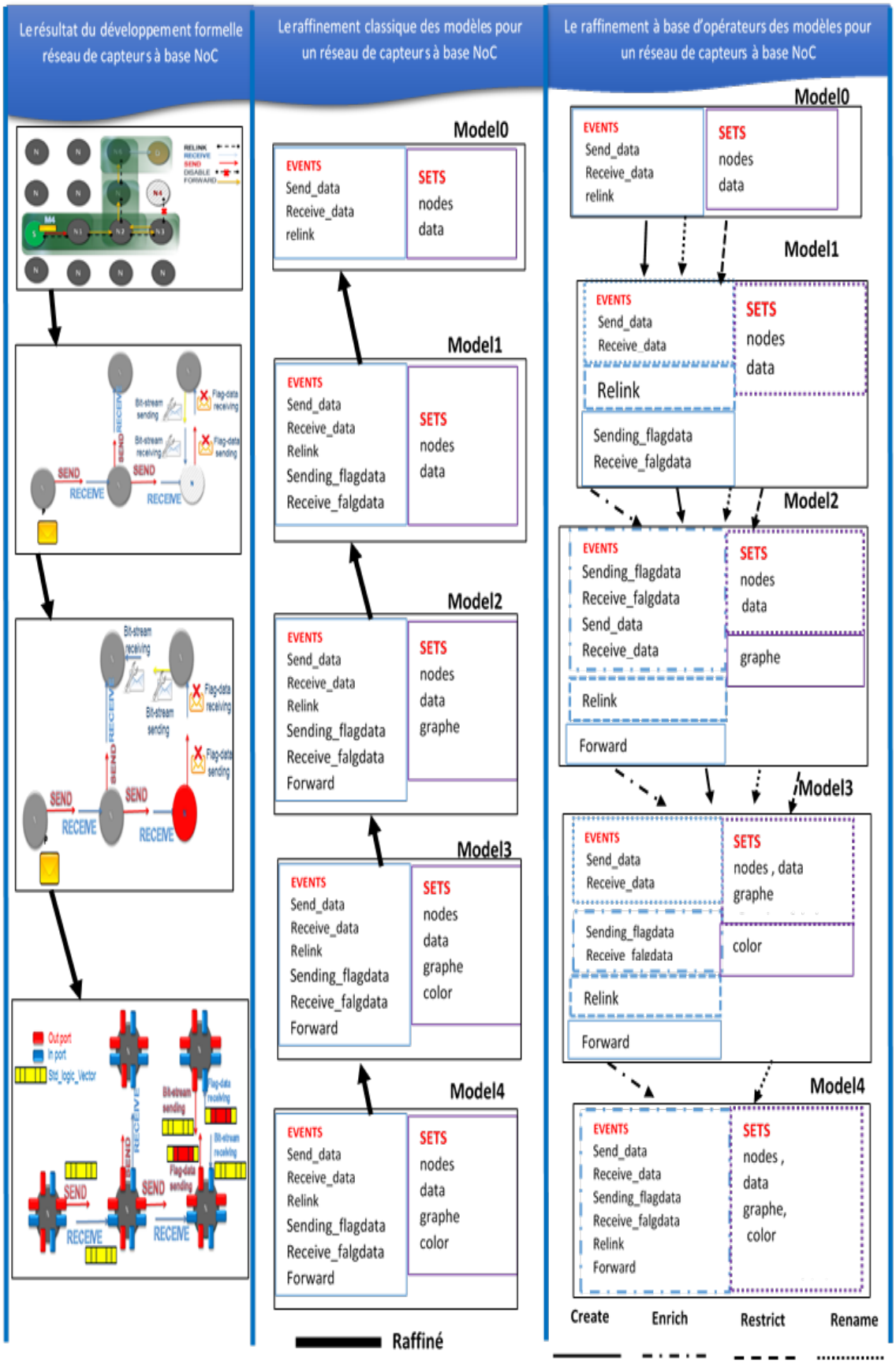


FIGURE IV.25 – Le raffinement classique vs. Le notre proposé pendant la modélisation de système à base NoC.

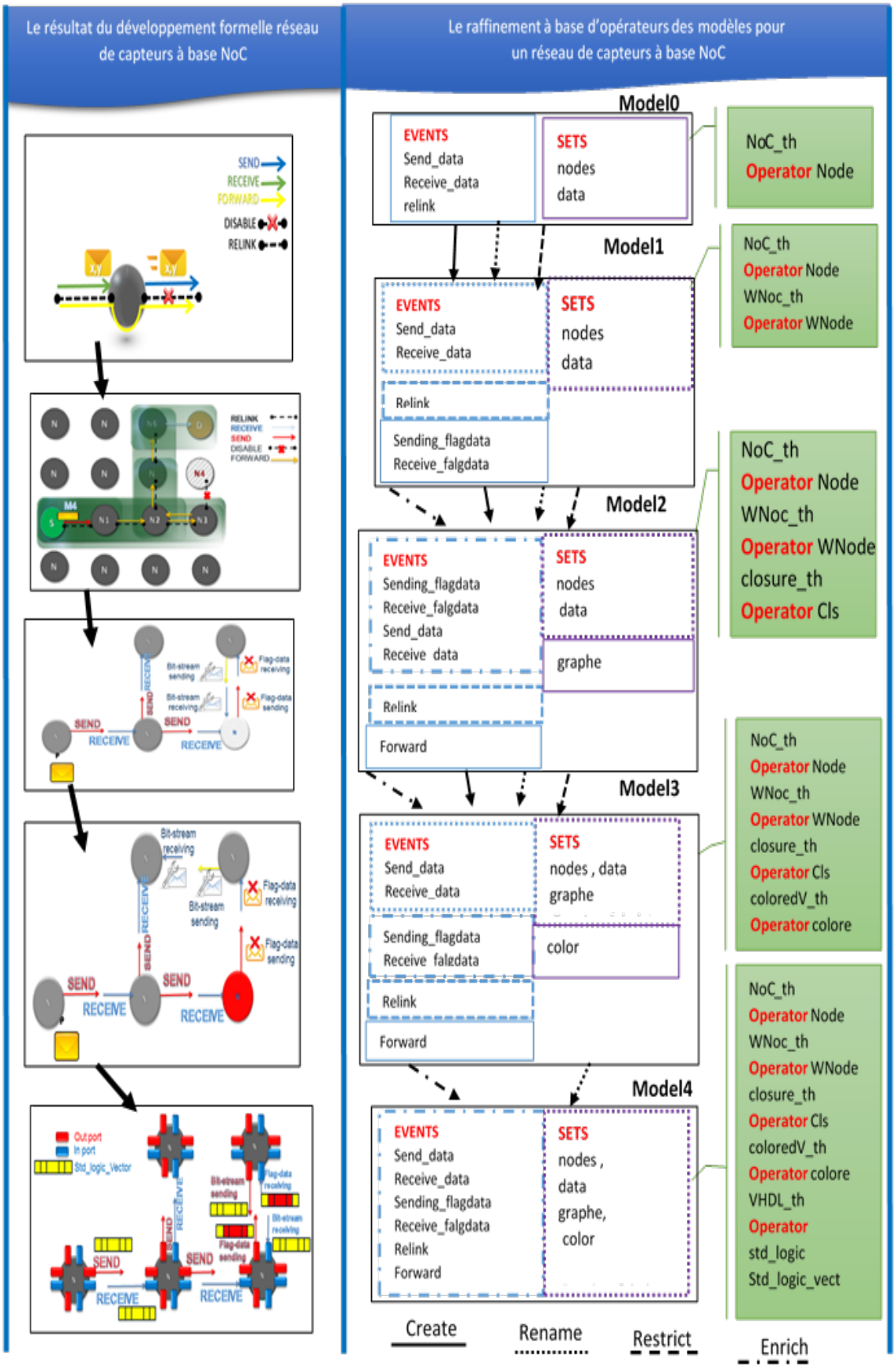


FIGURE IV.26 – Le processus de raffinement à base opérateurs pendant la modélisation à base théorie.

4.3.5.1 Opérateur de création

Au début de la modélisation en utilisant NoC théorie NoC_th pour présenter de nouveaux événements Sending _ data et Receiving _ data ($model0$ voir figure) nous pouvons aussi voir une création de nouvelle couleur de consigne $model4$ dans le moment que

4.3.5.2 Opérateur de renommage

Dans le second modèle il n'y a pas une telle différence entre l**'évènement Sending _** data dans $model0$ (qui décrit les propriétés NoC) en utilisant la théorie NoC_th et $model1$ (le modèle décrit les propriétés de reconfiguration par la théorie $WSNoC\_th$) **et cele nous permet de renommer cet événement**, nous pouvons dire la même chose chose sur **l'évènement** $receiving\_data$.

4.3.5.3 Opérateur d'Enrichissement

Après l'introduction de la théorie des graphes en $model1$ (ce modèle représente la théorie $WNoC\_th$ de récupération de NoC-commutateurs propriétés), nous ajoutons des actions aux événements sending _ data et $sending\_flagdata$ pour obtenir des événements sur $model2$, prendre un autre exemple présenté dans le dernier niveau de $model4$ (que la théorie $VHDL\_th$ utilisée), tous les événements sont enrichis de leur version abstraite de $Model3$ (en utilisant la théorie $ColoredV\_th$).

4.3.5.4 Opérateur de Restriction

L'exemple de l'utilisation **de l'opérateur** de restreindre présentée après l'introduction de la théorie des graphes en $model2$ pour assurer que chaque nouveau lien couvre toujours la propriété de fermeture de sorte que nous ajoutons plus guare pour l'événement $RELINK$ du $model1$ (que théorie couverture de $WNoC\_th$).

4.3.6 Résultats et performance

La méthode Event-B génère des obligations de preuve(voir le Tableau IV.1) . Le résumé de ces obligations de preuve libéré automatiquement ou de manière interactive est une mesure de la complexité du développement lui-même et on peut dire que la modélisation à base théorie nous facilite de façon compréhensive et automatique la validation des modèles pour ce système et cela est bien vu par le nombre de preuve interactive pendant la génération des obligations de preuve.
Dans le but de vérifier la bonne-fonctionnement de ce type de réseau sans fil des théories sont proposées pour être utilisé comme base pour modéliser le réseau . Ces modèles ont été créés par la combinaison de ces théories (bien expliqué dans les sections précédemment) : *la théorie des graphes colorés*, *La théorie de NoC*, *la théorie de WNoC*.Ensuite, la présente section nous conduit à valider également ce comportement est décrit dans la structure de VHDL en utilisant la théorie de VHDL déjà introduit auparavant. Après la modélisation de ce système en combinant ces théories, il obtient des preuves interactives déchargées de sorte qu**'on** peut dire clairement la stratégie de la modélisation est renforcée.
Pour vérifier ce système à partir de la théorie des graphes qui représentent le système distribué de NoC puis nous recueillons à l'ensemble **des propriétés des graphes colorés** pour valider la gestion de la stratégie de reconfiguration pour le réseau de sans fil NoC et de finaliser notre travail en incluant la théorie de VHDL pour obtenir une validation formelle du comportement VHDL de notre système.

4.3.6.1 Preuve de nivau abstrait

Commençant toujours avec *la théorie de NoC*, le nœud peut envoyer, recevoir et transmettre des paquets ; des OPs dans cette théorie sont déchargées manuellement (Figure IV.27), mais la plupart

Element Name	Total	Auto	Manual	Reviwed	Undischarged
Color_WNoC	61	47	14	0	0
ModC001	17	15	2	0	0
ModC002	10	10	0	0	0
ModM001	17	8	9	0	0
ModM002	7	5	2	0	0
ModM003	5	5	0	0	0
ModM004	5	4	1	0	0

TABLEAU IV.1 – Les statistiques de preuves de la modélisation SONoC à base théories.

des règles relatives au système NoC sont Convenant pour le prouveur RODIN et par le résultat des propriétés NoC pourrait être appliqué.

Element Name	Total	Auto	Manual	Reviewed	Undischarged
dftM0	**17**	**15**	**2**	**0**	**0**
INITIALISATI...	5	5	0	0	0
inv1	0	0	0	0	0
inv2	0	0	0	0	0
inv3	3	3	0	0	0
inv4	3	3	0	0	0
inv5	3	3	0	0	0
inv6	2	1	1	0	0
inv7	2	1	1	0	0
SEND	4	3	1	0	0
RECIEVE	5	4	1	0	0
FORWARD	1	1	0	0	0
inv8	0	0	0	0	0
inv9	0	0	0	0	0
inv10	0	0	0	0	0

FIGURE IV.27 – La preuve de l'application de la théorie NoC sur le système SONoC en Event-B.

4.3.6.2 Preuve du premier raffinement

Selon *la théorie de NoC* un noeud peut traiter à l'intérieur d'un réseau de switches à base de NoC qui appliquent toutes les propriétés du graphe en utilisant *la théorie de fermeture closure*, dans ce niveau le nombre de POs dans la preuve manuel déchargée sont en augmentation en raison de la complexité des propriétés du réseau combiné avec les propriétés NoC(Figure IV.28).

4.3.6.3 Preuve de deuxième raffinement

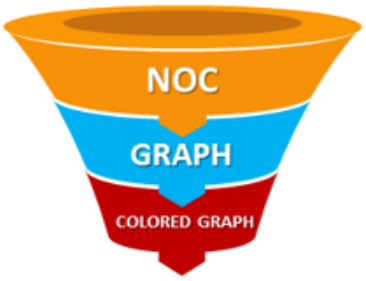
Le réseau basé sur NoC peut devenir dans l'état d'échec, de sorte que la théorie de la coloration (étendue de la théorie de fermeture) gérer la zone de défaillance (un certain nombre de noeuds défectueux) et colorés par les quatre couleurs disponibles ou en ajoutant à une liste dans l'attente d'être colorés , voici le nombre de POs pendant la preuve manuelle (interactive) encore c'était un grand effort de convincre le prouveur dans RODIN en fonction de la nouvelle stratégie de la manipulation d'un échec (Figure IV.29).

4.3.6.4 Preuve du troisième et raffinement

La théorie de WNoC exprimer l'émission des données de reconfiguration à **un nœud spécifique** $reconfig\_node$ **et la reception de bitstream et la transmission des paquets jusqu'à un chaque nœud défectueux devient un nœud à l'état initial et par conséquence le système renforce avec**

Element Name	Total	Auto	Manual	Reviewed	Undischarged
dftM01	**10**	**10**	**0**	**0**	**0**
INITIALISATI...	6	6	0	0	0
SEND	0	0	0	0	0
RECIEVE	0	0	0	0	0
COLORE_FAU...	3	3	0	0	0
FORWARD	0	0	0	0	0
inv1	0	0	0	0	0
inv2	0	0	0	0	0
inv3	2	2	0	0	0
inv4	1	1	0	0	0
inv5	2	2	0	0	0
inv6	0	0	0	0	0
inv7	2	2	0	0	0
inv8	1	1	0	0	0
inv10	1	1	0	0	0

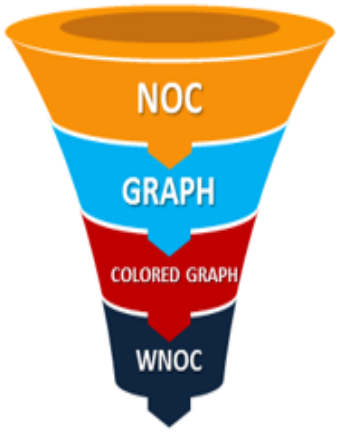
FIGURE IV.28 – La preuve de l'application de la théorie des graphes et la théorie NoC sur le système SONoC en Event-B.



Element Name	Total	Auto	Manual	Reviewed	Undischarged
dftM02	**17**	**8**	**9**	**0**	**0**
INITIALISATI...	3	2	1	0	0
SEND	0	0	0	0	0
RECIEVE	0	0	0	0	0
COLORE_FAU...	1	0	1	0	0
inv1	0	0	0	0	0
inv2	0	0	0	0	0
RECONFIG_N...	3	2	1	0	0
inv3	0	0	0	0	0
inv4	3	1	2	0	0
inv5	1	1	0	0	0
RECEIVING_BI...	5	2	3	0	0
inv6	1	1	0	0	0
MANAGE_CO...	3	0	3	0	0
FORWARD	0	0	0	0	0

FIGURE IV.29 – La preuve de la modélisation par la théorie des graphes colorés et la théorie NoC du système SONoC.

de façon que tous les nœuds sont corrects, il reste encore à colorer les nœuds défaillants qui sont en attente **pour** être récupéré, les POs sont tous prouvés automatiquement ce qui signifie que les règles de coloration et les propriétés NoC qui sont introduites précédemment rendu le système plus approprié (Figure IV.30).

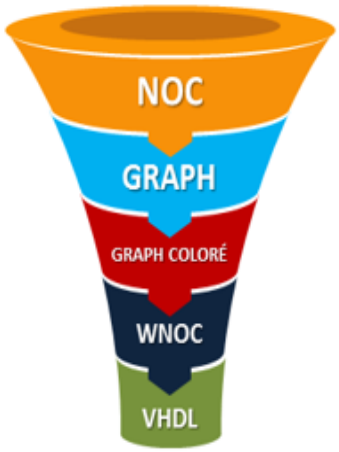


Element Name	Total	Auto	Manual	Reviewed	Undischarged
dftM03	**7**	**5**	**2**	**0**	**0**
INITIALISATI...	2	2	0	0	0
SEND	0	0	0	0	0
RECIEVE	0	0	0	0	0
COLORE_FAU...	1	1	0	0	0
RECONFIG_N...	0	0	0	0	0
FORWARD	0	0	0	0	0
CORRECTED_...	3	1	2	0	0
MANAGE_CO...	1	1	0	0	0
RECEIVING_BI...	0	0	0	0	0
inv1	4	4	0	0	0
inv2	0	0	0	0	0

FIGURE IV.30 – L'application de la théorie WNoC sur le système WSONoC en Event-B.

4.3.6.5 Preuve du quatrième raffinement

Après avoir combiné toutes les théories pour exprimer les propriétés de ce système, les variables VHDL sont introduites pour représenter le comportement VHDL de ce système et il est clair que toutes les POs sont automatiquement déchargées **et cela** signifie que les systèmes est prêt à l'étape de génération de code (Figure IV.31).



Element Name	Total	Auto	Manual	Reviewed	Undischarged
dftM04	**5**	**5**	**0**	**0**	**0**
INITIALISATI...	1	1	0	0	0
SEND	0	0	0	0	0
RECIEVE	0	0	0	0	0
COLORE_FAU...	1	1	0	0	0
RECONFIG_N...	0	0	0	0	0
FORWARD	0	0	0	0	0
CORRECTED_...	1	1	0	0	0
MANAGE_CO...	1	1	0	0	0
RECEIVING_BI...	0	0	0	0	0
inv1	5	5	0	0	0
COLORE_WAI...	1	1	0	0	0

FIGURE IV.31 – La preuve de l'application de la théorie de VHDL sur le système SONoC en Event-B.

4.3.6.6 L'animation de la stratégie par le plug-in ProB

Afin de valider notre stratégie, on effectue des scénarios de test de déroulement des évènements du système, il y a généralement des facteurs importants concrètement dans chaque événement :

- **Préservation des invariants :**(Voir Figure IV.32) c'est une obligation que la définition de chaque invariant doit être respecté dans tous les évènements dont il l'utilise. C-à-d chaque évènement concret dans un niveau N doit franchir tous les guards de l'évènement abstrait dans le niveau N-1 et les nouveaux guards introduits.

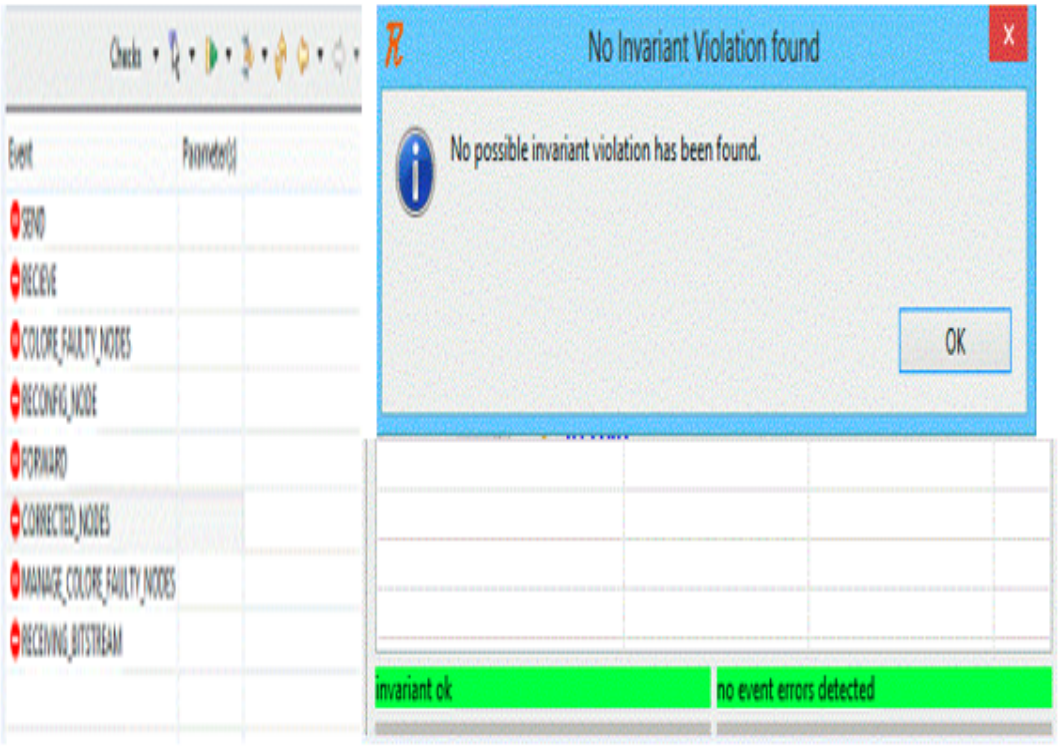


FIGURE IV.32 – L'animation par la préservation des invariants dans ProB.

- **L'obligation de preuve de raffinements :** exige que chaque événement raffiné à partir d'autre évènement de la machine abstraite doit être prouvé et respecte l'hiérarchie de règles de la précédente obligation de preuve(voir Figure IV.33).
- **L'animation par évènement pour arriver à dérouler tous les événements d'un model :** permet de détecter la défaillance de la stratégie de notre système décrite précédemment. Prenant l'évènement' $Colore\_Faulty\_node$ '(voir figure IV.34) :
- On remarque que une fois l'évènement est déclenché les actions de cet évènement sont exécutées, le nœud défaillant soit coloré, ajouté à l'ensemble des nœuds colorés, puis il incrémente le compteur.

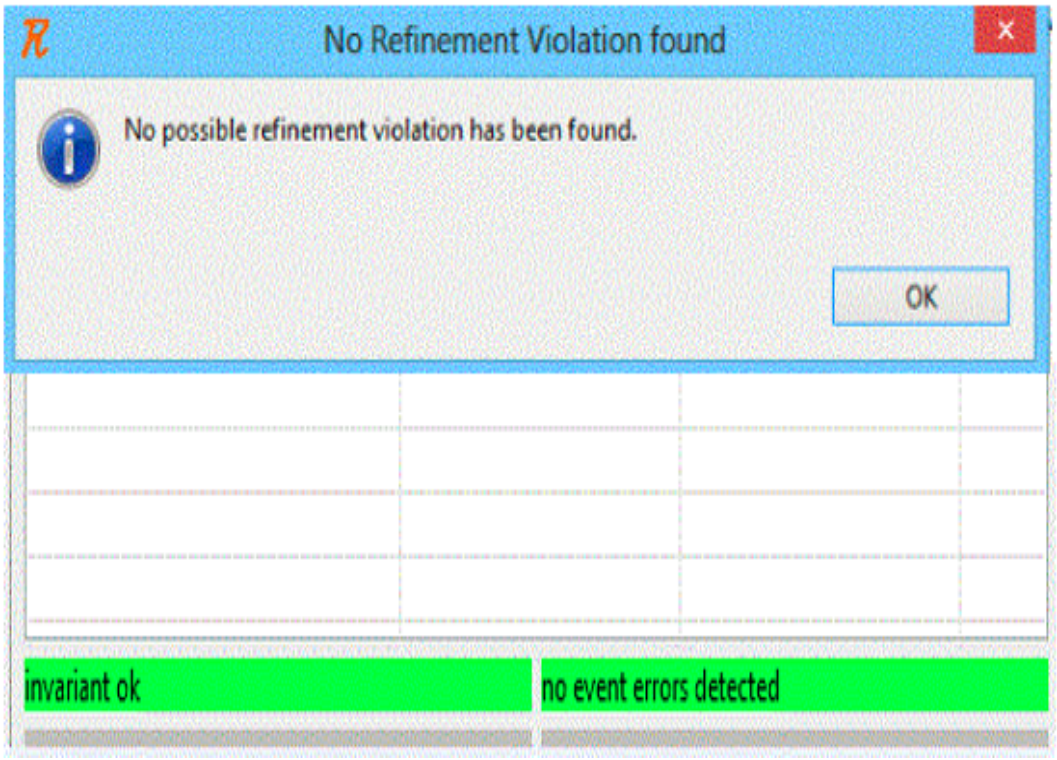


FIGURE IV.33 – L'animation par l'obligation de violation de raffinement des invariants dans ProB.

