Numéro d'ordre : 4545

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Mohamed TOUNSI**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : **INFORMATIQUE**

Preuves d'algorithmes distribués par raffinement

Directeur de thèse : **Mohamed MOSBAH**

Soutenue le 04 Juillet 2012
Devant la Commission d'Examen

JURY

MERY	Dominique	Professeur	Président
AIT-AMEUR	Yamine	Professeur	Rapporteur
COUVREUR	Jean-Michel	Professeur	Rapporteur
METIVIER	Yves	Professeur	Examinateur
MOSBAH	Mohamed	Professeur	Directeur de thèse

Remerciements

Ce travail n'aurait pu aboutir sans l'aide et le soutien de nombreuses personnes, je leurs exprime ici mes sincères remerciements et ma profonde reconnaissance. Je tiens à remercier particulièrement :

Monsieur Mohamed MOSBAH pour son encadrement, son soutien et sa disponibilité. Ses conseils, ses suggestions de lecture, ses commentaires, ses corrections et ses qualités scientifiques ont été très précieux pour mener à bien ce travail.

Monsieur Dominique MERY, pour son suivi et l'attention qu'il y a porté et sa disponibilité. Ses remarques, ses conseils et ses qualités scientifiques m'ont permis d'améliorer la qualité de ce mémoire.

Mes remerciements les plus distingués sont adressés aux membres du jury qui m'ont fait l'honneur de bien vouloir accepter d'évaluer ce travail de recherche.

Je n'oublie pas les membres du laboratoire LaBRI, mes amis et les agents administratifs du LaBRI et de l'enseirb.

Je remercie les membres de ma famille pour leur incessant soutien et plus particulièrement mes parents qui m'ont guidé sur le chemin des études. La liste n'est pas exhaustive : Je remercie toutes les personnes, des amis aux inconnus, qui, même au cours d'une rencontre fugace, m'ont fait découvrir le plaisir d'apprendre.

Table des matières

Table des figures

Table des figures

Liste des tableaux

Liste des tableaux

Introduction générale

Dans notre vie quotidienne, les systèmes informatiques sont de plus en plus présents. Depuis les architectures centralisées, en arrivant aux architectures réparties, ces systèmes ne cessent pas d'évoluer tant au niveau logiciel que matériel. Les dernières décennies ont été marquées par une révolution dans le domaine de l'informatique distribuée. Le commerce électronique, la recherche d'informations, la gestion de réseaux ou le travail coopératif sont des exemples type d'applications développées sous ces systèmes. Elles sont devenues des outils indispensables pour plusieurs secteurs notamment la finance, l'économie, etc.

Couramment, un système distribué est défini par une collection d'entités de calcul autonomes qui communiquent ensemble pour accomplir une tâche commune. Dans un tel système, une entité ne peut interagir qu'avec l'ensemble de ses voisins directs. Cette définition englobe les réseaux géographiquement étendus et également les réseaux locaux de multiprocesseurs [1, 2]. Bien que les systèmes centralisés connaissent une certaine maturité que ce soit sur le niveau de conception ou celui de la preuve, les systèmes distribués requièrent encore beaucoup d'efforts pour maîtriser leur complexité. Car généralement, les systèmes distribués sont composés d'un grand nombre d'éléments qui sollicitent souvent des coordinations et des interactions assez complexes. Par conséquent, leur conception et la preuve de leur correction deviennent des tâches très difficiles, et la moindre erreur peut avoir des conséquences extrêmement graves. Afin de réduire cette complexité, il est de plus en plus admis que les modèles formels jouent un rôle dans la spécification, l'analyse et la preuve de correction des systèmes et des applications répartis. Ceci continue de stimuler une intense activité de recherche autour des modèles formels pour l'algorithmique distribuée qui constitue le fondement des applications et des systèmes distribués. Plusieurs travaux ont été proposés dans la littérature pour ramener l'étude d'un algorithme distribué à l'étude du modèle qui le code [3, 4].

Les systèmes de réécriture de graphes, et plus généralement les calculs locaux [5], constituent un modèle formel solide pour exprimer à un haut niveau d'abstraction les algorithmes distribués. Dans ce modèle, un tel système est représenté par un graphe étiqueté, connexe, et non orienté, où les nœuds correspondent aux processeurs et les arêtes aux ca-

naux de communication. L'étiquette d'un nœud [resp. une arête] code l'état du processeur [resp. lien de communication] correspondant. Un calcul sur un système distribué (ou un algorithme distribué), qui consiste donc à faire évoluer ses états, peut être décrit par des règles de réécriture d'étiquettes de graphes. Chaque règle, définie par un graphe connexe associé à deux étiquetages, traduit un pas élémentaire du calcul effectué sur une portion locale du graphe. Cette représentation de haut niveau a permis d'étudier, d'analyser et d'obtenir des résultats nouveaux pour plusieurs problèmes de l'algorithmique distribuée comme l'élection, le calcul d'un arbre recouvrant, la détection de la terminaison ou bien la reconnaissance de certaines propriétés du réseau. De plus, cette approche permet d'offrir un codage unifié et relativement intuitif pour prouver les algorithmes distribués. Ceci est fait par des preuves de terminaison de systèmes de réécritures et des preuves de validité d'invariants. Le système de calculs locaux constitue le cadre théorique de cette thèse.

Preuve formelle des algorithmes distribués

Diverses méthodes et approches formelles ont été proposées pour répondre à un besoin de modélisation à la fois des différentes composantes d'un système distribué et de leurs interactions respectives. Dans ce contexte, plusieurs travaux se sont focalisés sur la preuve de correction des algorithmes distribués. Ces travaux peuvent être structurés en deux grandes approches formelles : La première a pour objectif de montrer, en utilisant une famille de techniques inhérentes à la méthode de vérification *Model Checking* [6], qu'un algorithme distribué satisfait bien sa spécification. La deuxième approche, fondée sur des techniques de preuves, consiste à formaliser mathématiquement l'algorithme distribué et à démontrer sa correction par des théorèmes.

La vérification par *Model Checking* consiste à construire un modèle fini du système et à vérifier si une propriété donnée peut se reproduire. Cette approche se manifeste par une exploration automatique et exhaustive de l'espace d'états du modèle. En général, les modèles sont spécifiés avec un langage de spécification abstrait, comme *Promela* pour Spin [7], ou TLA+ pour TLC [3]. A titre d'exemple, nous citons le travail de Garavel H. et al. [8] qui appliquent l'approche *Model Checking* pour vérifier les algorithmes distribués d'élection dans un réseau ayant comme topologie l'anneau unidirectionnel. Dans ce travail, les auteurs ont étudié la robustesse de cette classe d'algorithmes en présence de canaux de communication et/ou de machines non fiables. Ces algorithmes ont été formellement décrits à l'aide du langage de spécification LOTOS [9] et vérifiés grâce à la boite à outils CADP [1].
Bien qu'elle soit simple et ayant une tendance à être exécutée automatiquement, cette première approche soulève un fameux problème qui est l'explosion des états. Ce problème est notamment exacerbé lorsque l'approche est appliquée aux logiciels. Ceci est dû à la quantité de détails dans un logiciel qui est naturellement plus importante que les modèles abstraits.

1. http ://www.inrialpes.fr/vasy/cadp

La deuxième approche est basée sur la preuve des théorèmes. La première étape dans cette approche consiste à décrire le système et toutes ses propriétés avec une logique mathématique. La deuxième étape est de prouver les propriétés de correction avec des théorèmes qui se basent sur les axiomes du système modélisé. Le plus souvent, la preuve des théorèmes est faite par des assistants de preuve (automatiques ou semi-automatiques) comme ISABELLE/HOL [10, 11], COQ [12], PVS [13], etc. Dans cette thèse, nous avons choisi les techniques de preuve par théorèmes pour être en mesure de démontrer les propriétés génériques de l'algorithmique distribuée. Parmi les travaux de preuve par théorème, des algorithmes codés par les calculs locaux, nous citons les deux principaux travaux suivants :

Le premier travail de P. Castéran et al. [14] propose une formalisation en Coq [12] du modèle de calculs locaux. Dans ce travail, les auteurs ont développé une bibliothèque qui décrit toute la théorie inhérente à la preuve formelle des systèmes de calcul locaux. En d'autres termes, cette bibliothèque comprend des preuves générales de réalisation de tâches par des systèmes de ré-étiquetage. Aussi, elle englobe des lemmes génériques portant sur des classes de systèmes de ré-étiquetage associées aux divers types de synchronisation et de modes de détection de la terminaison. Ainsi, ce premier travail est complémentaire à nos travaux de thèse car il adopte une approche ascendante (Bottom-up) alors que la notre peut être vue comme descendante (Top-down). En effet, avec une approche ascendante, la preuve est faite depuis un niveau local (on ne considère que la vision locale des nœuds) pour vérifier la correction du niveau global (l'objectif de l'algorithme).

Le deuxième travail, inscrit dans le cadre du projet RIMEL [2], est proposé par D. Méry et D. Cansell [15]. Ce travail montre, à travers des études de cas, l'efficacité de la méthode *B événementiel* et ses techniques de preuve dans le développement de systèmes distribués corrects par construction. En d'autres termes, les auteurs ont prouvé qu'un système obtenu après une suite de raffinements successifs est conforme à sa spécification initiale. En outre, leurs travaux visent à systématiser des développements guidés par la relation de raffinement pour assurer le maintien de propriétés entre un modèle abstrait et un modèle concret. Les algorithmes, ainsi développés, ont été tous validés à l'aide de l'outil Rodin. Ces travaux sont considérés comme le point de départ de nos études de recherches dans cette thèse. D'une part, nous considérons que toutes nos contributions sont complémentaires à ce deuxième travail, car elles s'inscrivent dans le même objectif tracé par le projet RIMEL. D'autre part, nos travaux de recherches sont des généralisations des travaux de D. Méry et D. Cansell [15], car nous nous intéressons aux preuves de classes d'algorithmes distribués. De ce fait, les chapitres de cette thèse vont insister sur la combinaison entre d'une part la méthodologie formelle de construction d'algorithmes distribués corrects validés par la méthode *B événementiel*, et d'autre part les calculs locaux comme un puissant outil de codage et de preuve d'algorithmes distribués.

2. http ://rimel.loria.fr

Objectifs de la thèse

L'objectif de cette thèse est d'étudier et de développer, dans ce contexte, un environnement de preuve pour les algorithmes distribués codés par les calculs locaux. Il s'agit de définir une méthodologie générale de preuve en s'appuyant sur des assistants de preuve pour automatiser la vérification et la validation de ces preuves. Également, l'ambition de cette thèse est de proposer des propriétés génériques relatives à une classe de systèmes de réécriture. A cet effet, nous utilisons la méthode *B événementiel* et la plate-forme Visidia. Dans ce contexte, le projet RIMEL nous fournit un cadre idéal d'expérimentation et de confrontation aux besoins industriels pour les systèmes distribués. En résumé, l'objectif de nos travaux de recherche consiste à :

1. Étudier l'existant des travaux liés aux techniques formelles et aux algorithmes distribués.
2. Caractériser de façon formelle et incrémentale une démarche générale de preuve d'algorithmes distribués.
3. Valider des solutions proposées dans le cadre d'un patron et implémenter un outil de preuve d'algorithmes distribués.

Contribution et structure de la thèse

Le travail de recherche élaboré depuis le début de cette thèse, a fait l'objet de six chapitres répartis comme suit :

Le premier chapitre introduit principalement le cadre théorique de la thèse. Il présente les modèles de calculs locaux, et les calculs distribués soutenus par ces modèles. Il décrit la plate-forme Visidia qui permet d'implémenter, de simuler, de visualiser et d'expérimenter les algorithmes distribués codés par des calculs locaux.

Le deuxième chapitre présente la méthode *B événementiel*. Dans ce chapitre, nous donnons un aperçu sur le langage *B événementiel* ainsi que son assistant de preuve automatique *Rodin*. Nous montrons que cette méthode s'inscrit complètement dans l'approche *correct-par-construction*. Nous clôturons ce chapitre par un état de l'art des principaux travaux réalisés pour prouver la correction des algorithmes distribués avec la méthode *B événementiel*.

Le troisième chapitre présente notre première contribution qui consiste à proposer un patron de spécification formelle d'algorithmes distribués codés par les calculs locaux. Ce patron consiste en une description générale et générique des solutions possibles à des problèmes de spécification et de preuves d'algorithmes distribués. Nous illustrons ce patron par deux exemples d'algorithmes distribués en l'occurrence le calcul d'arbre recouvrant et la coloration d'un anneau. A travers ce patron, nous soulignons l'importance du raffinement dans la diffusion de la complexité des preuves et la validation de l'intégration des

invariants du système.

Notre deuxième contribution est présentée au quatrième chapitre. Dans ce chapitre nous proposons une approche qui introduit en plus du raffinement, la décomposition pour surmonter la complexité du développement formel des algorithmes distribués. Ainsi, le raffinement et la décomposition peuvent être utilisés simultanément pour prouver la correction des algorithmes distribués. L'approche proposée se limite aux algorithmes distribués codés par le modèle de calculs locaux de type handshake. Notre approche simplifie la preuve en réutilisant certaines parties déjà prouvées et rend la spécification plus lisible. Nous illustrons notre approche par un exemple de calcul d'arbre recouvrant avec détection de la terminaison globale. L'algorithme sera décomposé en deux sous-algorithmes : le calcul d'arbre recouvrant et le Dijkstra-Scholten.

Notre troisième contribution sera décrite dans le cinquième chapitre. Elle porte sur la proposition d'une approche de preuve de correction d'algorithmes de synchronisation. Cette approche développe graduellement un algorithme de synchronisation depuis un niveau abstrait jusqu'à un niveau concret probabiliste. Les premiers niveaux sont spécifiés et prouvés par la méthode *B événementiel* tandis que le dernier niveau est implémenté et vérifié par un model-checker quantitatif appelé Prism. Nous illustrons cette approche avec un nouvel algorithme de synchronisation de type handshake que nous proposons. Nous montrons à l'aide d'une étude analytique et expérimentale que notre nouvel algorithme est plus performant par rapport aux algorithmes qui existaient déjà. Concrètement, nous montrons que notre algorithme est tolérant aux pannes, accomplit des synchronisations avec un nombre minimal de tentatives, et réduit le nombre de messages transmis sur le réseau.

Notre dernière contribution fait l'objet du sixième chapitre. Elle consiste à proposer une nouvelle approche, appelée $B2Visidia$, qui permet de simuler une spécification *B événementiel* d'un algorithme distribué. Cette approche est implémentée par un outil appelé aussi $B2Visidia$ qui permet de générer automatiquement un code Java à partir d'une spécification *B événementiel* d'un algorithme distribué codé par les calculs locaux. Le code généré est destiné à être exécuté seulement sous la plate-forme $Visidia$. À cette fin, nous allons définir un langage de spécification propre à $B2Visidia$ et basé sur celui du *B événementiel*. Le nouveau langage peut spécifier des algorithmes codés par les calculs locaux. Ainsi, ce travail montre qu'il est possible de spécifier formellement et de mettre en œuvre une grande famille d'algorithmes distribués à l'aide d'un cadre formel commun.

Enfin, ce document s'achève par une conclusion générale et quelques perspectives de cette thèse.

Liste des tableaux

Chapitre I

Calculs Locaux : un formalisme pour décrire les algorithmes distribués

1 Introduction

Dans des systèmes distribués, plusieurs dimensions de variabilité peuvent cœxister. Il s'agit notamment de la topologie du réseau, les mécanismes de communication inter-processus, les mécanismes de sécurité, etc. Ces dimensions ont permis aux systèmes distribués de gagner, au fils des années, de plus en plus d'importance, mais hélas elles ont rendu de plus en plus complexe l'exercice de la modélisation et de la preuve. Plusieurs travaux ont été proposés dans la littérature pour ramener l'étude d'un algorithme distribué à l'étude du modèle qui le code.
Les modèles sont des abstractions simples qui aident à comprendre le mode de fonctionnement du système, tout en préservant ses caractéristiques essentielles. L'objectif d'un modèle est de rendre explicite l'ensemble des hypothèses pertinentes sur les systèmes que nous modélisons. Aussi, ces modèles permettent de formuler des généralisations sur ce qu'il est possible ou impossible, eu égard à ses hypothèses [16]. Dans un contexte réparti, les systèmes de réécriture de graphe et, plus généralement, les calculs locaux sont considérés comme des modèles qui fournissent des outils puissants pour l'encodage des algorithmes distribués et la preuve de leurs exactitudes.
Ce chapitre traite principalement les modèles de calculs locaux, et les calculs distribués soutenus par ces modèles. Étant donné que l'étude de ces modèles exige nécessairement une certaine expertise en matière de théorie des graphes, nous présentons, dans un premier volet de ce chapitre, quelques définitions et notations standard des graphes. Ensuite, nous introduisons les systèmes de réétiquetage de graphes et les calculs locaux. Parmi les paradigmes associés aux calculs locaux, nous présentons un exemple d'algorithme traitant le problème de calcul d'arbre recouvrant. Enfin, nous présentons la plateforme Visidia qui permet d'implémenter, simuler, visualiser et expérimenter les algorithmes distribués codés par des calculs locaux.

2 Définitions et notations standards des graphes

Dans cette thèse, nous nous sommes fixés des hypothèses et nous avons établi des choix sur la nature et les terminologies des graphes à utiliser. Les définitions, que nous présentons dans cette section, sont très courantes et proviennent essentiellement de [5, 17]. Tous les graphes que nous considérons sont finis, non orientés et simples. Formellement, nous représentons un graphe G par un couple $G = (V, E)$, où V est un ensemble fini de nœuds et E est un ensemble des arêtes $E = \{\{v, v'\} | v, v' \in V, v' \neq v\}$.

Définition I.1 (Taille d'un graphe)**.** *La taille du graphe G est égale au nombre de ses nœuds.*

Définition I.2 (Arêtes adjacentes)**.** *Deux arêtes sont adjacentes si elles partagent un nœud commun.*

Définition I.3 (Nœuds voisins)**.** *Deux nœuds v et v' sont voisins s'ils ont une arête commune* e={v, v'} *; on dit que e est incidente à v et v'.*

Définition I.4 (Degré d'un nœud)**.** *Soit v un nœud du graphe G et $N(v)$ l'ensemble des voisins de v ; le degré de v, noté $d(v)$, est le nombre d'éléments de $N(v)$ (défini aussi comme le nombre d'arêtes incidentes à v). Les nœuds de degré 1 sont appelés des feuilles, cependant les autres sont appelés des nœuds internes.*

Définition I.5 (Chemin)**.** *Un chemin P de v_1 à v_i dans G est une suite* P = v_1, e_1, v_2, e_2, ..., e_{i-1}, v_i *de nœuds et d'arêtes, tel que, pour chaque j ; $1 \leq j < i$ alors e_j est une arête incidente aux nœuds v_j et v_{j+1}. La longueur de P est égale à $i-1$. Si $v_1 = v_i$ alors P est un cycle. P est simple si aucun nœud n'apparaît deux fois dans P.*

Définition I.6 (Graphe connexe)**.** *Deux nœuds v et w sont connectés, s'il existe un chemin de v à w. Un graphe est connexe si tous ses nœuds sont deux à deux connectés.*

Définition I.7 (Distance entre deux nœuds)**.** *Soit v et v' deux nœuds connectés, la distance entre v et v', notée $d(v, v')$, est la longueur minimale d'un chemin simple de v à v'.*

Définition I.8 (Diamètre du graphe)**.** *Le diamètre de G, noté $D(G)$, est la plus grande distance entre deux nœuds quelconques du graphe.*

Définition I.9 (Arbre)**.** *Un arbre est un graphe connexe qui ne contient aucun cycle. Dans un arbre, deux nœuds quelconques sont donc connectés par un seul simple chemin.*

Définition I.10 (Sous-graphe)**.** *Soient G et G' deux graphes ; G' est un sous-graphe de G si $V' \subseteq V$ et $E' \subseteq E$ (G est dit un super-graphe de G'). Le sous-graphe de G induit par V', noté $G[V']$, est un graphe qui renferme l'ensemble des nœuds V' (V' est un sous-ensemble de V) et contient toutes les arêtes de G tel que,* E' ={{u, v} ∈ E | u, v ∈ V'}.