★ dftM01			
★ colored	{(NODES1↦red)}	ø	
★ count_colored	1	0	
faulty_node	NODES1	NODES1	
★ has_colored	{NODES1}	ø	
wait_to_be_colore	ø	ø	

FIGURE IV.34 – Le résultat de l'animation des évènements dans ProB.

4.4 La génération de code

Une des particularités du problème que nous tentons de résoudre ici est que les modèles Event-B qu'on est en train de travailler sur sont des modèles prouvés, ce qui mène à une translation valide. Une part importante de ce travail est donc la réalisation d'outil de traduction automatique d'un modèle mathématique vers une description VHDL. La première étape c'est d'extraire les fichiers XML des modèles Event-B. Une série de transformations est alors appliquée à l'aide d'un parseur des fichiers XML. Cette extraction précède la deuxième partie qui consiste à faire la correspondance entre le modèle Event-B et VHDL suivant des règles de génération.

4.4.1 Notre solution à base des fichiers XML

XML, acronyme de : eXtensible Markup Language (langage à balisage extensible), est un standard de structuration et d'échange de données, normalisé en 1996 par le . Le est un organisme dont l'objectif principal est la mise au point de recommandations (qui s'imposent souvent comme des standards) et de protocoles ouverts et libres, dans un souci d'interopérabilité maximale. Les principaux atouts de XML sont :

- La lisibilité : aucune connaissance ne doit théoriquement être nécessaire pour comprendre un contenu d'un document XML.
- Auto descriptif et extensible.
- Une structure arborescente : permettant de modéliser la majorité des problèmes informatiques.
- Universalité et portabilité : les différents jeux de caractères sont pris en compte.
- Intégrabilité : un document XML est utilisable par toute application pourvue d'un parseur (c'est-à-dire un logiciel permettant d'analyser un code XML).
- Extensibilité : un document XML doit pouvoir être utilisable dans tous les domaines d'applications, particulièrement adapté à l'échange de données et de documents.

XML permet de définir la structure du modèle Event-B uniquement, ce qui permet d'une part de pouvoir définir séparément la présentation de ce modèle, d'autre part d'être capable de récupérer les données pour les utiliser. L'idée est d'échanger les informations entre un modèle Event-B et un parseur en utilisant le langage XML. Cette méthode est déjà très largement utilisée pour l'interopérabilité entre systèmes d'information hétérogène. Elle est assez facile à mettre en œuvre.

4.4.1.1 Fichiers XML générés par l'outil RODIN

La plate-forme RODIN manipule plusieurs types de fichiers qui sont généralement distingués par leurs extensions. Le tableau suivant (Tableau IV.2) illustre quelques-unes. Les extensions suivantes ont une forme d'un fichier XML : **.bum**, **.buc**, **.bcc**, **.bpo**, **.bpr** (ainsi explique dans le chapitre 2) :

Extension Du fichier	**Type d'élément racine**	**Contenu**
.bum	IMachineRoot	Une Machine Event-B
.buc	IContextRoot	Un contexte Event-B
.bcm	ISCMachineRoot	Une Machine Event-B vérifié Statiquement
.bcc	ISCContextRoot	Un contexte Event-B vérifié Statiquement
.bpo	IPORoot	Les Obligations de Preuve Event-B
.bpr	IPRRoot	Les Preuves Event-B
.bps	IPSRoot	Les Statues de la Preuve Event-B

TABLEAU IV.2 – Les types de fichiers utilisés par la plateforme RODIN.

4.4.1.2 Parseur XML

XML est uniquement un langage de structuration et de représentation de données. Il ne comporte pas d'instructions de contrôle et ne permet donc pas d'exploiter directement les données. Pour réaliser notre application basée sur ces fichiers XML, il faut donc avoir recourt aux langages de programmation classiques (Java dans notre cas), un analyseur syntaxique XML (ou parseur Voir IV.35), permet de récupérer dans une structure XML, des balises, leur contenus, leurs attributs et de les rendre accessibles.
Le parseur sert donc à "découper" le fichier de données en un ensemble de "mots" et à les mettre à disposition d'applications. Il est associé à un module de traitement qui a lui un rôle plus important le processeur XML. Il en existe deux types :

- Les processeurs SAX (Simple API for XML), orientés événement.
- Les processeurs DOM (Document Object Model) orientés hiérarchie.

DOM le parseur qu'on a exploité (selon [354]) dans notre travail est une API basée sur une arborescence pour accéder aux documents XML.

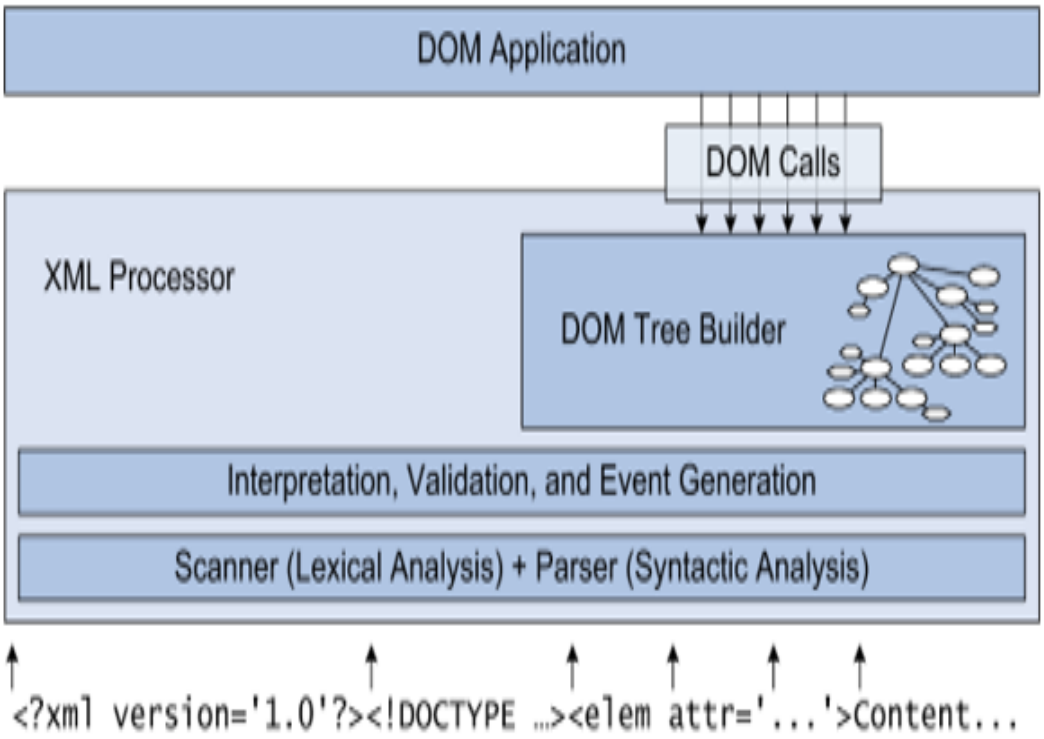


FIGURE IV.35 – Le parseur DOM .

4.4.2 Notre mécanisme de translation automatique

Cette section présente les différentes étapes qu'on a suivre (voir IV.36) afin d'accomplir une translation respectant les règles de la traduction entre un modèle Event-B et une description VHDL. Le processus de traduction consiste à transformer la partie concrète d'un projet Event-B dans un texte sémantiquement équivalent écrit en langage VHDL. Nous proposons une architecture pour le générateur Event-B.

4.4.2.1 Génération du code VHDL

Dans cette dernière phase(voir IV.35) On est sure que les déclarations pour un modèle Event-B (machine ou contexte) toujours reste sous types d'une déclaration, les instruction sont traduit

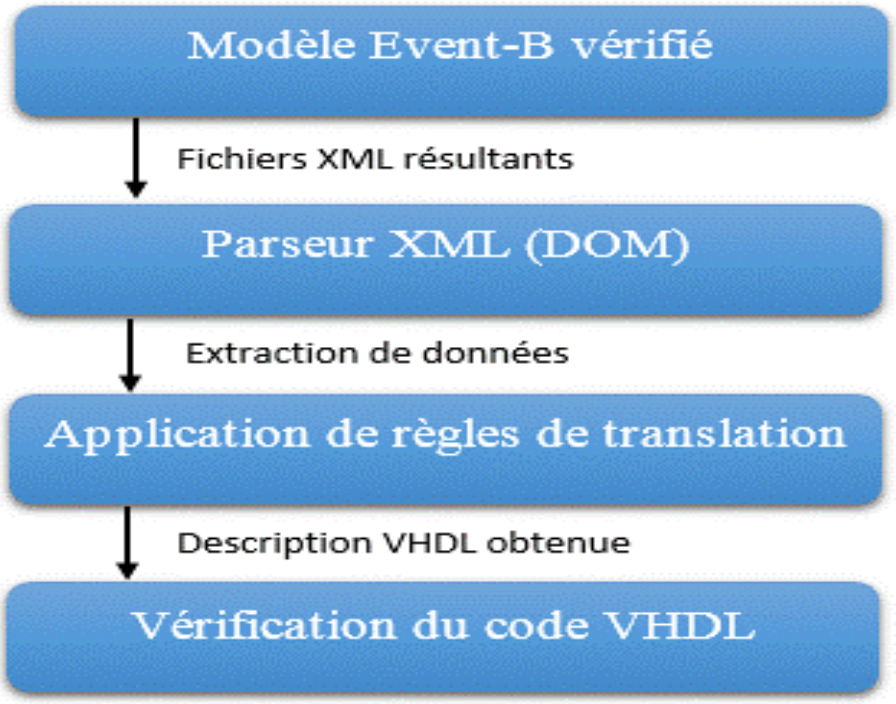


FIGURE IV.36 – Spécification du processus de la génération automatique du code VHDL.

à partir des partie dynamiques comme les évènement et les assertions (gardes, axiomes, etc. ..). Les deux tableaux ci-dessus englobent les translations faites (soient pour les déclarations soient des instructions) de l'Event-B vers VHDL. Chaque déclaration de VHDL pourrait être décrit en utilisant la théorie de l'Event-B et ici une correspondance entre les deux, d'une part, nous montrons comment représenter les types en VHDL utilisant Event-B, puis nous montrons les instructions similitudes entre Event-B et VHDL et enfin nous montrons le composants VHDL dans Event-B. L'utilisation de la $VHDL\_Th$ théorie dans chaque modèle que nous appelons en invariant avec l'un de ces structures dans Event-B cela signifie que nous représentons l'une des déclarations de VHDL représentés sur le tableau suivant. Les événements qui utilisent un invariant qui comprennent

Event-B structure (VHDL theory)	**VHDL Statement in Code**
$VHDL\_int$	$Integer$
std_logic	std_logic
$std\_logic\_vector(\cdots)$	std_logic_vector

TABLEAU IV.3 – Les structures de l'Event-B générés En VHDL.

un type de la théorie de $VHDL\_th$ pour définir un ensemble d'actions ces derniers doit être une instruction de VHDL que nous essayons de montrer dans le prochain : Depuis une machine peut avoir différentes variables qui représentent les entrées et sorties ainsi que des données internes, il est une nécessaire pour les distinguer. Ceci est très clair en utilisant la théorie VHDL_th qui fait la différence entre une entité des variables de l'architecture comme suit :

4.4.2.2 Les règles de génération de code VHDL

Pour activer une génération du code VHDL à partir du modèle, le modèle doit être déterministe. En d'autres termes, la dernière étape de **raffinement** doit contenir seulement des opérations déterministes sur les variables qui ont présenté **par des data-**types. Par conséquent, avoir le modèle de

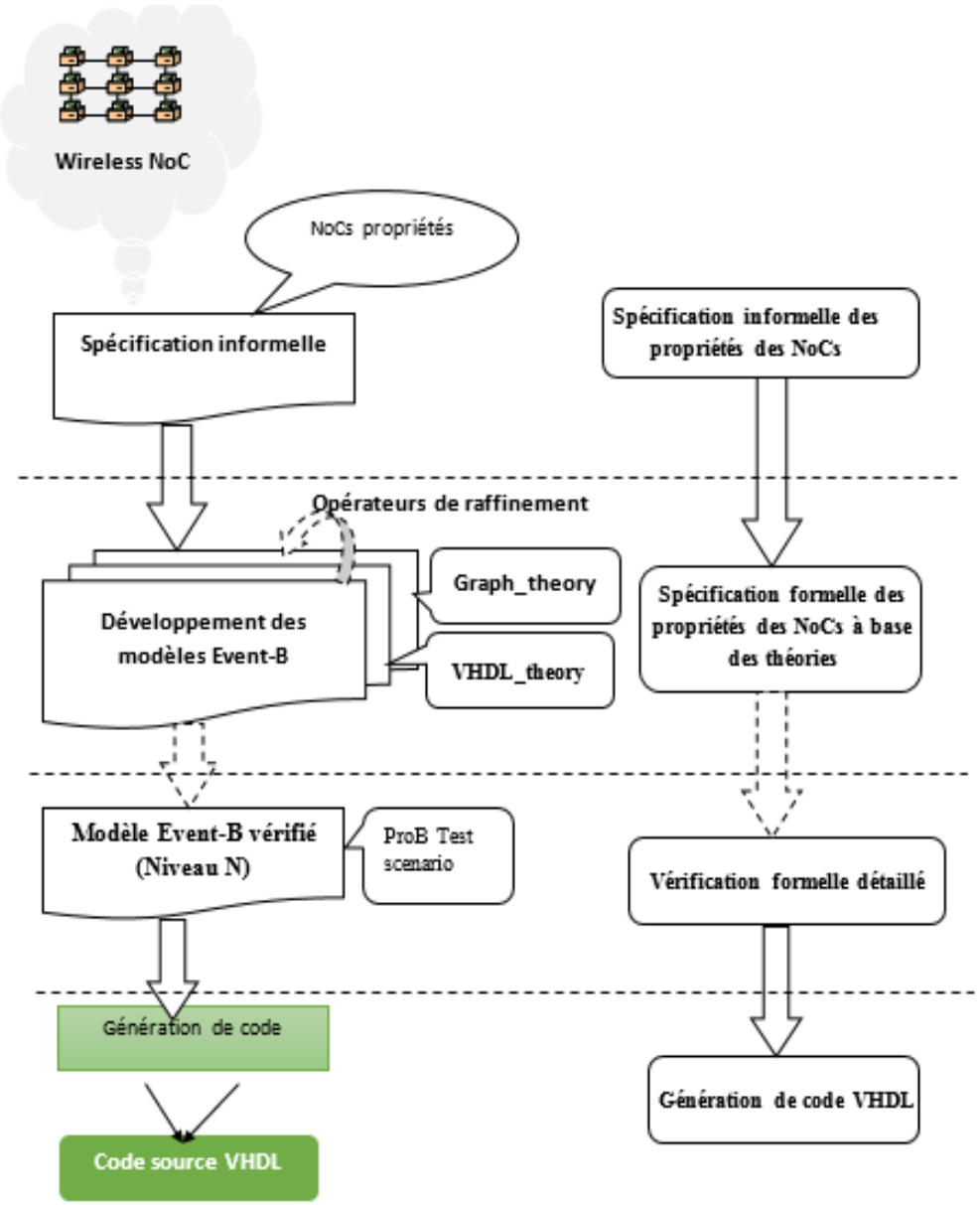


FIGURE IV.37 – Le placement de la phase de génération de code dans notre approche.

Event-B action		**VHDL instruction**	**Comment**
$std\_mult(a,b)$	$\Longmapsto$	a*b	a,b : integer
$a := b$	$\Longmapsto$	$a := b$	a,b : integer
$assign\_vect(c,d)$	$\Longmapsto$	$c <= d$	c,d : std_logic
$vhdl\_plus(a,d)$	$\Longmapsto$	a + b	a,d :integer
$assign\_vect(c,0)$	$\Longmapsto$	$c <= (OTHERS =>' 0')$	c : $std\_logic\_vector()$
$a := 0$	$\Longmapsto$	$a := 0$	a : integer
$\begin{cases} (h := 0 \\ h := 1) \end{cases}$	$\Longmapsto$	$\begin{cases} (h :=' 0' \\ h :=' 1') \end{cases}$	h : std_logic

TABLEAU IV.4 – Les actions de l'Event-B générés en VHDL.

Event-B Invariants		**VHDL component**

MACHINE **ModName**
INVARIANTS
Outputs ∈ port_decl(out, std_logic)
Inputs ∈ port_decl(in , std_logic_vector(max_buff ,std_logic))
EVENTS
INITIALISATION≙
 	
begin
Intputs := *def_val*
//Initialization of the other variables
⋮
end
⋮
	⟾	*ENTITY* **ModName** *is*
		port
		(−− *Inputs and outputs, their types*
		−− *and the default value*,
		
		Inputs : in std_logic := def_val ;
		Outputs : out std_logic_vector(max_buff-1 downto 0)
		);
		end **ModName**;

MACHINE **ModName**
INVARIANTS
bitstream ∈ arch_decl(vector(max_buff ,std_logic))
⋮
EVENTS
INITIALISATION≙
begin
btistream := *def_val*
//Initialization of the other variables
⋮
end
*evt*1≙
any ⋯
where G
then A
end
⋮
	⟾	ARCHITECTURE a of **ModName** is
		Begin
		Bitstream : std_logic_vector(max_buff-1 downto 0)
		Bitstream <= *def_val*;
		evt1 : process (clk,reset) is
		Begin
		if G
		then A
		end if
		
		end process ;
		end a

TABLEAU IV.5 – La génération d'une machine en Event-B vers le code VHDL.

l'Event-B final (le raffinement final) nous générons **de ce modèle un code VHDL en respectant** ces règles. L'algorithme de génération de code comprend les étapes suivantes :

- Chaque variable est transformée en signaux qui ont le type défini dans le Table précédant.
- Toutes les variables qui font partie de $port\_decl$ sont mis en clause de port de l'entité et sont fournis avec le mot clé qui reflète la direction de chaque signal (ou arrière) correspondant.
- Les variables qui sont parte de $arch\_decl$ sont représentés en tant que signaux internes et sont mis en disposition de l'architecture.
- **Comme toutes les variables en Event-B sont initialisées et ils sont générées comme des signaux en VHDL, donc ils(les signaux) sont mis en état initial**.
- **Chaque variable en Event-B notamment introduit son correspondant en VHDL est ajouté à l'architecture de la liste de sensibilité contenant tous les signaux**.
- Chaque événement du modèle se transforme en **instruction « si alors » en VHDL**.
- **L'événement qu'on introduit dont elle englobe tout les modèles raffinés (le dernier Niveau de raffinement noté par N) est transformé à un clause spécial d'instruction "si" et les variables que l'évènement utilise sont encore transformés à des entrées et des signaux en VHDL**.

Conclusion

Dans ce chapitre, nous avons présenté une approche qui améliore l'extensibilité de **langage en Event-B et son prouveur**. La construction de la théorie est utilisée pour définir et valider les règles de réécriture ainsi que théorèmes polymorphes. Les obligations de preuve sont générées pour veiller à ce que la solidité est maintenue. Nous avons montré comment la construction de la théorie peut être utilisée pour spécifier des règles de réécriture afin d'améliorer les capacités de réécriture de RODIN.
Durant ce chapitre, nous avons fourni un aperçu étendu du plug-in de la théorie. Nous avons décrit la composante de la théorie qui agit comme un support de place pour les différentes extensions. La vérification statique et la génération d'obligations de preuve sont étendus à vérifier et valider les théories. **Le déploiement rend les** théories immédiatement utilisables dans les modèles et les preuves. Le plugin de la théorie met en œuvre les idées présentées dans cette thèse, et il peut également être utilisé à d'autres fins telles que la génération de code [355]. Le plug-in de la théorie a été développée dans le cadre de l'ensemble de l'outillage du projet DEPLOY [356], qui vise à faciliter le développement de méthodes formelles dans l'industrie. **Le paquetage est concentré d'améliorer** le support d'outil pour l'Event-B au moyen de RODIN et d'autres plug-ins utiles.La description de l'architecture logicielle **par un langage cible qui nous intéresse. Elle est basée** sur un type abstrait de données structuré sous la forme de théories polymorphes **et cela donne l'Event-B** une puissance expressive importante, ce qui permet notamment **la spécification des** architectures dynamiques.
En outre, nous devons également définir comment passer d'un modèle à base de théorie à l'autre, **c.à.d** comment la description de l'architecture abstraite sera raffinée pour donner spécification formelle concrète du niveau suivant. Nous proposons donc de mettre en place un processus formel de raffinements successifs, écrit dans une langue officielle appropriée. Nous avons besoin de décrire formellement cette série d'améliorations par un procédé présenté par quatre opérateurs : créer, enrichir, renommer, restreindre.
La validation formelle d'un réseau de capteurs composé de l'ensemble des nœuds à base NoC auto-organisé consiste de montrer l'utilité de notre approche pour entamer à améliorer la phase de la génération du code VHDL correcte en terme de sémantique.
Cette validation a utlisé plusieurs théories : *théorie des graphes, théorie du langage VHDL* exprimé en Event-B via le plug-in de la théorie. L'intérêt d'utiliser Event-B est le support

d'outils qui aide à une modélisation assistée pour l'objectif d'atteindre des modèles "sains" et bien-définis. Après une série de raffinements générés par la présence des opérateurs qui facilitent le processus de raffinement en terme d'ergonomie et de la solidité de preuves.

Notre approche permet d'établir un modèle final prêt d'être transformer en code VHDL (importation des fichier de projet sous forme des fichier XML et les parser pour transformer vers un code VHDL en respectant les règles de la génération) ainsi qu'elle sert à adapter le code générer selon chaque amélioration locale établie pour enrichir le système global de développement de ce genre de systémes micro-éléctronique basés sur les NoC.

Conclusions & future travaux

Comme il **est** décrit par avant, la portée de cette thèse unifie les méthodes formelles, la logique et le génie logiciel. Notre travail vise à fournir un mécanisme utilisable dans la pratique par l'outil « tool-set » RODIN par laquelle la méthodologie formelle Event-B peut être profondément étendu. Plus succinctement, cette thèse fait les contributions suivantes :

(a) Le flot de conception d'une application **basé sur la technologie FPGA est habituellement réalisé en plusieurs étapes et catégorisé en deux fameuse approches classique ou moderne.Ces genre de systèmes sont souvent utilisés soit pour le prototypage des systèmes complexes comme les Système à base NoC, soit comme des systèmes dédiés au traitement du signal et d'image**. Après la description du produit **selon les besoins** du marché on **obtient** un revenu maximum mais le délai de la simulation discret de tous les cas possibles dans une architecture peut causer une perte de revenus vu le temps de déclin du marché. **Á cet** effet nous **avons** menés de trouver des solutions pour améliorer le facteur *nombre de revenue /temps de validation* pour l'ensemble de produit de qualité et cela nous à guider directement à l'une de méthodes formelles qu'elle a une grande expertise dans la validation industrielle : Event-B.

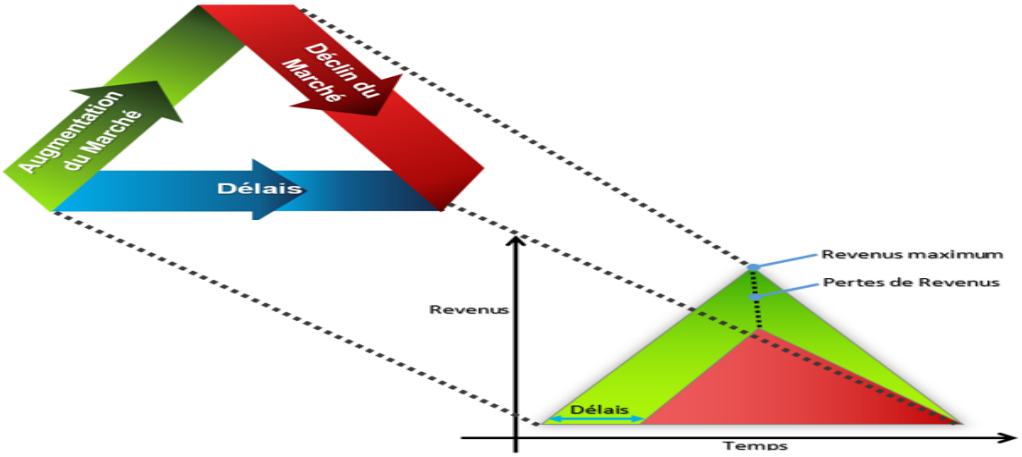


FIGURE XX.1 – Schéma du temps de mettre en marché une architecture développée.

(b) On a montré comment l'extensibilité et la reconfigurabilité de RODIN peuvent être exploitées pour ajouter des fonctionnalités utiles à l'outil **dédié à** l'Event-B. **L'outil** RODIN et la notation de modélisation Event-B **ont** été conçu avec l'esprit de l'extensibilité et l'adaptabilité [260, 261]. Nous soutenons que ces aspects de l'architecture RODIN ont beaucoup aidé dans la réalisation des idées présentées dans cette thèse. L'utilisation d'un analyseur dynamique de l'arbre de syntaxe abstraite en Event-B (AST) a permis de l'ajout de nouveaux opérateurs et types de données. La facilité de preuve avec laquelle l'outillage (par exemple, l'outil de vérification statique) est **augmentée de** grande partie en raison de la haute reconfigurabilité de la plateforme RODIN. D'autres aspects sur l'architecture **de l'outil RODIN réservé pour l'Event-B qui sont décrits au chapitre 2**.

(c) Dans le chapitre 3 nous avons traité l'opération plus élémentaire dans cette thèse, la transformation d'une description d'architecture logicielle **vers** une description formelle dans la notation d'une méthode formelle classique en renforçant les liens entre les étapes de description architecturale et de spécification classique par l'intermédiaire de processus de raffinement. Notre approche de raffinement (voir Tableau XX.1) en Event-B est appliquée pendant la phase de conception architecturale et assure la traduction des constructions d'un langage **vers** un autre qui ne manipule pas les mêmes concepts. Pour ce faire, chaque "transformation élémentaire" est représentée par une ou plusieurs règles sous forme des théories polymorphes. Nous effectuons, par ailleurs, une **une amélioration pour** le raffinement supporté par la méthode Event-B dans la décomposition architecturale :

Place du raffinement	Type de processus de raffinement	Relation de raffinement
Conception Architecturale	Transformation par opérateurs génériques utilisant des théories polymorphiques	affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme) par des règles pendant la manipulation des évènements selon l'état de ses invariants préservées et leur définitions préservés dans les contextes (axiomes, ensembles, etc.).

TABLEAU XX.1 – Les critères principaux de raffinement de notre approches.

Distinction RH \ RV	OUI
Raffinement Horizontal (RH)	OUI
Raffinement Vertical (RV)	OUI
Raffinement de Données	OUI
Raffinement Fonctionnel	OUI
Raffinement Comportemental	OUI
Raffinement Compositionnel	OUI
Génération de code	NON (mais générateur créé)
Multiples niveaux d'abstraction	OUI
Réutilisation	OUI
Préservation des propriétés inhérentes	OUI
Préservations des propriétés définies par l'utilisateur	OUI
Multi-langages	OUI

TABLEAU XX.2 – Les critéres auxiliaires de raffinement dans notre approche.

— Un raffinement horizontal et celui que nous effectuons pour transformer une architecture en des machines composées avec des contextes pour créer des modèles de la méthode formelle Event-B.

— Un raffinement vertical (voir Tableau XX.2). Cette transformation **est destinée pour** modifier les données de la spécification, **et encore plus, assurer le comportement de raffinement** par l'utilisation de quatre opérateurs de création, d'enrichissement, de restriction et de renommage (create, enrich, restrict, rename) à cet instant, nous pourrions opérer le raffinement de modèles par des nouvelles décompositions, et **les composer passant par** des différentes étapes de raffinements verticaux. **Généralement** nous gérons le raffinement de l'architecture jusqu'à la génération de code VHDL, **mais pour notre cas la génération de code** peut être effectuée dans le cadre d'un développement formel **dès que la spécification en Event-B est établie. L'intérêt de faire passer traitement conçue pour modéliser par de théories polymorphes et types abstrait de données nous a aidé à la fois d'obtenir** une génération de cadre syntaxique du code VHDL et pour respecter le cadre sémantique pour le comportement des systèmes à base NoC (par exemple stockages, algorithme de routage, l'adaptation de système), **et plus précisément modéliser des spécification par le passage à base des opérateur** nous pouvons garantir la conservation des propriétés des descriptions raffinées (Nouveau mécanisme d'auto-organisation). En outre, elle nous permettrait également de vérifier de nouvelles propriétés interprétés sous formes des ADT (des datatype et operators dans une théorie), qui seraient, par exemple, réutilisé au développement formel en Event-B.

(d) Nous avons montré ainsi comment l'ancien paradigme existant utilisé dans les développements **en** Event-B peut être utilisé pour la méta-raisonnement **dès** que la modélisation en Event-B est réalisée au moyen de contextes et de machines.**Nous avons décrit que** les obligations de preuve sont générées pour vérifier la cohérence du système par rapport à une certaine sémantique comportementale (propriétés de système NoC, propriétés de graphe, les propriétés de réseau sans fil). La méta-raisonnement peut être effectuée à l'aide du composant de la théorie à spécifier **le langage destiné pour le développement d'un système quelconque** et des extensions de preuve. Les obligations de preuve sont ensuite utilisées pour assurer les extensions sont conservatrices par rapport à la logique qui sous-tend le langage mathématique Event-B. Nous soutenons que la familiarité de notre approche peut être considérée comme un aspect important de la facilité d'utilisation de notre outil **proposé pour améliorer l'outil RODIN**. Cette contribution est décrite au chapitre 3 et 4.

(e) **Sachant que les extensions en Event-B sont basés sur le concept de la méta-raisonnement alors que leurs solidité est assurée par l'utilisation des obligations de preuve**. Pour chaque extension de preuve et langage mathématique, certains contrôles statiques sont effectués. Ces contrôles de solidité sont effectués par des moyens de la génération d'obligation de preuve. **Nous pouvons énoncés depuis une étude des travaux de Schmalz [35] que le méta-raisonnement disponible dans l'outil RODIN grâce au plug-in de la théorie "Theory plug-in"** ne correspond pas à la disposition d'un méta-modèle pour l'Event-B. Un tel effort est considéré comme un plongement peu profond de l'Event-B avec Isabelle / HOL [35]. Et ceci qui est décrite au chapitre 4.

(f) **Aussi dans le chapitre 4, nous expliquons comment les nouveaux opérateurs polymorphes peuvent être définies dans le volet théorique par une Prédicat (par exemple, la formule) et une expression (le terme) et comment elles peuvent être spécifiés pour former la composante de la théorie en Event-B. Les opérateurs doivent être validées au moyen d'obligations de preuve. L'argument qui affirme la reutilisabilité de l'Event-B est que même les obligations de preuve relatives aux opérateurs nouvellement introduites sont justifiées dans cette thèse en se basant sur les travaux [35].** En Event-B la théorie est La composante qui peut être utilisée pour spécifier des théorèmes polymorphes et les règles de preuve. Les théories polymorphes seront utilisées et manipulées par des opérateurs de raffi-

nement pendant la concrétisation d'une description formelle en Event-B pour un système par exemple l'émission dans un réseau à base NoC est enrichis pour respecter les critères de la nouvelle stratégie d'auto- reconfiguration, le raffinement pendant la modélisation en Event-B utilise l'opérateur ENRICH **pour renforcer les critères du modèle abstrait représentés par la théorie NoC par une théorie qui respecte les critères du système à base WNoC**.Les théorèmes polymorphes sont des formules dans l'Event-B qui peuvent être utilisés dans les preuves à condition qu'une instanciation de type approprié est fourni. **Dans certaines théorie exprimées en Event-B on est besoin de plus en plus des règle de réécriture et d'inférences pour générer les obligations de preuve. Durant l'utilisation de ces théories par des modèles, les obligations de preuve relatifs à des théorèmes , de règles de réécriture ou des règles d'inférences assurent qu'ils sont valides et bien défini.** L'adéquation des différentes manipulations (en terme des opérateurs) des extensions du langage à base des théories sont justifiées dans les chapitres 3 et 4.

(g) L'utilisation d'obligations de preuve est primordiale pour assurer que la solidité est préservée. Ceci est évident d'après les résultats du chapitre 4 et bien expliqué par l'exemple de réseau à base NoC. En générale, cette thèse a contribué le bénéfice de la réutilisabilité de l'extensibilité de l'Event-B qui maintient les exigences importantes suivantes :

i. ***La facilité d'utilisation :*** le support d'outil fournit un moyen efficace et un mécanisme utilisable dans la pratique de la spécification appliquée sur les extensions. L'approche proposée durant ce travail permet la réutilisation des définitions, et réduit l'effort de preuve dans les développements multiples.

ii. ***La solidité de préservation :*** l'utilisation des obligations de preuve sur les extensions garantit que l'utilisateur est au courant de toutes les extensions **potentiellement non** solides.

(h) Notre approche est fondée après un rappel utile des caractéristiques des différents formalismes existants. Alors que précédemment nous **avons présenté** l'état de l'art sur différents perfectionnements des méthodes dans un cadre assez général, nous nous concentrons sur un problème plus spécifique le mécanisme de raffinement en Event-B. En amont nous proposons le défait d'un développement formel classique de la description architecturale du langage de haut niveau VHDL, pour cette raison nous allons proposer d'améliorer la logique de développement dans la méthode Event-B. Dans un premier temps, une description architecturale du système provient de la phase d'analyse des besoins sous forme de théories. **Ensuite le raffinement de la description est déclenché en se basant sur cahier des charges d'un langage abstrait formel qui conduisent par la suite à la mise en œuvre du système au moyen d'un développement formel classique**. En outre, nous devons également définir comment faire passer d'un niveau à l'autre, autrement dit comment la description de l'architecture abstraite sera raffinée pour donner spécification formelle plus concrète. Nous proposons donc de mettre en place un processus formel de raffinements successifs, décrivent les fonctionnalités d'un système à partir le cahier de charge jusqu'à l'implémentation en code VHDL. Nous avons besoin de décrire formellement cette série d'améliorations par un procédé présenté par quatre opérateurs : créer, enrichir, renommer, restreindre. La description de l'architecture logicielle qui nous intéresse est basé sur un type de données abstrait ADT structuré sous la forme de théories polymorphes et cela donne à l'Event-B une puissance expressive importante permet notamment l'expression des architectures dynamiques complexes comme le système à basé-NoC . **Cette approche qui nous entraine d'expliquer est très utile lors de la génération de code et le développement de ce type de réseau. Notre cas d'étude est un système qui est un réseau composé par des nœuds sans fil auto-organisés, nous avons mené dans cette approche a utilisé une théorie des graphes pour gérer les propriétés de ce réseau et de la théorie de VHDL modélisées avec Event-B dans le plug-in de la théorie pour exprimer le comportement des noeud en code VHDL. L'Event-B est en effet le grand avantage**

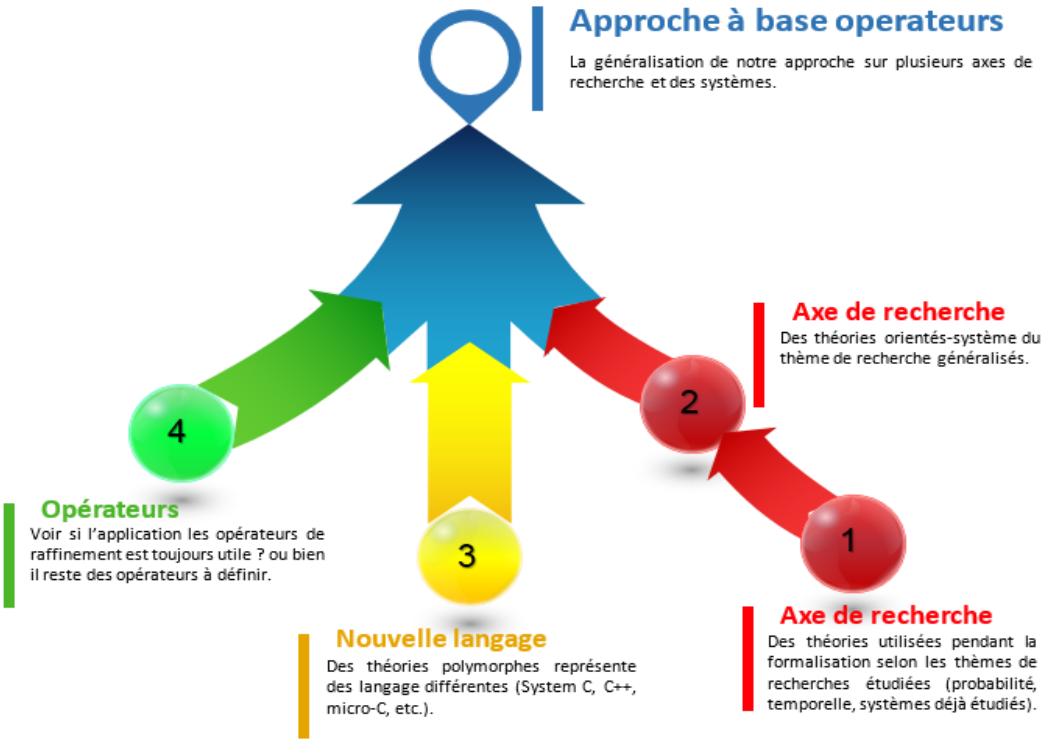


FIGURE XX.2 – Les perspectives de notre travail.

d'avoir des outils (RODIN Toolset) pour aider à l'élaboration et du code VHDL à partir des spécifications formelles.

— **Les perspectives de futur**
Les éléments suivants décrivent les domaines dans lesquels des recherches supplémentaires peut être réalisée comme une extension de notre travail.

i ***Robustesse de l'approche de raffinement à base operateur***
Les perspectives ouvertes par nos travaux peuvent se grouper selon trois axes (cf. figure XX.2) : la continuation du travail , des nouvelles applications et des nouveaux thèmes de recherche ouverts. Nous allons par la suite développer les perspectives développées selon les trois axes :

(I) Le premier pas dans la continuation de notre travail concerne le développement de l'environnement permettant à l'utilisateur de raffiner en utilisant des opérateurs en Event-B. En effet, nous n'avons pas encore éprouvé notre approche sur un nombre significatif de systèmes. **L'application** de opérateurs au raffinement en Event-B de descriptions d'architectures différentes, de plus grande taille, pourrait ainsi en faire ressortir d'autres avantages et inconvénients, par exemple pour des aspects non fonctionnels comme l'ergonomie de l'interface, mais aussi pour confirmer la complétude de définitions liées relativement aux descriptions d'architectures logicielles en Event-B **développées en utilisant les opérateurs de raffinement précédemment proposées.**

(II) Cette approche présentée est dépendante à la fois de la méthode Event-B et des ADTs polymorphe Néanmoins, nous aurions pu choisir un autre langage de description ou une autre méthode formelle. En effet, **nos objectifs de transformations s'articulent autour d'un format architectural, basé sur les constituants élémentaires de toute architecture logicielle. Les deux langages, source et cible, se trouvent fortement découplés et si on change un il ne posera d'autre problème que lors**

l'écriture d'un nouveau but pour les règles de passage pour assurer que la transformation relie le nouveau langage utilisé à cette forme polymorphique.

(III) L'approche que nous proposons dans cette thèse est **liée** par plusieurs éléments, et **elle** fournit dans le même temps des structures (ADT ou théories en Event-B) génériques. Cette proposition aussi affirme que notre travail ouvre de nouvelles perspectives, que ce soit dans la dissociation des raffinements horizontaux et verticaux, **soit** dans la transformation d'un formalisme vers un autre. L'application de notre approche à d'autres langages peut être envisagée à court terme, mais l'écriture "manuelle et intuitive" de nouveaux raffinements nécessite un degré d'expertise **et certain logique pour opérer** le passage du nouveau langage source au cible considéré. C'est pourquoi nous devons, à moyen terme, après avoir pris un peu plus de recul et d'expérience, considérer la définition d'un méta-modèle **pour le processus de transformation d'une spécification vers un langage de description d'architecture quelconque, c'est à dire d'un langage formel vers une spécification dans un autre langage formel. C'est une tâche qui s'annonce difficile, mais qui s'inscrit en priorités dans les travaux menés au laboratoire pour cerner à améliorer les concepts** dans le domaine du raffinement d'architectures logicielles [357].

ii ***Création d'une bibliothèque de théories***
Les formalismes établis comme Isabelle ont une riche ensemble de bibliothèques allant de la théorie simple jusqu'à les théories de mathématiques continus complexes. La création d'une bibliothèque **similaire en Event-B** peut fournir un ensemble standard de théories qui peuvent être utilisées pour enrichir l'activité de modélisation. Une attention particulière devrait être accordée pour assurer **que les théories qui** sont définies dans certaines hiérarchies sont bien compris pour faciliter **la tâche de la validation des modèles**.

iii ***Amélioration de Soutien aux Data-types***
Actuellement, **le Theory plug-in ne prend en charge les définitions de types de données énumérés et simples. Cependant, mutuellement les définitions de types de données récursives pourraient également être prise en charge dans les futures versions**. En outre, une étude fondamentale des types de données dans la logique de l'Event-B pourrait servir à la poursuite de base des travaux sur ce sujet.

iv ***L'Event-B2VHDL***
La phase de la génération automatique d'une description formelle en Event-B vers un code VHDL dans notre cas était par l'exemple **pour le** système à base NoC ou on a pu élaborer des règles qui assurent la bonne génération de code VHDL. **cet effet** a comme avantage en plus que la validation de l'implémentation de solutions embarquées, l'optimisation du temps de réponse et du temps de la réalisation du système. La standardisation de cette étape cela sera beaucoup plus fiable et de valeur primordiale si **elle** sera intégré étant un plug-in dans RODIN qui est plus proche de la sémantique de système modélisé que l'outil EHDL [358].

Bibliographie

[1] J. Bergeron, *Writing testbenches : functional verification of HDL models*. Springer Science & Business Media, 2012. 6

[2] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007. 6

[3] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992. 6

[4] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 269–274, 1995. 6

[5] D. E. Perry, "State of the art : Software architecture," in *International Conference on Software Engineering*, vol. 19, pp. 590–591, IEEE COMPUTER SOCIETY, 1997. 6

[6] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 3, pp. 213–249, 1997. 6

[7] D. Compare, P. Inverardi, and A. L. Wolf, "Uncovering architectural mismatch in component behavior," *Science of Computer Programming*, vol. 33, no. 2, pp. 101–131, 1999. 6

[8] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems and Their Applications*, vol. 14, no. 3, pp. 54–62, 1999. 6

[9] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of the 41st annual Design Automation Conference*, pp. 409–414, ACM, 2004. 6

[10] A. Romanovsky, "Deploy : Industrial deployment of advanced system engineering methods for high productivity and dependability," in *Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pp. 117–119, ACM, 2008. 6, 33, 68

[11] M. Ziane, "Towards tool support for design patterns using program transformations," *Langages et Modèles à Objets (LMO)*, vol. 7, pp. 199–214, 2001. 6

[12] J.-R. Abrial, *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010. 6, 9, 59, 60, 61, 71, 103

[13] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models : Application to event-b," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1–28, 2007. 6

[14] J.-R. Abrial, "A system development process with event-b and the rodin platform," in *International Conference on Formal Engineering Methods*, pp. 1–3, Springer, 2007. 6, 124

[15] J.-R. Abrial and J.-R. Abrial, *The B-book : assigning programs to meanings*. Cambridge University Press, 2005. 6, 8, 52, 75

[16] S. Hallerstede, "On the purpose of event-b proof obligations.," *ABZ*, vol. 5238, pp. 125–138, 2008. 7, 130

[17] M. Butler and S. Hallerstede, "The rodin formal modelling tool," in *BCS-FACS Christmas 2007 Meeting-Formal Methods In Industry, London.*, 2007. 7, 66, 124, XXXI

[18] J.-R. Abrial and D. Cansell, "Click'n prove : Interactive proofs within set theory," in *TPHOLs*, vol. 2758, pp. 1–24, Springer, 2003. 7, 9, 127

[19] R. A. Riemenschneider, "Correct transformation rules for incremental development of architecture hierarchies," tech. rep., Working Paper DSA-98-01, Dependable System Archiecture Group, Computer Science Laboratory, SRI International, Menlo Park, CA, 1998. 7, 8

[20] P. C. Clements, "A survey of architecture description languages," in *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, pp. 16–25, IEEE, 1996. 7, 87

[21] D. Garlan, "Style-based refinement for software architecture," in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pp. 72–75, ACM, 1996. 7

[22] L. A. Clarke, "Improving architectural description languages to support analysis better," in *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA 1998), Marsala, Sicily, Italy*, pp. 78–80, 1998. 7

[23] T. Bolusset, *β-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B*. PhD thesis, Université de Savoie, 2004. 7

[24] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000. 7, 87

[25] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE transactions on software engineering*, vol. 21, no. 4, pp. 356–372, 1995. 8

[26] M. Moriconi and X. Qian, "Correctness and composition of software architectures," in *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 164–174, ACM, 1994. 8

[27] M. Moriconi and R. A. Riemenschneider, "Introduction to sadl 1.0 : A language for specifying software architecture hierarchies," tech. rep., Technical Report SRI-CSL-97-01, SRI International, 1997. 8

[28] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–354, 1995. 8

[29] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, 1995. 8

[30] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz, "Partial orderings of event sets and their application to prototyping concurrent, timed systems," *Journal of Systems and Software*, vol. 21, no. 3, pp. 253–265, 1993. 8

[31] J. Abrial, "R. ; system and software engineering in modelling in event-b," 2010. 8, 33

[32] A. Hariche, M. Belarbi, and H. Daoud, "A new operators-based approach for the event-b refinement : Qnoc case study," in *Microelectronics (ICM), 2013 25th International Conference on*, pp. 1–4, IEEE, 2013. 8, 100

[33] F. Mehta, "Supporting proof in a reactive development environment," in *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pp. 103–112, IEEE, 2007. 8, 70, 131

[34] F. D. Mehta, *Proofs for the working engineer*. PhD thesis, 2008. 9, 10, 70, 71, 73, 74, 126, 127, 128

[35] M. Schmalz, *The logic of Event-B*. 2011. 10, 45, 46, 84, 105, 172

[36] I. Maamria and M. Butler, "Rewriting and well-definedness within a proof system," *arXiv preprint arXiv :1012.4897*, 2010. 10, 44, 45, 74, 126, 127, 128

[37] I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh, "On an extensible rule-based prover for event-b," *Abstract State Machines, Alloy, B and Z, volume 5977 of Lecture Notes in Computer Science*, 2009. 10, 44, 45, 74, 126, 127, 128, 139

[38] F. Mehta, "A practical approach to partiality-a proof based approach.," in *ICFEM*, vol. 5256, pp. 238–257, Springer, 2008. 10, 46

[39] T. S. Hoang and J.-R. Abrial, "Event-b decomposition for parallel programs," in *International Conference on Abstract State Machines, Alloy, B and Z*, pp. 319–333, Springer, 2010. 11

[40] M. J. Butler, "Decomposition structures for event-b.," in *IFM*, vol. 9, pp. 20–38, Springer, 2009. 11, 108

[41] A. HARICHE, M. BELARBI, and H. Daoud, "Based-b extraction of qnoc architecture properties," *Models & Optimisation and Mathematical Analysis Journal*, vol. 1, no. 2, pp. 8–13, 2012. 11

[42] R. M. Burstall and J. A. Goguen, "The semantics of clear, a specification language," in *Abstract Software Specifications*, pp. 292–332, Springer, 1980. 11, 97, 106

[43] D. Sannella and M. Wirsing, "A kernel language for algebraic specification and implementation extended abstract," in *International Conference on Fundamentals of Computation Theory*, pp. 413–427, Springer, 1983. 11

[44] A. HARICHE, M. BELARBI, and A. CHOUARFIA, "Embedded systems design using event-b theories," *Int. J. Com. Dig. Sys*, vol. 5, no. 2, 2016. 11, 140, XXXI

[45] P. Castéran, V. Filou, and M. Mosbah, "Formal proofs of local computation systems," tech. rep., LaBRI - University of Bordeaux, 2009. 11

[46] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast, "Formal verification of fault tolerant noc-based architecture," in *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, 2012. 11

[47] A. Edmunds, M. Butler, I. Maamria, R. Silva, and C. Lovell, "Event-b code generation : type extension with theories," *Abstract State Machines, Alloy, B, VDM, and Z*, pp. 365–368, 2012. 11

[48] F. Maraninchi and P. Caspi, "La place de l'informatique dans l'enseignement des logiciels et systemes embarqués," tech. rep., Laboratoire Verimag, 2003. 14

[49] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE micro*, vol. 26, no. 2, pp. 10–24, 2006. 15

[50] J. Yoshida, "Texas instruments launches davinci platform," *EE Times*, 2005. 15

[51] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network : Built for speed," *IEEE micro*, vol. 26, no. 3, pp. 10–23, 2006. 15

[52] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A gals infrastructure for a massively parallel multiprocessor," *IEEE Design & Test of Computers*, vol. 24, no. 5, 2007. 15

[53] L. Benini and G. De Micheli, "Networks on chip : a new paradigm for systems on chip design," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pp. 418–419, IEEE, 2002. 15

[54] L. Benini and D. Bertozzi, "Network-on-chip architectures and design methods," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 261–272, 2005. 16