Définition I.11 (Boule). *Soit v un nœud et k un entier positif ; la boule de rayon k et de centre v, notée $B_G(v,k)$, est le sous-graphe de G induit par l'ensemble des nœuds* V'={v' ∈ V | d(v,v') ≤ k}.

Définition I.12 (Homomorphisme). *Un homomorphisme d'un graphe G sur un autre G′ est une application γ : V → V' tels que pour toute arête $\{u,v\}$ de G ; $\{\gamma(u),\gamma(v)\}$ est aussi une arête de G′. γ est dite un isomorphisme si γ est bijective et γ^{-1} est aussi un homomorphisme.*

3 Système de réétiquetage de graphe

3.1 Graphe étiqueté

Un graphe *L-étiqueté* est un graphe dont les nœuds et les arêtes sont tous étiquetés par des labels qui appartiennent à un ensemble d'alphabets finis L. Ce type de graphe est représenté par (G,λ) où G et $\lambda : V(G) \cup E(G) \to L$ désignent respectivement un graphe et une fonction d'étiquetage qui associe à chaque nœud et arête de G un élément de L. G est appelé un graphe sous-jacent de (G,λ), et λ est un étiquetage de G. Soient *(G, λ)* et *(G', λ')* deux graphes étiquetés ; alors, *(G, λ)* est dit un sous-graphe de *(G', λ')*, si, et seulement si, G est un sous-graphe de G' et λ est la restriction de λ' dans $V(G) \cup E(G)$. La relation de sous-graphe entre G et G' est notée par $(G,\lambda) \subseteq (G',\lambda')$. Un morphisme d'un graphe étiqueté (G,λ) à un autre (G',λ'), est une fonction $\phi : V \to V'$ qui préserve les relations d'adjacence présentes dans G. Autrement dit, si *{x, y}* est une arête dans G alors $\{\phi(x),\phi(y)\}$ est une arête dans G' et ϕ commute avec λ et λ'.

3.2 Définitions formelles

Après avoir introduit le principe de graphe étiqueté, nous abordons, dans cette section, l'aspect formel d'un système de réétiquetage de graphe.

Définition I.13 (Definition 1). *Une règle de réétiquetage est un triplé* R=(G, λ, λ') *où* (G, λ) *et* (G, λ') *sont deux graphes étiquetés. Les graphes étiquetés* (G, λ) *et* (G, λ') *sont respectivement le coté gauche et le coté droit de la règle R.*

Définition I.14 (Definition 2). *Un Système de Réétiquetage de Graphes (*GRS*) est un triplet* R=(L, I, P) *où L est un ensemble de labels, I est un sous ensemble de L, et P est un ensemble fini de règles de réétiquetage. I est l'ensemble des labels indispensables pour l'étiquetage de l'état initial du graphe. Dans ce qui suit, nous désignons par une étape de calcul, une étape de réétiquetage.*

Définition I.15 (Definition 3). *Une étape de* R-réétiquetage *est un 5-uplet* (G, λ, R, φ, λ') *où R est une règle de réétiquetage qui appartient à P, et φ est à la fois une occurrence de* (G_R, λ_R) *dans* (G, λ) *et une occurrence de* (G_R, λ) *dans* (G, λ'). *Intuitivement parlant, un étiquetage λ′ de G, obtenu à partir de λ, est le résultat d'une modification de tous les*

labels de $\varphi(G_R, \lambda_R)$ suivant l'étiquetage λ'_R. Une étape de réétiquetage sera notée par la fonction suivante : (G,λ) $\rightarrow_{R,\varphi}$ (G, λ'). *Ainsi, la notion de calcul dans un* GRS *correspond à une séquence de réétiquetage.*

Définition I.16 (Definition 4)**.** *Une séquence de réétiquetage est un uplet (G, λ_0, R_0, φ_0, λ_1, R_1, φ_1, λ_2, R_2, φ_2, ... λ_{n-1}, R_{n-1}, φ_{n-1}, λ_n) qui vérifie la propriété suivante : quelque soit i tel que $0 \leq i < n$ alors $(G, \lambda_i, R_i, \varphi_i, \lambda_{i+1})$ est une étape de* R-réétiquetage. *L'existence d'une telle séquence de réétiquetage est notée par la fonction (G, λ_0) $\rightarrow^*_R$ (G, λ_n).*
Le calcul distribué se termine au moment où le graphe devient irréductible.

Définition I.17 (Definition 5)**.** *Un graphe étiqueté (G, λ) est dit irréductible lorsque il n'existe aucune occurrence de (G_R, λ_R) dans (G, λ) pour toute règle de réétiquetage R. Pour tout graphe étiqueté (G, λ) dans G_I (la classe de graphes $I-$étiqueté), nous désignons par $Irred_R(G, \lambda)$ l'ensemble des graphes R-irréductible (G, λ') tel que (G, λ) $\rightarrow^*_R$ (G, λ'). Autrement dit, l'ensemble $Irred_R(G, \lambda)$ englobe tous les étiquetages finals qui peuvent être obtenus à partir d'un graphe $I-$étiqueté, et après l'application des règles de réétiquetage inclues dans P.*

Afin d'atteindre un niveau d'expressivité satisfaisant d'un *GRS*, deux mécanismes de contrôle locaux sont introduits. Ces mécanismes permettent de restreindre d'une certaine manière l'applicabilité des règles de réétiquetage. Le premier mécanisme introduit une relation de priorité sur l'ensemble de règles réétiquetage. Ainsi, un système de réétiquetage de graphe avec priorité est défini comme un 4-uplet $R = (L, I, P, >)$ où (L, I, P) est un *GRS*, et $>$ est un ordre partiel défini sur l'ensemble P. Le deuxième mécanisme interdit l'application de quelques règles de réétiquetage sous certaines conditions. Le principe de ce mécanisme est simple, il permet d'empêcher l'application d'une règle de réétiquetage quand l'occurrence correspondante est inclue dans des configurations particulières. Ces configurations forment un ensemble de cas de figures appelé un contexte interdit.

4 Modèle de calculs locaux

4.1 Définitions formelles de calculs locaux

Un système de ré-étiquetage de graphes est en fait une illustration d'un mécanisme plus général appelé les calculs locaux. Dans la suite, nous allons présenter la notion de séquences de ré-étiquetage qui va permettre de définir le modèle des calculs locaux. Ces définitions sont générales ; elles considèrent les calculs locaux sur des boules de rayon $\geq$1. Toutefois, notre travail considère seulement les calculs locaux sur des boules de rayon 1.

Définition I.18 (Definition 1)**.** *Une* relation de réécriture *de graphe est une relation binaire et isomorphe $R \subseteq G_L \times G_L$. Une chaîne de* R-réécriture *est une séquence de graphes étiquetés* (G_1, λ_1), (G_2,λ_2),...,(G_n,λ_n) *telle que, pour tout* i, 1$\leq$ i $<$ n *alors* (G_i,λ_i) R

(G_{i+1},λ_{i+1}). *Une séquence de longueur 1 est dite une étape de* R-réécriture *(ou étape de calcul tout court).*

Définition I.19 (Definition 2)**.** *Soit $R \subseteq G_L \times G_L$ une* relation de réécriture *de graphe, et k un entier positif (mesure le rayon d'une boule), alors on a :*

1. *R est une* relation de ré-étiquetage*, si pour tous graphes étiquetés G et H, l'implication suivante est toujours vraie :*

$$(G,\lambda)R(H,\lambda') \Rightarrow G = H$$

Si R est une relation de ré-étiquetage*, alors nous adoptons l'appellation de* chaînes de R-ré-étiquetage *au lieu de* chaînes de R-réécriture*.*

2. *Une* relation de ré-étiquetage R *est* k-locale*, si pour toute* ré-étiquetage *$(G,\lambda)R(G,\lambda')$, les étiquetages λ et λ' ne se diffèrent que sur certaines boules de rayon k :*

$$\exists \text{ v} \in \text{V(G) tel que } \forall \text{ x} \notin V(B_G(v,k)) \cup E(B_G(v,k)),\ \lambda(\text{x}) = \lambda'(x)$$

La relation R est dite locale si elle est $k-$locale pour certains $k > 0$.

3. *Une forme* R-normale *de (G, λ) $\in G_L$ est un graphe (G,λ'), tel que si* (G, λ) R* (G, λ') *et* (G, λ') R (G, λ'') *alors (G, λ'') $\notin G_L$ (la fermeture transitive de R est notée R$*$). R est dit noethérien, si pour tout graphe (G, λ) $\in G_L$ il n'existe pas une chaîne de* R-ré-étiquetage *finie et issue de (G,λ). Ainsi, si R est une relation de* ré-étiquetage *noethérien, alors tout graphe étiqueté a une forme* R-normale*.*

Définition I.20 (Definition3)**.** *Soit R une relation de* ré-étiquetage *et k un entier positif : la relation R est* k-localement *générée, si pour tous graphes étiquetés (G,λ), (G,λ'), (H,η), (H,η'), et pour tous nœuds $v \in V(G)$, $w \in V(H)$, tels que les boules $B_G(v,k)$ et $B_H(w,k)$ sont isomorphes via $\phi : V(B_G(v,k)) \to V(B_H(w,k))$ et $\phi(v) = w$; ces trois conditions suivantes impliquent que $(G,\lambda)R(G,\lambda')$ si et seulement si $(H,\eta)R(H,\eta')$.*

1. $\forall x \in V(B_G(v,k)) \cup E(B_G(v,k)), \lambda(x) = \eta(\phi(x))$ *et* $\lambda'(x) = \eta'(\phi(x))$,
2. $\forall x \notin V(B_G(v,k)) \cup E(B_G(v,k)), \lambda(x) = \lambda'(x)$,
3. $\forall x \notin V(B_H(w,k)) \cup E(B_H(w,k)), \eta(x) = \eta'(x)$

La relation R est localement générée si elle est $k-$localement générée pour certains k>0.

La notion de *séquence de ré-étiquetage* définie ci-dessus correspond à la notion du calcul séquentiel. Notons également, qu'une *relation de ré-étiquetage k-localement générée* permet entre autres, une réécriture parallèle, puisque les k-boules non chevauchées peuvent être ré-étiquetées de manière indépendante. Ainsi, un calcul distribué peut être aperçu comme deux étapes de ré-étiquetage consécutives, des k-boules non-chevauchées, pouvant s'appliquer dans un ordre quelconque. A proprement parlé, la notion de *séquence de ré-étiquetage*, adopté par ce rapport, est considérée comme une sérialisation de calculs distribués suivant un modèle clairement asynchrone.

4.2 Techniques de preuve

Les questions relatives à la preuve de correction des systèmes distribués représentent l'intérêt majeur des systèmes de réécriture de graphes. En effet, ces systèmes utilisent des techniques de preuve issues de la théorie classique des systèmes de réécriture pour prouver des propriétés de corrections des algorithmes distribués. Les plus importantes propriétés sont : la correction, la terminaison, la détection de la terminaison et la complexité temporelle. Nous illustrons, ci-dessous, la correspondance entre les propriétés des algorithmes distribués et les systèmes de ré-étiquetage de graphes.

Soit A un algorithme distribué codé par un graphe de ré-étiquetage R :

- L'algorithme A est correct si pour tout graphe étiqueté (G, λ) alors chaque graphe étiqueté appartenant à $Irred(G, \lambda)$ correspond à une solution valide.
- L'algorithme A termine toujours si, et seulement si, il n'existe pas une chaîne infinie de R-ré-étiquetage ou, de façon équivalente, si le système R est noethérien.
- La mesure de la complexité temporelle de l'algorithme A peut être obtenue en considérant la longueur maximale d'une chaîne de R-ré-étiquetage.
- L'algorithme A permet de détecter localement la terminaison globale s'il existe une propriété P et un entier positif K vérifiant : (1) P est vrai si le graphe est en configuration finale. (2) si v est un nœud dans G, alors toute boule $B(v, K)$ satisfait P est $R-irréductible$. Dans ce cas, le nœud v est en mesure de détecter localement la terminaison de A.

Afin de démontrer la correction d'un algorithme distribué, ou de manière équivalente un système de ré-étiquetage, il est indispensable de définir un ensemble des propriétés invariantes, et des propriétés de la configuration finale. L'ensemble des invariants doit satisfaire la configuration initiale du système et doit aussi être préservé après chaque étape de ré-étiquetage. Les propriétés de la configuration finale permettent de conclure que lorsque le graphe est R-irréductible le système de ré-étiquetage est correct et représente une solution valide.

5 Algorithmes de synchronisation

Lynch, N. [18] a prouvé que de nombreux problèmes n'ont pas encore de solutions dans un environnement distribué. En particulier, il est pratiquement impossible de concevoir des algorithmes déterministes pour un réseau asynchrone de processus anonymes qui adoptent un modèle de communication basé sur l'échange de messages asynchrones [19]. De ce fait, la solution d'introduire des algorithmes de synchronisation probabiliste peut rendre possible l'implémentation de certains algorithmes distribués. Par définition, *un algorithme probabiliste laisse "au hasard" un certain nombre de décisions ou d'instructions qui le composent* [20]. Ce hasard est matérialisé par la théorie des probabilités et, plus simplement, par une suite de tirages à pile ou face (0 et 1) intervenant au cours de l'algorithme, telle décision (instruction) étant déterministe, à un moment donné, par la suite de $\{0, 1\}^*$ obtenue [20].

Dans [21, 22], les auteurs distinguent trois grandes classes des calculs locaux. Chaque

classe est caractérisée par un type de règle de ré-étiquetage et implémentée par un algorithme de synchronisation probabiliste. Ces règles se distinguent par leurs champs d'application (une boule de rayon 1 ou deux voisins adjacents) et par leurs ré-étiquetages autorisées. L'exécution d'un algorithme de synchronisation est désormais une étape préliminaire et indispensable pour implémenter des algorithmes déterministes. Car, une fois les nœuds sont synchronisés, ils seront prêts pour appliquer des règles de ré-étiquetage. Ci-dessous, nous présentons les trois classes de calculs locaux : Handshake (LC0), Calculs Locaux de type 1 (LC1) et de type 2 (LC2).

5.1 Handshake (LC0)

La classe d'algorithme $LC0$ est décrite par un système de ré-étiquetage dont les règles sont toutes de type $LC0$. Une telle règle ne peut être applicable que sur deux nœuds adjacents du graphe. Elle autorise un changement d'état des deux nœuds et de l'arête qui les relient. Ce changement est accompli si, et seulement si, les étiquettes des nœuds et de l'arête correspondent bien à la condition de déclenchement de la règle. La règle $LC0$ est décrite ci-dessous par la figure V.2.

Figure I.1 – Règle de type LC0

Plusieurs algorithmes de synchronisation de type $LC0$ ont été proposés ; ils visent tous à préparer deux nœuds adjacents quelconques à une étape de calcul. Citons deux exemples : Le premier algorithme, proposé par Elhibaoui, A. et al. [23], est un algorithme synchrone basé sur des délais aléatoires et générés au hasard et d'une manière uniforme. Autrement dit, un handshake peut avoir lieu entre deux nœuds voisins s'ils sont libres au moment de la génération des délais.

Le deuxième algorithme, proposé par Métivier. Y. et al. [22], est un algorithme synchrone et probabiliste. L'exécution de l'algorithme est rythmée par des rounds : dans chaque round, chaque nœud p choisit un de ses voisins $c(p)$ au hasard. Il y'a un handshake entre p et $c(p)$ si, et seulement si, les deux nœuds se choisissent mutuellement. Dans ce cas un échange de messages entre p et $c(p)$ suivi par un éventuel changement de leurs étiquettes peuvent avoir lieu. Ci dessous nous présentons le deuxième algorithme (voir Algorithm.1).

5.2 LC1

La règle $LC1$ opère seulement sur une boule de rayon 1 (soit c le centre de la boule). L'application d'une règle de ré-étiquetage de type $LC1$ peut modifier l'état du nœud c et des arêtes issues de c. En revanche, les étiquettes des feuilles ne peuvent pas être modifiées.

Algorithm 1 Algorithme de synchronisation de type *Handshake*

Chaque nœud p répète infiniment les actions suivantes :
  Le nœud p choisit d'une façon aléatoire et uniforme un de ses voisins $c(p)$
  Le nœud p envoie 1 à $c(p)$
  Le nœud p envoie 0 à tous les autres nœuds voisins différents de $c(p)$; $w \neq c(p)$
  Le nœud p reçoit un message de chaque nœud voisin
  si Le nœud p reçoit 1 de $c(p)$ **alors**
    Le nœud p et $c(p)$ sont maintenant synchronisés
  fin si

La garde de la règle porte sur les étiquettes des nœuds de la boule et des arêtes issues du nœud centre. Une règle de ré-étiquetage de type $LC1$ est représentée par Figure.I.2 :

Figure I.2 – Règle de type LC1

Une implémentation possible de l'algorithme de synchronisation de type $LC1$ a été proposée dans [21]. L'algorithme proposé est appelé *algorithme d'élection locale* car il permet d'élire un nœud centre dans une boule. L'algorithme est synchrone et est donc rythmé par des rounds. Pour chaque round, un nœud quelconque p du graphe choisi aléatoirement un entier positif $rand(p)$ de l'ensemble $\{1,..N\}$ où N est un entier suffisamment supérieur à 1. Le nœud p envoie à tous ses voisins la valeur de $rand(p)$. p est élu dans la boule $B(p,1)$, si pour chaque nœud w appartenant à $B(p,1)$ et différent de p : $rand(p) > rand(w)$. Dans ce cas une étape de calcul peut avoir lieu dans $B(p,1)$. Nous présentons ci-dessous l'algorithme de synchronisation de $LC1$ (voir Algorithm.2).

Algorithm 2 Algorithme de synchronisation de type $LC1$

Chaque nœud p répète infiniment les actions suivantes :
  p tire au hasard un entier positif $rand(p)$
  p envoie $rand(p)$ à tous ses nœuds voisins
  p reçoit les entiers de tous ses nœuds voisins
  si $rand(p)$ est strictement supérieur à tous les entiers reçus par p **alors**
    p est le centre d'une synchronisation de type $LC1$
  fin si

5.3 LC2

Comme pour $LC1$, les règles de ré-étiquetage de type $LC2$ opèrent aussi sur une boule de rayon 1 (Soit c, le centre de la boule). Ainsi, une règle $LC2$ permet, en plus de toutes

les modifications autorisées par celle de $LC1$, de mettre à jour des états respectifs aux feuilles de la boule. La Figure.I.3 présente une règle de ré-étiquetage de type $LC2$.

Figure I.3 – Règle de type LC2

Métivier. Y. et al. ont proposé dans [21], un algorithme synchrone et probabiliste pour implémenter les synchronisations de type $LC2$. Dans chaque round, tout nœud p du graphe doit choisir aléatoirement un entier positif $rand(p)$ appartenant à l'ensemble $\{1,..N\}$ où N est un entier suffisamment supérieure à 1. Le nœud p envoie la valeur de $rand(p)$ à tous ses voisins. Après la réception des entiers de tous ses voisins, p envoie à chaque voisin w le maximum parmi les entiers qu'il a reçu de tous ses voisins excepté w. Le nœud p est élu dans $B(p,2)$, si $rand(p)$ est strictement supérieure à $rand(w)$ pour chaque nœud w de la boule de rayon 2 et de centre p. Dans ce cas une étape de calcul peut se réaliser sur la boule $B(p,1)$. Durant cette étape de calcul, il y aura un échange des labels et éventuellement un changement des états de tous les nœuds et les arêtes de la boule $B(p,1)$. L'Algorithm.3 présente une description de l'algorithme de synchronisation de type $LC2$.