[55] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004. 16

[56] D. B. Gustavsion *et al.*, "Ieee standard for scalable coherent interface (sci)," *IEEE Computer Society, IEEE Std*, pp. 1596–1992. 16

[57] H. Hellwagner and A. Reinefeld, *SCI : Scalable Coherent Interface : architecture and software for high-performance compute clusters*. Springer Science & Business Media, 1999. 16

[58] W.-D. Weber, "Enabling reuse via an ip core-centric communications protocol : Open core protocoltm," *Proceedings of the IP*, pp. 20–22, 2000. 16

[59] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes : an infrastructure for low area overhead packet-switching networks on chip," *INTEGRATION, the VLSI journal*, vol. 38, no. 1, pp. 69–93, 2004. 16

[60] D. Bertozzi and L. Benini, "Xpipes : A network-on-chip architecture for gigascale systems-on-chip," *IEEE circuits and systems magazine*, vol. 4, no. 2, pp. 18–31, 2004. 16

[61] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pp. 117–124, IEEE, 2002. 16

[62] A. Andriahantenaina and A. Greiner, "Micro-network for soc : Implementation of a 32-port spin network," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, p. 11128, IEEE Computer Society, 2003. 16

[63] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon : a novel on-chip communication network," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, p. 15, IEEE, 2004. 16

[64] A. Banerjee, R. Mullins, and S. Moore, "A power and energy exploration of network-on-chip architectures," in *Proceedings of the First international Symposium on Networks-on-Chip*, pp. 163–172, IEEE Computer Society, 2007. 16

[65] J. Borel, "System on a chip (soc) and design methodology challenges," *Microelectronic engineering*, vol. 54, no. 1-2, pp. 15–22, 2000. 16

[66] E. I. Moreno, K. M. Popovici, N. L. V. Calazans, and A. A. Jerraya, "Integrating abstract noc models within mpsoc design," in *Rapid System Prototyping, 2008. RSP'08. The 19th IEEE/IFIP International Symposium on*, pp. 65–71, IEEE, 2008. 16

[67] W. R. Ashby, "Principles of the self-organizing dynamic system," *The Journal of general psychology*, vol. 37, no. 2, pp. 125–128, 1947. 16

[68] W. R. Ashby, "Principles of the self-organizing system," in *Facets of Systems Science*, pp. 521–536, Springer, 1991. 16

[69] G. Lendaris, "On the definition of self-organizing systems," *Proceedings of the IEEE*, vol. 52, no. 3, pp. 324–325, 1964. 17

[70] M. B. L. Dempster, "Aself-organizing systems perspective on planning for sustainability," tech. rep., University of Waterloo, 1998. 17

[71] F. Heylighen and C. Gershenson, "The meaning of self-organization in computing," *IEEE Intelligent Systems*, vol. 18, no. 4, 2003. 17

[72] T. De Wolf and T. Holvoet, "Emergence versus self-organisation : Different concepts but promising when combined," in *International Workshop on Engineering Self-Organising Applications*, pp. 1–15, Springer, 2004. 17

[73] C. Tanougast, *Méthodologie de partitionnement applicable aux systèmes sur puce à base de FPGA, pour l'implantation en reconfiguration dynamique d'algorithmes flot de données*. PhD thesis, Nancy 1, 2001. 17

[74] M. B. Gokhale, *Splash : A reconfigurable linear logic array*. Supercomputing Research Center, 1990. 17

[75] P. Bertin, *Mémoires actives programmables : conception, réalisation et programmation*. PhD thesis, Paris 7, 1993. 17

[76] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable active memories : reconfigurable systems come of age," in *Readings in hardware/software co-design*, pp. 611–624, Kluwer Academic Publishers, 2001. 17

[77] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2 : FPGAs in a custom computing machine*, vol. 9. Wiley-IEEE Computer Society Press, 1996. 17

[78] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," *IEE Proceedings-Computers and Digital Techniques*, vol. 153, no. 3, pp. 157–164, 2006. 17

[79] M. Hübner and J. Becker, "Exploiting dynamic and partial reconfiguration for fpgas : toolflow, architecture and system integration," in *Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pp. 1–4, ACM, 2006. 18

[80] L. Kessal, N. Abel, and D. Demigny, "03-traitement temps réel des images en exploitant la reconfiguration dynamique : architecture et programmation," 2006. 18

[81] H. Guermoud, Y. Berviller, E. Tisserand, and S. Weber, "Architecture à base de fpga reconfigurable dynamiquement dédiée au traitement d'image sur flot de données," in *16° Colloque sur le traitement du signal et des images, FRA, 1997*, GRETSI, Groupe d'Etudes du Traitement du Signal et des Images, 1997. 18

[82] M. Hiibner, C. Schuck, M. Kiihnle, and J. Becker, "New 2-dimensional partial dynamic reconfiguration techniques for real-time adaptive microelectronic circuits," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, pp. 6–pp, IEEE, 2006. 18

[83] H. Amano, "A survey on dynamically reconfigurable processors," *IEICE transactions on Communications*, vol. 89, no. 12, pp. 3179–3187, 2006. 18

[84] A. Ahmadinia, C. Bobda, and J. Teich, "A dynamic scheduling and placement algorithm for reconfigurable hardware," *Organic and Pervasive Computing–ARCS 2004*, pp. 443–465, 2004. 18

[85] F. Dittmann, "Reconfiguration time aware processing on fpgas," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006. 18

[86] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Design, Automation, and Test in Europe*, pp. 111–123, Springer, 2008. 19

[87] F. Karim, A. Nguyen, S. Dey, and R. Rao, "On-chip communication architecture for oc-768 network processors," in *Proceedings of the 38th annual Design Automation Conference*, pp. 678–683, ACM, 2001. 19

[88] F. Karim, A. Nguyen, and S. Dey, "An interconnect architecture for networking systems on chips," *IEEE micro*, vol. 22, no. 5, pp. 36–45, 2002. 19

[89] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage, "Guaranteeing the quality of services in networks on chip," in *Networks on chip*, pp. 61–82, Springer, 2003. 19

[90] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip," *IEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 5, pp. 294–302, 2003. 19

[91] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Qnoc : Qos architecture and design process for network on chip," *Journal of systems architecture*, vol. 50, no. 2, pp. 105–128, 2004. 19

[92] A. Lines, "Asynchronous interconnect for synchronous soc design," *IEEE Micro*, vol. 24, no. 1, pp. 32–41, 2004. 20

[93] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar, "An asynchronous router for multiple service levels networks on chip," in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, pp. 44–53, IEEE, 2005. 21

[94] O. Lysne, T. M. Pinkston, and J. Duato, "A methodology for developing dynamic network reconfiguration processes," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 77–86, IEEE, 2003. 21

[95] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 233–242, IEEE, 2007. 21

[96] M. B. Stensgaard and J. Sparsø, "Renoc : A network-on-chip architecture with reconfigurable topology," in *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pp. 55–64, IEEE, 2008. 21

[97] F. M. Vallina, N. Jachimiec, and J. Saniie, "Nova interconnect for dynamically reconfigurable noc systems," in *Electro/Information Technology, 2007 IEEE International Conference on*, pp. 546–550, IEEE, 2007. 21

[98] T. Pionteck, R. Koch, and C. Albrecht, "Applying partial reconfiguration to networks-on-chips," in *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pp. 1–6, IEEE, 2006. 21

[99] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "Dynoc : A dynamic infrastructure for communication in dynamically reconfugurable devices," in *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 153–158, IEEE, 2005. 21

[100] M. Majer, C. Bobda, A. Ahmadinia, and J. Teich, *Packet routing in dynamically changing networks on chip*. 21

[101] S. Jovanović, C. Tanougast, C. Bobda, and S. Weber, "Cunoc : A dynamic scalable communication structure for dynamically reconfigurable fpgas," *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 24–36, 2009. 21

[102] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda, "A new deadlock-free fault-tolerant routing algorithm for noc interconnections," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 326–331, IEEE, 2009. 21

[103] S. Jovanovic, C. Tanougast, S. Weber, and C. Bobda, "Cunoc : A scalable dynamic noc for dynamically reconfigurable fpgas," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 753–756, IEEE, 2007. 21

[104] S. Jovanovic, C. Tanougast, C. Bobda, and S. Weber, "A scalable dynamic infrastructure for dynamically reconfigurable systems," *Re-CoSoC07, Montpellier, France*, 2007. 21

[105] S. B. Furber, *ARM system-on-chip architecture*. pearson Education, 2000. 23

[106] P. R. Panda, N. D. Dutt, and A. Nicolau, *Memory issues in embedded systems-on-chip : optimizations and exploration*. Springer Science & Business Media, 1999. 23

[107] L. Benini and G. De Micheli, "Networks on chips : A new soc paradigm," *computer*, vol. 35, no. 1, pp. 70–78, 2002. 23

[108] S. Srikanteswara, J. H. Reed, P. Athanas, and R. Boyle, "A soft radio architecture for reconfigurable platforms," *IEEE Communications Magazine*, vol. 38, no. 2, pp. 140–147, 2000. 23

[109] I. XILINX, "Xilinx : The programmable logic data book," tech. rep., 2000. 23, 24

[110] S. Brown and J. Rose, "Fpga and cpld architectures : A tutorial," *IEEE design & test of computers*, vol. 13, no. 2, pp. 42–57, 1996. 24

[111] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou, "The systolic ring : A dynamically reconfigurable architecture for embedded systems," in *Field-Programmable Logic and Applications*, pp. 409–419, Springer, 2001. 24

[112] "Atmel 40k datasheet," tech. rep. 24

[113] "Altera databook," tech. rep. 24

[114] B. Haberman and M. Trakhtenbrot, "An undergraduate program in embedded systems engineering," in *Software Engineering Education & Training, 18th Conference on*, pp. 103–110, IEEE, 2005. 25

[115] P. P. Pande, C. Grecu, A. Ivanov, R. Saleh, and G. De Micheli, "Design, synthesis, and test of networks on chips," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 404–413, 2005. 25

[116] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006. 25

[117] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Proceedings of the 38th annual Design Automation Conference*, pp. 667–672, ACM, 2001. 25

[118] A. Jantsch, "Models of computation for networks on chip," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 165–178, IEEE, 2006. 25

[119] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, pp. 890–895, IEEE, 2004. 25

[120] S. F. Nielsen and J. Sparsø, "Analysis of low-power soc interconnection networks," in *IEEE 19th Norchip Conference*, pp. 77–86, 2001. 25

[121] C. Grecu, A. Ivanov, R. Saleh, E. S. Sogomonyan, and P. P. Pande, "On-line fault detection and location for noc interconnects," in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, pp. 6–pp, IEEE, 2006. 25

[122] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli, "Analysis of error recovery schemes for networks on chips," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 434–442, 2005. 25

[123] M. K. Schafer, T. Hollstein, H. Zimmer, and M. Glesner, "Deadlock-free routing and component placement for irregular mesh-based networks-on-chip," in *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pp. 238–245, IEEE, 2005. 25

[124] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu, "Deadlock prevention in the æthereal protocol," in *CHARME*, pp. 345–348, Springer, 2005. 25

[125] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing message-dependent deadlock free networks on chips for application-specific systems on chips," in *Very Large Scale Integration, 2006 IFIP International Conference on*, pp. 158–163, IEEE, 2006. 25

[126] K. Lahiri, A. Raghunathan, and S. Dey, "Evaluation of the traffic-performance characteristics of system-on-chip communication architectures," in *VLSI Design, 2001. Fourteenth International Conference on*, pp. 29–35, IEEE, 2001. 25

[127] A. S. Berger, *Embedded systems design : an introduction to processes, tools, and techniques*. Focal Press, 2002. 26

[128] T. Vallius and J. Roning, "Embedded object architecture," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pp. 102–107, IEEE, 2005. 26

[129] Q. Deng, H. Xu, S. Wei, Y. Han, and G. Yu, "An embedded sopc system using automation design," in *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pp. 232–239, IEEE, 2005. 26

[130] C. Talarico, A. Gupta, E. Peter, and J. W. Rozenblit, "Embedded system engineering using c/c++ based design methodologies," in *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pp. 81–88, IEEE, 2005. 27

[131] A. Fraboulet, *Optimisation de la mémoire et de la consommation des systèmes multimédia embarqués*. PhD thesis, Villeurbanne, INSA, 2001. 27

[132] A. Mignotte, *Compilation sur silicium ou conception conjointe matérielle logicielle*. PhD thesis, Université Claude Bernard de Lyon, 1999. 27

[133] G. erard Berry, "The foundations of esterel," 1998. 27

[134] H. Ishikawa and T. Nakajima, "Earlgray : a component-based java virtual machine for embedded systems," in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pp. 403–409, IEEE, 2005. 29

[135] A. Courbot, M. Pavlova, G. Grimaud, and J.-J. Vandewalle, "A low-footprint java-to-native compilation scheme using formal methods," in *CARDIS*, pp. 329–344, Springer, 2006. 29

[136] E. A. Lee, "What's ahead for embedded software ?," *Computer*, vol. 33, no. 9, pp. 18–26, 2000. 29

[137] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified modeling language reference manual, the*. Pearson Higher Education, 2004. 29

[138] G. Booch, *The unified modeling language user guide*. Pearson Education India, 2005. 29

[139] H. Espinoza, J. Medina, H. Dubois, S. Gérard, and F. Terrier, "Towards a uml-based modelling standard for schedulability analysis of real-time systems," in *MARTES Workshop at MODELS Conference*, pp. 79–90, 2006. 29

[140] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems : Formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997. 29

[141] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy : A framework for simulating and prototyping heterogeneous systems," 1994. 30

[142] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE micro*, vol. 14, no. 4, pp. 26–36, 1994. 30

[143] P. H. Feiler, D. P. Gluch, J. J. Hudak, and B. A. Lewis, "Pattern-based analysis of an embedded real-time system architecture.," in *IFIP-WADL*, pp. 51–65, 2004. 30

[144] R. Allen and D. Garlan, "The wright architectural specification language," *Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science*, 1996. 30

[145] D. Luckham, "Rapide : A language and toolset for simulation of distributed systems by partial orderings of events," 1996. 30

[146] D. Garlan, R. Monroe, and D. Wile, "Acme : An architecture description interchange language," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, p. 7, IBM Press, 1997. 30

[147] G. Zelesnik, "The unicon language reference manual," *School of Computer Science Carnegie Mellon, Pittsburgh, Pennsylvania (May 1996)*, 1996. 30

[148] S. Vestal, "Software programmer's manual for the honeywell aerospace compiled kernel (metah language reference manual)," *Honewell Systems and Research Center, Technical Report*, 1993. 30

[149] S. I. A. S. D. A. A.-C. Subcommittee, "Avionics Architecture Description Language Standard. SAE Document AS 5506." http://www.sae.org, 2004. [Online ; Consulté en Nov. 2017]. 30

[150] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya, "Scalable and flexible cosimulation of soc designs with heterogeneous multi-processor target architectures," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pp. 63–68, ACM, 2001. 31

[151] P. Feautrier, "Les compilateurs," *TSI. Technique et science informatiques*, vol. 19, no. 1-3, pp. 223–232, 2000. 31

[152] S. Derrien, A.-C. Guillou, P. Quinton, T. Risset, and C. Wagner, "Automatic synthesis of efficient interfaces for compiled regular architectures," in *Proc. Int. Samos Workshop Systems, Architectures, Modeling and Simulation*, pp. 127–150, 2003. 31

[153] T. Risset, "Contribution à la compilation de nids de boucles sur silicium," 2000. 31

[154] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International journal of parallel programming*, vol. 28, no. 5, pp. 469–498, 2000. 31

[155] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, pp. 79–103, Springer, 1996. 31

[156] F. Rastello, *Partitionnement : optimisations de compilation et algorithmique hétérogene*. PhD thesis, Lyon, École normale supérieure (sciences), 2000. 31

[157] C. Kern and M. R. Greenstreet, "Formal verification in hardware design : a survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999. 31

[158] T. Kropf, *Introduction to formal hardware verification*. Springer Science & Business Media, 2013. 31

[159] N. AFNOR, "Iso/afnor.," *Dictionnaire de génie logiciel.*, 1997. 32, 43

[160] M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot, and M. Lemoine, "Précis de génie logiciel," 1996. 32

[161] C. Choppy, *Spécifications algébriques : Prototypage et validation*. PhD thesis, Habilitation à diriger des recherches—Université Paris-Sud, 1994. 32

[162] J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods," *Computer*, vol. 28, no. 4, pp. 56–63, 1995. 32

[163] J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods... ten years later," *Computer*, vol. 39, no. 1, pp. 40–48, 2006. 32

[164] J. P. Bowen and M. G. Hinchey, "Seven more myths of formal methods," *IEEE software*, vol. 12, no. 4, pp. 34–41, 1995. 32

[165] E. M. Clarke and J. M. Wing, "Formal methods : State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996. 32

[166] J.-F. Monin, *Comprendre les méthodes formelles : panorama et outils logiques*. Masson, 1996. 32, 124

[167] H. Saiedian, "An invitation to formal methods," *Computer*, vol. 29, no. 4, pp. 16–17, 1996. 32

[168] D. Hoffman and P. Stooper, *Fundamentals of Software Design and Verification*. McGraw Hill, 1994. 32

[169] D. H. Craigen, S. L. Gerhart, and T. J. Ralston, "An international survey of industrial applications of formal methods. volume 2. case studies," tech. rep., NAVAL RESEARCH LAB WASHINGTON DC, 1993. 33

[170] A. Hall, "Seven myths of formal methods," *IEEE software*, vol. 7, no. 5, pp. 11–19, 1990. 33, 105, 106

[171] N. Dellsie and D. Gartan, "A formal specification of an oscilloscope," *IEEE software*, vol. 7, no. 5, pp. 29–36, 1990. 33

[172] I. Houston and S. King, "Cics project report experiences and results from the use of z in ibm," in *VDM'91 Formal Software Development Methods*, pp. 588–596, Springer, 1991. 33, 64

[173] S. Prehn, *VDM'91. Formal Software Development Methods. 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 21-25, 1991. Proceedings : Volume 1 : Conference Contributions*, vol. 1. Springer Science & Business Media, 1991. 33, 124

[174] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, and D. L. Parnas, "Software requirements for the a-7e aircraft.," tech. rep., NAVAL RESEARCH LAB WASHINGTON DC, 1992. 33

[175] D. L. Parnas, "Some theorems we should prove," in *Higher Order Logic Theorem Proving and Its Applications*, pp. 155–162, Springer, 1994. 33

[176] G. Barrett, "Formal methods applied to a floating-point number system," *IEEE transactions on software engineering*, vol. 15, no. 5, pp. 611–621, 1989. 33

[177] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen, "The b-method," in *International Symposium of VDM Europe*, pp. 398–405, Springer, 1991. 33, 43

[178] P. Behm, P. Desforges, and J.-M. Meynadier, "Météor : An industrial success in formal development," in *International Conference of B Users*, pp. 26–26, Springer, 1998. 33

[179] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models : Application to event-b," *Fundamenta Informaticae*, vol. 77, no. 1-2, pp. 1–28, 2007. 34

[180] F. Bellegarde, J. Julliand, and O. Kouchnarenko, "Synchronized parallel composition of event systems in b," in *International Conference of B and Z Users*, pp. 436–457, Springer, 2002. 34

[181] S. Merz, "Model checking techniqes for the analysis of reactive systems," *Synthese*, vol. 133, no. 1-2, pp. 173–201, 2002. 34

[182] J. Rushby, "Theorem proving for verification," in *Summer School on Modeling and Verification of Parallel Processes,MOVEP 2000*, pp. 39–57, Springer, 2000. 34

[183] L. Lamport, *Specifying systems : the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002. 34, 41, 82

[184] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International journal on software tools for technology transfer*, vol. 1, no. 1-2, pp. 134–152, 1997. 34

[185] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997. 34

[186] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994. 34

[187] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos : A model-checking tool for real-time systems," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 298–302, Springer, 1998. 34

[188] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech : A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997. 34

[189] B. Barras, S. Boutin, C. Cornes, J. Courant, B. Barras, S. Boutin, C. Cornes, J. Courant, S. Conchon, S. Conchon, *et al.*, "The {Coq Proof Assistant Reference Manual} version 6.2," *Computers and Mathematics with Applications*, vol. 8, pp. 79–94, 2007. 34

[190] J. Harrison, "Hol light tutorial (for version 2.20)," *Intel JF1-13, Section*, vol. 18, 2006. 34

[191] S. Owre, J. M. Rushby, and N. Shankar, "Pvs : A prototype verification system," in *International Conference on Automated Deduction*, pp. 748–752, Springer, 1992. 34, 38, 124, 132

[192] A.-e.-P. ClearSy, "Atelier b, 2002." 34, 61

[193] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*, pp. 575–607, Springer, 2002. 36

[194] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections : an effective verification process," *IEEE software*, vol. 6, no. 3, pp. 31–36, 1989. 36

[195] M. E. Fagan, "Advances in software inspections," in *Pioneers and Their Contributions to Software Engineering*, pp. 335–360, Springer, 2001. 36

[196] M. Bush, "Improving software quality : The use of formal inspections at the jet propulsion laboratory," in *Software Engineering, 1990. Proceedings., 12th International Conference on*, pp. 196–199, IEEE, 1990. 36

[197] E. F. Weller, "Lessons from three years of inspection data (software development)," *IEEE software*, vol. 10, no. 5, pp. 38–45, 1993. 36

[198] M. Dyer, *The cleanroom approach to quality software development*. John Wiley & Sons, Inc., 1992. 37

[199] G. M. Schneider, J. Martin, and W.-T. Tsai, "An experimental study of fault detection in user requirements documents," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 2, pp. 188–204, 1992. 37

[200] H. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, pp. 19–25, sep 1987. 37

[201] K. L. Heninger, "Specifying software requirements for complex systems : New techniques and their application," *IEEE Transactions on Software Engineering*, no. 1, pp. 2–13, 1980. 37

[202] P. Henderson, *Object-oriented Specification and Design with C++*. McGraw-Hill, 1993. 37

[203] A. J. Van Schouwen, "The a-7 requirements model : Re-examination for real-time systems and an application to monitoring systems.," tech. rep., Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1992. 37

[204] A. J. van Schouwen, D. L. Parnas, and J. Madey, "Documentation of requirements for computer systems," in *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pp. 198–207, IEEE, 1993. 37

[205] C. B. Jones, *Systematic software development using VDM*, vol. 2. Prentice Hall Englewood Cliffs, 1990. 37, 126

[206] H. Alexander and V. Jones, *Software Design and Prototyping using me too*. Prentice-Hall, Inc., 1989. 38

[207] S. Hekmatpour and D. C. Ince, *Software prototyping, formal methods and VDM*. Addison-Wesley, 1988. 38

[208] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of obj2," in *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 52–66, ACM, 1985. 38

[209] I. J. Hayes and C. B. Jones, "Specifications are not (necessarily) executable," *Software Engineering Journal*, vol. 4, no. 6, pp. 330–339, 1989. 38

[210] R. S. Boyer and J. S. Moore, *A computational logic handbook : Formerly notes and reports in computer science and applied mathematics*. Elsevier, 2014. 38

[211] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink, "Eves : An overview," in *International Symposium of VDM Europe*, pp. 389–405, Springer, 1991. 38

[212] J. Cook, I. FILIPPENKO, B. LEVY, L. MARCUS, and T. MENAS, "Formal computer verification in the state delta verification system (sdvs)," in *8th Computing in Aerospace Conference*, p. 3715, 1991. 38

[213] M. J. Gordon and T. F. Melham, "Introduction to hol a theorem proving environment for higher order logic," tech. rep., Cambridge University Press, Cambridge, UK, 1993. 38

[214] G. Leduc and F. Germeau, "Verification of security protocols using lotos-method and application," *Computer Communications*, vol. 23, no. 12, pp. 1089–1103, 2000. 39

[215] T. Worm, "Using metapatterns with sdl," in *SDL'99*, pp. 355–372, Elsevier, 1999. 39

[216] D. Harel, "On the formal semantics of statecharts," in *IEEE Symposium on Logic in Computer Science, 1987*, pp. 54–64, 1987. 39

[217] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate : A working environment for the development of complex reactive systems," *IEEE Transactions on software engineering*, vol. 16, no. 4, pp. 403–414, 1990. 39

[218] D. Harel and M. Politi, "The languages of statemate," tech. rep., Technical Report, I-Logix, Inc., Andover, MA (250 pp.), 1991. 39

[219] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994. 39

[220] G. Gössler and J. Sifakis, "Composition for component-based modeling," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 161–183, 2005. 39

[221] "Research topics." [Online ; Consulté en Nov. 2017]. 40

[222] M. Poulhies, J. Pulou, C. Rippert, and J. Sifakis, "A methodology and supporting tools for the development of component-based embedded systems," in *Monterey Workshop*, pp. 75–96, Springer, 2006. 40

[223] A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis, "Using bip for modeling and verification of networked systems–a case study on tinyos-based networks," in *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, pp. 257–260, IEEE, 2007. 40