Algorithm 3 Algorithme de synchronisation de type $LC2$

Chaque nœud p répète infiniment les actions suivantes :
  Le nœud p tire au hasard un entier positif $rand(p)$
  Le nœud p envoie $rand(p)$ à tous ses nœuds voisins
  Pour tout voisin w ; soit $max_{p;w}$ le max des entiers que p a reçu des nœuds voisins différents de w
  Le nœud p envoie $maxp;w$ à tous ses nœuds voisins w
  Le nœud p reçoit les entiers de ses nœuds voisins
  si $rand(p)$ est est strictement supérieur à tous les entiers reçus par p **alors**
    Le nœud p est le centre d'une synchronisation $LC2$
  fin si

6 Calcul distribué d'un arbre recouvrant

Le but de cet algorithme est de créer une arborescence hiérarchique sur l'ensemble du réseau. Il détermine tous les chemins redondants entre les processus et choisit un seul d'entre eux. L'algorithme est implémenté avec des règles $LC1$ et permet une détection locale de la terminaison globale. Formellement, il peut être encodé par le système de réécriture de graphe avec contexte interdit S= (L,I,P) où L = {A, A1, F, END, N, 0,

1}, I={A, N, 0} et P={R1, R2, R3, R4}. L désigne l'ensemble des labels des nœuds et des arêtes, I est l'ensemble des étiquettes initiales et R1..R4 sont les règles de réécriture. Initialement, nous supposons qu'un seul nœud est dans un état "actif" (codé par le label A), tous les autres nœuds étant dans un état "neutre" (label N) et que toutes les arêtes sont dans un état "passif" (label 0). Un nœud étiqueté A peut activer un de ses voisins neutre et marquer l'arête correspondante (règle $R1$). La nouvelle étiquette du nœud voisin est $A1$. Marquer une arête correspond à changer son label de 0 à 1. A son tour, un nœud étiqueté $A1$ peut poursuivre le calcul en activant un de ses voisins neutres et marquer l'arête correspondante (règle R2). Une fois, un nœud étiqueté $A1$ n'a plus de voisins neutres ou étiqueté $A1$ alors il peut changer son étiquette à F (règle $R3$). La dernière règle $R4$ s'applique une seule fois quand les voisins du nœud étiqueté A sont tous étiquetés F. Le calcul s'arrête quand l'étiquette du premier nœud actif (étiqueté A) devient END et tous les autres sont étiquetés F. Ainsi, l'arbre résultant, qui est constitué par les arêtes marquées, couvre entièrement le graphe. Graphiquement, les règles de réécriture sont représentées comme suit :

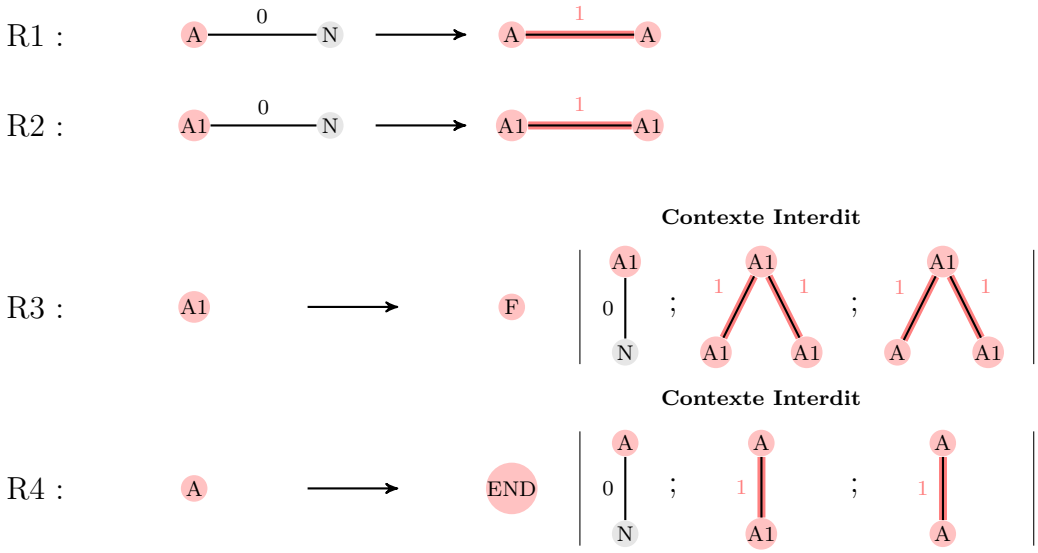


Figure I.4 – Règles de ré-étiquetage

Dans la figure I.5, nous schématisons un scénario d'exécution de l'algorithme. Nous, précisons que durant l'exécution, plusieurs nœuds peuvent simultanément appliquer les règles de ré-étiquetage dans des parties disjointes du graphe. L'arbre recouvrant est calculé, si le graphe devient irréductible : aucune règle n'est applicable.

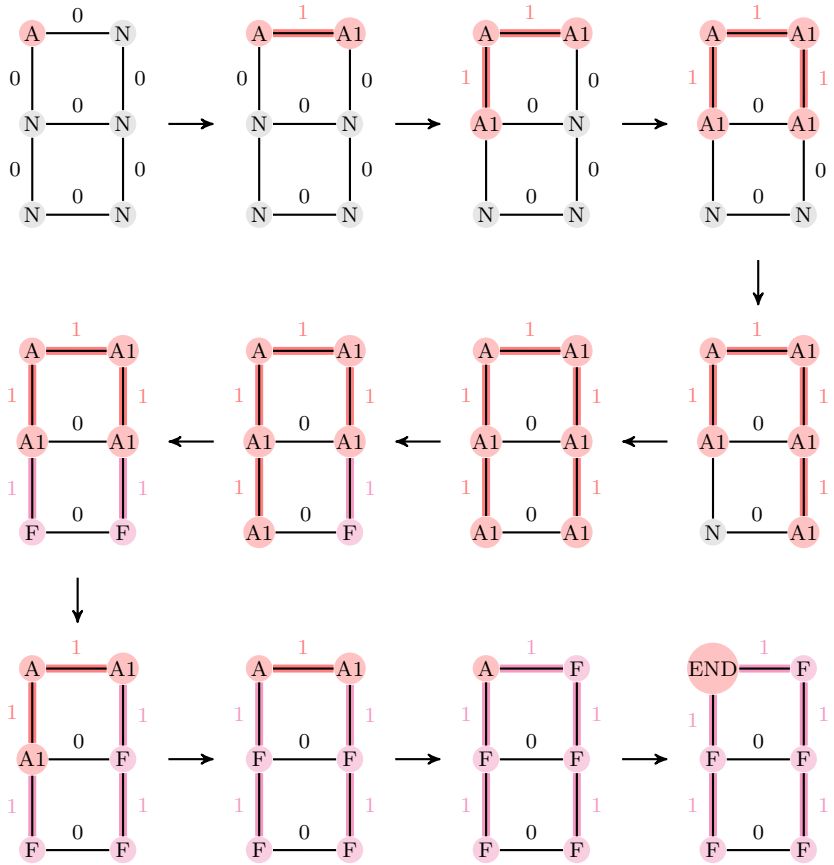


Figure I.5 – Calcul d'arbre recouvrant

7 Visidia : une plate forme de preuve et de simulation des algorithmes distribués

La palteforme Visidia [24–26] est conçue pour implémenter, simuler, visualiser et expérimenter des algorithmes distribués. Elle est basée sur un fondement théorique solide développé par plusieurs chercheurs du LaBRI [1]. Les équipes de recherche qui ont contribué à ce projet sont l'équipe de *combinatoire et algorithmique* et l'équipe des *méthodes formelles*. Ces équipes ont beaucoup collaboré pour que ce projet progresse et gagne de plus de plus d'intérêts. Leur motivation est de mettre en oeuvre d'une façon concrète les résultats théoriques importants portant sur la puissance des calculs locaux dans le codage des algorithmes distribués. Dans cette section, nous présentons les principales caractéristiques de Visidia ainsi que les autres outils qui s'inscrivent dans le même contexte. Par la suite, nous détaillons l'architecture logiciel de l'outil.

1. www.labri.fr

7.1 Présentation

Visidia permet aux utilisateurs de concevoir un graphe, implémenter leurs propres algorithmes et lancer la simulation. Doté d'un API assez rigoureux, cet outil offre un support complet pour implémenter des algorithmes distribués avec les modèles d'échange de messages, d'agents mobiles et de réseaux de capteur. Le premier modèle, simule les processus par des nœuds et les liens de communications par des arêtes. Ainsi, les nœuds vont pouvoir exécuter des algorithmes par le moyen des primitives offertes par Visidia comme celles conçues pour l'envoi et la réception de messages et de changement d'états. Dans un modèle à base d'agents mobiles, Visidia considère les sommets du graphe comme des états passifs, et les agents mobiles comme des entités de calcul qui se déplacent d'un nœud à un autre. Cependant, les travaux réalisés sur les réseaux de capteur sont assez récents, l'apport de Visidia à ce niveau n'est pas volumineux en terme de quantité. Néanmoins, les travaux en cours ont réussi de se débarrasser des réseaux statiques de Visidia et d'implémenter des capteurs qui se déplacent et communiquent avec leurs voisins sur une topologie dynamique. Peu importe le modèle choisi, l'animation d'un algorithme distribué sous Visidia s'exécute en temps réel. Autrement dit, l'animation est une reproduction immédiate du changement des états qui s'effectue dans le système avec la possibilité de visualiser l'échange des messages et aussi le déplacement des agents mobiles.

Visidia comporte deux versions différentes : une locale et une autre totalement distribuée. La première version s'exécute sur une simple machine où les nœuds du graphe sont simulés par des threads Java. La version distribuée de Visidia s'exécute sur un réseau réel de plusieurs machines distinctes. Ainsi, cette deuxième version peut effectuer des expérimentations sur des graphes de grande taille qui peuvent s'étaler sur plus de 1200 nœuds. La bibliothèque *RMI* de Java a été utilisée pour étendre la distribution de Visidia [27]. L'outil Visidia est codé en Java et peut être installé et exécuté très facilement. Le code source du logiciel peut être téléchargé gratuitement depuis le site web : `http://visidia.labri.fr`. Éventuellement, Visidia peut être exécuté en ligne en tant qu'une applette Java. Le site propose, en plus, plusieurs exemples d'algorithmes distribués ainsi que la documentation (licence GPL).

Pour la simulation et la visualisation des algorithmes distribués, d'autres plateformes intéressantes peuvent être citées. A titre d'exemple nous présentons les plateformes **VADE** [28], **PARADE** [29] et **LYDIAN** [30].

- **VADE** (Visualisation of Algorithms in Distributed Environnements) : est un outil qui permet la visualisation des algorithmes distribués dans un système asynchrone. L'architecture de VADE est basée sur le modèle client-serveur : l'algorithme s'exécute sur un serveur distant tandis que la visualisation se déroule sur une machine client. Cette architecture offre une grande accessibilité, une protection du code et un coût de communication assez faible. Cependant, cet outil n'a pas évolué depuis sa publication.
- **PARADE** (parallel program animation development environment) : est un outil qui permet la visualisation des algorithmes parallèles et distribués. L'architecture de l'outil est décomposée en trois parties principales, à savoir un programme de

surveillance des traces, un système de visualisation, et une application qui permet de visualiser les actions du programme.

– **LYDIAN** (Library of Distributed Algorithms and Animations) : est un environnement de simulation et de visualisation des algorithmes distribués. Il est destiné essentiellement au milieu académique. Il offre aux étudiants un environnement expérimental pour tester et visualiser le comportement des algorithmes distribués. LYDIAN supporte la simulation et l'animation des algorithmes asynchrones qui adoptent un modèle de communication basé sur l'échange de messages. Cet outil est considéré comme un outil flexible parce qu'il permet aux utilisateurs de combiner des algorithmes et des animations avec des structures de réseau arbitraire. LYDIAN est un logiciel libre et facile à installer sur les plateformes Linux et UNIX.

7.2 Architecture générale

L'architecture de l'outil se compose de trois parties principales, à savoir, l'interface graphique, le simulateur et la bibliothèque d'algorithmes distribués. Cet aspect modulaire offre la possibilité de modifier et de développer chaque module de manière indépendante.

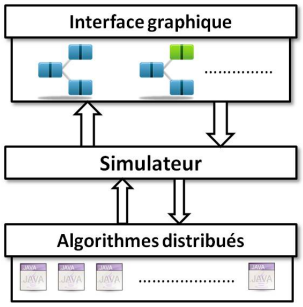


Figure I.6 – Architecture de Visidia

7.2.1 Interface graphique

L'interface graphique permet de dessiner un graphe et de visualiser l'exécution de l'algorithme. L'interface graphique de la dernière version de Visidia a nettement évolué ; elle passe d'un affichage multi-fenêtres à un affichage multi-onglets. Désormais, plus besoin d'ouvrir une fenêtre pour chaque simulation ; une seule fenêtre peut inclure plusieurs onglets dont chacun correspond à une simulation. Autrement dit, une fois le réseau est dessiné par le moyen d'un onglet de construction des graphes, plusieurs simulations peuvent être lancées sur des onglets différents. Cependant, le contenu de la barre d'outils et les fonctionnalités proposées dépendent strictement de l'onglet actif qui peut être soit l'édition d'un graphe, soit la visualisation d'une simulation. L'édition d'un graphe est facile, elle est basée sur une simple manipulation de la souris "Drag & Drop". L'utilisateur peut, facilement, ajouter et supprimer des nœuds et des arêtes et attribuer des étiquettes,

des couleurs, etc... Comme l'édition des graphes, le contrôle d'une simulation est assez simple et paramétrable : l'utilisateur peut par exemple modifier la vitesse, arrêter et faire une pause de l'animation. Une autre fonctionnalité importante supportée par Visidia est l'importation des graphes réalisés par d'autres éditeurs (par exemple GML).

7.2.2 Le simulateur

Le simulateur est un module intermédiaire entre l'interface graphique et la bibliothèque des algorithmes. Il contrôle les échanges de messages entre les threads (les threads implémentent les processeurs), ainsi que la visualisation des événements. Il permet aussi de modéliser un réseau de processus asynchrones.

7.2.3 Bibliothèque d'algorithmes distribués

Un algorithme distribué est implémenté par une classe Java et recopié sur tous nœuds du graphe pour être exécuté de manière asynchrone. A son tour, un nœud est implémenté par une classe Java qui contient son identifiant, son propre état, son degré et la taille du graphe. Souvent, ces propriétés sont accessibles via des primitives qui permettent aussi d'impléménter la communication entre les nœuds. La dernière version de la bibliothèque est assez riche, beaucoup d'algorithmes distribués décrits par les calculs locaux sont implémentés et prêts à être réutilisés par les utilisateurs. Aujourd'hui, une grande partie de cette bibliothèque a été formellement prouvée dans le cadre de cette thèse et aussi par le projet *LOCO* [14] qui a pour but la formalisation en *Coq* [12] du modèle des calculs locaux. Cette formalisation, permet non seulement de prouver la correction d'algorithmes, mais aussi de comparer divers modes de synchronisation et de détection de la terminaison des calculs.

8 Discussion

Il est intéressant de constater que les calculs locaux ont apporté beaucoup de solutions à des problèmes divers en algorithmique distribuée. Les calculs locaux ont été largement adoptés comme des outils de codage et de preuve des algorithmes distribués tels que, [31–33]. La puissance du modèle des calculs locaux résulte de la simplicité relative du modèle lui-même et du cadre théorique qui le supporte. Cela dit, la simplicité du modèle suppose une facilité d'utilisation des notions algorithmiques qui l'incorpore. A titre d'exemple, nous citons les types de synchronisation, la détection de la terminaison, les contextes interdits, etc.

Quoique, dans [34] les auteurs ont pu extraire quelques limites des calculs locaux dans un graphe par le moyen des preuves d'impossibilité. A titre d'exemple, ils ont prouvé qu'il n'existe aucune relation de ré-étiquetage, généréé localement, qui reconnaît les graphes planaires. Toutefois, un résultat d'impossibilité est une déclaration au sujet de tous les algorithmes possibles autorisées dans un système. Ceci ne diminue dans aucun cas la puissance expressive du modèle mais au contraire il prouve que ce dernier est suffisamment

précis pour décrire les propriétés pertinentes de tous les algorithmes admis. Cependant, nous avons constaté que la procédure de preuve du modèle des calculs locaux est souvent longue et manuelle. En particulier, cette particularité a soulevé le degré de complexité de preuve et de vérification des algorithmes distribués. Également, nous avons conclu que la preuve est loin d'être une tâche manuelle, les algorithmes sont tellement complexes qu'ils nécessitent une méthodologie générale de preuve. Cette méthodologie doit s'appuyer sur des outils d'aide à la preuve existants pour automatiser la vérification et la validation de ces preuves. De ce fait, il nous semble important de construire un environnement d'aide à la preuve de correction d'algorithmes distribués, mais également des propriétés génériques relatives à une classe de systèmes de réécriture. Cet environnement doit être capable de discerner entre deux différents niveau d'abstraction : la preuve formelle et la simulation d'algorithmes.

En vue de fournir un support de raisonnement rigoureux et de prouver les propriétés qui garantissent la correction des algorithmes, nous avons choisi d'utiliser : (1) l'approche formelle de preuve et vérification "correcte par construction" et (2) l'environnement de preuve, d'implémentation et de simulation des algorithmes distribués Visidia. La présentation de l'approche "correct-par-construction" fera l'objet du deuxième chapitre.

9 Conclusion

A proprement parlé, ce chapitre constitue un cadre théorique de la thèse. Il a comme objectif principal, la présentation d'un état de l'art des systèmes de ré-étiquetage de graphes et des calculs locaux. Afin d'atteindre ce but, nous avons essayé, tout d'abord, de définir quelques notations standard des graphes. Ensuite, nous avons donné un aperçu sur les systèmes des ré-étiquetage de graphe. Nous avons montré, via des définitions formelles, la puissance de ce système dans l'encodage des algorithmes distribués et nous avons mis en première priorité son intérêt majeur dans la preuve de correction des calculs locaux. Par la suite, nous avons essayé d'illustrer les notions théoriques avec un exemple qui implémente un algorithme de calcul d'arbre recouvrant dans un graphe connexe. Enfin, nous avons présenté Visidia qui est un environnement d'implémentation, de simulation et de prototypage des algorithmes distribués conçus suivant le modèle des calculs locaux.

Le chapitre suivant va permettre de présenter la méthode et l'approche de preuves que nous allons adopter pour prouver des algorithmes distribués.

Chapitre II

Méthode de spécification formelle *B événementiel*

1 Introduction

Les problèmes qui surgissent dans les systèmes informatiques découlent souvent d'erreurs et d'insuffisance dans leurs spécifications de départ [35]. Dans un contexte distribué, ces problèmes sont amplifiés à cause de la complexité croissante des applications à savoir le non-déterminisme dans l'exécution des processus, le manque de l'information globale, etc. Toutefois, l'utilisation des méthodes formelles peut les résoudre en écartant toute ambiguïté et imprécision dans la spécification et en offrant une assistance complète dans la preuve du système.

Dans cette optique, nous avons choisi la méthode formelle *B événementiel* (*Event-B*) pour spécifier et prouver la correction des calculs locaux. En effet, l'abstraction des algorithmes distribués, fournie par les modèles de calculs locaux, peut être facilement exprimée en *B événementiel*. Aussi, ces modèles offrent un complément graphique qui peut être exploité par cette méthode. D'autres part, cette méthode fournit un cadre général de preuve qui assure une bonne maîtrise des techniques de preuves issues de la théorie classique des systèmes de réécriture.

Dans la suite de ce chapitre, nous développons d'avantage le cadre général de cette méthode. D'abord, nous présentons le langage *B événementiel*. Ensuite, parmi les assistants de preuve automatique qui supportent la méthode *B événementiel*, nous présentons la plate-forme *Rodin*. Enfin, pour bien illustrer la méthode *B événementiel*, nous spécifions l'algorithme distribué présenté dans le chapitre précédent.

2 Présentation générale

La méthode *B événementiel* est à la fois un langage et une méthode de spécification et de preuve de logiciels informatiques. Elle a été récemment proposée par Jean-Raymond

Abrial [36, 37] comme une évolution de la méthode B classique [38]. La nouvelle méthode a préservé en même temps la puissance et la simplicité de l'ancienne, et elle a apporté également des améliorations sur plusieurs aspects, notamment la spécification des systèmes réactifs. Le processus de développement en *B événementiel* suppose que chaque spécification est associée à une preuve de correction basée elle même sur le fondement mathématique du langage. Il est à noter que ce dernier est fondé sur la logique du premier ordre et la théorie des ensembles.