[224] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis, "Incremental component-based construction and verification of a robotic system.," in *ECAI*, vol. 178, pp. 631–635, 2008. 40

[225] "Deliverable : D 2.2 : concept for partitioning, mapping, and tracing for multi- and many-core systems," Aug 2016. [Online ; Consulté en Nov. 2017]. 40

[226] "Openembedd platform." [Online ; Consulté en Nov. 2017]. 40

[227] G. Berry and G. Gonthier, "The esterel synchronous programming language : Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992. 40

[228] G. Berry, "The foundations of esterel.," in *Proof, language, and interaction*, pp. 425–454, 2000. 40

[229] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre : A declarative language for programming synchronous systems," in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, vol. 178, p. 188, 1987. 40

[230] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations : the signal language and its semantics," *Science of computer programming*, vol. 16, no. 2, pp. 103–149, 1991. 40

[231] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems : a survey of current trends," in *Current trends in Concurrency*, pp. 510–584, Springer, 1986. 41

[232] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994. 41

[233] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *25 Years of Model Checking*, pp. 196–215, Springer, 2008. 41

[234] E. Allen Emerson and J. Halpern, "Sometimes" and "not never" revisited : On branching times versus linear time," *Journal of the ACM*, vol. 33, pp. 151–178, 1986. 41

[235] S. Owre, N. Shankar, and J. Rushby, "User guide for the pvs specification and verification system (beta release)," *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1993. 42

[236] G. Barthe, J. Forest, D. Pichardie, and V. Rusu, "Defining and reasoning about recursive functions : a practical tool for the coq proof assistant," in *International Symposium on Functional and Logic Programming*, pp. 114–129, Springer, 2006. 42

[237] S. Merz, "An encoding of tla in isabelle," *Available in Isabelle distribution*, 1999. 42

[238] P. De Groote and S. Salvati, "Higher-order matching in the linear λ-calculus with pairing," in *International Workshop on Computer Science Logic*, pp. 220–234, Springer, 2004. 42

[239] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL : a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002. 42, 45, 46, 105, 125

[240] P. De Groote, "Partially commutative linear logic : sequent calculus and phase semantics," in *Third Roma Workshop : Proofs and Linguistics Categories–Applications of Logic to the analysis and implementation of Natural Language*, pp. 199–208, 1996. 42

[241] B. Rossman, D. Rosenzweig, Y. Gurevich, and A. Blass, "Interactive small-step algorithms ii : Abstract state machines and the characterization theorem," *Logical Methods in Computer Science*, vol. 3, 2007. 43

[242] D. Bjørner and C. B. Jones, "He vienna development method : The meta-language," *Lecture Notes in Computer Science 61*, 1978. 43

[243] A. JR, *Specication language*, p. 342–410. CUP Archive (Cambridge University Press), 1980. 43, 60

[244] J.-R. Abrial, "B# : Toward a synthesis between z and b," in *International Conference of B and Z Users*, pp. 168–177, Springer, 2003. 43

[245] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, "Meteor : A successful application of b in a large project," in *International Symposium on Formal Methods*, pp. 369–387, Springer, 1999. 43, 65

[246] L. Casset, "Development of an embedded verifier for java card byte code using formal methods," in *International Symposium of Formal Methods Europe*, pp. 290–309, Springer, 2002. 43

[247] G. Pouzancre, "How to diagnose a modern car with a formal b model ?," in *International Conference of B and Z Users*, pp. 98–100, Springer, 2003. 43

[248] G. Pouzancre and J.-P. Pitzalis, "Modélisation en b évenementiel des fonctions mécaniques, électriques et informatiques d'un véhicule," *TSI. Technique et science informatiques*, vol. 22, no. 1, pp. 119–128, 2003. 43

[249] S. Hallerstede, "Parallel hardware design in b," in *International Conference of B and Z Users*, pp. 101–102, Springer, 2003. 43

[250] J.-R. Abrial, "Extending b without changing it (for developing distributed systems)," in *1st Conference on the B method*, vol. 11, 1996. 43

[251] R.-J. Back and K. Sere, "Stepwise refinement of action systems.," *Struct. Program.*, vol. 12, no. 1, pp. 17–30, 1991. 43, 60

[252] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975. 43

[253] S. Nakajima, "A refinement planning sheet,," in *Rodin Workshop User and Developer*, (University of Duesseldorf, Germany), 2010. 44

[254] B. Bicknell, K. Kanso, N. Rampton, and D. McLeod, "Advance in smart grids," tech. rep., In ADVANCE Industry Day 2014, House of the University,Duesseldorf, Germany, 2014. 44

[255] F. Mejia, M.-T. Khuu, and M. Leuschel, "Wp1 : Railway case study," tech. rep., In ADVANCE Industry Day 2014, House of the University,Duesseldorf, Germany, 2014. 44

[256] F. Garillot and B. Werner, "Simple types in type theory : Deep and shallow encodings," in *Theorem Proving in Higher Order Logics*, pp. 368–382, Springer, 2007. 45

[257] R. Milner, *The definition of standard ML : revised*. MIT press, 1997. 46, 125

[258] A. Matoussi, *Construction de spécifications formelles abstraites dirigée par les buts*. PhD thesis, Université Paris-Est, 2011. 59, 64, 65

[259] J. Misra, "Parallel program design : a foundation," 1988. 60

[260] S. Hallerstede, "Justifications for the event-b modelling notation," in *International Conference of B Users*, pp. 49–63, Springer, 2007. 62, 63, 68, 69, 70, 128, 170

[261] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for event-b," in *International Conference on Formal Engineering Methods*, pp. 588–605, Springer, 2006. 66, 68, 70, 131, 170

[262] E. Foundation., "Eclipse platform," 2011. [Consulté en nov. 2016]. 66

[263] D. L. Detlefs, "An overview of the extended static checking system," in *Proceedings of the First Workshop on Formal Methods in Software Practice*, pp. 1–9, Citeseer, 1996. 66

[264] B. Atelier, "the industrial tool to efficiently deploy the b method," *URL : http ://www. atelierb. eu/index-en. php (access date 22.03. 2015)*, 2008. 69, 75

[265] M. Schmalz, "Export to isabelle," 2009. [Consulté en nov. 2016]. 69, 127

[266] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Supporting reuse in event b development : modularisation approach," in *International Conference on Abstract State Machines, Alloy, B and Z*, pp. 174–188, Springer, 2010. 70

[267] C. Snook, "Event-b records extension," *URL : http ://wiki.event-b.org/index.php/Records_-Extension. php, University of Southampton, UK*, 2009. 70

[268] F. Systerel, "Rodin 2.0 release notes," 2010. 70

[269] R. Milner, "Logic for computable functions description of a machine implementation," tech. rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972. 73

[270] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of jml tools and applications," *International journal on software tools for technology transfer*, vol. 7, no. 3, pp. 212–232, 2005. 74, 126

[271] DEPLOY, "Rodin : le support officiel," 2010. 82

[272] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia, "La validation de modèles event-b avec le plug-in prob pour rodin," *TSI*, pp. 1065–1084, 2008. 82

[273] R. Gorrieri and A. Rensink, "Action refinement," in *Handbook of process algebra*, pp. 1047–1147, Elsevier, 2001. 86

[274] A. Egyed, N. Mehta, and N. Medvidovic, "Software connectors and refinement in family architectures," in *International Workshop on Software Architectures for Product Families*, pp. 96–106, Springer, 2000. 86

[275] A. Kerschbaumer, "Non-refinement transformation of software architectures," in *Proceedings of the ZB2002 International Workshop on Refinement of Critical Systems : Methods, Tools and Experience, Grenoble*, 2002. 86

[276] T. Vos, S. Swierstra, and W. Prasetya, "Yet another program refinement relation," in *RCS'02 International Workshop on Refinement of Critical Systems : Methods, Tools and Experience, IMAG, Grenoble*, 2002. 86

[277] J. Magee and J. Kramer, "Darwin : An architectural description language," tech. rep., Specifying Distributed Software Architectures, Department of Computing, Imperial College, Londres, 1998. 87

[278] "Darwin language, version 3d," 19987. 87

[279] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992. 89

[280] P. Bon, *Du cahier des charges aux spécifications formelles : une méthode basée sur les réseaux de Petri de haut niveau*. PhD thesis, Lille 1, 2000. 92

[281] R. M. Kamenoff, E. Meyer, N. Lévy, and J. Souquières, "Utilisation de patterns dans la construction de specifications en uml et b," in *Approches Formelles dans l'Assistance au Developpement de Logiciels-AFADl'2000*, pp. 26–28, 2000. 92

[282] R. Sanlaville, "Description d'architecture logicielles : Utilisation du formalisme wright pour l'interconnexion de machines abstraites b. report for dea d'informatique : Systems et communications," *DEA, LSR, IMAG, Grenoble*, 1997. 92

[283] J.-R. Abrial, "Event driven sequential program construction," *Approches Formelles dans l'Assistance au Développement des Logiciels (AFADL 2001), Nancy*, 2001. 92

[284] T. Lecomte, "Event driven b : methodology, language, tool support and experiments," in *Workshop on Refinement of Critical Systems*, 2002. 92

[285] R. M. Burstall and J. A. Goguen, "Putting theories together to make specifications," in *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2*, pp. 1045–1058, Morgan Kaufmann Publishers Inc., 1977. 95, 96, 100

[286] R. M. Burstall and J. A. Goguen, *The semantics of Clear, a specification language*. 1980. 95, 96, 98, 99, 100

[287] M. A. Arbib and E. G. Manes, *Arrows, structures, and functors : The categorical imperative*, vol. 101. Academic Press New York, 1975. 96

[288] H. Herrlich and G. Strecker, "Category theory," 1973. 96

[289] B. Liskov and S. Zilles, "Specification techniques for data abstractions," in *ACM SIGPLAN Notices*, vol. 10, pp. 72–87, ACM, 1975. 96

[290] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta informatica*, vol. 10, no. 1, pp. 27–52, 1978. 96

[291] J. Goguen, J. Thatcher, and E. Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types, ibm res," *Rep. RC6487, Yorktown Heights*, vol. 156, 1976. 96, 100

[292] D. J. Lehmann and M. B. Smyth, "Data types," tech. rep., Department of Computer Science,University of Warwick,England, 1977. 96

[293] J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Data type specification : Parameterization and the power of specification techniques," in *Proceedings of the tenth annual ACM symposium on Theory of computing*, pp. 119–132, ACM, 1978. 96

[294] H. Ehrig, H.-J. Kreowski, and P. Padawitz, "Stepwise specification and implementation of abstract data types," in *International Colloquium on Automata, Languages, and Programming*, pp. 205–226, Springer, 1978. 96

[295] H. Ehrich and V. Lohberger, "Parametric specification of abstract data types, parameter substitution and graph replacements," in *Proc. of the Workshop 'Graphentheoretische Konzepte in der Informatik', Applied Computer Science, Carl Hanser Verlag, Munich-Vienna*, 1978. 96

[296] M. Honda and R. Nakajima, "Interactive theorem proving on hierarchically and modularly structured set of very many axioms," in *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 1*, pp. 400–402, Morgan Kaufmann Publishers Inc., 1979. 96

[297] L. Robinson, K. Levitt, and B. Silverberg, "The hdm handbook, vol. i-iii," *SRI International*, 1979. 96

[298] J.-R. Abrial, *Specification Language Z : Basic Library*. Oxford University Computing Laboratory, 1980. 96

[299] H. Genrich, *The Petri net representation of mathematical knowledge*. GMD, Ges. für Math. und Datenverarb., 1976. 96

[300] M. Caplain, *Langage de specifications*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1978. 96

[301] O.-J. Dahl, *Can Program Proving be Made Practical ? : Lectures Presented at the Eec-crest Course on Programming Foundations, Toulouse 1977.(rev. May 1978)*. Universitetet i Oslo. Institute of Informatics, 1978. 96

[302] J. C. Reynolds, "Reasoning about arrays," *Communications of the ACM*, vol. 22, no. 5, pp. 290–299, 1979. 96

[303] B. Liskov and V. A. Bērzinš, *An appraisal of program specifications*. Massachusetts Institute of Technology, Laboratory for Computer Science, 1977. 96

[304] P. D. Mosses, "Making denotational semantics less concrete," in *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, no. 41, pp. 102–109, 1977. 96

[305] F. W. Lawvere, "Functorial semantics of algebraic theories," *Proceedings of the National Academy of Sciences*, vol. 50, no. 5, pp. 869–872, 1963. 98, 100

[306] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, *An introduction to categories, algebraic theories and algebras*. IBM Thomas J. Watson Research Division, 1975. 100

[307] R. M. Burstall and D. Rydeheard, "The free algebraic theory on a signature (unpublished draft report)," tech. rep., 1979, institution=Dept. of Computer Science, University of Edinburgh. 100

[308] R. Godement, *Théorie des faisceaux*. Hermann, 1964. 100

[309] S. Eilenberg, J. C. Moore, *et al.*, "Adjoint functors and triples," *Illinois Journal of Mathematics*, vol. 9, no. 3, pp. 381–398, 1965. 100

[310] E. G. Manes, *Algebraic theories*, vol. 26. Springer Science & Business Media, 2012. 100

[311] D. Sannella, "Algebraic specification and program development by stepwise refinement," in *International Workshop on Logic Programming Synthesis and Transformation*, pp. 1–9, Springer, 1999. 102

[312] C. Métayer and L. Voisin, "The event-b mathematical language," *Systerel, March*, 2009. 102

[313] K. Robinson, "Reconciling axiomatic and model-based specifications reprised," in *International Conference on Abstract State Machines, B and Z*, pp. 223–236, Springer, 2008. 105, 128

[314] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada, "Maude : Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002. 105, 124, 125

[315] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler, "An introduction to obj 3," in *International Workshop on Conditional Term Rewriting Systems*, pp. 258–263, Springer, 1987. 105, 125

[316] S. Owre, N. Shankar, J. M. Rushby, and D. W. Stringer-Calvert, "Pvs language reference," *Computer Science Laboratory, SRI International, Menlo Park, CA*, vol. 1, no. 2, p. 21, 1999. 105, 125, 126

[317] N. Volker and F. Hagen, "On the representation of datatypes in isabelle/hol," in *First Isabelle Users Workshop*, Citeseer, 1995. 105

[318] J. Loeckx, H.-D. Ehrich, and M. Wolf, *Specification of abstract data types*. John Wiley & Sons, Inc., 1997. 105

[319] D. Sannella and M. Wirsing, "A kernel language for algebraic specification and implementation extended abstract," in *International Conference on Fundamentals of Computation Theory*, pp. 413–427, Springer, 1983. 106

[320] G. Dos Reis and J. Järvi, "What is generic programming ?," *Library-Centric Software Design (LCSD'05)*, p. 1, 2005. 106

[321] J. Gibbons and R. Paterson, "Parametric datatype-genericity," in *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pp. 85–93, ACM, 2009. 106

[322] A. S. Fathabadi, M. Butler, and A. Rezazadeh, "A systematic approach to atomicity decomposition in event-b," in *International Conference on Software Engineering and Formal Methods*, pp. 78–93, Springer, 2012. 108

[323] J.-R. Abrial, "Formal methods : Theory becoming practice.," *J. UCS*, vol. 13, no. 5, pp. 619–628, 2007. 124

[324] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods : Practice and experience," *ACM computing surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009. 124

[325] L. C. Paulson *et al.*, "The isabelle reference manual," tech. rep., Citeseer, 1993. 124

[326] N. Martí-Oliet and J. Meseguer, "Rewriting logic as a logical and semantic framework," in *Handbook of Philosophical Logic*, pp. 1–87, Springer, 2002. 125

[327] D. Sannella, *Formal program development in Extended ML for the working programmer*. University of Edinburgh, Department of Computer Science. Laboratory for Foundations of Computer Science, 1989. 125

[328] S. Kahrs, D. Sannella, and A. Tarlecki, "The definition of extended ml : A gentle introduction," *Theoretical Computer Science*, vol. 173, no. 2, pp. 445–484, 1997. 125

[329] J.-R. Abrial, D. Cansell, and G. Laffitte, ""higher-order" mathematics in b," in *International Conference of B and Z Users*, pp. 370–393, Springer, 2002. 125

[330] J. Harrison, "Metatheory and reflection in theorem proving : A survey and critique," tech. rep., Citeseer, 1995. 125, 126

[331] M. Gordon, "Hol-a machine oriented formulation of higher order logic," 2001. 125

[332] A. Armando, A. Cimatti, and L. Viganò, "Building and executing proof strategies in a formal metatheory," in *Congress of the Italian Association for Artificial Intelligence*, pp. 11–22, Springer, 1993. 126

[333] D. Griffioen and M. Huisman, "A comparison of pvs and isabelle/hol," in *International Conference on Theorem Proving in Higher Order Logics*, pp. 123–142, Springer, 1998. 126

[334] C. B. Jones, K. D. Jones, P. Lindsay, and R. Moore, *Mural : a formal development support system*. Springer Science & Business Media, 2012. 126

[335] D. R. Cok and J. R. Kiniry, "Esc/java2 : Uniting esc/java and jml," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 108–128, Springer, 2004. 126

[336] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie : A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, pp. 364–387, Springer, 2005. 126

[337] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system : An overview," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pp. 49–69, Springer, 2004. 126

[338] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc : A practical system for verifying concurrent c," in *International Conference on Theorem Proving in Higher Order Logics*, pp. 23–42, Springer, 2009. 126

[339] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal system development with kiv," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 363–366, Springer, 2000. 126

[340] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, *et al.*, "The key tool," *Software & Systems Modeling*, vol. 4, no. 1, pp. 32–54, 2005. 126

[341] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt, "The key system : Integrating object-oriented design and formal methods," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 327–330, Springer, 2002. 126

[342] M. Giese, "Taclets and the key prover," *Electronic Notes in Theoretical Computer Science*, vol. 103, pp. 67–79, 2004. 126

[343] L. C. Paulson, "Isabelle : The next seven hundred theorem provers," in *International Conference on Automated Deduction*, pp. 772–773, Springer, 1988. 127

[344] G. Luttgen, C. Munoz, R. Butler, B. DiVito, and P. Miner, "Towards a customizable pvs," tech. rep., INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING HAMPTON VA, 2000. 127

[345] F. Systerel, "Smt solvers plug-in," 2011. Consulté en nov. 2016. 127

[346] L. C. Paulson, "The foundation of a generic theorem prover," *Journal of Automated Reasoning*, vol. 5, no. 3, pp. 363–397, 1989. 128, 129

[347] D. Benanav, D. Kapur, and P. Narendran, "Complexity of matching problems," *Journal of symbolic computation*, vol. 3, no. 1-2, pp. 203–216, 1987. 139

[348] S. Eker, "Associative-commutative matching via bipartite graph matching," *The Computer Journal*, vol. 38, no. 5, pp. 381–399, 1995. 139

[349] S. Eker, "Associative-commutative rewriting on large terms," in *International Conference on Rewriting Techniques and Applications*, pp. 14–29, Springer, 2003. 139

[350] M. Tounsi, A. H. Kacem, M. Mosbah, and D. Méry, "A refinement approach for proving distributed algorithms : Examples of spanning tree problems," *Integration of Model based Formal Methods and Tools (IMFMT 2009), Düsseldorf Allemagne*, vol. 2, 2009. 140

[351] M. Heil, C. Tanougast, C. Killian, and A. Dandache, "Self-organized reliability suitable for wireless networked mpsoc," in *Microelectronics (ICM), 2013 25th International Conference on*, pp. 1–4, IEEE, 2013. 141

[352] DEPLOY, "Deploy project rodin tools documentation," 2014. [Consulté en nov. 2016]. 142

[353] DEPLOY, "Deploy project rodin theory plug-in," 2014. [Consulté en nov. 2016]. 142, XXXI

[354] R. van Engelen, "Code generation techniques for developing light-weight xml web services for embedded devices," in *Proceedings of the 2004 ACM symposium on Applied computing*, pp. 854–861, ACM, 2004. 163

[355] A. Edmunds, M. Butler, I. Maamria, R. Silva, and C. Lovell, "Event-b code generation : type extension with theories," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pp. 365–368, Springer, 2012. 167, XXXII

[356] DEPLOY, "Deploy : Industrial deployment of system engineering methods providing high dependability and productivity," 2008. 167, XXXII

[357] K. Megzari, *Refiner : Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture*. PhD thesis, Université Savoie Mont Blanc, 2004. 175

[358] S. Ostroumov, L. Tsiopoulos, K. Sere, and J. Plosila, "Generation of structural vhdl code with library components from formal event-b models," in *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 111–118, IEEE, 2013. 175

Annexe A

Notation de l'Event-B et notions de bases

(a) ***Notation de l'Event-B*** les notation de l'Event-B sont présentés respectivement(Voir les tables suivantes) :

i. *Symbole logique*,

Symbole logique	**symbôle**	**Signification**
$\forall$	!	quantificateur 'pour tout'
$\exists$	#	quantificateur 'il existe'
.	.	point qui suit les quantificateurs
λ	%	définition de fonction
$\neg$	*not*	symbole négation logique
$\wedge$	&	symbole et logique
$\vee$	*or*	symbole ou logique
$\Rightarrow$	=>	symbole implique logique
$\Leftrightarrow$	<=>	symbole equivalent logique
$=$	$=$	égalité de deux valeurs
$\neq$	$/=$	non égalié de deux valeurs
$btrue$		prédicat constant "vrai"
$bfalse$		prédicat constant "faux"

ii. *Constructeurs d'ensembles*,

Constructeurs d'ensembles	**symbôle**	**Signification**
$\emptyset$	{}	ensemble vide
$\times$	*	produit cartésien
$\mapsto$	|->	maplet (doublet de valeurs)
$\mathbb{P}$	POW	parties d'un ensemble
$\mathbb{P}_1$	$POW1$	parties non vides d'un ensemble
$\mathbb{F}$	FIN	parties finies d'un ensemble
$\mathbb{F}_1$	$FIN1$	parties finies non vides
..	..	intervalle

iii. *Relations et fonctions*,

iv. *Prédicats sur les ensembles*,

v. *Opérations sur les ensembles*,

vi. *Opérations sur les relations et les fonctions*,

vii. *Ensembles et constantes arithmétiques prédéfinis*,

Relations et fonctions	**symbôle**	**Signification**
$\leftrightarrow$	$<->$	ensemble des relations
$\rightarrow$	$-->$	ensemble des fonctions totales
$\nrightarrow$	$+->$	$\cdots$ fonctions partielles
$\rightarrowtail$	$>->$	$\cdots$ fonctions injectives totales
$\nrightarrowtail$	$>+>$	$\cdots$ fonctions injectives partielles
$\twoheadrightarrow$	$-->>$	$\cdots$ fonctions surjectives totales
$\nrightarrow\!\!\!\!\twoheadrightarrow$	$+->>$	$\cdots$ fonctions surjectives partielles
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$>->>$	$\cdots$ fonctions bijectives totales
$\nrightarrowtail\!\!\!\!\twoheadrightarrow$	$>+>>$	$\cdots$ fonctions bijectives partielles

Prédicats sur les ensembles	**symbôle**	**Signification**
$\in$	$:$	appartient
$\notin$	$/:$	n'appartient pas
$\subseteq$	$<:$	est inclus dans
$\not\subseteq$	$/<:$	n'est pas inclus dans
$\subset$	$<<:$	est strictement inclus dans
$\not\subset$	$/<<:$	n'est pas strictement inclus dans

Opérations sur les ensembles	**symbôle**	**Signification**
$\cup$	$\backslash/$	union d'ensembles
$\cap$	$/\backslash$	intersection d'ensembles
$-$	$-$	soustraction d'ensembles
union	*union*	union d'un ensemble d'ensembles
inter	*inter*	intersection d'un ensemble d'ensembles
$\bigcup$	$UNION$	union quantifiée
$\bigcap$	$INTER$	intersection quantifiée

Opérations sur les relations et les fonctions	**symbôle**	**Signification**
dom	dom	ensemble domaine
ran	ran	ensemble rang ou codomaine
id	id	relation identité
prj_1	$prj1$	première projection d'une relation
prj_2	$prj2$	deuxième projection
rel	rel	transformée d'une fonction en relation
fnc	fnc	transformée d'une relation en fonction
;	;	composition séquentielle des relations
$\otimes$	><	produit direct de relations
$\parallel$	||	produit paralléle de relations
R^{-1}	$R^{\sim}$	relation inverse
R^*	$closure(R)$	fermeture réflexive transitive
R^+	$closure1(R)$	fermeture transitive
$\lhd$	<|	restriction de domaine
$\not\lhd$	<<|	soustraction de domaine
$\rhd$	|>	restriction de codomaine
$\not\rhd$	|>>	soustraction de codomaine
$\Leftarrow\!\!\!+$	< +	overriding ou surcharge
$r[s]$	r[s]	image de l'ensemble s par la relation r
$f(x)$	f(x)	valeur de la fonction f au point x

Ensembles et constantes arithmétiques prédéfinis	**symbôle**	**Signification**
$\mathbb{N}$	$NATURAL$	entiers naturels
$\mathbb{N}^+$	$NATURAL1$	entiers positifs
$\mathbb{Z}$	$INTEGER$	entiers relatifs
NAT	NAT	entiers naturels représentables
NAT_1	$NAT1$	entiers positif représentables
INT	INT	entiers relatifs représentables
$minint$	$MININT$	entier minimum représentable
$maxint$	$MAXINT$	entier maximum représentable

Opérations arithmétiques	symbôle	Signification
$+$	$+$	addition
$-$	-	soustraction et moins unaire
$\times$	$*$	multiplication
$/$	/	quotient de la division entière
mod	mod	reste de la division entière
x^y	$x ** y$	puissance entière
$succ$	succ	fonction succ des entiers
$pred$	pred	fonction pred des entiers
Σ	SIGMA	somme quantifiée d'entiers
Π	PI	produit quantifié d'entiers
$card$	card	cardinal d'un ensemble
min	min	minimum d'un ensemble d'entiers
max	max	maximum d'un ensemble d'entiers

viii. *Opérations arithmétiques*,

ix. *Prédicats arithmétiques*.

Prédicats arithmétiques	symbôle	Signification
$\leq$	<=	plus petit ou égal
$<$	<	strictement plus petit
$\geq$	>=	plus grand ou égal
$>$	>	strictement plus grand

(b) ***Calcul dans les substitutions généralisées***

i. **Terminaison**
On caractérise le fait qu'une substitution S "termine" par un prédicat note $trm(S)$. On donne ci-après la définition théorique du prédicat trm. Le prédicat $btrue$ est la constante de prédicat "vrai", alors que $bfalse$ est la constante prédicat "faux".

Notation	Définition
$trm(S)$	$[S]\, btrue$

On calcule facilement les résultats de terminaison des substitutions primitives et de quelques formes usuelles. Le prédicat prd_x est défini au paragraphe A.2.3.

$$
\begin{array}{l}
trm(x := E) \Leftrightarrow btrue \\
trm(skip) \Leftrightarrow btrue \\
trm(P \mid S) \Leftrightarrow P \wedge trm(S) \\
trm(P \Rightarrow S) \Leftrightarrow P \Rightarrow trm(S) \\
trm(S[T) \Leftrightarrow trm(S) \wedge trm(T) \\
trm(@z \cdot S) \leftrightarrow \forall z \cdot trm(S) \\
trm(x :\in E) \Leftrightarrow btrue \\
trm(x : P) \Leftrightarrow btrue
\end{array}
$$

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

$$\begin{aligned} trm(W(P,S,J,V)) \Leftrightarrow & J \wedge \forall x \cdot ((J \wedge P) \Rightarrow [S]J) \wedge \\ & \forall x \cdot (J \Rightarrow V \in N) \wedge \\ & \forall x \cdot ((J \wedge P) \Rightarrow [n := V][S](V < n)) \end{aligned}$$

ii. **Faisabilité**

Certaines substitutions généralisées peuvent ne pas avoir de sens. Cela arrive en particulier lorsqu'on a une substitution gardée, dont la garde est toujours fausse : $bfalse \Rightarrow S$. En effet, dans ce cas quelle que soit la post-condition R, cette substitution établit R, puisque $[bfalse \Rightarrow S]R \Leftrightarrow bfalse \Rightarrow [S]R$ qui est effectivement vrai. C'est théoriquement une substitution "miraculeuse" puisqu'elle permet de spécifier n'importe quel programme. Malheureusement (ou plutôt heureusement) c'est une substitution irréalisable ou non "faisable". Le prédicat $fis(S)$ caractérise les valeurs pour lesquelles S est faisable. On

Notation	Définition
$fis(S)$	$\neg[S]\,bfalse$

calcule facilement les résultats de faisabilité des substitutions primitives et de quelques formes usuelles. Ce calcul est celui de la "garde" d'une substitution, aussi dans certains articles, on trouve la notation $grd(S)$. Dans la substitution $x : P$, la variable $x\$0$, que l'on peut trouver dans$P$, représente la valeur de x avant la substitution. W, la variable $x\$0$, que l'on peut trouver dans J ou V, représente la valeur de x avant l'entrée dans la boucle.

$fis(x := E) \Leftrightarrow btrue$
$fis(skip) \Leftrightarrow btrue$
$fis(P \mid S) \Leftrightarrow P \Rightarrow trm(S)$
$fis(P \Rightarrow S) \Leftrightarrow P \wedge fis(S)$
$fis(S[T) \Leftrightarrow trm(S) \vee trm(T)$
$fis(@z \cdot S) \leftrightarrow \forall z \cdot fis(S)$
$fis(x :\in E) \Leftrightarrow E \neq \emptyset$
$fis(x : P) \Leftrightarrow \exists x' \cdot [x\$0, x := x, x']P$
$fis(S;T) \Leftrightarrow (trm(S) \Rightarrow \exists x' \cdot (prd\,x(S) \wedge [x := x']fis(T)))$ $fis(W(P,S,J,V)) \Leftrightarrow$
$trm(W(P,S,J,V)) \Rightarrow \exists x' \cdot ([x\$0, x := x, x'](J \wedge \neg P))$

iii. **Prédicat avant-après**

Le prédicat avant-après permet de déterminer la relation entre les valeurs d'une variable x avant et après une substitution S. D'une manière classique, la valeur avant est notée x et la valeur après est notée x' (il s'agit toujours d'un nouvel identificateur par rapport aux variables de la machine courante). On voit donc que la ou les variables dont on veut parler sont un paramètre supplémentaire du prédicat. On note $prd_x(S)$ la relation avant-après des variables x pour la substitution S. La double négation est nécessaire à cause du non-déterminisme des substitutions. La formule signifie que x' est une des valeurs possibles de x après l'opération. Le calcul donne en particulier les résultats suivants

Notation	Définition
$prd_x(S)$	$\neg[S](x' \neq x)$

(pour la notation $x\$0$, voir les commentaires au paragraphe précédent) :

$$
\begin{array}{l}
prd_x(x := E) \Leftrightarrow x' = E \\
prd_{x,y}(x := E) \Leftrightarrow x', y' = E, y \\
prd_x(skip) \Leftrightarrow x' = x \\
prd_x(P \mid S) \Leftrightarrow P \Rightarrow prd_x(S) \\
prd_x(P \Rightarrow S) \Leftrightarrow P \wedge prd_x(S) \\
prd_x(S[T) \Leftrightarrow prd_x(S) \vee prd_x(T) \\
prd_x(@z \cdot S) \Leftrightarrow \exists z \cdot prd_x(S) siz \backslash x' \quad (*) \\
prd_x(@y \cdot T) \Leftrightarrow \exists y, y' \cdot prd_{x,y}(T) siy \backslash x' \quad (**) \\
prd_x(x :\in E) \Leftrightarrow x' \in E \\
prd_x(x : P) \Leftrightarrow [x\$0, x := x, x']P \\
prd_x(S;T) \Leftrightarrow (trm(S) \Rightarrow \exists x'' \cdot ([x' := x'']prd_x(S) \wedge [x := x'']prd_x(T))) \\
prd_x(W(P,S,J,V)) \Leftrightarrow trm(W(P,S,J,V)) \Rightarrow [x\$0, x := x, x'](J \wedge \neg P)
\end{array}
$$

Le prédicat avant-après d'une substitution de choix non borné "$@z \cdot S$" est divisé en deux : dans le cas marqué (*), la variable z n'est pas modifiée par S. Dans le cas marqué (**), la variable y est modifiée et le développement du prédicat sur T est donc quantifié par y et y' qui vont intervenir dans la relation $prd_{x,y}(T)$. Une propriété intéressante est de relier la faisabilité et l'existence d'une valeur "après" du prédicat avant-après. On peut en effet démontrer :

$$fis(S) \Leftrightarrow \exists x' \cdot prd_x(S)$$

iv. **Forme normalisée**
Toute substitution généralisée peut se mettre sous la forme :

$$S = P \mid @x' \cdot (Q \Rightarrow x := x') six' \backslash P$$

Dans le B-Book [Abr96], la preuve de cette forme normalisée est donnée en la calculant pour chacune des substitutions primitives et en appliquant des transformations syntaxiques. En fait, on peut montrer que toute substitution généralisée est équivalente à la forme suivante, construite avec trm et de prd_x :

$$S = trm(S) | @x' \cdot (prd_x(S) \Rightarrow x := x')$$

La plus faible précondition d'une forme normalisée est :
$[P \mid @x' \cdot (Q \Rightarrow x := x')]R \Leftrightarrow P \wedge \forall x' \cdot (Q \Rightarrow [x := x']R)$
On a les équivalences :

$$
\begin{array}{l}
trm(P \mid @x' \cdot (Q \Rightarrow x := x') \Leftrightarrow P \\
fis(P \mid @x' \cdot (Q \Rightarrow x := x') \Leftrightarrow P \Rightarrow \exists x' \cdot Q \\
prd_x(P \mid @x' \cdot (Q \Rightarrow x := x') \Leftrightarrow P \Rightarrow Q
\end{array}
$$

On peut définir l'égalité de deux substitutions par : En se basant sur la forme normali-

$S = T$	$[S]R \Leftrightarrow [T]R$	pour tout prédicat R

sée, ce résultat conduit à : Dans le tableau qui suit, R et Q sont des prédicats et S une substitution généralisée quelconque. Les propriétés suivantes font partie de la théorie des

$S = T$	$(trm(S) \Leftrightarrow trm(T)) \wedge ((trm(S) \Rightarrow prd_x(S)) \Leftrightarrow (trm(T) \Rightarrow prd_x(T)))$

$[S](R \wedge Q) \Leftrightarrow [S]R \wedge [S]Q$	Distributivité
$\forall x \cdot (R \Rightarrow Q) \Rightarrow ([S]R \Rightarrow [S]Q)$	Monotonie

Totalité	$prd_x(S) \vee trm(S)$
Terminaison	$[S]R \Rightarrow trm(S)$

substitutions généralisées et plus généralement des transformateurs de prédicats positivement conjonctifs : Les substitutions généralisées satisfont les propriétés : Preuve : On a : $x = x' \vee btrue$, c'est-à-dire $x \neq x' \Rightarrow btrue$. Et, par la propriété de monotonie :

$$\begin{aligned} [S](x6 = x') &\Rightarrow [S]btrue \\ &\Leftrightarrow \neg[S](x \neq x') \vee [S]btrue \\ &\Leftrightarrow prd_x(S) \vee trm(S) \end{aligned}$$

La seconde propriété se démontre de la même manière.

Annexe B

Le raffinement dans les méthodes formelles

Refinement property	Darwin,MetaH,UniCon,Weaves	Rapide	UML-based Catalysis
Place du raffinement	implément-ation	conception	Document-ation a posteriori
Type de Processus de Raffinement	Compilateur (excepté Weaves)	Comparaison de traces a posteriori	Document-ation
Relation de Raffinement	Compilation	Inclusion de la trace abstraite dans la trace concrète	NON
Distinction RH / RV	NON	-	OUI
Raffinement Horizontal (RH)	-	-	OUI
Raffinement Vertical (RV)	OUI	-	OUI
Raffinement de Données	-	NON	OUI
Raffinement fonctionnel	-	NON	Limité
Raffinement Comportemental	-	OUI	NON
Raffinement compositionnel	NON	NON	
Génération de Code	OUI	NON	NON
Multiples niveaux d'abstraction	NON	-	OUI
Réutilisation	NON	NON	NON
Préservation de propriétés "inhérentes"	OUI	OUI mais juste 1 niveau	NON
Préservation de propriétés "définies par l'utilisateur"	NON	NON	NON
Multi-langages	-	Plusieurs sous-Langages	-

Refinement property	VDM	Z
Place du raffinement	conception	Conception
Type de Processus de Raffinement	Comparaison a posteriori des modèles possibles	Vérification a posteriori de conditions mathématiques (logiques)
Relation de Raffinement	L'implémentation doit avoir une fonctionnalité vue comme implémentant les constructions spécifiées de façon moins précise	Affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme), lorsque l'abstraction se termine
Distinction RH / RV	-	-
Raffinement Horizontal (RH)	-	-
Raffinement Vertical (RV)	NON pas réellement	OUI
Raffinement de Données	NON	OUI
Raffinement fonctionnel	OUI	OUI
Raffinement Comportemental	OUI (modèles possibles)	Limité
Raffinement compositionnel	OUI	OUI
Génération de Code	NON	NON
Multiples niveaux d'abstraction	Pas très distincts	Pas très distincts
Réutilisation	NON	NON
Préservation de propriétés "inhérentes"	OUI (modèles possibles)	OUI (obligations de preuves)
Préservation de propriétés "définies par l'utilisateur"	NON	NON
Multi-langages	NON	NON

Refinement property	Méthode B	Event-B
Place du raffinement	Conception	Conception
Type de Processus de Raffinement	différentiel	Différentiel (extension du B classique)
Relation de Raffinement	Affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme) : comportement possible de l'abstraction	Affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme)
Distinction RH / RV	-	NON
Raffinement Horizontal (RH)	-	OUI (mais pas sous forme de composant et connecteur)
Raffinement Vertical (RV)	OUI (mais limité surtout aux corps d'opérations)	OUI (mais limité surtout aux corps d'opérations)
Raffinement de Données	Limité	Limité
Raffinement fonctionnel	OUI (modification des corps d'opérations)	OUI modification des corps d'opérations
Raffinement Comportemental	Limité	Limité
Raffinement compositionnel	OUI (excepté les implémentations)	OUI (excepté les implémentations) : possibilité d'enchaîner raffinements et décompositions
Génération de Code	OUI	OUI
Multiples niveaux d'abstraction	OUI	OUI
Réutilisation	NON	NON
Préservation de propriétés "inhérentes"	OUI (obligations de preuve)	OUI (obligations de preuve sur des états et des événements)
Préservation de propriétés "définies par l'utilisateur"	NON (pas automatisée)	NON (pas automatisée)
Multi-langages	Mots-clés réservés en fonction du niveau d'abstraction	Mots-clés réservés en fonction du niveau d'abstraction

Refinement property	B et CSP	CSP2B
Place du raffinement	Conception	Conception
Type de Processus de Raffinement	Différentiel (raffinement classique en B : la partie CSP est conservée dans les différents Niveaux d'abstraction)	Différentiel (raffinement classique en B : les machines B et CSP sont raffinées indépendamment)
Relation de Raffinement	Affaiblissement des préconditions et renforcement des post-conditions (plus de déterminisme) : comportement possible de l'abstraction	Affaiblissement des préconditions et renforcement des post-conditions
Distinction RH / RV	-	-
Raffinement Horizontal (RH)	-	-
Raffinement Vertical (RV)	OUI (mais limité surtout aux corps d'opérations)	OUI (mais limité surtout aux corps d'opérations)
Raffinement de Données	Limité	Limité
Raffinement fonctionnel	OUI modification des corps d'opérations	OUI modification des corps d'opérations
Raffinement Comportemental	Limité	Limité
Raffinement compositionnel	OUI (excepté les implémentations)	OUI
Génération de Code	NON	OUI (conversion des machines CSP en B)
Multiples niveaux d'abstraction	OUI	OUI
Réutilisation	NON	NON
Préservation de propriétés "inhérentes"	OUI (obligations de preuve)	OUI (obligations de preuve)
Préservation de propriétés "définies par l'utilisateur"	NON	NON (pas automatisée)
Multi-langages	OUI : CSP et un sous ensemble de B	OUI (CSP et B)

	Refinement property	Réduction pour la Méthode B	Calcul de raffinement
Place du raffinement		Conception	Conception/ implémentation
Type de Processus de Raffinement		Différentiel mais le résultat est la spécification d'un nouveau problème	Transformation et raffinement de comportement ou justification de programmation
Relation de Raffinement		Renforcement des préconditions et affaiblissement des post-conditions	Lois de raffinement (renforcement de post-conditions,⋯) et invariants
Distinction RH / RV		-	NON
Raffinement Horizontal (RH)		-	possible
Raffinement Vertical (RV)		NON pas vraiment (ajout d'opération et changement de paramètres)	possible
Raffinement de Données		Limité	OUI
Raffinement fonctionnel		OUI	OUI (utilisation d'invariants locaux)
Raffinement Comportemental		Limité	OUI
Raffinement compositionnel		NON	NON
Génération de Code		NON	OUI (lois de raffinement)
Multiples niveaux d'abstraction		NON	OUI
Réutilisation		NON	OUI (lois de raffinement)
Préservation de propriétés "inhérentes"		OUI mais limitée aux relations de recouvrement	OUI (invariants)
Préservation de propriétés "définies par l'utilisateur"		NON	NON
Multi-langages		NON	NON (mais différents vocabulaires)

Refinement property	SADL	Event-B à base d'opérateur
Place du raffinement	Conception	*Conception architecturale*
Type de Processus de Raffinement	Mise en correspondance de styles	*Transformation par opérateurs génériques utilisant des théories polymorphiques.*
Relation de Raffinement	Traduction selon la mise en correspondance : exactement la même chose	*Traduction des concepts de contrôle architecturaux par des théories peuvent être appelés dans des modèles en Event-B*
Distinction RH / RV	Distinction entre décomposition et raffinement de style	*OUI*
Raffinement Horizontal (RH)	Composants composites Statiques	*OUI (mais pas sous forme de composant et connecteur)*
Raffinement Vertical (RV)	Changement de styles architecturaux entiers	*OUI*
Raffinement de Données	Pour ce qui peut être défini dans les Styles	*Limité*
Raffinement fonctionnel	Pour ce qui peut être défini dans les styles	*OUI*
Raffinement Comportemental	NON	*OUI*
Raffinement compositionnel	OUI	*OUI*
Génération de Code	NON	*Oui (générateur basé sur syntaxe orienté théorie)*
Multiples niveaux d'abstraction	Si les styles sont considérés ainsi	*OUI*
Réutilisation	OUI patrons	*OUI*
Préservation de propriétés "inhérentes"	OUI	*OUI*
Préservation de propriétés "définies par l'utilisateur"	NON	*OUI*
Multi-langages	NON	*OUI*

Annexe C

Le language VHDL

Le langage VHDL est utilisé pour de nombreuses applications, c'est un langage de description de matériel qui utilisé pour la spécification (description du fonctionnement) ; la simulation et la preuve formelle d'équivalence de circuits. Ensuite il a aussi été utilisé pour la synthèse automatique. L'abréviation VHDL ou 'VHSIC Hardware Description Langage' est développé dans les années 80 aux États-Unis, le langage de description VHDL est ensuite devenu une norme IEEE numéro 1076 en 1987. Révisée en 1993 pour supprimer quelques ambiguïtés et améliorer la portabilité du langage, cette norme est vite devenue un standard en matière d'outils de description de fonctions logiques. A ce jour, on utilise le langage VHDL pour :

- Concevoir des ASIC,
- Programmer des composants programmables du type PLD, CPLD et FPGA,
- Concevoir des modèles de simulations numériques ou des bancs de tests.

Le VHDL est un langage normalisé, cela lui assure une pérennité. Il est indépendant d'un fournisseur d'outils. Il est devenu un standard tous les vendeurs d'outils EDA. Cela permet aux industriels d'investir sur un outil qui n'est pas qu'une mode éphémère, c'est un produit commercialement inévitable. Techniquement, il est incontournable car c'est un langage puissant, moderne et qui permet une excellente lisibilité, une haute modularité et une meilleure productivité des descriptions. Il permet de mettre en œuvre les nouvelles méthodes de conception. Ce langage s'adresse à des concepteurs de systèmes électroniques, qui n'ont pas forcément de grandes connaissances en langage de programmation. D'autres fausses idées circulent sur le langage VHDL. Celui-ci n'assure pas la qualité du résultat, la portabilité et la synthèse des descriptions. Une méthodologie est indispensable pour combler ces lacunes. Certaines instructions du VHDL sont clairement non synthétisables. Il y a, par exemple, l'instruction « Wait for X ns » qui modélise un temps. Mais bien souvent, c'est la manière d'utiliser une instruction qui rend la description non synthétisable. La connaissance seule des instructions de leur syntaxe ne suffit pas. Il faut connaitre leur utilisation dans le cadre de la synthèse. A l'inverse, la connaissance de l'ensemble du langage n'est pas nécessaire pour écrire des descriptions synthétisables. En 1999 IEEE a édité la norme 1076.6 qui définit le sous ensemble synthétisable. Cette norme définit l'ensemble des syntaxes autorisées en synthèse.Nous pouvons affirmer que depuis l'année 2000 l'ensemble synthétisable du langage VHDL est mieux en mieux défini. La majorité des outils de synthèse sont compatibles avec cette norme. Un véritable standard est donc établi pour la synthèse automatique avec le langage VHDL.

— Contrairement à C ou PASCAL, VHDL est un langage qui comprend le « parallélisme », c'est à dire que des blocs d'instructions peuvent être exécutés simultanément, par opposition à séquentiellement comme dans un langage procédural traditionnel. Autant ce parallélisme est fondamental pour comprendre le fonctionnement d'un simulateur logique, et peut-être déroutant pour un programmeur habitué au déroulement séquentiel des instructions qu'il écrit, autant il est évident que le fonctionnement d'un circuit ne dépend pas de l'ordre dans lequel

ont été établies les connexions. L'utilisateur de VHDL gagnera beaucoup en ne se laissant pas enfermer dans l'aspect langage de programmation, en se souvenant qu'il est en train de créer un vrai circuit. Les parties séquentielles du langage, car il y en a, doivent, dans ce contexte, être comprises soit comme une facilité offerte dans l'écriture de certaines fonctions, soit comme le moyen de décrire des opérateurs fondamentalement séquentiels : les opérateurs synchrones.

— La modélisation correcte d'un système suppose de prendre en compte, au niveau du simulateur, les imperfections du monde réel. VHDL offre donc la possibilité de spécifier des retards, de préciser ce qui se passe lors d'un conflit de bus etc. Pour simuler toutes ces vicissitudes, le langage offre toute une gamme d'outils : signaux qui prennent une valeur inconnue, messages d'erreurs quand un « circuit » détecte une violation de setup time, changements d'état retardés pour simuler les temps de propagation. Toutes les constructions associées de ce type ne sont évidemment pas synthétisables. La difficulté principale est que, suivant les compilateurs, la frontière entre ce qui est synthétisable et ce qui ne l'est pas n'est pas toujours la même, même pour des compilateurs qui respectent la norme IEEE-1076. Avant d'utiliser un outil de synthèse, le concepteur de circuit a tout à gagner à lire très attentivement la présentation du sous-ensemble de VHDL accepté par cet outil.

(a) ***Les éléments de bases du langage VHDL***

Trois classes de données existent en VHDL : les constantes, les variables et les signaux. La nature des signaux ne présente aucune ambiguïté, ce sont des objets qui véhiculent une information logique tant du point de vue simulation que dans la réalité. Les signaux qui ont échappé aux simplifications logiques, apportées par l'optimiseur toujours présent, sont des vraies équipotentielles du schéma final. Les variables sont destinées, comme dans tout langage, à stocker temporairement des valeurs, dans l'optique d'une utilisation future, sans chercher à représenter la réalité. Certains compilateurs considèrent que les variables n'ont aucune existence réelle, au niveau du circuit, qu'elles ne sont que des outils de description fonctionnelle. D'autres transforment, éventuellement (cela dépend de l'optimiseur), les variables en cellules mémoires··· Pendant l'implémentation en VHDL il faut :

— n'utiliser dans les identificateurs que des lettres et des chiffres.

— définir si vous écrivez les identificateurs en majuscules ou en minuscules.

Afin de simplifier la lecture des descriptions VHDL, nous avons défini une convention pour les noms des signaux. Le tableau ci-après vous donne les suffixes utilisés :

Objet	Suffixe
Port d'entrée	_i
Port de sorti	_o
Port entrée /sortie	_io
Signal interne architecture textuelle	_s
Signal interne schéma bloc	Aucun
Constante	_c
Variable	_v
Spécification pour un blanc de test (test- bench)	
Signaux de stimuli	_sti
Signaux observés	_obs
Signaux de référence	_ref

(b) ***Les types d'objets***

Le VHDL est un langage fortement typé. Tout objet manipulé doit être déclaré et avoir un type avant sa première utilisation. Indépendamment de son type, un objet appartient à une classe. Nous pouvons résumer cela en disant que le type définit le format de l'objet et que la classe spécifie le comportement de celui-ci. Le langage distingue quatre catégories de type :

— les types scalaires, cela comprend les types énumérés et numériques.
— les types composés (tableau et enregistrement), ceux-ci possèdent sous-éléments.
— les types accès, qui sont des pointeurs.
— le type fichier, qui permet de générer des fichiers.