Toutefois, la méthode *B événementiel* est directement liée à l'approche de conception de programmes : "*correct-par-construction*". Cette méthode garantit qu'au moment où le développement formel du système est terminé que celui-ci soit déjà vérifié. Cette approche consiste à débuter le développement par un modèle abstrait du système et rajouter progressivement des détails pour générer à la fin un modèle concret très proche de l'implémentation. Formellement, cette construction incrémentale du système est guidée par la technique de raffinement qui permet de préserver les propriétés du système, y compris les propriétés de correction et celle de la terminaison. Le processus de développement engendre plusieurs obligations de preuve qui garantissent sa correction. Ces obligations sont prouvées par des outils de vérification qui incorporent souvent des procédures de preuve automatique et interactive.

Les outils dédiés au développement en B classique et *B événementiel* sont multiples. A titre d'exemple, nous citons les outils *Atelier B* [39], *B4free*[1] et *Rodin* [40]. l'*Atelier B*, développé par la société *ClearSy*, est le premier outil industriel qui permet une utilisation opérationnelle de la méthode B. Dans un autre projet, la même société a proposé *B4free* comme un outil de preuve destiné pour l'enseignement de la méthode B en milieu académique. Enfin, la plate-forme *Rodin* permet de spécifier, d'animer et de prouver des modèles *B événementiel*. Ces outils s'appuient sur les mêmes prouveurs qui proviennent de l'*atelier B*. Comme nous avons précisé précédemment, nous allons, dans cette thèse, se limiter à l'utilisation de la plate-forme *Rodin*.

3 Langage B événementiel

Un développement *B événementiel* est une spécification incrémentale de plusieurs machines. Il débute par une spécification mathématique abstraite du système et s'achève par le code informatique correspondant. Ainsi, la machine est un composant indispensable pour la construction formelle d'un système en *B événementiel* ; elle spécifie sa partie dynamique. Elle inclut des éléments comme les variables, les invariants et les événements qui établissent le changement d'état du système. Souvent, les machines font recours à des éléments statiques du système tels que les constantes, les ensembles et les axiomes. Ces éléments sont décrits par un deuxième composant du langage *B événementiel* appelé "*contexte*". Les relations machine-contexte, machine-machine et contexte-contexte sont définies comme suit : une machine peut voir un ou plusieurs contextes, une machine peut raffiner une autre machine, et un contexte peut étendre un ou plusieurs autres contextes.

1. http ://www.b4free.com

Quand une machine voit un contexte, cela veut dire qu'elle peut utiliser les constantes et les propriétés de ce dernier mais sans pouvoir les changer. L'extension des contextes consiste tout simplement à rajouter des nouveaux ensembles, constantes et propriétés. Bien que, le raffinement soit différent de l'extension, cependant tous les deux sont considérés comme des constructions similaires d'un point de vue méthodologique.

Dans la suite de cette section, les deux composants, machine et contexte, seront présentés avec plus de détails.

3.1 Contexte

Le contexte permet de décrire l'ensemble des données statiques du système. Il se compose des éléments suivants : un nom, les noms des contextes à étendre, une liste d'ensembles non vides, une liste de constantes, et une liste de propriétés sur les constantes. Les ensembles, déclarés dans la clause *SETS*, permettent de typer tous les éléments du modèle. Un ensemble peut être soit un type énuméré (c'est à dire un ensemble de constantes), soit un ensemble fini dont les éléments sont inconnus. Les constantes, déclarées dans la clause *CONSTANTS*, sont typées et définies dans la clause *AXIOMS*. Dans la clause *THEOREMS*, nous examinons des propriétés déduites à partir de celles présentes dans la clause *AXIOMS*.

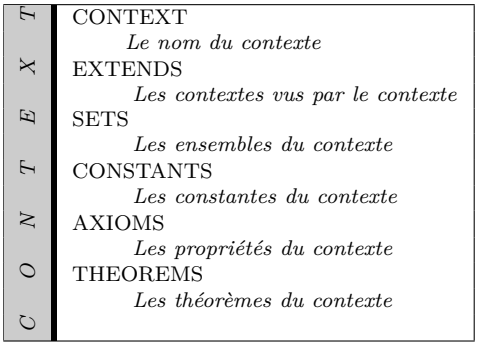


Figure II.1 – Structure du contexte

3.2 Machine

Une machine est constituée d'un nom, des variables v, des invariants $I(v)$ et d'un ensemble d'événements. Les variables définissent l'état du système à spécifier. Elles sont déclarées dans la clause *VARIABLES*. Les invariants sont des prédicats qui demeurent valides tout au long de l'exécution du système. Ils servent à typer les variables de la machine et à spécifier les propriétés du système. Ils sont déclarés dans la clause *INVARIANT*. Également, dans la même clause, il est possible de définir des théorèmes qui servent à démontrer des propriétés de correction du système. Souvent, ces théorèmes ne constituent pas nécessairement des invariants inductifs, mais ils représentent une preuve

de validité supplémentaire de la machine. Le variant, déclaré dans la clause *VARIANT*, est une mesure qui décroît après toute application d'un événement convergent. Le variant est donc considéré comme un moyen pour prouver la terminaison du calcul. Les événements permettent de spécifier le comportement dynamique du système et d'initialiser ses variables. L'initialisation est accomplie par le moyen d'un événement spécial appelé événement d'*Initialisation*. Un événement est défini par des gardes et des actions. Les gardes déterminent les conditions nécessaires pour le déclenchement de l'événement, et les actions permettent de spécifier le changement des valeurs des variables. Les actions sont exprimées à l'aide d'un langage de substitutions généralisées. Un événement ne peut se déclencher que lorsque sa garde est vraie. Cependant, deux événements différents ne peuvent pas se déclencher en même temps (même si leurs gardes sont vraies au même moment) ; ils doivent être exécutés alternativement selon un ordre indéterminé. Les événements sont introduits dans la clause *EVENTS*.

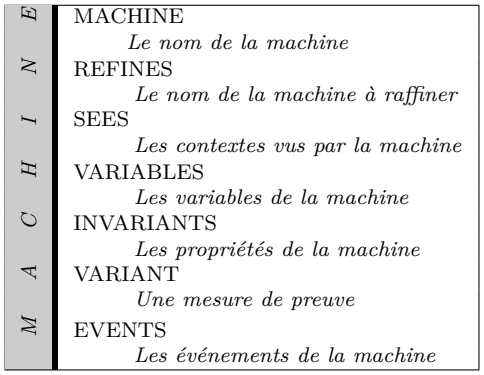


Figure II.2 – Structure d'une machine

3.2.1 Événement

Formellement, nous distinguons trois types d'événement : indéterministe, gardé et simple. Un événement indéterministe est composé de quatre éléments : un nom, des variables locales l, des gardes et des actions. Les gardes, désignées par $G(v,l)$, sont des prédicats qui dépendent à la fois des variables de l'événement et celles de la machine. Les actions, désignées par $S(v,l)$, sont des substitutions généralisées. L'aspect indéterministe de l'événement est basé sur le principe du déclenchement de ce dernier qui dépend fortement de l'existence des valeurs de variables locales l satisfaisant $G(v,l)$. La structure de cet événement est représentée par la figure Fig.II.3 (voir l'événement à gauche). Un événement gardé est un événement qui ne dispose pas des variables locales ; il est simplement composé de gardes $G(v)$ et de substitutions généralisées $S(v)$. La structure de cet événement est représentée par la figure Fig.II.3 (voir l'événement au milieu). Un événement est dit simple s'il ne contient aucune garde. Autrement dit, la garde est toujours vraie et l'événement peut toujours se déclencher. Il est important de signaler que l'initialisation est

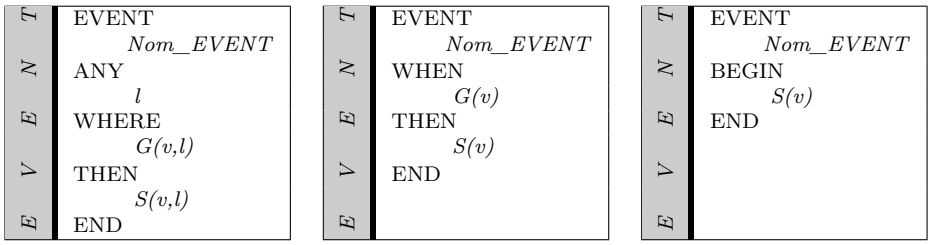


Figure II.3 – Différents types d'événements

un événement simple qui ne se déclenche qu'une seule fois. La structure de cet événement est représentée par la figure Fig.II.3 (voir l'événement à droite).

3.2.2 Substitution généralisée

Nous distinguons trois formes possible de substitutions généralisées [41] : déterministe, non-déterministe et vide. Ces différentes formes sont présentées dans le Tableau.II.1.

Une substitution déterministe est une substitution simple qui ressemble beaucoup à une affectation directe dans un langage de programmation impératif. Elle permet de changer les valeurs d'un ou de plusieurs variables d'état ; le plus souvent en fonction de leurs états postérieures. Nous désignons par x l'ensemble des variables qui vont changer de valeurs et qui appartiennent à v (l'utilisation de la notion d'ensemble est mise en vigueur pour simplifier la présentation). Les nouvelles valeurs que peuvent prendre x sont désignées par $E(v)$. En effet, $E(v)$ se réfère à un ensemble d'expressions ayant le même nombre de variables dans x. De même, chaque expression appartenant à $E(v)$ doit avoir le même type du variable qui lui correspond dans x.

La substitution non-déterministe est la forme la plus générale des substitutions. Elle exprime un changement de valeurs d'un ou de plusieurs variables d'état. Ce changement est réalisable si, et seulement si, des conditions, portant à la fois sur les variables locales de l'événement et celles de la machine, soient toutes vraies. Les conditions sont exprimées par des prédicats $P(t, v)$ où t est un ensemble de variables locales. $F(t, v)$ est un ensemble d'expressions dont chaque élément correspond à une variable de x. Dans une substitution aussi bien non-déterministe que déterministe, les variables et les expressions sont séparées par l'opérateur d'affectation ":=". Le choix de cet opérateur a pour but la simplification de la présentation de cette section ; car en effet, d'autres opérateurs peuvent être utilisés. Nous citons, l'opérateur ":$\in$" qui permet d'affecter arbitrairement une valeur à partir d'un ensemble, et l'opérateur ":|" qui permet d'affecter une valeur sous certaines conditions.

Enfin, une substitution vide est spécifiée par l'opérateur "*skip*". Elle désigne une substitution qui n'opère aucun changement sur les variables de la machine. Autrement dit, nous parlons ici d'un type d'événements qui n'a aucune action.

Déterministe	$x := E(v)$
Vide	*skip*
Non-déterministe	**any** t **where** $P(t,v)$ **then** $x := F(t,v)$ **end**

Tableau II.1 – Types de substitutions généralisées

4 Principe de raffinement

Dans le cadre de notre travail, le raffinement est la base de l'approche "correcte par construction". Il constitue aussi le fondement du développement formel en *B événementiel*. D'une part, il permet de construire progressivement et de manière incrémentale des programmes corrects à partir des spécifications abstraites. D'autre part, il assure que chaque étape préserve bien les propriétés de correction vis-à-vis de l'étape précédente.

A partir de l'étude que nous avons menée sur la technique de raffinement, nous avons pu en déduire les avantages suivants :

- La simplification de la preuve de correction des programmes informatiques.
- Le partage de la complexité de développement formel.
- La possibilité de réutiliser des propriétés déjà prouvées.

La notion de raffinement a été introduite la première fois par Dijkstra [42], ensuite elle a fait l'objet de plusieurs travaux de recherche, parmi lesquels nous citons : Morgan [43], Back [44] et Abrial [38]. Dans [45] (un rapport qui regroupe plusieurs aspects liés à la technique de raffinement), les auteurs définissent trois propriétés importantes pour qu'une relation de raffinement soit satisfaite : la réflexivité, la transitivité et la monotonie. La réflexivité assure qu'un programme est le raffinement de lui-même. La transitivité garantie qu'un programme peut être raffiné par étapes successives. La monotonie admet que les parties d'un programme peuvent être séparément raffinées.

Dans un développement *B événementiel*, il n'y a que les machines qui peuvent évoluer par raffinement. En effet, un raffinement d'une machine abstraite permet de renforcer les invariants et ajouter des détails à la spécification du système. Formellement, il est matérialisé par (1) l'introduction des nouvelles variables d'état (éventuellement, le remplacement des variables abstraites) et (2) par l'ajout de nouveaux événements (éventuellement, la modification des événements abstraits). Deux machines successives sont liées par le moyen d'un invariant spécial appelé invariant de collage. Cet invariant exprime le lien entre les variables abstraits et concrets. Le raffinement des événements permet de renforcer les gardes et aussi d'affaiblir les actions. Le processus de développement par raffinement génère un ensemble d'obligations de preuves qui permettent de vérifier sa correction. Ces obligations sont présentées dans la section 5.2.

5 Les obligations de preuve

Une obligation de preuve est une proposition mathématique à démontrer pour prouver que les différents modules ainsi que le processus de spécification sont corrects. Formellement, elle est codée par un ensemble d'hypothèses et par une conclusion. La démonstration d'une obligation nécessite obligatoirement la preuve de correction de sa conclusion eu égard à ses hypothèses.

La méthode *B événementiel* contient deux grandes classes d'obligation de preuve : les obligations de preuve du modèle et les obligations de preuve du raffinement. Souvent, ces obligations sont générées par l'outil de spécification formelle d'une façon automatique. Toutefois, certaines obligations ne peuvent pas être automatiquement démontrées et alors l'intervention de l'utilisateur demeure indispensable. Dans la suite de cette section, nous présentons d'une manière détaillée les différentes classes d'obligation de preuves.

5.1 Les obligations de preuve du modèle

Nous distinguons 3 principaux types d'obligation de preuve du modèle : la préservation de l'invariant, l'initialisation et la faisabilité des événements. Afin de bien expliquer les obligations, nous avons choisi de décrire les événements par des prédicats *avant-après*. Cette description permet de tracer la relation entre les valeurs des variables avant et après la substitution.

La terminologie, utilisée dans cette section, est définie comme suit :

```
s : Les ensembles et les constantes du contexte.
v : Les variables d'état de la machine.
A(s) : Les axiomes et les théorèmes du contexte.
I(s, v) : Les invariants et les théorèmes de la machine.
x : Les variables locales de l'événement.
G(x, s, v) : Les gardes de l'événement.
PAA (x, s, v, v') : Le prédicat avant-après de l'événement.
```

5.1.1 Préservation de l'invariant

Un invariant est une propriété du système qui doit être toujours vraie. Concrètement, le système doit préserver cette propriété après tout déclenchent d'événement. L'hypothèse de l'obligation de preuve correspondante est définie par la conjonction des déclarations du contexte $A(s)$, d'invariants $I(s,v)$, de garde de l'événement $G(x,s,v)$ et du prédicat avant-après $PAA(x,s,v,v')$. La conclusion de cette obligation est l'invariant I(s,v').

$$\begin{array}{|l} A(s) \\ I(s,v) \\ G(x,s,v) \\ PAA(x,s,v,v') \\ \hline\hline I(s,v') \end{array}$$

5.1.2 Initialisation

L'initialisation est un événement particulier qui permet d'affecter des valeurs initiales à toutes les variables de la machine. L'obligation correspondante permet de vérifier si la substitution de l'événement implique la correction de l'invariant. Formellement, cette obligation est présentée comme suit :

$$\begin{array}{|l} A(s) \\ G(x,s,v) \\ PAA(x,s,v,v') \\ ---------- \\ I(s,v') \end{array}$$

5.1.3 Faisabilité des événements

Une machine *B événementiel* doit assurer que chaque substitution généralisée non-déterministe soit faisable. Autrement dit, si la garde de l'événement est vraie, alors l'action doit être réalisable. Ainsi, l'objectif de l'obligation de preuve correspondante est de garantir qu'il existe une nouvelle variable locale v' qui satisfait le prédicat avant-après $PAA(x,s,v,v')$. Formellement, cette obligation est représentée comme suit :

$$\begin{array}{|l} A(s) \\ I(s,v) \\ G(x,s,v) \\ ---------- \\ \exists v' \cdot PAA(x,s,v,v') \end{array}$$

5.2 Les obligations de preuve du raffinement

Lors d'un raffinement, il est indisponible de vérifier que la machine concrète ne se contredit dans aucun cas avec l'abstraite. Dans cette optique, plusieurs obligations de preuve doivent être générées et déchargées pour valider la correction du raffinement. Les principales obligations que nous présentons dans cette section sont : le renforcement de la garde, la simulation des actions, la correction des événements convergents et la correction des témoins (witness). Comme pour les obligations de preuve du modèle, nous décrivons, dans cette section, les événements par des prédicats *avant-après* et nous ajoutons la terminologie suivante :

```
w : Les variables concretes de la machine.
J(s,v,w) : Les invariants et les théorèmes concrets de la machine.
y : Les variables locales de l'événement concret.
H(y, s, w) : Les gardes de l'événement concret.
BA1(v, v') : Les actions de l'événement abstrait.
BA2(w, w') : Les actions de l'événement concret.
n(s, v) : Le variant (le cas d'un entier naturel).
t(s, c, v) : Le variant (le cas d'un ensemble).
W(x, y, s, w) : Les témoins (witness) de l'événement concret.
```

5.2.1 Renforcement de la garde

Le renforcement de la garde s'inscrit dans le cadre de la préservation de l'état abstrait par l'événement concret. L'obligation de preuve, qui correspond à ce principe, veille à ce que les gardes d'un événement concret renforcent correctement les gardes abstraites. Autrement dit, cette obligation garantit que lorsqu'un événement concret est déclenché, alors son correspondant abstrait est automatiquement déclenché. L'obligation de preuve relative au renforcement de la garde est représentée comme suit :

$$\begin{array}{|l} A(s) \\ I(s,v) \\ J(s,v,w) \\ H(y,s,w) \\ W(x,y,s,w) \\ ---------- \\ g(x,s,v) \end{array}$$

5.2.2 Simulation des actions

En plus du renforcement de la garde, un événement concret doit garantir une simulation correcte des actions pour assurer la préservation de l'état abstrait. L'obligation de preuve, relative à la simulation des actions, doit garantir que lorsqu'un événement concret est "exécuté", alors ses actions ne doivent pas être contradictoires avec ceux de l'événement abstrait. Formellement, cette obligation est représentée comme suit :

$$\begin{array}{|l} A(s) \\ I(s,v) \\ J(s,c,w) \\ H(y,s,w) \\ W1(x,y,s,w) \\ W2(y,v',s,w) \\ BA2(w,w',...) \\ ---------- \\ BA1(v,v',...) \end{array}$$

5.2.3 Correction des événements convergents

Afin de garantir la cohérence de la machine concrète avec la machine abstraite, nous devons prouver que les événements concrets ne divergent pas et ne prennent pas le contrôle du système raffiné indéfiniment. Pour pallier à ce problème, la définition d'un variant est peut être une solution pour exercer un contrôle sur les événements concrets (les événements convergents en particulier). Cette définition génère deux obligations de preuve : la première obligation assure que chaque événement convergent décroît le variant (si le variant est une mesure numérique), ou le diminue (si le variant est un ensemble). La deuxième obligation veille à ce que le variant proposé soit un entier positif.

$A(s)$	$A(s)$	
$I(s,v)$	$I(s,v)$	$A(s)$
$J(s,v,w)$	$J(s,v,w)$	$I(s,v)$
$G(x,s,w)$	$G(x,s,v)$	$J(s,v,w)$
$PAA(x,s,w,w')$	$PAA(x,s,w,w')$	$G(x,s,v)$
$----------$	$----------$	$----------$
$n(s,w') < n(s,w)$	$t(s,w') \subset t(s,w)$	$n(s,v) \in \mathbb{N}$

5.2.4 Correction des témoins (witness)

Un raffinement d'un événement indéterministe peut garder les mêmes variables locales, comme il peut aussi les remplacer par d'autres variables plus concrètes. Pour cette dernière possibilité, il est indispensable de définir des témoins (Witness) pour prouver la correction de l'événement. Les témoins sont des prédicats qui permettent de définir les variables locales abstraites en fonction des concrètes. Souvent, un témoin est formellement spécifié par une simple égalité. L'obligation de preuve, ci-dissous présentée, permet de prouver que chaque témoin proposé dans la section $WITH$ d'un événement concret a une existence effective dans le niveau abstrait :

$A(s)$
$I(s,v)$
$J(s,v,w)$
$H(y,s,w)$
$----------$
$\exists x \cdot W(x,y,s,w)$

6 Spécification formelle d'un algorithme distribué

Pour bien illustrer la méthode *B événementiel*, nous allons spécifier, dans cette section, l'algorithme de calcul d'arbre recouvrant présenté dans la section 6 du chapitre I. La stratégie de raffinement que nous allons adopter dans le développement de l'algorithme est présentée comme suit :

- La première machine spécifie l'objectif global de l'algorithme sans entrer dans le détail du calcul distribué.
- La deuxième machine est un raffinement de la première machine. Elle introduit les propriétés inductives qui permettent d'exprimer le calcul de l'arbre.
- La troisième machine raffine celle d'avant. Elle spécifie localement le calcul de l'arbre. En d'autres termes, cette machine permet de spécifier des événements qui correspondent exactement à l'ensemble des règles de ré-étiquetage.