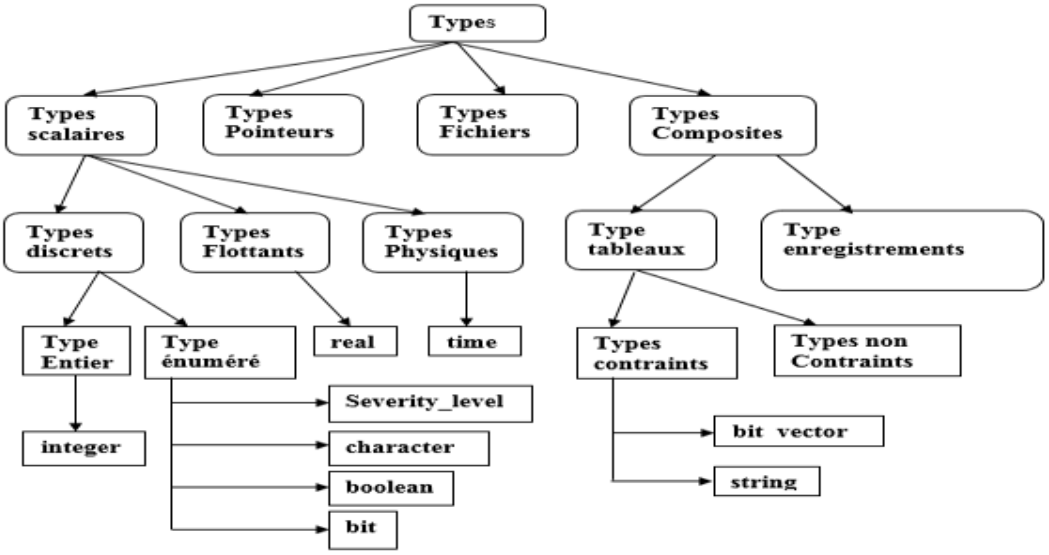


Figure 1 type de package.

- Nous allons uniquement utiliser, dans le cadre du VHDL en vue de la synthèse, les deux premiers types. Les deux autres types ne peuvent pas être utilisés pour la synthèse.
- Les types scalaires sont constitués par une liste de valeurs (types énumérés) ou par un domaine de variation (types numériques).
- Les types composés sont constitués d'éléments scalaires tous de même type dans le cas de tableaux (array) et de types différents dans le cas d'enregistrements (record).
- Le type enregistrement sera utilisé uniquement pour les blancs de test.
- Le type de tableau que nous utiliserons le plus souvent est le **Std_Logic_Vector**. Celui-ci est un ensemble de **Std_Logic.**

```
Type Std_Logic_Vector is array (Natural range <>) of  Std_Logic ;
Type Unsigned  is  array (Natural range <>) of Std_Logic ;
```

(c) ***Les Opérateurs***
En VHDL, il existe un vaste choix d'opérateurs :

— **Opérateurs logiques :**

Opérateurs	**Description**
And	ET logique
Or	OU logique
Nand	Non-ET logique
Nor	Non-OU logique
Xor	OU exclusive logique
Xnor	Non-OU exclusive logique

Opérateurs	Description
=	égal
/=	différent
<	strictement inférieur
>	strictement supérieur
>=	supérieur ou égal
<=	inférieur ou égal

Opérateurs	Description
sll	décalage logique à gauche
srl	décalage logique à droite
sla	décalage arithmétique à gauche
sra	décalage arithmétique à droite
rol	rotation à gauche
ror	rotation à droite

— **Opérateur de test :**

— **Opérateur de décalage :**

— **Opérateur arithmétique :**

Opérateurs	Description
+	addition
-	soustraction
/	division
*	multiplication
&	concaténation

- Et il a y d'autres : opérateurs relationnels, opérateurs de signe, opérateurs multiplicatifs, opérateurs divers.
- l'affectation d'une variable se fait à l'aide de l'opérateur :=
  Nom_Variable := expression ;
- l'affectation d'un signal se fait à l'aide de l'opérateur <=
  Nom_Signal <= expression ;

(d) ***Limitations du langage***

Lors des descriptions VHDL en vue de synthèse, c'est le style d'écriture et lui seul qui va guider le synthétiseur dans ses choix d'implantation au niveau circuit. Il est donc nécessaire de produire des instructions ayant une équivalence non ambiguë au niveau porte.
Le synthétiseur est un compilateur un peu particulier susceptible au fil des ans d'améliorer sa capacité à implanter des fonctions de plus en plus abstraites. Cependant, il reste des règles de bon sens comme « le retard des opérateurs est d'ordre technologique » ou bien « un fichier n'est pas un circuit » etc. En conséquence, les limitations du langage du niveau RTL les plus courantes sont :

— Un seul WAIT par PROCESS

— Les retards sont ignorés (pas de sens)

— Les initialisations de signaux ou de variables sont ignorées

— Pas d'équation logique sur une horloge (conseillé)

— Pas de fichier ni de pointeur

- — Restriction sur les boucles (LOOP)
- — Restriction sur les attributs de détection de fronts (EVENT, STABLE)
- — Pas de type REAL.
- — Pas d'attributs BEHAVIOR, STRUCTURE, LAST_EVENT, LAST_ACTIVE, TRANSACTION.
- — Pas de mode REGISTER et BUS.

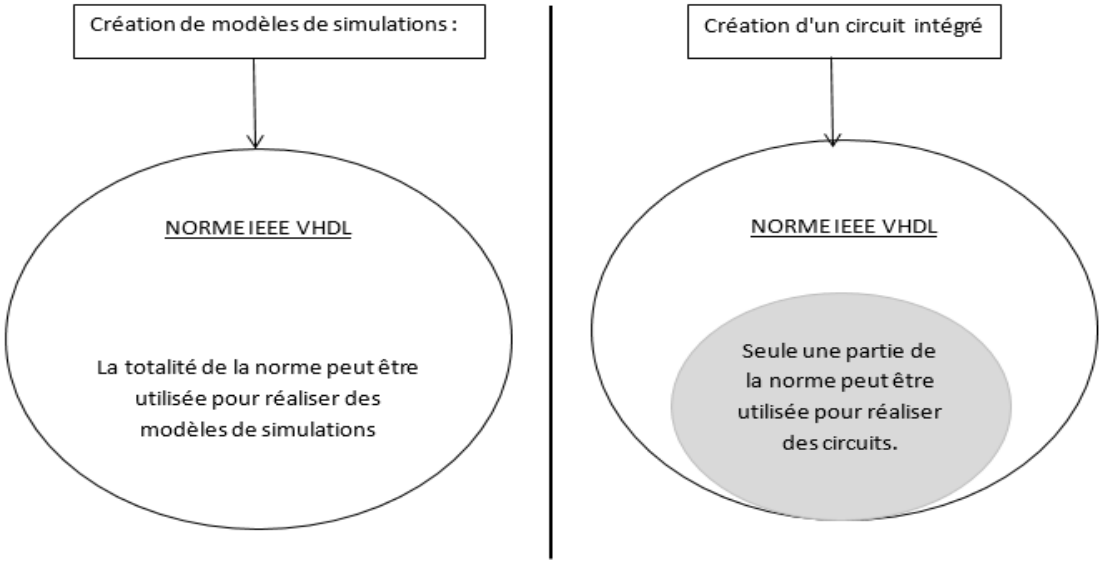
(e) ***Les Avantages***

Les intérêts majeurs du langage sont :

- — Des niveaux de description très divers : VHDL permet de représenter le fonctionnement d'une application tant du point de vue système que du point de vue circuit, en descendant jusqu'aux opérateurs les plus élémentaires. A chaque niveau, la description peut être structurelle (portrait des interconnexions entre des sous-fonctions) ou comportementale (langage évolué).
- — Son aspect « non-propriétaire » : le développement des circuits logiques a conduit chaque fabricant à développer son propre langage de description. VHDL a été, initialement, conçu comme un langage de simulation, il est fortement marqué par cet héritage très informatique, ce qui est parfois un peu déroutant pour l'électronicien, proche du matériel, qui n'est pas toujours un spécialiste des langages de programmation. Citons quelques exemples :
- — Niveaux de description très divers : VHDL permet de représenter le fonctionnement d'une application aussi bien du point de vue élémentaire que du point de vue système. A chaque niveau, la description peut être structurelle ou comportementale (langage évolué).
- — Portabilité des descriptions VHDL, c'est-à-dire, possibilité de cibler une description VHDL dans le composant ou la structure que l'on souhaite en utilisant l'outil que l'on veut.
- — Augmentation de la qualité des designs.

(f) ***La simulation en VHDL***

La norme qui définit la syntaxe et les possibilités offertes par le langage de description VHDL est très ouverte. Il est donc possible de créer une description VHDL de systèmes numériques non réalisable, tout au moins, dans l'état actuel des choses. Il est par exemple possible de spécifier les temps de propagations et de transitions des signaux d'une fonction logique, c'est à-dire créer une description VHDL du système que l'on souhaite obtenir en imposant des temps précis de propagation et de transition. Or les outils actuels de synthèses logiques sont incapables de réaliser une fonction avec de telles contraintes. Seuls des modèles théoriques de simulations peuvent être créés en utilisant toutes les possibilités du langage.
Cette situation peut donc se résumer de la façon suivante :


Création de modèles de simulations :
NORME IEEE VHDL
La totalité de la norme peut être utilisée pour réaliser des modèles de simulations
Création d'un circuit intégré
NORME IEEE VHDL
Seule une partie de la norme peut être utilisée pour réaliser des circuits.

Annexe D

Les théories en Event-B développées

(a) ***La théorie NoC***

Le réseau NoC est un ensemble de nœuds qui pourraient avoir tant de rôles comme étant une *src* source de transition de paquets ou la destination *rcv* ou une *dst* intermédiaire lors de la transition de paquets, même les nœuds pourraient également avoir aucun rôle *nop*.

$datatype\ role$
$constructors\ nop, src, rcv, dst$

$operator\ node$
$prefix$
$args\ r : role, nod : \mathbb{P}(S)$
$condtition\ nd \in nod$
$definition\ nd$

Le NoC est présenté comme un réseau (un graphe) entre les sources et les destinations de paquets. Ce graphe est non vide, non-transitive et symétrique.

$$thm : \forall net, T \cdot net \in \mathbb{P}(S \times S) \wedge T \in \mathbb{P}(S) \wedge any \in role \wedge nodes \in node(any, T) \Rightarrow sym(net, nodes) \wedge cls(net) \wedge irreflexive(net, nodes) \wedge card(net) \neq 0$$

Ce graphe peut être représenté comme une grille avec un $Netsize$ très spécifique de la taille.

$operator\ int\_max$
$definition\ 1000$
`network with 1000 nodes`

$operator\ netsize$
$definition\ 1..int\_max$

Toutes les opérations à l'intérieur du réseau NoC est lié à la nature d'accès d'entrée et de sortie (représentée par *pio*) des nœuds représentés par $portsNoc$ et les buffers (représentés par $buffersNoc$ pour les buffers et $buffchnlNoc$ pour les buffers utilisés crédit pour envoyer des données) pour stocker les données transmises l'intérieur du réseau.

datatype *iop*
constructors *in*, *out*

---

operator *sent*
prefix
args $a : role$
$b : \mathbb{P}(S)$
$m : \mathbb{P}(T)$
condtition $any \in role$
$\wedge nd \in \mathbb{P}(S)$
$\wedge a \in node(src, nd)$
$\wedge b \in node(any, nd)$
definition $a \mapsto m$

---

operator *forward*
prefix
args $a : role$
$b : \mathbb{P}(S)$
$m : \mathbb{P}(T)$
condtition $any \in role$
$\wedge nd \in \mathbb{P}(S)$
$\wedge a \in node(dst, nd)$
$\wedge b \in node(any, nd)$
definition $b \mapsto m$

---

operator *receive*
prefix
args $a : role$
$b : \mathbb{P}(S)$
$m : \mathbb{P}(T)$
condtition $any \in role$
$\wedge nd \in \mathbb{P}(S)$
$\wedge a \in node(any, nd)$
$\wedge b \in node(rcv, nd)$
definition $b \mapsto m$

---

operator *store*
prefix
args $a : \mathbb{P}(S)$
$m : \mathbb{P}(T)$
condtition $any \in role$
$\wedge nd \in \mathbb{P}(S)$
$\wedge a \in node(any, nd)$
definition $a \mapsto m$

---

operator *chanel*
prefix
args $a : role$
$b : \mathbb{P}(S)$
$m : \mathbb{P}(T)$
condtition $any \in role$
$\wedge nd \in \mathbb{P}(S)$
$\wedge a \in node(any, nd)$
$\wedge b \in node(any, nd)$
definition $a \mapsto b$

---

operator *max_places*
definition 3

---

operator *buffersNoc*
prefix
args $pin : iop$
$b : \mathbb{P}(S \times S)$
$m : \mathbb{P}(T)$
condtition $anypin \in iop$
$\wedge ports \in$ portsNoc(anypin,r)
definition $ports \mapsto m$

---

operator *portNoc*
prefix
args $pin : iop$
$r : \mathbb{P}(S \times S)$
definition r

---

operator *bufferchnlNoc*
prefix
args $pin : iop$
$r : \mathbb{P}(S \times S)$
$crd : \mathbb{P}(R)$
$m : \mathbb{P}(T)$
condtition $anypin \in iop$
$\wedge buff \in$ buffersNoc(anypin, r,m)
definition $buff \mapsto crd$

---

operator *coord*
prefix
args $xy : \mathbb{N} \leftrightarrow \mathbb{N}$
condtition $xy \in netsize \leftrightarrow netsize$
definition xy

operator $position$
prefix
args $nod : \mathbb{P}(S)$
condtition $any \in role$
$\wedge net \in node(any, nd)$
$\wedge net \in cls(nod \times nod)$
$\wedge pos \in \mathbb{N} \leftrightarrow \mathbb{N}\ xy \in coord(pos)$ *defintion*
$\{n, f \cdot n \in net \wedge f \in n \rightarrow xy \mid f\}$

---

operator $availableplce$
prefix
args $r : \mathbb{P}(S \times S)$
condtition $any \in \mathbb{P}(S \times S)$
$\wedge any \in portsNoc(out, r)$
defintion $1..max\_place$

---

operator $sizeplaces$
prefix
args $r : \mathbb{P}(S \times S)$
$pin : iop, nbrfree \mathbb{N}_1 :$
condtition $anyp \in \mathbb{P}(S \times S)$
$\wedge anyp \in portsNoc(pin, r)$
defintion $nbrfree$

Quand il est à propos de l'ensemble des processus que chaque réseau NoC utilisé doit expliquer tout d'abord l'ensemble des opérateurs suivants :

sent : pour représenter la source de données transmises.
receive : représenter la destination des données transmises.
forward : pour représenter le nœud qui transfère les données pour sa destination.
chanel : pour représenter le couple de nœuds pendant la transmission de données.
store : pour représenter le nœud qui stocke les données transmises et les données stockées.
sizeplace : pour représenter le nombre de places libres pour le port de sortie pour transmettre des données.
coord : représenter entier coordonnées.
position : représenter les coordonnées de deux nœuds dans une fermeture.
availableplace : pour représenter le maximum de buffers libres à l'intérieur du nœud.

Chaque nœud dans le réseau de réseau pourrait faire sur des événements suivants :

i. **Data sending** Lorsqu'une source envoie un paquet m, le paquet est placé dans le réseau d'un port d'entrée IP pour le switch de source s.

$EVENT\ Data\_sending \hat{=}$
$\vdots$
then $snt = snt \cup \{s \mapsto m\}$
$\wedge destin = destin \cup (d \times \{m\})$
$\wedge inputbuffer = inputbuffer \cup \{ip \mapsto m\}$
end

ii. **Credit-data receiving** Un paquet de crédit crd est reçu à partir d'un port de sortie op, s'il y a une place disponible $places$ dans le buffer.

$EVENT\ Credit\_data\_receiving \hat{=}$
$\vdots$
then buffcrdchn= buffcrdchn$\backslash\{ch \mapsto bfc\}$
$\wedge$ sizeplaces(out,op,places) = sizeplaces(out,op,places)+1
end

iii. **Data receiving** Un paquet est reçu par son destinataire, si le paquet a atteint la destination.

$EVENT\ Data\_receiving \hat{=}$	buffcrdchn= buffcrdchn
⋮	$\cup\{portsNoc \sim (in, ip) \mapsto bfc\}$
$then\ rcvd = rcvd \cup \{d \mapsto m\}$	end
$\wedge$ inputbuffer= inputbuffer $\setminus\{ip \mapsto m\}$	

iv. **Data forward**Dans le réseau, un paquet m transite par un noeud x vers un autre noeud y, jusqu'à ce qu'il atteigne sa destination d. Ce passage est de la logique de sortie op de passer à un canal $switchchcontrol$.

$EVENT\ Data\_forward \hat{=}$	$\wedge$ outputbuffer= outputbuffer $\setminus\{op \mapsto m\}$
⋮	end
$then$	
switchcontrol = switchcontrol$\setminus\{x \mapsto m\}$	

v. **Switch contrôle** Modélise le passage d'un paquet p, à partir d'un port d'entrée ip d'un commutateur x, à un port de de sortie op menant à un noeud y lorsque le noeud x est plus occupé, il envoie un signal de libération au noeud précédent relié au port d'entrée.

$Event\ switch\_control \hat{=}$	$\wedge$ outputbuffer =outputbuffer $\cup\{op \mapsto m\}$
⋮	$\wedge$ buffcrdchn = buffcrdchn $\cup\{portsNoc \sim (in, ip) \mapsto bfc\}$
$then$ availableplace(op) = availableplace(op)-1	end
$\wedge$inputbuffer =inputbuffer $\setminus\{ip \mapsto m\}$	

Lorsque chaque nœud dans le réseau NoC a quatre directions du noeud peut contrôler quatre cas pour acheminer entrant et sur les données provenant de ces cas sont expliqués comme suit :

— *à gauche Cas 1 :* un paquet m est transmis, à partir d'un port d'un noeud x d'entrée, à un port de sortie, ce qui conduit à un voisin y. Cet événement est déclenché si les coordonnées x de la destination d (du paquet m) est inférieure à la coordonnée x du noeud courant x.

EVENTswitch_control_West $\hat{=}$	$\wedge$ prj2(position (y)) = prj2(position (x))
$Refines\ switch\_control$	… $then$
$where$ …	**same actions of switch_control**
prj1(position(x)) > prj1(position (d))	end
$\wedge$ prj1(position (y)) = prj1(position (x))-1	

— *à droite Cas 2 :* un paquet m est transmis, à partir d'un port de sortie d'un noeud x, à un port d'entrée d'un voisin y. Cet événement est déclenché si les coordonnées x de la destination d (du paquet m) est supérieure à la coordonnée x du noeud courant x.

EVENTswitch_control_Est ≙
Refines switch_control
where ...
prj1(position(x)) < prj1(position (d))
∧ prj1(position (y)) = prj1(position (x))+1
∧ prj2(position (y)) = prj2(position (x))
... *then*
same actions of switch_control
end

— *vers le haut cas 3 :* un paquet *m* est transmis, à partir d'un port de sortie d'un noeud *x*, à un port d'entrée d'un voisin *y*. Cet événement est déclenché si la coordonnée y de la destination *d* (du paquet *m*) est supérieure à la coordonnée y du noeud courant *x*, et soit,si les coordonnées x de la destination *d* est égale à la coordonnée x du noeud courant *x*, ou si le paquet *m* ne peut pas le transit le long de l'axe *X*.

EVENTswitch_control_North ≙
Refines switch_control
where ...
(((prj1(position(x)) = prj1(position(d)))
∨ (prj1(position(x)) > prj1(position(d))
∧*x* ↦ position~(prj1(position(x))-1 ↦ prj2(position(x))) ∉ gr) ∨ (prj1(position(x)) < prj1(position(d))
∧*x* ↦ position~(prj1(position(x))+1 ↦ prj2(position(x))) ∉ gr)))
∧ prj2(position(x)) < prj2(position(d))
∧ prj1(position(y)) = prj1(position(x))
∧ prj2(position(y)) = prj2(position(x))+1
...*then*
same actions of switch_control
end

— *vers le bas de cas 4 :* un paquet *m* est transmis, à partir d'un port de sortie d'un noeud *x*, à un port d'entrée d'un voisin *y*. Cet événement est déclenché si la coordonnée y de la destination *d* (du paquet *m*) est inférieure à la coordonnée y du noeud courant *x*, et soit, si la coordonnée x de la destination *d* est égale à la coordonnée x du noeud courant *x*, ou si le paquet *m* ne peut pas le transit le long de l'axe *X*.

EVENTswitch_control_South ≙
Refines switch_control
where ...
(((prj1(position(x)) = prj1(position(d)))
∨ (prj1(position(x)) > prj1(position(d))
∧ *x* ↦ position ~ (prj1(position(x))-1 ↦ prj2(position(x))) ∉ gr)
∨ (prj1(position(x)) < prj1(position(d))
∧ *x* ↦ position~ (prj1(position(x))+1 ↦ prj2(position(x))) ∉ gr)))
∧ prj2(position(x)) < prj2(position(d))
∧ prj1(position(y)) = prj1(position(x))
∧ prj2(position(y)) = prj2(position(x))-1
...*then*
same actions of switch_control
end

vi. **Data passage from a buffer to a node** La transition d'un paquet *m* à partir d'un port de sortie *op*, à un canal *ch* menant à un noeud de destination *n*.

$EVENT\ buff\_to\_node \triangleq$
$\vdots$
$then$
$chan = chan \cup \{ch \mapsto m\}$
$\wedge outputbuffer = outputbuffer \setminus \{op \mapsto m\}$
end

vii. **Data passage from a node to a buffer** Pour transférer un paquet m à partir d'un canal ch à un noeud connecté n. Cette transition d'un paquet m à partir d'un canal ch à un port d'entrée ip du noeud cible n.

$EVENT\ node\_to\_buff \triangleq$
$\vdots$
$then$
$chan = chan \cup \{ch \mapsto m\}$
$\wedge inputbuffer = inputbuffer \cup \{ip \mapsto m\}$
end

viii. **Disabling of node link** Un noeud désactivé sd n'a pas le droit de communiquer avec ses voisins (échec, etc.), de sorte ce fil à retirer son lien $lnbg$ forment la fermeture du réseau gr.

$EVENT\ Disable \triangleq$
$\vdots$
$then$
$gr = cls(gr \setminus (lnbg \cup lnbg\sim))$
end

ix. **Relink a node** La reconfiguration du réseau est un processus dans lequel les nœuds en état faulty sont réactivées : les liens entre eux et leurs voisins sont restaurés, permettant ainsi des communications et des paquets transferts donc c'est le sens d'ajouter le lien $lnbg$ du nœud n à la fermeture du réseau gr.

$EVENT\ Relink \triangleq$
$\vdots$
$then$
$gr = cls(gr \cup (lnbg \cup lnbg\sim))$
sizeplaces(out, op,places) =
sizeplaces(out, (lnbg∪ lnbg∼)◁ op,availableplc((lnbg ∪ lnbg ∼) ◁ op)
end

(b) ***La théorie WNoC***

Au cours de la modélisation pour ce réseau particulier utilisé NoC, elle doit présenter le mode de reconfiguration pour tout nœud défectueux et pour cette nouvelle contrainte, il ajoute une nouvelle extension de la théorie NoC, cette nouvelle théorie Wireless-NoC contient : Le "**état**" nouveau type de données est utilisé pour représenter l'état de chaque nœud. Les prochains opérateurs sont utilisés pour clarifier davantage la nouvelle théorie qui respecte toutes les propriétés de fiabilités :

WNocStat : créer un ensemble de nœuds avec son état.
ocpy : indique que le nœud est occupé par l'envoi ou la réception de données.
free : mentionne que le noeud est pas occupé.
nodeState : cet opérateur présente l'état de nœud nod spécifique.
packet : pour représenter la nouvelle structure de données qui contient un drapeau flg

pour spécifier le type de données dans le cas de défaillance d'un nœud de cette flg aura la valeur 1.

datatype $state$
constructors $ocpy$, $free$

---

operator $packet$
prefix
args $flg : \mathbb{N}$, $data : \mathbb{P}(T)$
condtition $flg \in 0..1$
definition $data$

---

operator $WSnode$
prefix
args $state$, $nod : \mathbb{P}(S)$
condtition $any \in role$
$\wedge net \in node(any, nd)$
definition net

---

operator $Statenode$
prefix
args $nod : \mathbb{P}(S)$
condtition $any \in role$
$\wedge net \in node(any, nd)$
$\wedge s \in state$
definition s

Ainsi, il peut présenter ce NoC particulier en utilisant définition de la théorie WNoC comme dans la suite :

i. **data sending** Quand une source $Srcnod$ envoie un paquet msg, le paquet doit avoir une valeur de 0 pour l'argument flg et les états des noeuds $Srcnod$, $Desnod$ doit être changé avec la valeur $ocpy$.

ii. **Data receiving** Un paquet est reçu par sa destination, si le paquet a atteint la destination (noeud avec un rôle égal à rcv cas 3 dans le prédicat) .Ce paquet doit avoir une valeur de 0 pour l'argument flg et les états des noeuds $Srcnod$, $Desnod$ doit être changé avec la valeur $free$.

iii. **Data forward** Dans le réseau, un paquet msg transite à partir d'un noeud $Srcnod$ vers un autre nœud $Desnod$ (noeud avec un rôle égal à dst Cas 2 prédicat) .Ce paquet doit avoir une valeur de 0 pour l'argument flg et l'état de le $Srcnod$ de noeud doit être changé avec la valeur $free$ lorsque le $Desnod$ nœud aura la valeur $ocpy$.

$EVENT\ SEND \triangleq$
⋮
then $send = send \cup \{Srcnod \mapsto msg\}$
$\wedge srcstate = ocpy$
$\wedge Desstate = ocpy$
end

$EVENT\ FORWARD \triangleq$
⋮
then $frwd = frwd \cup \{Desnod \mapsto msg\}$
$\wedge srcstate = free$
$\wedge Desstate = ocpy$
⋮
end

$EVENT\ RECEIVE \triangleq$
⋮
then $rcvd = rcvd \cup \{Desnod \mapsto msg\}$
$\wedge srcstate = free$
$\wedge Desstate = free$
⋮
end

Dans le cas optimal ces derniers événements pourraient représenter la communication entre les différents nœuds sinon certains noeuds pourraient être en état d'échec de sorte qu'ils doivent informer les autres nœuds en envoyant un $FlagData$ avec $flg = 1$. (plus de détail voir Flag-data sending)

iv. **Flag-data sending** Dans le cas d'un noeud $Fnod$ Le défaut qui ne peut recevoir un paquet msg, les $FlagData$ de paquets avec une valeur de 1 pour l'argument flg est envoyé si le $Fnod$ de noeud défaillant encore être occupé (valeur $ocpy$).

$EVENT\ Send\_flg\_Data \triangleq$	$send = send \cup \{Fnod \mapsto$FlagData$\}$
$\vdots$	$\wedge srcstate = ocpy$
*where*FlagData$\in$packet(1,$\mathbb{P}$(T))	$\wedge Failstate = ocpy$
$\wedge$srcstate=Statenode(Srcnod)	*end*
$\wedge$Failstate=Statenode(Fnod)	
then	

Lorsque les données de signalisation est reçu (voir Flag-data receiving), l'un des noeuds correctes prennent la mission de reconfiguration (voir Config-data receiving) pour les noeuds défectueux après avoir vérifié ces règles suivantes :

- Chaque noeud est pas un noeud défaillant,
- Chaque noeud a les ressources matérielles pour mettre en œuvre nœud IP,
- Chaque noeud a le paquet Bitstream de configuration IP,
- Chaque noeud ne pas être occupé par une priorité.

v. **Flag-data receiving** Un paquet $FlagData$ est reçu par le noeud $Srcnod$, si l'on envoie à partir d'un noeud défaillant $Fnod$.

$EVENT\ Receive\_flg\_Data \triangleq$	$\wedge srcstate = free$
$\vdots$	$\wedge Failstate = free$
then	*end*
$rcvd = rcvd \cup \{Desnod \mapsto msg\}$	

vi. **Data-Config sending** Dans le cas d'un noeud $Srcnod$ peut vérifier les règles de la possibilité de reconfiguration (décrit précédemment), il envoie des données de configuration $Bitstream$ au noeud défaillant $Fnod$.

$EVENT\ Send\_Conf\_Data \triangleq$	$send = send \cup \{Srcnod \mapsto$Bitstream$\}$
$\vdots$	$\wedge srcstate = free$
where	$\wedge Failstate = ocpy$
Bitstearm$\in$packet(0,$\mathbb{P}$(T))	*end*
$\wedge$FlagData$\in$packet(1,$\mathbb{P}$(T))	
then	

vii. **Data-Config receiving** Un paquet $FlagData$ est reçu par le noeud $Srcnod$, si l'on envoie à partir d'un noeud défaillant $Fnod$.

$EVENT\ Receive\_Conf\_Data \triangleq$	$\wedge srcstate = free$
$\vdots$	$\wedge Failstate = free$
then	*end*
$rcvd = rcvd \setminus \{Fnod \mapsto$Bitstream$\}$	

(c) ***La théorie de coloration*** En raison de la gestion des nœuds échecs et la voie de l'auto-récupération, on peut dire :

- Les algorithmes de graphes de coloration [44] peuvent être utilisés pour contrôler un ensemble de nœuds : Il peut y avoir deux nœuds qui ont le même travail, mais deux nœuds adjacents ne peuvent même pas fixer le nœud défaillant en même temps [17].
- Extension de Math est une bibliothèque standard fournit la fermeture [353] comme une théorie qui est presque similaire à notre architecture sans fil NoC mais il faut ajouter les règles de coloration dans le contexte de couvrir tous les bienséances Courant de notre théorie des graphes.

Lors des événements de raffinement qui sont les variables d'application représentés par une théorie des graphes et leurs variables d'environnement d'exécution doivent être manipulés dans le même modèle en introduisant la théorie VHDL qui pourrait être utilisé dans tous les modèles d'événement comme de nouvelles variables VHDL représentés avec Event-B. Donc, la coloration couvrira tous les états possibles pour les événements qui peuvent faire des nœuds.

datatype $Colors$
constructors
Yellow,
Red,
Green,
Blue,

none

---

operator $Colorenode$
prefix
args $c : Colors, nod : \mathbb{P}(S)$
definition nod

Donc, un nœud qui ne peut pas envoyer et recevoir des données peut être marqué avec Bleu, Jaune, Vert, Rouge et incolorés a même réussi le multi-échec de nœuds qui viaduc plus de 4 nœuds en comptant comme nœud dans la liste d'attente pour être dans la prochaine couleur en raison de la couleur disponible.

EVENT $colored\_faulty \widehat{=}$
⋮
where clr∈Colors∧$count \in \mathbb{N}$
∧$count \leq 4$
⋮
then
$colored = colored \cup \{Fnod\}$
∧$colored(Desnod) = clr$
∧$count = count + 1$
∧$has\_colored(Desnod) =$
$has\_colored \cup \{Fnod\}$
end

EVENT $faulty\_waiting\_color \widehat{=}$
⋮
where clr∈Colors∧$Count \in \mathbb{N}$
∧$count > 4$
⋮
then
$wait\_to\_be\_colored = wait\_to\_be\_colored \cup$
$\{Fnod\}$
∧$count = count + 1$
∧$colored(Desnod) = clr$
end

i. **Flag-data sending** Dans le cas d'un noeud $Fnod$ Le défaut qui ne peut pas recevoir un paquet, il doit colorer le nœud qui envoie un $FlagData$ en rouge.

EVENT $Send\_flg\_Data \widehat{=}$
⋮
then
$Send = Send \cup \{Fnod \mapsto FlagData\}$
∧$colored(Fnod) = clr$
∧$has\_colored = has\_colored \cup \{Fnod\}$
end

ii. **Data-Config sending** Le Bitstream-paquet (Bitstream) est reçu par un noeud défaillant (Fnod), si elle est l'envoi d'un IP de réconfiguration $Rfcgnod$ dans la mémoire de buffer, puis pour cette raison les deux nœuds $Fnod$ et $Rcfgnod$ doivent être incolorés.

$EVENT\ Send\_Conf\_Data \hat{=}$
$\vdots$
$then$
$rcvd = rcvd \setminus \{Fnod \mapsto bitstream\}$
$\wedge colored(Fnod) = none$
$\wedge has\_colored = has\_colored \cup \{Fnod\}$
end

(d) ***La theorie VHDL***

La VHDL_th Théorie utilisée au cours de ce réseau de modélisation auto-organisé est composé d'une entité et un ensemble d'architectures qui contient un ensemble de variables, l'entité contient des ports dans la direction en dehors ou in_out, à partir de ce que les deux opérateurs sont créés ($arch\_decl$ et $ports\_decl$) quand on a un paramètre de type de données pio, dans cette étude de cas de VHDL, nous avons eu des signaux std_logic et les std_logic_vectors, le dernier doit avoir la nouvelle théorie pourrait être utilisé dans le VHDL_th, il est une structure similaire d'un tableau présenté dans [355, 356].

$THEORY\ VHDL\_th$

$DATATYPES$
$pio \hat{=} in, out, inout$

$OPERATORS$
$\bullet arch\_decl :$
$arch\_decl(s : \mathbb{P}(T))$
direct definition
$arch\_decl(s : \mathbb{P}(T)) \hat{=} s$

$\bullet port\_decl :$
$port\_decl(p : pio, s : \mathbb{P}(T))$
direct definition
$port\_decl(p : pio, s : \mathbb{P}(T)) \hat{=} s$

$\bullet std\_logic : std\_logic$
direct definition

$std\_logicl \hat{=} 0..1$

$\bullet vector :$
$vector(s : \mathbb{P}(T))$
direct definition
$vector(s : \mathbb{P}(T)) \hat{=} \{n, f \cdot n \in \mathbb{N}$
$\wedge f \in 0..(n-1) \rightarrow s | f\}$

$\bullet std\_logic\_vector :$
$vector(length : \mathbb{N}, s : \mathbb{P}(T))$
well-definedness condition
$length \in \mathbb{N}$
$finite(s)$
direct definition
$std\_logic\_vector(length : \mathbb{N},$
$s : \mathbb{P}(T)) \hat{=}$
$\{v | v \in vector(s) \wedge card(s) = length\}$

L'application de la théorie de VHDL sur système WSNoC dans l'Event-B. La variable en VHDL doit avoir un type et dans cette structure par exemple le type entier est représenté comme un opérateur et veiller à ce que ne pouvait pas être plus $INT\_MAX = 2^{32}$ et toutes les opérations qui utilisent ce type de variable ne doit jamais déborder cette valeur de sorte que le l'obligation de preuve garantira l'erreur non-débordement pour des variables entières dans le code VHDL.

Au cours de la modélisation de notre théorie VHDL qu'il va introduire dans le modèle Event-B l'ensemble des opérations créées dans nos nouvelles théories et de prendre certains cas comme les opérations de VHDL (+, *, -, /), ou l'affectation de variables en VHDL , aussi quelques preuves devaient être rejetées au cours de cette modélisation de WSNoC sera un bon début pour la génération de code après avoir vérifié la définition du bien de tout le système.

Annexe E

Liste des abréviations

AD Atomicity Decomposition. 97, 103

ADL Architecture Description Language. 28, 97, 99, 103

ADT Abstract Data Type. 92, 96, 97

AECB Atomic Energy Control Board. 31

ASCII American Standard Code for Information Interchange. 46

ASIC Application-Specific Integrated Circuit. 21

ASM Abstract State Machine. 40

AST Abstract Syntax Tree. 63, 67

BIP Behaviour Interactive Property. 36, 37

CTL Computational Tree Logic. 39

CU Communication Unit. 19

DSF Dependable Software Forum. 41

DSP Digital Signal Processor. 21

FPGA Field Programmable Gate Array. 4, 12, 15, 16, 19, 21, 23, 44

GCJ GNU Compiler for Java. 27

GPS Global Positioning System. 12

GQAM Generic Quantitative Analysis Modeling. 27

HDL Hardware Description Langages. 25

IDE Interactive Development Environment. 62, 63, 66

IP Intellectual Property. 14, 21, 127

ISS Instruction Set Simulator. 28

JML Java Modellig Language. 7, 71, 108

JSD Jackson Structure Diagram. 97

JVM Java Virtual Machine. 25, 29

LCF Logic of Computable Functions. 70, 106–108

LUT Look Up Table. 21, 23

MARTE Modeling and Analysis of Real Time and Embedded systems. 27, 29

ML Mono-Lemma prover. 7, 43, 66, 72, 109, 110

MPSoC Multi-Processor System on Chips. 4, 13

MPU Microprocessing Unit. 21

MUI Modelling User Interface. 63

NFP Non Functional Property. 27

NI Network Interface. 14

NoC Network on Chips. 4, 8–10, 12–14, 16–21, 23, 24, 44, 92, 106, 113, 122–124, 126, 129, 134, 135, 139, 140

OCP Open Core Protocol. 14

PC Personal Computer. 12

PE Processing Element. 13, 17, 19

PM Proof Manager. 67

PO Proof Obliagtions. 71, 72, 77, 139, 140

POG Proof Obliagtions Generator. 63

POM Proofs Obligations Manager. 63, 67

PP Predicate Prover. 7, 66, 71, 72, 109, 110

PUI Proving User Interface. 63

QoS Quality of Service. 13, 18, 19

RISC Reduced Instruction Set Computing. 20

RTEM Real-Time and Embedded Modeling. 27

RTL Register Transfer Level. 25

RTOS Real-Time Operating System. 13, 29

SC Static Checker. 63, 119

SCI Scalable Coherent Interface. 14

SCR Software Cost Reduction. 35

SEQP SEQuent Prover. 63

SLDL System Level Design Language. 25, 29

SoC System on Chips. 12–14, 17

SONoC Self-Roganised Network on Chips. 9, 21, 81, 122

SRAM Static Random Access Memory. 21

TCRM Time, Concurrency and Resources Modeling. 27

TLA+ Temporal Logic of Actions. 39, 79

UML Unified Modeling Language. 27, 29, 79

VDM Vienna Development Method. 35, 40, 41, 86, 87, 108

VHDL Very High Speed Integrated Circuit Hardware Description Language. 4, 5, 8–10, 25, 28, 29, 44, 80, 81, 95, 106, 122, 128, 129, 134, 135, 137, 139, 140

WD Well-Definedness. 8, 118

WNoC Wireless Network on Chips. 122

XML eXtensible Markup Language. 63, 64, 144

Approche à base d'opérateurs pour la validation formelle de systèmes micro-électroniques orientés NoC.

Résumé : Les travaux de recherche menés par cette thèse s'effectuent dans le cadre des projets au niveau du laboratoire LIM (Laboratoire de la recherche Informatique et Mathématique Université de Ibn Khaldoun de Tiaret). Ils s'inscrivent dans l'utilisation de la méthode formelle Event-B et l'outil de développement RODIN en vue d'effectuer la réalisation des systèmes microélectroniques embarqués. L'idée à développer consiste à enrichir le processus de conception d'un système embarqué par la notion de preuve formelle délivrée par l'outil RODIN. Plus précisément, il s'agit de développer un flot de conception permettant de générer de manière incrémentale (à base des opérateurs de raffinement qui manipulent des théories) et prouvée (couvre le code VHDL et les propriétés de système) une architecture microélectronique synthétisable. L'application durant ce projet et le développement d'un système de communication à base des réseaux sur puce tolérant aux fautes pour les systèmes multiprocesseurs sur puce (Multi-Processor System on Chip –MPSoC) à base de technologie FPGA. L'objectif principal de la thèse est d'introduire la notion d'opérateurs au niveau du concept de raffinage utilise dans la méthode Event-B. Le domaine d'applications concerne des systèmes micro-électroniques notamment les NoC (Network-On-Chip) développés en utilisant les technologies FPGA pour créer des systèmes auto-organisés. Le travail de recherche à mener consiste à introduire un ensemble d'opérateurs algébriques dans le processus de spécification qui commence à partir d'un modèle abstrait de haut niveau vers un modèle concret qui est représenté par une description VHDL ou RTL. L'objectif est de mettre en place un ensemble de règles formelles permettant de générer automatiquement ou semi automatiquement certaines actions de raffinages qui se terminent par l'étape de la génération de code. La contribution de cette thèse consiste à commander le processus de raffinage et proposer un ensemble de choix aux concepteurs de circuits comme une large solution permettant d'optimiser l'architecture cible à base de la technologie FPGA.
Les systèmes à base NoC : architectures à base de FPGA tolérantes aux fautes, seront adoptés comme exemple d'explication plus qu'une étude de cas de validation de l'approche proposés. D'un autre coté plusieurs formalismes seront explorés dans la littératures tels que BIP, CSP pour avoir une formalisation fine de la notion de raffinages. L'introduction de la notion d'opérateurs de raffinement (créer, enrichir, restreindre, renommer) au niveau de Event-B par l'intermédiaire des théories implique la révision du système de façon plus générique, par l'ensemble des obligations de preuves développées au niveau de l'outil RODIN dédié à la méthodologie Event-B.

Mots clés : Event-B, BIP, Génération de code, NoC, Raffinement, Opérateurs, Théories, RODIN.

Operator-based approach for the formal validation of NoC-oriented microelectronic systems.

Abstract : The research conducted by this thesis is one of the LIM research laboratory projects (Computer Research Laboratory and Mathematics University of Ibn Khaldoun Tiaret). This work is part of the use of formal method Event-B and the RODIN development tool to ease the realization of embedded microelectronic systems. The idea to develop is to enrich the design process of an embedded system checked by the notion of formal proof issued by the RODIN tool. More specifically, it is to develop a design flow to generate incrementally (refinement based on operators that are handling theories) and proven (covers VHDL code and system properties) a synthesizable microelectronics architecture. The application for this project presented on the development of a SoC-based network communication system for fault tolerant multiprocessor systems on chip (MP-SoC) technology-based FPGA. The main aim of the thesis is to introduce the concept of operators in the refining concept used in the Event-B method. The applications field concerns microelectronic systems including NoC (Network-On-Chip) developed using FPGA technology to create self-recover systems. Carrying out this research work is to introduce a set of algebraic operators in the specification process that starts from a high level abstract model to a concrete model which is represented by one VHDL or RTL description. The aim is to develop a set of formal rules to automatically or semi-automatically generate certain actions of refining processes till the last step of code generation. The contribution of this thesis is to control the refining process and propose a set of choices for circuit designers as a wise way for optimizing the FPGA-based technology of a target architecture.
NoC-based systems represent FPGA-based fault-tolerant architectures ; such systems will be adopted as an example of explanation than a case study of validation of the proposed approach. On the other hands several formalisms will be explored in the literature such as BIP, CSP to have a well-studied formalization of the concept of refining processes. The introduction of the concept of refinement operators (create, enrich, restrict, rename) in Event-B through theories involves the more generically review of system developed by the generation of the set of proof obligations in the tool RODIN dedicated to the methodology Event-B.

Keywords : Event-B, BIP, Code generation, NoC, Refinement, Operators, Theories, RODIN.

النهج القائم على المعاملات للتحقق من جودة تصميم أنظمة الـNoC للإلكترونيات الدقيقة المدمجة على الرقاقة.

ملخص

البحث المقدم من خلال هذه الأطروحة هو واحد من المشاريع المباشَرة في مختبر الأبحاث LIM (مختبر البحوث في الإعلام الآلي و الرياضيات التابعة لكلية الرياضيات والإعلام الآلي بجامعة ابن خلدون- تيارت)، هذا العمل يبين المنفعة الكامنة من استخدام واحد من الأساليب الشكلية وهو طريقة: الـ Event-B المرفقة بالوسيلة البرمجية مفتوحة المصدر"RODIN " لتسهيل مرحلة التحقق من جودة تصميم أنظمة الإلكترونيات الدقيقة المدمجة، فالمعنى المجرد لتطوير هذه الطريقة هو إثراء لعملية التصميم لمثل هذا النظام وتعتبر جزءا لا يتجزأ من مهمة فحصها بفضل آلية البراهين المثبتة من قبل الأداة RODIN المزودة بكم من البرامج المساعدة أو الإضافات، وبشكل أكثر تحديدا، فحوى فكرتنا هو تطوير منهج التصميم للحصول تدريجيا (على أساس عامل توليد الأفكار الذي يتعامل مع النظريات) وبصفة مُثْبَتة (يغطي لغة الكود VHDL لخصائص النظام) على التركيب الخاص بهندسة الالكترونيات الدقيقة لأي نظام، يوضّح فكرة هذا المشروع مثال هدفه تطوير نظام شبكة الاتصالات القائمة على نظام الشبكة على رقاقة NoC لأنظمة متعددة المعالجات المتسامحة مع الأخطاء المدمجة على رقاقة MPSoC والقائمة على تكنولوجية الـ FPGA فالهدف الرئيسي من هذه الرسالة إذن هو إدخال مفهوم "المعامل" أو "عامل" في مفهوم توليد النماذج المستخدمة في الطريقة Event-B.وكما ذكرنا آنفا مجال تطبيق فكرتنا يأخذ نظم الإلكترونيات الدقيقة بما في ذلك نظام الـ NoC (الشبكة على رقاقة) ثم تطويرها باستخدام تقنية FPGA لتحقيق أنظمة "ذاتية إعادة التنظيم"، أمّا الجانب التطبيقي من البحث فيسعى إلى تقديم مجموعة من المعاملات الجبرية في عملية التوصيف التي تبدأ من نموذج مجرد وسطحي على مستوى عال من الوصف إلى نموذج متماسك معمّق التفصيل الذي يمثله "وصف بلغة الكود VHDLأو بنموذج الـRTL " بهدف وضع مجموعة من القواعد الرسمية تنشئ إجراءات معينة تولّد النماذج في مرحلة التوصيف تلقائيا أو شبه تلقائيا الذي يشمل مرحلة إنتاج الأكواد، وعليه فالمغزى الرئيسي من طرح الإشكالية في هذه الرسالة هو إمكانية السيطرة على آلية عملية توليد الأفكار المترجمة إلى نماذج واقتراح مجموعة من البراهين لمصمّمي الدوائر الالكترونية كخيار واسع لتحسين هندسة أيّ نظام حتى ذلك القائم على تكنولوجيا الـFPGA .

ستُعتَمد النظم القائمة على الـ NoCوالبنى المتسامحة مع الخطأ المستندة إلى تقنية FPGA كمثال لشرح و دراسة إمكانية المصادقة على النهج المقترح في هذه الأطروحة ، وذلك بعد استكشاف العديد من الأساليب الشكلية المعتمدة من قبل مصممي الأنظمة نأتي على ذكرBIP ، CSP..وغيرها ضمن فصول هذه المذكرة لإضفاء طابع البحث الأكاديمي خلال دراسة مفهوم عمليات توليد النماذج، بغيةعرض مفهوم عاملات(**إنشاء، إثراء، حصر، تسمية**) توليد النماذج المُمثّلة بنظريات في الـ Event-Bالتي تنطوي على مراجعة أكثر نوعية للنظام بفضل جميع متطلبات البراهين المنجزة من قبل الأداة RODIN المخصصة للمنهجية Event-B .

الكلمات المفتاحية: Event-B، BIP، إنتاج الأكواد ، نظام الشبكة على رقاقة NoC، عاملات، توليد، نظريات ،RODIN.

Approche à base d'opérateurs pour la validation formelle de systèmes micro-électroniques or ientés NoC.

Résumé :

Les travaux de recherche menés par cette thèse s'effectuent dans le cadre des projets au niveau du laboratoire LIM (Laboratoire de la recherche Informatique et Mathématique Université de Ibn Khaldoun de Tiaret). Ils s'inscrivent dans l'utilisation de la méthode formelle Event-B et l'outil de développement RODIN en vue d'effectuer la réalisation des systèmes micro-électroniques embarqués. L'idée à développer consiste à enrichir le processus de conception d'un système embarqué par la notion de preuve formelle délivrée par l'outil RODIN. Plus précisément, il s'agit de développer un flot de conception permettant de générer de manière incrémentale (à base des opérateurs de raffinement qui manipulent des théories) et prouvée (couvre le code VHDL et les propriétés de système) une architecture microélectronique synthétisable. L'application durant ce projet et le développement d'un système de communication à base des réseaux sur puce tolérant aux fautes pour les systèmes multiprocesseurs sur puce (Multi-Processor System on Chip–MPSoC) à base de technologie FPGA. L'objectif principal de la thèse est d'introduire la notion d'opérateurs au niveau du concept de raffinage utilise dans la méthode Event-B.Le domaine d'applications concerne des systèmes micro- électroniques notamment les NoC (Network-On-Chip) développés en utilisant les technologies FPGA pour créer des systèmes auto-organisés.

Le travail de recherche à mener consiste à introduire un ensemble d'opérateurs algébriques dans le processus de spécification qui commence à partir d'un modèle abstrait de haut niveau vers un modèle concret qui est représenté par une description VHDL ou RTL.

L'objectif est de mettre en place un ensemble de règles formelles permettant de générer automatiquement ou semi automatiquement certaines actions de raffinages qui se terminent par l'étape de la génération de code. La contribution de cette thèse consiste à commander le processus de raffinage et proposer un ensemble de choix aux concepteurs de circuits comme une large solution permettant d'optimiser l'architecture cible à base de la technologie FPGA.

Les systèmes à base NoC : architectures à base de FPGA tolérantes aux fautes, seront adoptés comme exemple d'explication plus qu'une étude de cas de validation de l'approche proposés.D'un autre coté plusieurs formalismes seront explorés dans la littératures tels que BIP, CSP pour avoir une formalisation fine de la notion de raffinages. L'introduction de la notion d'opérateurs de raffinement (créer, enrichir, restreindre, renommer) au niveau de Event-B par l'intermédiaire des théories implique la révision du système de façon plus générique, par l'ensemble des obligations de preuves développées au niveau de l'outil RODIN dédié à la méthodologie Event-B.

Mots clés : Event-B, BIP, Génération de code, NoC, Raffinement, Opérateurs, Théories, RODIN.