6.1 Le contexte *graphe*

Le contexte *graphe* spécifie les propriétés d'un réseau sur lequel notre algorithme est censé fonctionner. Formellement, un réseau peut être modélisé par un graphe connecté,

non-orienté et simple. Les nœuds du graphe désignent les processeurs, et les arêtes correspondent aux liens de communications directes. Un graphe est simple s'il n'a ni arêtes multiples ni boucles (axm4 et axm3). Un graphe non orienté signifie que ses arêtes sont symétriques (axm3). Un graphe est connecté, si et seulement si, pour chaque paire de nœuds il existe un ensemble d'arêtes qui les relient (axm5). Nous précisons que dans la spécification de ce contexte, nous avons repris en partie les définitions usuelles de la théorie des graphes et les travaux de Jean-Raymond Abrial et al. [36, 46]. Ainsi, un graphe g est modélisé par un ensemble de nœuds ND peut être présenté comme suit :

$$\begin{array}{l} axm1 : g \subseteq ND \times ND \\ axm2 : dom(g) = ND \\ axm3 : g = g^{-1} \\ axm4 : id(ND) \cap g = \varnothing \\ axm5 : \forall s.s \subseteq ND \wedge s \neq \varnothing \wedge g[s] \subseteq s \Rightarrow ND \subseteq s \end{array}$$

6.2 Le contexte *Arbre*

Un arbre recouvrant d'un graphe connexe est un sous-graphe connexe et acyclique qui contient tous les nœuds du graphe. Afin de spécifier un arbre, nous choisissons un nœud quelconque r ($r \in ND$) pour qu'il soit la racine de l'arbre résultant tr. En effet, r est appelé racine de l'arbre tr si, et seulement si, il existe un chemin unique joignant r à chaque nœud de l'arbre tr. Formellement, nous spécifions l'ensemble des chemins, dont r est une racine, par une fonction t définie comme suit :

$$t \in ND \setminus \{r\} \rightarrow ND$$

La fonction t précise que chaque nœud du graphe g possède un parent unique, à l'exception de la racine r. De ceci, nous pouvons définir un cycle dans un graphe fini g comme un sous-ensemble de nœuds c dont ses éléments sont des membres de l'image inverse de c sous t :

$$c \subseteq t^{-1}[c]$$

Afin de garantir la non existence d'un cycle dans un arbre, nous devons prouver que l'ensemble (c) est égal à l'ensemble vide. Formellement, nous décrivons cette propriété de la façon suivante :

$$\forall c.(c \subseteq ND \wedge c \subseteq t^{-1}[c] \Rightarrow c = \varnothing)$$

Afin de pouvoir utiliser cette propriété comme une règle d'induction, Abrial J-R. et al. [46] ont proposé une autre définition, équivalente à l'axiome précédent, présentée comme suit :

$$\forall q.(q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q)$$

Enfin, un arbre recouvrant d'un graphe connexe g est défini comme suit :

$$tr = \{t \in ND - \{r\} \rightarrow ND \wedge \forall q.(q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q) \wedge t \subseteq g\}$$

6.3 Le contexte *Label*

Le contexte *label* permet de spécifier l'ensemble des labels qui peuvent encoder les états des nœuds et des arêtes. Dans notre cas de figure, les états des arêtes peuvent être encodés soit par TRUE, soit par FALSE, et donc seuls les labels des nœuds qui vont être spécifiés dans ce contexte. Nous désignons par *label* l'ensemble qui regroupe tous les labels possibles des nœuds. L'ensemble *label* est spécifié comme suit :

$$partition(label, N, A, A1, F, END)$$

6.4 La première machine

La première machine permet de spécifier l'objectif de l'algorithme d'une façon très abstraite. Cette machine voit les contextes *graphe* et *Arbre*. Elle est constituée d'un seul événement, nommé *span*, qui calcule la solution dans une seule étape. Cet événement permet de retourner un arbre recouvrant quelconque du graphe g. Dans ce niveau, nous avons besoin d'une seule variable *st* pour sauvegarder l'arbre recouvrant résultant.

$$st \in ND \leftrightarrow ND$$

EVENT span
BEGIN
$act1 : st :\in tr$
END

6.5 La deuxième machine

La deuxième machine raffine la première décrite ci-dessus. Elle introduit des nouvelles variables et de nouveaux événements qui codifient les nouveaux comportements qui peuvent être aperçus dans ce niveau. Nous désignons par *new_tree* l'arbre en cours de construction. Nous définissons les variables : *tr_nodes* est l'ensemble des nœuds qui appartiennent à *new_tree*, à l'opposé de l'ensemble *remaining_nodes*. Les invariants suivants permettent de typer les nouvelles variables :

$inv1 : tr\_nodes \subseteq ND$
$inv2 : remaining\_nodes \subseteq ND$
$inv3 : tr\_nodes \cap remaining\_nodes = \varnothing$
$inv4 : tr\_nodes \cup remaining\_nodes = ND$
$inv5 : new\_tree \subseteq g$
$inv6 : new\_tree \in tr\_nodes \setminus \{r\} \rightarrow tr\_nodes$
$inv7 : \forall q \cdot q \subseteq tr\_nodes \wedge r \in q \wedge new\_tree^{-1}[q] \subseteq q \Rightarrow tr\_nodes \subseteq q$

Le nouvel événement *progrees* permet de calculer progressivement l'arbre recouvrant. Bien entendu, cet événement va permettre de spécifier un calcul distribué mais sans tenir compte des règles de ré-étiquetage. Ces dernières seront développées, plus tard, dans la troisième machine. Formellement, l'événement progress est décrit comme suit :

EVENT progrees
ANY $s1, s2$
WHERE
$grd1 : s1 \in tr\_nodes$
$grd2 : s2 \in remaining\_nodes$
$grd3 : s1 \mapsto s2 \in g$
THEN
$act1 : new\_tree := new\_tree \cup \{s2 \mapsto s1\}$
$act2 : tr\_nodes := tr\_nodes \cup \{s2\}$
$act3 : remaining\_nodes := remaining\_nodes \setminus \{s2\}$
END

6.6 La troisième machine

Dans ce raffinement, nous allons passer d'un calcul global à un calcul local de l'arbre recouvrant. Autrement dit, nous allons pouvoir localiser toutes les transitions des états des nœuds. Concrètement, nous introduisons la variable *lab* pour encoder les labels des nœuds, et nous ajoutons le contexte *Label* à l'ensemble des contextes qui peuvent être vus par la machine. Suite à l'ajout de cette variable, nous serons capables d'observer plus d'événements, à savoir les règles de ré-étiquetage. Les variables abstraites *tr_nodes* et *remaining_nodes* seront remplacées. Maintenant, *remaining_nodes* constitue l'ensemble des nœuds étiquetés N, et *tr_nodes* constitue l'ensemble des nœuds étiquetés A, $A1$, F, ou END. Les invariants $inv2$ et $inv3$ présentés ci-dessous sont appelés des invariants de collage.

$inv1 : lab \in ND \rightarrow label$
$inv2 : tr\_nodes = lab^{-1}[\{END\}] \cup lab^{-1}[\{F\}] \cup lab^{-1}[\{A\}] \cup lab^{-1}[\{A1\}]$
$inv3 : remaining\_nodes = lab^{-1}[\{N\}]$

Nous renforçons la garde de l'événement *span* par une nouvelle condition ($grd1$) qui montre que le calcul de l'arbre recouvrant est fini si, et seulement si, il existe un nœud étiqueté END. L'événement *progress* est désormais raffiné par d'autres événements, il est remplacé par les événements $règle1$ et $règle2$ qui décrivent simultanément les règles de ré-étiquetage 1 et 2. Le nouvel événement $règle3$ spécifie la troisième règle de ré-étiquetage. Les gardes 2, 3 et 4 de l'événement $regle3$ spécifient le contexte interdit de la règle.

EVENT regle1
ANY $s1, s2$
WHERE
$grd1 : lab[\{s1\}] = \{A\}$
$grd2 : lab[\{s2\}] = \{N\}$
$grd3 : s1 \mapsto s2 \in g$
THEN
$act1 : new\_tree := new\_tree \cup \{s2 \mapsto s1\}$
$act2 : lab := (lab \setminus \{s2 \mapsto N\}) \cup \{s2 \mapsto A1\}$
END

EVENT regle2
ANY $s1, s2$
WHERE
$grd1 : lab[\{s1\}] = \{A1\}$
$grd2 : lab[\{s2\}] = \{N\}$
$grd3 : s1 \mapsto s2 \in g$
THEN
$act1 : lab := (lab \setminus \{s2 \mapsto N\}) \cup \{s2 \mapsto A1\}$
$act2 : new\_tree := new\_tree \cup \{s2 \mapsto s1\}$
END

EVENT regle3
ANY $s1$
WHERE
$grd1 : lab[\{s1\}] = \{A1\}$
$grd2 : \neg(\exists x \cdot s1 \mapsto x \in g \Rightarrow lab[\{x\}] = \{N\})$
$grd3 : \neg(\exists x, y \cdot x \neq y \wedge s1 \mapsto x \in g \wedge s1 \mapsto y \in g \Rightarrow lab[\{x\}] = \{A1\} \wedge lab[\{y\}] = \{A1\})$
$grd4 : \neg(\exists x, y \cdot x \neq y \wedge s1 \mapsto x \in g \wedge s1 \mapsto y \in g \Rightarrow lab[\{x\}] = \{A\} \wedge lab[\{y\}] = \{A1\})$
THEN
$act1 : lab := (lab \setminus \{s1 \mapsto A1\}) \cup \{s1 \mapsto F\}$
END

Si l'événement $regle4$ survient, le graphe devient irréductible (aucune règle ne peut être appliquée), l'arbre recouvrant est calculé et l'algorithme est considéré terminé.

EVENT regle4
ANY $s1$
WHERE
$grd1 : lab[\{s1\}] = \{A\}$
$grd2 : \neg(\exists x \cdot s1 \mapsto x \in g \Rightarrow lab[\{x\}] = \{N\})$
$grd3 : \neg(\exists x \cdot s1 \mapsto x \in g \Rightarrow lab[\{x\}] = \{A1\})$
$grd4 : \neg(\exists x \cdot s1 \mapsto x \in g \Rightarrow lab[\{x\}] = \{A\})$
THEN
$act1 : lab := (lab \setminus \{s1 \mapsto A\}) \cup \{s1 \mapsto END\}$
END

7 La plateforme Rodin

Rodin est une plateforme extensible qui a pour objectif le développement des modèles *B événementiel* et la preuve de leurs corrections. Dans cette section, nous allons expliquer le mode d'utilisation, ainsi que l'architecture de la plateforme.
L'utilisation de Rodin est extrêmement simple. D'abord, l'utilisateur est invité à écrire ou importer sa spécification, ce qui déclenche systématiquement une vérification statique. Cette dernière désigne la vérification lexicale et syntaxique de la spécification et l'analyse des types. Une fois la vérification statique est terminée avec succès, Rodin peut passer à la génération des obligations de preuve qui seront par la suite transmises aux preuveurs automatiques. Le générateur d'obligation de preuve permet de produire des théorèmes spécifiques qui représentent, une fois démontrés, une preuve de la correction du modèle. Les prouveurs génèrent des inférences destinées à être utilisées pour démontrer les obligations de preuve. Cependant, une obligation de preuve ne peut pas être toujours démontrée automatiquement. Ceci peut être dû à une obligation de preuve incorrecte ou à une preuve qui n'a pas pu être terminée avec les hypothèses données. Dans un tel cas, il est important que l'utilisateur interagisse avec la palteforme Rodin. Il doit apporter quelques modifications au niveau de la spécification dans la cas où l'obligation de preuve est incorrecte. D'autre part, il est utile d'aider les prouveurs à démontrer les obligations de preuves en ajoutant d'autres hypothèses dans le cas où l'obligation de preuve est correcte. Toutefois, Rodin est capable de réutiliser autant de fois des anciennes preuves pour démontrer d'autres obligations. Enfin, une spécification est dite correcte si, et seulement si, toutes les obligations de preuves générées sont correctement démontrées. La figure II.4 illustre bien le processus de développement formel adopté par Rodin.

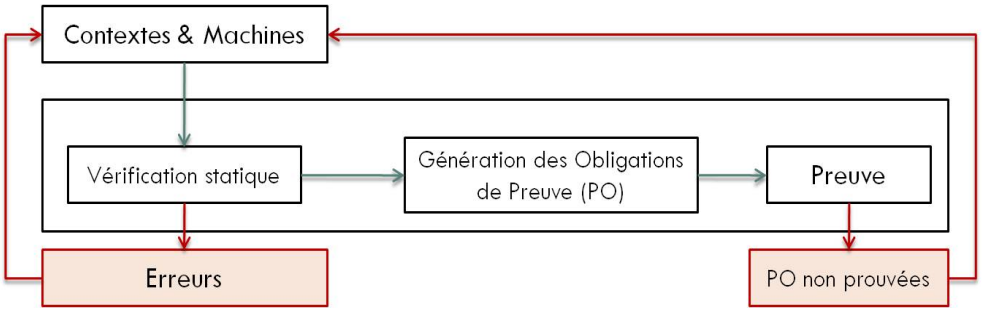


Figure II.4 – Structure de la plateforme Rodin

Techniquement, Rodin est un client lourd d'Eclipse [2]. Il est composé d'un ensemble de plug-ins dont les outils de spécification et de preuve et une interface utilisateur. Rodin se compose des plug-ins suivants :

- *Plateforme Rodin* : ce plug-in maintient les fonctionnalités de base de la plateforme Rodin : le stockage et la gestion des machines, des contextes, des obligations de preuve et des preuves.
- *Noyau B événementiel* : ce plug-in personnalise la plateforme Rodin pour le formalisme *B événementiel*. Il définit tous les éléments nécessaires à la modélisation d'un système *B événementiel* : Static Checker (vérificateur statique de la syntaxe et du type), générateur d'obligation de preuves et les proveurs.
- *Arbre syntaxique abstrait de B événementiel* : ce plug-in contient une bibliothèque qui permet de représenter et manipuler les formules mathématiques de *B événementiel* sous forme d'arbres syntaxiques abstraits.
- *Proveur des séquents B événementiel* : ce plug-in contient une bibliothèque qui permet de prouver les séquents.
- *Interface utilisateur* : ce plug-in offre deux interfaces graphiques différentes à savoir l'interface de la spécification et celle de la preuve.

8 Preuve des algorithmes distribués

Depuis l'apparition de la méthode *B événementiel*, de nombreux travaux de recherche ont été développés. Ces travaux couvrent un large spectre de domaines liés aux systèmes réactifs. Parmi eux, nous distinguons les domaines d'algorithmique distribuée, séquentielle et concurrente. Le livre [37], proposé par J-R. Abrial, fait l'objet d'une étude détaillée de tous ces domaines. L'auteur de ce livre a montré, non seulement, la possibilité d'aborder une multitude de domaines, mais aussi la puissance de sa méthode formelle dans la production des systèmes corrects par construction.

Dans cette section, nos intérêts portent essentiellement sur le domaine d'algorithmique distribuée. En effet, nous mentionnons qu'il existe, dans la littérature, un nombre

2. Eclipse est un Environnement de Développement Intégré (http://www.eclipse.org)

important d'exemples d'algorithme distribué qui ont été spécifiés avec la méthode *B événementiel*. Ces exemples ont permis de prouver que le système obtenu, après une suite de raffinements successives, est correct tout en tenant compte des propriétés du modèle initial. Parmi ces exemples, nous citons les suivants : le protocole de transfert de fichier [37], l'algorithme de découverte de la topologie [47], l'algorithme d'élection dans un anneau [15], l'algorithme de routage pour un agent mobile [37], l'algorithme de dénombrement des références [48], etc.

Dans [48], D. Cansell et al. ont présenté un développement formel guidé par la technique de raffinement de l'algorithme de dénombrement des références. Cet algorithme a été précédemment prouvé avec l'assistant de preuve Coq dans [49]. Ceci n'a pas empêché les auteurs de traiter encore une fois l'algorithme pour insister sur l'intérêt du raffinement dans la réduction de la complexité de preuve et la génération d'un système correct. Dans [47], T-S. Hoang et al. ont fourni une spécification, d'un algorithme de découverte de la topologie, qui inclut à la fois des propriétés de sécurité et de vivacité. Pour réaliser la preuve de ces propriétés, les auteurs ont combiné convenablement les preuves d'invariants, de raffinement, de convergences, et de non-blocage.

Dans le cadre du projet RIMEL [3], D. Méry et D. Cansell [15, 50, 51] ont affirmé, à travers des études de cas, l'efficacité de la méthode *B événementiel* et ses techniques de preuve dans le développement des systèmes distribués corrects par construction. Les études de cas choisies sont des algorithmes distribués codés par le modèle de calculs locaux et validées par l'outil Rodin. Les principaux algorithmes étudiés sont l'algorithme de calcul d'arbre recouvrant [46, 52] et l'algorithme d'élection d'un chef dans un arbre [15]. En outre, leurs travaux visent à systématiser des développements guidés par la relation de raffinement tout en assurant le maintien de propriétés entre les modèles abstraits et concrets. Ainsi, leurs travaux sont considérés comme le point de départ de nos études de recherche dans cette thèse. D'un certain point de vue, nous considérons que toutes nos contributions sont complémentaires à leurs travaux puisque elles s'inscrivent dans les mêmes objectifs tracés par le projet RIMEL. De ce fait, les suivants chapitres présentés dans cette thèse vont insister sur la combinaison de la méthode *B événementiel* avec les calculs locaux en tant qu'un puissant outil de codage et de preuve des algorithmes distribués.

9 Conclusion

L'objectif de ce chapitre est de montrer un état de l'art de la méthode formelle *B événementiel*. Pour le réaliser, nous avons, d'abord, essayé de situer la méthode dans le cadre de l'approche *correct-par-construction*. Ensuite, nous avons détaillé les éléments essentiels du langage B. Par le biais d'un exemple, nous avons essayé d'illustrer les concepts théoriques que nous avons présentés. Enfin, nous avons montré que des travaux existants ont réussi d'utiliser la méthode dans la preuve de correction des algorithmes distribués codés par le modèle des calculs locaux.

3. http ://rimel.loria.fr

En effet, ceci nous a permis de conclure que le degré de similitude entre les modèles des calculs locaux et ceux de la méthode *B événementiel* est assez élevé. Nous présentons ci-dessous les principaux points de similitude :

1. L'abstraction des modèles de calculs locaux est aisément capté par le formalise de développement *B événementiel*. Par exemple, une règle de ré-étiquetage est facilement représentée par un événement.
2. La méthode *B événementiel* peut étendre le raffinement des calculs locaux jusqu'au niveau d'échange des messages, en plus elle peut même résoudre des problèmes fondés sur un choix probabiliste [53].
3. La rigueur de raisonnement et de preuve, sur des propriétés du modèle des calculs locaux, offre un moyen simple pour vérifier la correction des algorithmes distribués avec la méthode *B événementiel*.

Dans le chapitre suivant nous essayons de rendre systématique, le plus possible, le développement incrémental des algorithmes distribués, soutenu par la méthode formelle *B événementiel*. Comme solution, nous élaborons un schéma de développement réutilisable appelé aussi patron de conception formelle.

Chapitre III

Techniques de preuves d'algorithmes distribués par raffinement

1 Introduction

Dans cette thèse, nous avons accumulé une expertise en matière de spécification et de preuve d'algorithmes distribués en *B événementiel*. En effet, notre expérience nous a permis de confronter des problématiques récurrentes dans le domaine de l'algorithmique distribuée et d'en concevoir chaque fois les solutions appropriées. Toutefois, il apparaît que les solutions que nous avons conçues partagent souvent des éléments communs que se soit sur la méthodologie ou sur le contenu des spécifications elles mêmes.

Notre objectif dans ce chapitre est de fournir une idée générale de la façon avec laquelle nous spécifions et nous prouvons la correction des algorithmes distribués codés par les calculs locaux. Nous allons, à cet effet, proposer un patron de spécification formelle des algorithmes distribués (Nous pouvons l'appeler aussi un patron de conception prouvée). Dans [54], Vincent Couturier définit le patron comme une abstraction de logiciels utilisés par des concepteurs et des programmeurs avancés dans leurs programmes. L'idée étant de capitaliser un savoir-faire et d'offrir aux usagers un gain de temps et d'efficacité en proposant des solutions déjà testées et expérimentées pour des problèmes récurrents. Dans notre contexte, le patron que nous proposons est une description générale et générique des solutions possibles à des problèmes de spécification et de preuves d'algorithmes distribués.

Formellement et en *B événementiel*, nous considérons un patron de spécification formelle comme un modèle *B événementiel* qui peut être instancié et/ou réutilisé en partie pour spécifier un algorithme distribué correct par construction. Le modèle est défini par un ensemble de contextes et par trois niveaux spécifiant le passage de la spécification de l'abstraction à la concrétisation. Chaque niveau est décrit par une machine générale qui peut être instanciée et raffinée par une ou plusieurs machines, et ceci dépend du problème traité.

Dans la suite de ce chapitre, nous présentons d'abord un aperçu sur l'ensemble des travaux connexes et nous nous focalisons sur les travaux qui ont proposé des patrons

de spécifications prouvés avec la méthode *B événementiel*. Ensuite, nous décrivons notre patron de spécifications des algorithmes distribués codés par les calculs locaux. Nous expliquons avec plus de détails chaque niveau du patron. Enfin, nous illustrons l'utilisation de notre patron avec deux exemples d'algorithmes distribués.

2 Patrons de conception formels

Dans [55], C. Alexander et al. attestent que le patron est un moyen de décrire le fondement des solutions possibles à un problème qui survient à plusieurs reprises dans notre environnement. Il peut être utilisé tant de fois afin de produire de différentes solutions à un problème donné (essentiellement l'architecture des bâtiments). Cette définition est aussitôt adoptée par E. Gamma et al. [56] pour l'appliquer au contexte de conception orienté objet. Dans un tel contexte, un patron détient quatre éléments essentiels [56] :

1. Le nom du patron : le nom est une annonce au sujet du problème traité par le patron.
2. Le problème traité : le problème sur lequel nous pouvons appliquer le patron.
3. La solution du problème : elle décrit les éléments de la solution qui composent la conception, leurs relations, etc.
4. Les conséquences : les conséquences d'un patron comprennent son impact sur la flexibilité, l'extensibilité, ou la portabilité d'un système.

Ainsi, selon les auteurs, l'utilisation d'un patron de conception fournit une aide considérable dans la conception des logiciels orientés objet. Dans la méthode B classique, l'idée de patron est implicitement incorporée. En effet, la réutilisation des composants est mise en œuvre par les clauses INCLUDES et IMPORTS qui sont respectivement utilisées au niveau de la spécification et de l'implémentation. Dans [57], S. Blazy et al. ont proposé une approche pour spécifier des systèmes conjointement en UML et en B classique à l'aide des patrons. Cette approche est illustrée par une étude de cas du contrôle d'accès. Le travail [58] introduit une démarche outillée pour réutiliser des patrons de conception formellement spécifiés avec la méthode B. Les auteurs définissent à cet effet trois niveaux de composition : la juxtaposition, la composition à l'aide de liens entre patrons et l'unification.

En *B événementiel*, J.-R. Abrial [37] a récemment suggéré l'introduction des patrons pour un développement formel basé sur l'approche " correct par construction ". Il a définit des patrons qui permettent de réutiliser un comportement basé sur l'action et la réaction pour systématiser un développement formel s'appuyant sur une validation par la preuve. Le patron a été appliqué sur une étude de cas d'un système de contrôle : commande d'une presse. Un autre patron appelé patron de réutilisation, a été suggéré par J-R. Abrial, D. Cansell et D. Méry dans [59, 60]. Dans [61], T. Lecomte et al. ont présenté un aperçu sur les travaux préliminaires concernant les patrons de conception prouvés. D'autre part, nous avons remarqué qu'une activité croissante sur les patrons de conception *B événementiel* [62, 63, 63] a été proposée pour étudier et examiner de nombreuses études de cas liées à

des domaines diversifiés. Cependant, nous avons remarqué qu'il n'existe aucun classement de patron de conception prouvé. Par ailleurs, il n'existe aucun référentiel réel des patrons de développement prouvé pour un langage formel spécifique.

Ainsi, et d'une façon générale, les patrons de conception prouvés issus du langage *B événementiel* se résument tout simplement par un ou plusieurs modèles. Un patron est considéré comme une forme générale et générique de la solution proposée pour exprimer l'aspect ou le comportement à spécifier [64]. Par la suite, l'application du patron à un problème donné, revient à instancier ses différents modèles. En effet, cette tâche d'instanciation ne réussit que si nous identifions rigoureusement les caractéristiques du système étudié. La documentation de référence du patron doit aider à réaliser cette tâche en décrivant le contexte et les caractéristiques des aspects à identifier [64].

Notre travail va permettre d'incorporer dans un patron, les principes de base d'un développement formel prouvé d'un algorithme distribué, et codé par les calculs locaux. Ainsi, le patron proposé permet d'encoder à la fois les propriétés de base des calculs locaux, le formalisme de la spécification et le mécanisme du raffinement. Dans la section suivante, nous présentons notre patron de conception formel des algorithmes distribués. Nous insistons sur le mécanisme de raffinement que nous utilisons pour transformer la spécification d'un niveau abstrait à un autre concret.

3 Présentation du patron

Dans cette section nous présentons notre patron de spécification formelle d'algorithmes distribués. Nous expliquons comment il pourra être utilisé pour spécifier un algorithme correct par construction. Rappelons que dans ce travail nous ne utilisons que le modèle des calculs locaux pour encoder les algorithmes distribués. Le patron que nous avons proposé s'articule autour de trois principaux niveaux. Chaque niveau est spécifié par une machine qui permet de capturer toutes les propriétés communes à des spécifications des algorithmes distribués. Dans la figure III.1, nous présentons la structure et les différents niveaux du patron.

Le niveau M0 : Ce niveau est concrétisé par une machine abstraite qui permet de calculer la solution de l'algorithme dans une seule étape. Concrètement, un seul événement est nécessaire pour spécifier la solution de l'algorithme. Cet événement modélise le passage direct de l'état initial à l'état final du système. Ce premier niveau permet de répondre à une toute première question : que fait l'algorithme ?

Le niveau M1 : Le raffinement de $M0$ en $M1$ permet d'introduire les propriétés inductives du système. Ces propriétés permettent d'exprimer un calcul global dans le système distribué. Le niveau $M1$ peut être instancié par plusieurs machines et ce en fonction de la complexité de l'algorithme traité. La deuxième question à laquelle ce deuxième niveau répond est : comment fait l'algorithme pour atteindre son objectif (d'un point de vue global) ?

Le niveau M-LC : Ce dernier niveau (Machine Local Computation) permet d'introduire toutes les règles de ré-étiquetage de l'algorithme. Ce niveau est instancié par

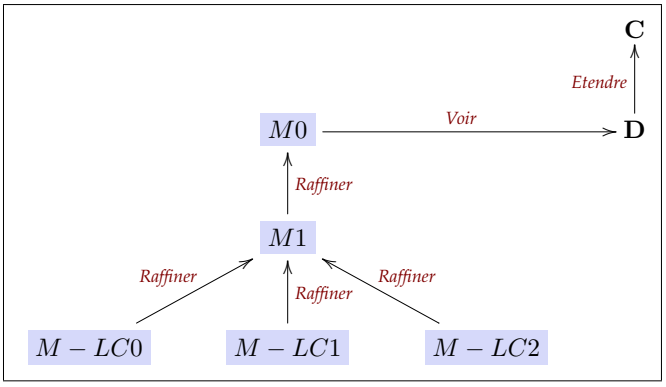


Figure III.1 – Patron de spécification formelle des algorithmes distribués

une seule machine qui décrit les changements locaux des états des nœuds et des arêtes. Avec ce niveau, nous serons capables de répondre à la question suivante : Comment fait l'algorithme pour atteindre son objectif d'un point de vue local ?

En plus des machines, il existe des contextes ayant des définitions particulières qui sont nécessaires et souvent présents dans une spécification *B événementiel* d'un algorithme distribué. Le premier contexte est appelé "C", il définit le domaine d'application de l'algorithme distribué : graphe, arbre, anneau... . Le second contexte est appelé "D", définit comme une extension du "C", inclut les propriétés locales des nœuds et des arêtes. Ces propriétés constituent l'ensemble des labels qui peuvent étiqueter les nœuds et les arêtes du graphe.

4 Développement formel des contextes

4.1 Le contexte C

Le contexte **C** spécifie les propriétés d'un réseau sur lequel les algorithmes distribués doivent fonctionner. Formellement, un réseau peut être modélisé par un graphe connexe, non-orienté et simple. Les nœuds du graphe désignent les processeurs. Les arêtes correspondent aux liens de communications directes. Un graphe est simple s'il n'a ni arêtes multiples ni boucles (axm4 et axm3). Un graphe non orienté signifie que ses arêtes sont symétriques (axm3). Un graphe est connexe, si et seulement si, pour chaque paire de nœuds il existe un ensemble d'arêtes qui les relient (axm5). Nous précisons que dans la spécification de ce contexte, nous avons repris en partie les définitions usuelles de la théorie des graphes et les travaux de Jean-Raymond Abrial et al. [36, 46]. Ainsi, un graphe g modélisé par un ensemble de nœuds ND, peut être présenté comme suit :

$axm1 : g \subseteq ND \times ND$
$axm2 : dom(g) = ND$
$axm3 : g = g^{-1}$
$axm4 : id(ND) \cap g = \varnothing$
$axm5 : \forall s.s \subseteq ND \wedge s \neq \varnothing \wedge g[s] \subseteq s \Rightarrow ND \subseteq s$

4.2 Le contexte D

Le contexte **D** est une extension du contexte **C**. Il définit tous les labels qui peuvent décorer la représentation abstraite du réseau. Il convient de rappeler que cette représentation est déjà définie dans le contexte **C**.

Dans la définition Def.I.13, nous avons vu qu'un système de ré-étiquetage de graphes (GRS) est un triplet composé de l'ensemble des labels possibles L, l'ensemble des labels de l'état initial I et l'ensemble des règles de ré-étiquetages P. Formellement, nous distinguons deux sous-ensembles dans L ; soient LN et LE qui spécifient l'ensemble de tous les labels qui peuvent étiqueter respectivement les nœuds et les arêtes. De la même façon, nous distinguons deux sous-ensembles dans I ; soient $Init\_LN$ et $Init\_LE$ qui spécifient l'ensemble des labels indispensables pour l'étiquetage respectivement de l'état initial des nœuds et des arêtes. Bien entendu, $init\_LE$ et $init\_LN$ doivent être non vides (axm6). Nous vérifions avec l'axiome $axm2$ que LN contient au moins deux labels. Ceci est dans le but d'assurer que GRS soit opérationnel, c.-à-d. les règles de ré-étiquetages appartenant à P peuvent opérer un calcul distribué ayant un sens.

Dans la définition Def.I.17, nous avons montré que l'ensemble des graphes irréductibles peut être obtenu après l'application des règles de ré-étiquetages sur une classe de graphe I-étiqueté. Afin de spécifier l'ensemble des graphes irréductibles, nous considérons deux nouveaux ensembles non vides $final\_LN$ et $final\_LE$. Ils représentent respectivement l'ensemble des labels qui peuvent étiqueter les nœuds et les arêtes des graphes irréductibles (axm3 et axm5). L'union des deux ensembles $init\_LN$ [resp. $init\_LE$] et $final\_LN$ [resp. $final\_LE$] est égale à LN [resp. LE] (axm7 et axm8). Cependant, l'intersection des ensembles $init\_LN$ et $final\_LN$ [resp. $init\_LE$ et $final\_LE$] n'est pas forcément différente de l'ensemble vide. Ainsi, nous spécifions l'ensemble des graphes irréductibles par un ensemble non vide (axm10) qui est inclu au produit cartésien des nœuds et des arêtes dans leurs états finals (axm9). Nous appelons cet ensemble "*solution*". Ci-dessous, nous présentons les principaux axiomes du contexte **D**.

$axm1 : finite(LN) \wedge finite(LE)$
$axm2 : card(LN) > 1 \wedge card(LE) \geq 1$
$axm3 : final\_LN \subseteq LN \wedge final\_LE \subseteq LE$
$axm4 : init\_LN \subseteq LN \wedge init\_LE \subseteq LE$
$axm5 : final\_LN \neq \varnothing \wedge final\_LE \neq \varnothing$
$axm6 : init\_LN \neq \varnothing \wedge init\_LE \neq \varnothing$
$axm7 : init\_LN \cup final\_LN = LN$
$axm8 : init\_LE \cup final\_LE = LE$
$axm9 : solution \subseteq (g \rightarrow final\_LE) \times (ND \rightarrow final\_LN)$
$axm10 : solution \neq \varnothing$

5 Développement formel des machines

Après avoir défini les contextes C et D, nous entamons maintenant la présentation des trois principaux niveaux de la partie dynamique d'une spécification formelle d'un algorithme distribué.

5.1 Le premier niveau

Le premier niveau est concrétisé par une seule machine. Celle-ci permet seulement de spécifier le résultat (l'objectif global) de l'algorithme sans entrer dans le détail du calcul distribué, vu qu'elle est placée dans un niveau d'abstraction très élevé. Un seul événement est nécessaire pour calculer le résultat prétendu de l'algorithme en une seule fois. Nous appelons cet événement *oneshot*. Dans un système de ré-étiquetage de graphes, l'événement *oneshot* permet de transformer immédiatement un graphe $I-étiqueté$ en un autre irréductible.

Dans la littérature, il existe des travaux qui ont adopté ce même principe de développement formel. A titre d'exemple, nous citons le développement de l'algorithme de MAZURKIEWEZ dans [51]. Aussi dans [15], les auteurs ont spécifié un algorithme d'élection (le protocole IEEE 1394) avec un seul événement de type *oneshot* dans la première machine. De même, nous avons appliqué ce principe dans le développement des algorithmes de calcul d'arbre recouvrant [65].

Formellement, nous définissions une variable appelée *result* afin de récupérer la solution retournée par l'algorithme. L'événement *oneshot* vérifie d'abord si la variable *result* est égale à l'ensemble vide comme elle a été instanciée, ensuite il choisi d'une façon indéterministe un élément de l'ensemble solution, et enfin il attribue cet élément à la variable *result*. Afin d'éviter des soucis de preuve, nous avons choisi de déclarer *result* par une paire ordonnée de deux éléments au lieu d'une variable atomique. De cette façon, les obligations de preuves générées par Rodin seront prouvées en utilisant seulement les prouveurs interactifs.

$result \in (g \leftrightarrow final\_LE) \leftrightarrow (ND \leftrightarrow final\_LN)$

Formellement, l'événement *oneshot* est présenté comme suit :

EVENT oneshot
  ANY
    x, y
  WHERE
    $grd1 : result = \varnothing$
    $grd2 : x \mapsto y \in solution$
  THEN
    $act1 : result := \{x \mapsto y\}$
END

5.2 Le deuxième niveau

L'objectif du deuxième niveau est d'introduire l'essence de l'algorithme : comment les nœuds convergent vers un état final qui représente une solution de l'algorithme. Nous rappelons que le calcul distribué des nœuds est vu seulement d'une façon globale. Nous décrivons ce niveau par une machine abstraite très générale. Cette machine peut être suivie par plusieurs raffinements afin d'en obtenir d'autres plus concrètes et plus adaptées pour l'introduction des règles de ré-étiquetages.

Bien entendu, le raffinement et le bon choix des invariants du système rendent plus simple la démonstration des obligations de preuves générées par l'outil RODIN. D'une part, le raffinement est basé sur un raisonnement approuvé et prouvé par le concepteur et donc issue d'une réflexion humaine. D'autre part, trouver les invariants convenables du système est à la fois une tâche ardue et non systématique car chaque algorithme a son spécificité. Pour ces raisons, l'automatisation d'un tel raisonnement est très difficile voire impossible. Par ailleurs, dans ce niveau nous n'allons pas se concentrer sur le raffinement, plutôt, nous allons essayer de définir une machine générale qui se place au dessus des spécificités des algorithmes distribués et qui retracent l'objectif de ce niveau.

Formellement, cette machine raffine la machine du niveau précédant. Elle raffine, alors, l'événement *oneshot* et ajoute deux autres appelés : *calcul+* et *calcul-*. Nous déclarons deux nouveaux variables *ln* et *le*. Ces variables sont ensuite définies dans la clause Invariants par deux fonctions. La fonction *ln* [resp. *le*] attribue un label à chaque nœud [resp. arêtes] du graphe. Formellement, *ln* et *le* sont spécifiées de la façon suivante :

$$ln \in ND \rightarrow LN$$
$$le \in g \rightarrow LE$$

L'événement d'instantiation permet d'étiqueter les nœuds [resp. arêtes] du graphe par des labels inclus dans *init_LN* [resp. *init_LE*]. La version concrète de l'événement *oneshot* remplace les variables locales x et y par respectivement deux instances de *le* et *ln*. Ces deux événements sont spécifiés comme suit :

EVENT INITIALISATION
BEGIN
$act1 : le :\in g \rightarrow init\_LE$
$act2 : ln :\in ND \rightarrow init\_LN$
END

EVENT oneshot
REFINES oneshot
WHEN
$grd1 : result = \varnothing$
$grd2 : le \mapsto ln \in solution$
WITNESSES
$x : x = le$
$y : y = ln$
THEN
$act1 : result := \{le \mapsto ln\}$
END

L'événement *calcul+* décrit le calcul progressif de la solution de l'algorithme. Il encode l'activité des nœuds dans leurs incursions vers un état final du graphe. Il spécifie un calcul distribué sur un nombre inconnu de nœuds dans le graphe. Les variables locales de cet événement sont :

- *nodes* : l'ensemble des nœuds qui vont changer leurs états.
- *edges* : l'ensemble des arêtes qui relient les nœuds de nodes et qui vont éventuellement changer leurs états.
- *new_label_nodes* : l'ensemble des labels indispensables pour l'étiquetage des états des nœuds inclus dans *nodes*.
- *new_label_nodes* : l'ensemble des labels indispensables pour l'étiquetage des états des arêtes inclues dans *edges*.
- *new_state_nodes* : le nouvel état des nœuds du graphe sous jacent composé par *nodes*.
- *new_state_edges* : le nouvel état des arêtes du graphe sous jacent composé par *nodes*.

Les conditions de l'événement *clacul+* sont spécifiées de la façon suivante :

$grd1 : le \mapsto ln \notin solution$
$grd2 : nodes \subseteq ND$
$grd3 : edges \subseteq g$
$grd4 : \exists a \cdot a \in nodes \wedge a \in ln^{-1}[init\_LN]$
$grd5 : \exists at \cdot at \in edges \wedge at \in le^{-1}[init\_LE]$
$grd6 : (\forall a \cdot a \in nodes \Rightarrow (\exists b \cdot b \in nodes \wedge a \mapsto b \in g)) \wedge (edges \subseteq nodes \lhd g \rhd nodes)$
$grd7 : new\_label\_nodes \subseteq final\_LN$
$grd8 : new\_label\_edges \subseteq final\_LE$
$grd9 : new\_state\_nodes \in nodes \nrightarrow new\_label\_nodes$
$grd10 : new\_state\_edges \in edges \nrightarrow new\_label\_edges$
$grd11 : new\_state\_nodes \neq \varnothing$
$grd12 : \exists x \cdot x \in dom(new\_state\_nodes) \Rightarrow ln(x) \neq new\_state\_nodes(x)$

La garde *grd1* permet de vérifier si le graphe est non irréductible et donc ne correspond

à aucune solution de l'algorithme. La garde *grd4* [resp. *grd5*] permet de s'assurer que parmi les nœuds de *nodes* il existe ceux qui sont étiquetés avec des labels de *init_LN* [resp. *init_LE*]. L'intérêt de cette garde est de montrer qu'il existe dans *nodes* des nœuds actifs qui vont éventuellement ré-étiqueter leurs états. La garde *grd6* permet de vérifier que le graphe sous jacent formé par les nœuds de *nodes* et les arêtes de *edges* est connexe. Pour la garde *grd9*, nous avons choisi de définir la variable locale *new_state_nodes* comme une fonction partielle entre les nœuds de *nodes* et l'ensemble des labels qui leurs sont associées. Ce choix est justifié par le fait que *nodes* peut inclure des nœuds actifs et autres passifs, et par conséquent le ré-étiquetage ne peut concerner qu'une partie de l'ensemble *nodes*. Pour la même raison, nous avons défini la variable locale *new_state_edges* comme une fonction partielle entre les arêtes de *edges* et les labels qui leurs sont associés (voir *grd10*).

La garde *grd11* évalue la variable *new_state_node*. Cette garde vérifie si l'ensemble *new_state_node* est différent de l'ensemble vide, et ce pour garantir que l'événement *calcul+* peut opérer un ré-étiquetage sur des nœuds du graphe. Toutefois, cette garde est insuffisante pour affirmer que l'événement réalise un vrai ré-étiquetage. Car le nouvel étiquetage du graphe sous jacent formé par *nodes* peut être égal à l'ancien étiquetage particulièrement dans le cas où il existe des labels commun entre *init_LN* et *final_LN*. Par conséquent, nous précisons dans la garde *grd12* que le nouvel état des nœuds est bel et bien différent de l'ancien.

Finalement les actions de l'événement permet de ré-étiqueter les états des nœuds et des arêtes inclus respectivement dans *nodes* et *edges*. Les substitutions généralisées de l'événement *calcul+* sont présentées comme suit :

$$act1 : ln := ln \lhd\!\!+ new\_state\_nodes$$
$$act2 : le := le \lhd\!\!+ new\_state\_edges$$

Par ailleurs, nous somme en mesure de confirmer que parfois certains algorithmes distribués peuvent considérer des règles de ré-étiquetages qui opèrent d'une manière opposée à ce que nous avons vu avec l'événement *calcul+*. Autrement dit, les nœuds peuvent ré-étiqueter leurs états et passer d'un étiquetage final (*final_LN*) à un autre initial (*init_LN*). Formellement, nous spécifions ce calcul distribué par l'événement *calcul-*. Cet événement permet de réaliser un ré-étiquetage des labels des nœuds et des arêtes de l'état final vers un état initial. Les variables locales, les conditions et les actions de cet événement seront formulées de la même façon que celles de *calcul+*. La différence entre les deux événements est donnée comme suit :

- *grd4* : les nœuds actifs sont étiquetés par des labels inclus dans l'ensemble *final_LN*.
- *grd5* : les arêtes sont étiquetées par des labels inclus dans l'ensemble *final_LE*.
- *grd7* : *new_label_nodes* est spécifié comme un sous ensemble de *init_LN*.
- *grd8* : *new_label_edges* est spécifié comme un sous ensemble de *init_LE*.

5.3 Le troisième niveau

L'intérêt de l'ajout de ce dernier niveau est de spécifier un calcul qui porte sur l'état local d'un nœud ou d'une arête du graphe. Toutefois, la spécification formelle, depend strictement du modèle de calculs locaux implémentés : LC0, LC1 ou LC2. Nous avons précisé dans le chapitre I, que chacun de ces modèles est caractérisé par un type particulier de règle de ré-étiquetage. La différence d'une règle par rapport à une autre se résume dans leurs champs d'application (une boule de rayon 1 ou deux nœuds adjacents) et dans leurs ré-étiquetages autorisés. Typiquement cette différence est aussitôt perceptible sur la spécification formelle d'un algorithme distribué. Autrement dit, chaque modèle de calculs locaux est spécifié d'une façon particulière. Dans la suite de cette section, nous allons essayer de faire apparaître les éléments communs et élémentaires dans la spécification de chaque modèle.

Dans ce niveau, nous spécifions chaque modèle par une machine résultant du raffinement de celle développée dans le niveau précédant. Dans la spécification des machines, nous gardons l'événement *oneshot* et nous remplaçons l'événement *calcul+* (éventuellement *calcul-*) par d'autres événements. Ces événements spécifient maintenant les règles de ré-étiquetages de l'algorithme et ayant le même nombre de celles-ci. Dans le développement d'un algorithme concret, et en fonction du modèle qui l'implémente, ce niveau sera concrétisé par une seule machine. Nous présentons dans la suite de cette section, les propriétés les plus importantes des événements qui caractérisent chaque machine.

5.3.1 Modèle de calculs locaux de type 0

La règle de type LC0 opère sur deux nœuds adjacents du graphe. Elle autorise un ré-étiquetage sur les deux nœuds adjacents et sur l'arête qui les relie. Ce type de règle est spécifié par un événement ayant les éléments de base suivants :

La clause variable locales : nous déclarons deux nœuds adjacents x et y qui vont exécuter une règle de ré-étiquetage. Nous déclarons aussi, les nouveaux labels de x, de y et de leur arête commune. Ces nouveaux labels sont respectivement nommés new_label_x, new_label_y et new_label_edge.

La clause *Witness* : Les *Witness* suivants permettent de remplacer les variables abstraites introduites dans l'événement *calcul+* par d'autres concrètes.

$edges : edges = \{x \mapsto y, y \mapsto x\}$
$nodes : nodes = \{x, y\}$
$new\_state\_nodes : new\_state\_nodes = \{x \mapsto new\_label\_x, y \mapsto new\_label\_y\}$
$new\_state\_edges : new\_state\_edges = \{(x \mapsto y) \mapsto new\_label\_edge\}$
$new\_label\_nodes : new\_label\_nodes = \{new\_label\_x, new\_label\_y\}$
$new\_label\_edges : new\_label\_edges = \{new\_label\_edge\}$

La clause garde : nous spécifions les conditions élémentaires et nécessaires pour exécuter une règle de ré-étiquetage de type LC0. D'autres éventuelles conditions auxiliaires utiles aux calculs peuvent être ajoutées lors de l'initiation du patron. Ces conditions sont présentées comme suit :

$$
\begin{array}{l}
grd1 : le \mapsto ln \notin solution \\
grd2 : x \mapsto y \in g \\
grd3 : new\_label\_x \in final\_LN \\
grd4 : new\_label\_y \in final\_LN \\
grd5 : new\_label\_edge \in final\_LE \\
grd6 : ln(x) \in init\_LN \\
grd7 : le(x \mapsto y) \in init\_LE
\end{array}
$$

La clause action : nous n'apportons aucune modification sur les substitutions généralisées de l'événement abstrait *calcul+*.

5.3.2 Modèle de calculs locaux de type 1

Une règle de type LC1 est une règle de ré-étiquetage qui opère sur une boule de rayon 1 et qui permet de réécrire les états du nœud centre de la boule, et des arêtes issues de ce nœud. Dans aucun cas, cette règle peut ré-étiqueter les feuilles de la boule.

Ainsi, pour spécifier cette règle nous raffinons l'événement *calcul+* et nous apportons les modifications suivantes : le nœud centre de la boule est déclaré par une simple variable appelée par exemple c. Par conséquent, les feuilles de la boule sont représentées par l'ensemble $g[\{c\}]$ et les arêtes issues de c par $\{c\} \lhd g$. Également, nous déclarons les nouveaux labels présumés du nœud c et des arêtes de la boule respectivement par les variables new_label_c et new_label_edge. La variable new_state_edges est raffinée et désormais remplacée par une nouvelle variable appelée $sphere\_edge$. Cette nouvelle variable permet d'associer les arêtes issues du nœud c à leurs nouveaux labels prétendus. Formellement, cette variable est définie comme suit : $sphere\_edge \in \{c\} \lhd g \nrightarrow new\_label\_edge$

L'ensemble des *Witness* de cette événement est défini comme suit :

$$
\begin{array}{l}
edges : edges = \{c\} \lhd g \\
nodes : nodes = \{c\} \cup g[\{c\}] \\
new\_state\_nodes : new\_state\_nodes = \{c \mapsto new\_label\_c\} \\
new\_state\_edges : new\_state\_edges = sphere\_edge \\
new\_label\_nodes : new\_label\_nodes = \{new\_label\_c\} \\
new\_label\_edges : new\_label\_edges = new\_label\_edge
\end{array}
$$

Les deux substitutions généralisées de l'événement sont données comme suit :

$$
\begin{array}{l}
act1 : ln := ln \lhd\!\!+ \{c \mapsto new\_label\_c\} \\
act2 : le := le \lhd\!\!+ sphere\_edge
\end{array}
$$

5.3.3 Modèle de calculs locaux de type 2

Les règles de ré-étiquetages de type LC2 opèrent aussi sur une boule de rayon 1. Ces règles se distinguent de celles de type LC1 par le fait qu'elles autorisent en plus le ré-étiquetage des feuilles de la boule. Formellement, la spécification d'une telle règle ne diffère pas trop de celle de type LC1. Plus précisément, nous ajoutons dans la spécification de l'événement "calcul+", la possibilité de ré-étiqueter aussi les feuilles de la boule. Pour cela, nous déclarons une variable, appelée *sphere_node*, pour associer à tous les nœuds de la boule leurs nouveaux labels une fois la règle est appliquée. Dans la clause $WHERE$ de l'événement *calcul+*, nous spécifions cette nouvelle variable comme suit : $sphere\_node : (g[\{c\}] \cup \{c\} \rightarrow new\_label\_node)$. La variable *new_label_node* représente l'ensemble des labels qui appartiennent à $final\_ln$ et qui peuvent être attribués aux nœuds de la boule. La variable *sphere_node* est différente de l'ensemble vide $sphere\_node \neq \varnothing$. Les *Witness* de l'événement sont spécifiés comme suit :

$$
\begin{array}{l}
edges : edges = \{c\} \lhd g \\
nodes : nodes = \{c\} \cup g[\{c\}] \\
new\_state\_nodes : new\_state\_nodes = sphere\_node \\
new\_state\_edges : new\_state\_edges = sphere\_edge \\
new\_label\_nodes : new\_label\_nodes = new\_label\_node \\
new\_label\_edges : new\_label\_edges = new\_label\_edge
\end{array}
$$

Dans la clause $THEN$ de l'événement *clacul+*, les éléments de *sphere_node* vont écraser les anciens éléments de *ln*, ce qui permet de ré-étiqueter les états des nœuds centre et feuilles de la boule.

$$
\begin{array}{l}
act1 : ln := ln \lhd\!\!+ sphere\_node \\
act2 : le := le \lhd\!\!+ sphere\_edge
\end{array}
$$

6 Exemples

Dans cette section, nous expliquons comment le patron proposé peut être appliqué pour spécifier des algorithmes distribués. A cet effet, nous avons choisi les deux algorithmes distribués suivants : le calcul d'arbre recouvrant et la coloration d'un anneau. Ces algorithmes sont codés respectivement par les modèles de calculs locaux LC0 et LC1. Dans [62], Abrial, J-R. et al. indiquent que l'usage d'un patron de conception "orienté objet" résulte des techniques d'adaptation et d'incorporation de certains de ses éléments prédéfinies dans le projet logiciel. Dans un contexte formel, ces deux techniques sont mises en œuvre de la façon suivante :

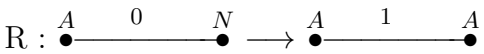
– L'adaptation d'un patron formel, spécifié en *B événementiel*, consiste essentiellement à instancier ses constantes, ses variables, et ses événements afin d'avoir certains éléments qui sont adaptés au problème de spécification.

– L'incorporation d'un patron formel, spécifié en *B événementiel*, dans une spécification consiste à intégrer certains de ses éléments dans le modèle traité.

Dans ce qui suit, nous allons essayer d'appliquer ces deux techniques pour spécifier les deux algorithmes choisis.

6.1 Calcul d'arbre recouvrant

L'algorithme de calcul d'arbre recouvrant permet de créer une arborescence hiérarchique sur l'ensemble du réseau. Il détermine tous les chemins redondants entre les processus et choisit un seul d'entre eux. L'algorithme que nous avons choisi est implémenté avec une seule règle de ré-étiquetage de type LC0. Initialement, nous supposons qu'un seul nœud est dans un état "actif" (codé par le label A), tous les autres étant dans un état "neutre" (codés par le label N) et toutes les arêtes ne sont pas marquées (codées par le label 0). Un nœud étiqueté A peut activer un de ses voisins neutres et marquer l'arête correspondante. Une fois tous les nœuds sont étiquetés A, l'arbre constitué par les arêtes marquées, est considéré calculé. La règle de ré-étiquetage est ci-dessous présentée :



Nous appliquons notre patron de conception formelle sur l'exemple ci-dessus présenté. Bien entendu, nous étendrons les contextes C et D, et nous spécifions chaque niveau du patron par une seule machine. Le but d'étendre le contexte C est de spécifier l'arbre qui est supposé être le résultat de l'exécution de l'algorithme. L'extension du contexte D permet de définir l'ensemble des labels des nœuds et des arêtes. La première machine ainsi que la deuxième sont des adaptations des machines proposées dans le premier et le deuxième niveau du patron. La spécification formelle de ces deux machines est déjà présentée dans la section 6 du chapitre II. Dans la suite de cette section, nous allons uniquement présenter la dernière machine.

6.1.1 Extension du contexte D

L'ensemble des labels qui peuvent être employés par l'algorithme sont $LN = \{A, N\}$ et $LE = \{0, 1\}$. Initialement, $init\_LN = LN$ et $init\_LE = \{0\}$. Quand tous les nœuds terminent l'exécution de l'algorithme alors $final\_LN = \{A\}$ et $final\_LE = LE$. La solution de l'algorithme est définie par un arbre étiqueté dont les arêtes et les nœuds sont respectivement "marquées" et "actifs" (voir $axm8$).

$axm1 : LN = \{A, N\}$
$axm2 : init\_LN = \{A, N\}$
$axm3 : final\_LN = \{A\}$
$axm4 : LE = \{0, 1\}$
$axm5 : init\_LE = \{0\}$
$axm6 : final\_LE = \{0, 1\}$
$axm7 : A \neq N$
$axm8 : solution = \{sol, a \cdot a \in trees \wedge$
$sol = \{((a \times \{1\}) \cup ((g \setminus a) \times \{0\}))\} \times \{ND \times \{A\}\} | sol\}$

6.1.2 Niveau 3

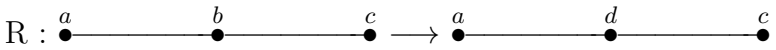
La dernière machine du modèle est une adaptation de la machine $M2 - LC0$. Elle contient les événements *oneshot*, *initialisation* et *Rule*. L'événement *Rule* est une spécification de la règle de ré-étiquetage de l'algorithme.

EVENT Rule
ANY
x, y
WHERE
$grd1 : le \mapsto ln \notin solution$
$grd2 : x \mapsto y \in g$
$grd3 : ln[\{x\}] = \{A\}$
$grd4 : le[\{x \mapsto y\}] = \{0\}$
$grd5 : ln[\{y\}] = \{N\}$
WITNESSES
$new\_label\_x : new\_label\_x = A$
$new\_label\_y : new\_label\_y = A$
$new\_label\_edge : new\_label\_edge = 1$
THEN
$act1 : ln := ln \Leftarrow (\{x \mapsto A, y \mapsto A\})$
$act2 : le := le \Leftarrow \{(x \mapsto y) \mapsto 1\}$
END

6.2 Coloration d'un anneau

Le deuxième exemple que nous avons choisi est l'algorithme de coloration d'un anneau. Une coloration d'un anneau est un étiquetage qui vérifie que tous les deux nœuds adjacents ont deux couleurs distinctes. Dans cet exemple, le nombre de couleurs qui correspond aussi au nombre de labels est fixé à trois. La seule règle de ré-étiquetage qui encode cet algorithme est présentée ci-dessous (nous supposons que x, y, et z sont les

trois couleurs possibles qui peuvent colorier les nœuds). Le développement formel de l'algorithme commence d'abord par la définition mathématique de l'anneau. Le contexte qui définit l'anneau résulte de l'extension du contexte **C**. Ensuite, nous définissons l'ensemble des couleurs (l'ensemble des labels qui peuvent être attribués aux nœuds) ainsi que l'ensemble des solutions possibles. Ceci est donné par une extension du contexte **D**. Enfin, nous développons les trois niveaux de la partie dynamique de la spécification.



a,b,c,d ∈ {x,y,z} ; b ∈ {a,c} ; d ∉ {a,c}

6.2.1 Extension du contexte C : le contexte anneau

Un anneau est un graphe constitué d'un cycle simple dont tous les nœuds sont de degré 2. Formellement, nous spécifions un anneau, appelé **rg**, par l'union de deux cycles orientés r et son image inverse r^{-1} (voir axm11).

$$axm10 : rg \subseteq ND \times ND$$
$$axm11 : rg = r \cup r^{-1}$$

r est alors un cycle orienté qui relie tous les nœuds du graphe deux à deux. Toutefois, la définition d'un cycle nécessite des preuves de fermeture transitives. A cet effet, nous déclarons la variable cr comme étant la fermeture transitive de r (voir les axim3, axm4, axm5, et axm6). Nous prouvons par un théorème que r est égal à cr, ceci nous permet de monter qu'il existe un chemin reliant n'importe quelles paires de nœuds du graphe. Autrement dit, le cycle r s'étale sur tout le graphe. Jusqu'ici nous avons une certitude sur le fait que r relie tous les nœuds du graphe, mais pas forcément deux à deux. A cet effet, nous ajoutons par le biais de l'axiome axm7 une autre propriété du cycle r : la relation bijective entre les nœuds. De cette façon, nous somme certain que chaque nœud possède un seul parent et un seul fils et donc deux nœuds voisins. Nous vérifions par l'axiome axm8 qu'il existe un cycle unique dans le graphe. Enfin, nous éliminons les boucles sur un nœud quelconque du graphe (voir axm9). Les axiomes du contexte anneau sont présentés comme suit :

$$axm1 : r \subseteq ND \times ND$$
$$axm2 : \forall s \cdot (s \subseteq ND \times ND \wedge ND \neq \varnothing \wedge s;r \subseteq s \Rightarrow ND \times ND \subseteq r)$$
$$axm3 : cr \subseteq ND \times ND$$
$$axm4 : r \subseteq cr$$
$$axm5 : cr;r \subseteq cr$$
$$axm6 : \forall u \cdot (u \subseteq ND \times ND \wedge r \subseteq u \wedge u;r \subseteq u \Rightarrow cr \subseteq u)$$
$$axm7 : r \in ND \rightarrowtail ND$$
$$axm8 : \forall s \cdot (s \subseteq ND \wedge s \neq \varnothing \wedge s \subseteq r^{-1}[ND] \Rightarrow ND \subseteq s)$$
$$axm9 : \forall i \cdot i \in ND \wedge i \in dom(r) \Rightarrow i \neq r(i)$$

6.2.2 Extension du contexte D : le contexte coloration

Dans ce contexte, nous définissons les labels qui peuvent être employés par l'algorithme dans l'étiquetage des nœuds de l'anneau. Rappelons que dans cet algorithme les arêtes ne sont pas étiquetées. Formellement, ce contexte est une extension du contexte "D". Nous déclarons la constante $COLORS$ comme l'ensemble des labels des nœuds de l'anneau. $COLORS$ est égale à l'ensemble LN qui est aussi égal à $init\_LN$ et $final\_LN$ car initialement les nœuds sont tous étiquetés (colorés) d'une façon aléatoire. Cet ensemble renferme les labels "BLUE", "WHITE" et "RED" (voir axm1). A l'aide de l'axiome axm2, nous spécifions toutes les solutions possibles de l'algorithme à savoir l'ensemble des étiquetages qui correspondent aux solutions de 3-colorations de l'anneau rg.

$axm1 : partition(COLORS, \{BLEU\}, \{WHITE\}, \{RED\})$
$axm2 : solution = \{x, ncolor \cdot x \in ND \wedge ncolor \in ND \rightarrow COLORS$
$\wedge ncolor(x) \neq ncolor(r(x)) \wedge ncolor(x) \neq ncolor(r^{-1}(x)) | ncolor\}$

6.2.3 Niveau 3

Le dernière machine est une instance de $M - LC1$. Elle permet de spécifier la seule règle de ré-étiquetage de l'algorithme. A cet effet, nous déclarons une fonction qui permet d'attribuer une couleur à chaque nœud du graphe. Formellement, cette fonction est décrite de la façon suivante : $t \in ND \rightarrow COLORS$. Le nouvel événement, appelé *coloring*, permet de vérifier si le nœud centre de la boule (la boule ne contient ici que trois nœuds), a une couleur identique au moins à une de ses deux voisins (grd2). Si c'est le cas, nous choisissons d'une façon aléatoire une autre couleur différente des celles des voisins du nœud c (grd3 et grd4). Cette couleur va être ensuite attribuée au nœud c dans la section *Action*. L'événement *coloring* est défini comme suit :

EVENT coloring
ANY
c, ac
WHERE
$grd1 : c \in ND$
$grd2 : t(c) \in t[rg[\{c\}]]$
$grd3 : ac \in COLORS$
$grd4 : ac \notin t[rg[\{c\}]]$
THEN
$act1 : t(c) := ac$
END

7 Conclusion

Dans ce chapitre, nous avons présenté un patron de conception formelle des algorithmes distribués. Notre contribution consiste essentiellement à définir un modèle général et réutilisable basé sur la technique de raffinement et l'encodage d'algorithmes distribués par les calculs locaux. Nous avons illustré notre modèle par deux exemples d'algorithmes distribués : le calcul d'arbre recouvrant et la coloration d'anneau.

Dans le suivant chapitre, nous allons essayer de renforcer notre modèle par une nouvelle technique de preuve à savoir la décomposition. D'une part, la combinaison du raffinement avec la décomposition permet de réduire la complexité du développement formel surtout pour les algorithmes distribués complexes. D'autre part, la décomposition facilite beaucoup la réutilisation des spécifications déjà prouvées et validées.

Ultérieurement, dans le chapitre VI, nous allons se baser sur ce modèle pour proposer une nouvelle approche pour le test et la simulation des spécifications formelles des algorithmes distribués. Cette approche, appelée B2Visidia, permet de générer automatiquement à partir d'une spécification *B événementiel* d'un algorithme distribué, un code Java destiné à être exécuté sous Visidia. Plus précisément, nous nous servons du $3^{ème}$ niveau du modèle pour définir un sous langage de *B événementiel* traduisible par B2Visidia.

Chapitre IV

Approche de décomposition d'algorithmes distribués

1 Introduction

La méthode *B événementiel* propose de nombreuses techniques pour réduire la complexité d'un développement formel. La technique la plus utilisée est le raffinement. Dans un travail précédant (voir chapitre III), nous avons proposé un patron de développement formel des algorithmes distribués basé sur le raffinement. Nous avons montré, avec des exemples, comment le raffinement permet de garantir la correction du développement via un processus de spécification incrémental. Aussi, nous avons souligné l'importance du raffinement dans la diffusion de la complexité des preuves et la validation de l'intégration des invariants du système.

Toutefois, le raffinement des modèles de grandes tailles peut s'avérer une tâche complexe et très fastidieuse. Dans un tel cas, la méthode *B événementiel* propose une autre technique qui permet de décomposer un modèle en plusieurs parties. Chaque partie peut être par la suite spécifiée séparément. Ces deux techniques peuvent être utilisées simultanément comme un outil pour prouver la correction des algorithmes distribués.

Dans ce chapitre, nous présentons une approche pour spécifier des algorithmes distribués en combinant des composants déjà prouvés. Techniquement, nous utilisons le raffinement et la décomposition pour surmonter la complexité de la preuve d'un algorithme distribué. L'approche proposée s'inscrit complètement dans l'approche "*correct-par-construction*" et elle se limite aux algorithmes distribués codés par le modèle de calculs locaux de type handshake.

Dans la suite de ce chapitre, nous présentons, dans la Section 2, un aperçu sur les travaux connexes. Ensuite, nous développons, dans la Section 3, les conditions suffisantes pour réussir la décomposition d'un algorithme distribué. Autrement dit, ces conditions assurent

que les exécutions entrelacées des sous-algorithmes, qui résultent de la décomposition, soient équivalentes à une exécution séquentielle. Par la suite, nous présentons dans la Section 4, notre approche de décomposition. Dans la Section 5, nous illustrons cette approche avec un algorithme de calcul d'arbre recouvrant avec détection locale de la terminaison globale. Cet algorithme sera décomposé en deux sous-algorithmes : le calcul d'arbre recouvrant et le Dijkstra-Scholten. Enfin, nous clôturons ce chapitre par une conclusion dans laquelle nous récapitulons la contribution et quelques travaux futures.

2 Les travaux connexes

Les efforts entrepris pour mettre en œuvre la réutilisation des composants logiciels formellement prouvés, ont fait apparaître deux principales approches : la décomposition et la composition. La première approche vise à diviser une spécification en plusieurs parties et à développer d'une façon indépendante chacune d'elles. La seconde approche consiste à définir des composants réutilisables et non spécifiques à un problème particulier. Une fois assemblées, ces parties forment une spécification du système traité. Dans ce contexte, les premiers et principaux travaux réalisés sont ceux de Abadi L. et al. [66] et Jones [67].

La méthode *B événementiel* ne gère pas la composition et la décomposition, contrairement à la méthode B classique, qui intègre des mécanismes de composition comme les clauses INCLUDES, EXTENDS et USES. Des travaux de recherche, relativement récents, ont été proposés pour que les mécanismes de composition et de décomposition soient une partie intégrante de la méthode *B événementiel*. Parmi ces deux mécanismes, notre intérêt va porter essentiellement, dans ce chapitre, sur l'approche de la décomposition. Deux approches pour décomposer les machines *B événementiel* ont été proposées, la première, développée par J.-R. Abrial et al. [59, 68], propose une décomposition basée sur les variables d'état. Tandis que, la deuxième, présentée par M. Butler et K. Damchoom dans [69], propose une décomposition basée sur les événements. Plus précisément, un événement atomique abstrait peut être décomposé en plusieurs sous-événements, dont au moins un qui raffine l'événement abstrait.

La décomposition d'un modèle traitant les algorithmes distribués apparaît dans certains travaux de recherches [70, 71]. T.S. Hoang et al. [71] présentent une méthode pour développer des programmes parallèles en utilisant le raffinement et la décomposition. E. Ball et al. [70] utilisent le mécanisme de décomposition pour développer des systèmes à base d'agents mobiles (MAS). Dans ce travail, les auteurs ont illustré leur approche par un protocole d'interaction MAS. Toutefois, à notre connaissance, aucun travail traitant le modèle des calculs locaux par les techniques de raffinement et de décomposition n'a été proposé. Nous proposons dans ce chapitre une approche de vérification et de preuve d'algorithmes distribués en combinant à la fois les mécanismes de raffinement et de décomposition. Les algorithmes considérés sont encodés par le modèle des calculs locaux de type handshake. Formellement, notre approche se base sur celle proposée par J.-R. Abrial

et al. [59, 68]. Elle propose de décomposer la spécification dans un niveau très abstrait et ce pour réduire la complexité de preuve et de vérification. Par ailleurs, les différentes étapes de l'approche ont été définies en respectant les propriétés de composition/décomposition présentées dans le cadre du projet LOCO [14]. Ce projet a cerné un ensemble de propriétés suffisantes qui permettent de réussir une composition/décomposition d'un ou de plusieurs algorithmes codés par le modèle de calculs locaux de type handshake. Les propriétés sont ensuite vérifiées et implémentées avec l'assistant de preuve COQ [12]. Une description détaillée de ces différentes propriétés sera présentée dans la section suivante.

3 Propriétés suffisantes pour la décomposition des algorithmes distribués

L'objectif de cette section est de définir un ensemble de propriétés suffisantes pour rendre possible la décomposition d'un algorithme distribué. Ces propriétés permettent de vérifier la correction d'une composition de plusieurs algorithmes distribués.

Le principe sur lequel se base ces propriétés est simple, il se résume comme suit : supposons qu'un algorithme C de type LC0 est décomposé en deux sous algorithmes A et B de même types, alors toute exécution entrelacée de A et B sur un même graphe est équivalente à une exécution séquentielle de A suivie par une exécution de B.

Nous supposons que les résultats d'exécution des deux algorithmes A et B sont respectivement T_A et T_B. De ce fait, T_B est dit compatible avec T_A si, et seulement si, pour toute exécution de A, T_A est un état initial pour l'algorithme B. La terminologie utilisée dans les définitions présentées ci-dessous peut être décrite de la manière suivante :

- $\mathcal{G}$, $\mathcal{G}'$, $\mathcal{G}_0$.. sont des graphes étiquetés.
- I_{T_A} et I_{T_B} sont deux états initiaux pour les algorithmes A et B.
- BA_T est une relation avant-après d'une exécution d'un algorithme distribué. Par exemple, $(\mathcal{G}, \mathcal{G}') \in BA_{T_A}$ signifie que $\mathcal{G}'$ est un résultat possible d'une exécution de l'algorithme A sur le graphe étiqueté $\mathcal{G}$.
- $\mathcal{G}_0 \xrightarrow{A} \mathcal{G}'_1$ correspond à un pas de calcul de l'algorithme A sur un graphe étiqueté $\mathcal{G}_0$.
- $\mathcal{G}_0 \xrightarrow{A \cup B^*} \mathcal{G}_1$ correspond à plusieurs pas de calcul des algorithmes A et B sur un graphe étiqueté $\mathcal{G}_0$.
- $NF(A)$ est la forme normale de l'algorithme A. C'est l'ensemble de toutes les solutions possibles (graphes étiquetés) issues des exécutions de l'algorithme A.

Définition IV.1. *Compatibilité des algorithmes :*
$\forall \mathcal{G}\ \mathcal{G}',\ \mathcal{G} \in I_{T_A} \wedge (\mathcal{G}, \mathcal{G}') \in BA_{T_A} \Rightarrow \mathcal{G}' \in I_{T_B}$

La compatibilité des algorithmes assure que les connaissances acquises lors de l'exécution du premier algorithme sont suffisantes pour résoudre le deuxième algorithme. A titre d'exemple, les connaissances acquises après l'exécution d'un algorithme de calcul d'arbre recouvrant sont suffisantes pour exécuter un algorithme d'élection.

La deuxième propriété assure que toute exécution entrelacée de A et B est équivalente à une exécution séquentielle des deux algorithmes. Cette propriété formalise le fait que les

algorithmes commutent. Concrètement, les algorithmes A et B commutent si les exécutions B puis A sur un même graphe étiqueté $\mathcal{G}_0$ produisent le même résultat $\mathcal{G}_2$.

Définition IV.2. *B commute avec A :*
$\forall\ \mathcal{G}_0\ \mathcal{G}_1\ \mathcal{G}_2,\ \mathcal{G}_0 \xrightarrow{B} \mathcal{G}_1 \xrightarrow{A} \mathcal{G}_2 \Rightarrow \exists \mathcal{G}'_1,\ \mathcal{G}_0 \xrightarrow{A} \mathcal{G}'_1 \xrightarrow{B} \mathcal{G}_2$

Enfin, afin de vérifier la correction de l'exécution séquentielle des deux algorithmes A et B, nous devons vérifier que l'algorithme B n'altère pas le résultat d'exécution de l'algorithme A. Autrement dit, nous devons garantir la stabilité de la forme normale de l'algorithme A.

Définition IV.3. *Stabilité de la forme normale :*
$\forall\ \mathcal{G}\ \mathcal{G}',\ \mathcal{G} \xrightarrow{B} \mathcal{G}' \wedge \mathcal{G}' \in NF(A) \Rightarrow \mathcal{G} \in NF(A)$

Les trois propriétés présentées ci-dessus permettent d'assurer que l'union des algorithmes A et B réalise respectivement T_A et T_B. Plus précisément :

Lemme IV.1. *Preuve de correction de l'union :*
$\forall \mathcal{G}_i \in I_{T_A},\ \forall \mathcal{G}_f \in NF(A \cup B), \mathcal{G}_i \xrightarrow{A \cup B^*} \mathcal{G}_f \Rightarrow \exists \mathcal{G} \in I_{T_B},\ (\mathcal{G}, \mathcal{G}_f) \in BA_{T_B}$

Une démonstration du Lemme IV.1 peut être présentée comme suit : soit $\mathcal{G}_f$ une forme normale de $A \cup B$ obtenue à partir des exécutions de A et B sur un graphe étiqueté $\mathcal{G}_i$. La commutativité de A et B implique qu'il existe un graphe $\mathcal{G}$ qui résulte d'une exécution de A sur un graphe $\mathcal{G}_i$, et un graphe $\mathcal{G}_f$ qui est le résultat d'une exécution de B sur $\mathcal{G}$. Étant donné que $\mathcal{G}_f \in NF(A \cup B) \Rightarrow \mathcal{G}_f \in NF(A)$ et d'après la définition Def.IV.3, nous savons que $\mathcal{G}$ est une forme normale de A, et donc un état initial de B. Puisque B réalise T_B, alors $(\mathcal{G}, \mathcal{G}_f) \in BA_{T_B}$. Enfin, nous remarquons que si A et B sont noethériens, alors la commutativité assure que leur union est noethérienne.

Ces trois propriétés sont suffisantes pour assurer la correction de la composition/décomposition. Toutefois, si nous voulons étudier des propriétés plus spécifiques, nous serons amenés à renforcer les hypothèses liées à l'exécution de A et de B. Nous introduisons à cet effet, le mode de détection de terminaison de l'algorithme A. La détection de terminaison est un problème bien connu dans les systèmes distribués. Y. Métivier et al. [72] ont défini plusieurs modes de détection de terminaison pour les systèmes de calculs locaux. Nous utilisons, dans ce chapitre deux modes de détection de terminaison : Détection de Terminaison Locale (LTD) et Détection de Terminaison Globale (GTD). Le mode LTD suppose qu'un nœud quelconque du graphe ne peut détecter que sa propre terminaison. Par contre, dans le mode GTD, un nœud peut détecter la terminaison de tous les autres nœuds du graphe.

Cependant, les notions générales de la commutativité et de la stabilité de la forme normale, peuvent être compliquées à prouver formellement. Dans la suite, nous détaillons les propriétés pour exprimer localement les conditions suffisantes pour assurer la composition/décomposition mais d'une façon locale. Les propriétés seront ainsi exprimées en

fonction du mode de détection de la terminaison, des variables partagées et de leurs modes d'accès (lecture/écriture). Dans le but de simplifier la présentation des propriétés, nous supposons que les états sur lesquels A et B opèrent sont des couples. L'élément gauche [resp. droit] de l'état représente les variables modifiées par A [resp. B]. Afin d'assurer la commutativité et la stabilité de la forme normale, nous introduisons les propriétés suivantes :

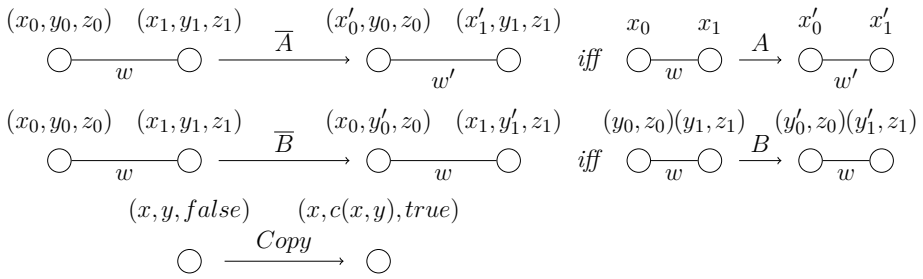
1. A ne modifie que la partie gauche des états des nœuds.
2. A ne dépend que de la partie gauche des états des nœuds.
3. Une fois que A a détecté la terminaison, les valeurs des variables gauches ne changent plus.
4. B ne modifie que la partie droite des états des nœuds.
5. B ne modifie pas les états des arêtes.
6. Les pas de calculs de B ne peuvent dépendre des variables de A que si la terminaison de l'algorithme A a été localement détectée.
7. B peut s'exécuter sur une arête si, et seulement si, la terminaison de l'algorithme A a été détectée sur une des extrémités de l'arête.

D'une façon générale, les algorithmes A et B sont relativement indépendants car ils sont souvent connectés par des variables partagées. Ces variables servent à transmettre les résultats de l'algorithme A à l'algorithme B. En vu de garder la conformité des propriétés locales (1) à (7), nous ajoutons le mécanisme de transfert des résultats suivant :

1. Les pas de calcul de B ne dépendent pas des variables partagées. Ils ne dépendent que des copies locales de ces variables.
2. Le transfert des résultats ne peut se déclencher que si la terminaison locale est détectée.

Formellement, la mise en œuvre du mécanisme de transfert dans la composition/décomposition des algorithmes distribués peut être définie comme suit : Nous supposons que les algorithmes A et B ont des états disjoints L_A et $L_B \times Bool\_var$. La variable booléenne $Bool\_var$ est égale à $true$ si le transfert des résultats est effectué et $false$ dans le cas contraire. Nous désignons par c une fonction dans $L_A \rightarrow L_B \rightarrow L_B$ qui représente l'opération de transfert de résultat de A vers B. Nous désignons par $\overline{A}$ et $\overline{B}$ l'implémentation des deux algorithmes A et B sur L_A et $L_B \times Bool\_var$. Nous désignons par x l'état de l'algorithme A, par y l'état de l'algorithme B et par z une variable booléenne. Après une étape de ré-étiquetage nous marquons les états réécrits par une apostrophe. Nous définissons la composition de A et B par le système de ré-étiquetage suivant :

Définition IV.4. *Composition/décomposition de A et B*



4 Présentation de l'approche

En se basant sur la définition IV.4, nous proposons une approche de développement, d'une classe d'algorithmes distribués, basée à la fois sur le raffinement et la décomposition. L'objectif étant de simplifier la preuve de correction. Ceci est assuré par le fait que les sous algorithmes qui résultent de la décomposition sont plus simples à spécifier et à vérifier. Dans le cas où ces sous-algorithmes ont été déjà spécifiés, alors leurs intégrations dans la spécification ainsi que la réutilisation de leurs obligations des preuves représentent une aide considérable dans la preuve de correction. Intuitivement, nous partons d'un algorithme distribué C qui peut être ensuite divisé en deux sous-algorithmes A et B. Formellement la spécification de C est décomposée en deux spécifications qui correspondent respectivement à A et B. Les différentes étapes suivantes, décrivent l'approche de décomposition que nous avons proposée. Cette dernière est graphiquement représentée par la figure Fig.IV.1.

- Les *Contextes* définissent les propriétés de graphes, ainsi que la solution de l'algorithme spécifié (arbre recouvrant, élection, ...).
- La machine $Machine_0$ spécifie la solution de l'algorithme C dans une seule étape. Concrètement, un seul événement est nécessaire pour modéliser un passage direct de l'état initial à l'état final du système.
- Le raffinement de $Machine_0$ en $Machine_d$ introduit la séquentialité entre les deux sous algorithmes A et B. Ainsi, la machine $Machine_d$ est considérée comme une étape primordiale pour la décomposition de la spécification.
- La machine $Machine_d$ est ensuite décomposée en trois sous-machines : $Machine_{A\_0}$, $Machine_{B\_0}$ et $Machine_{main}$. Les deux premieres sous-machines sont des spécifications abstraites de A et B. La machine $Machine_{main}$ permet de composer les résultats calculés par les deux algorithmes A et B, et représentent ainsi une spécification de leur composition.
- Les raffinements futurs de $Machine_{A\_0}$ et de $Machine_{B\_0}$ vont produire des spécifications concrètes des algorithmes A et B. Ces nouvelles machines vont ainsi permettre de spécifier toutes les règles de ré-étiquetage de A et B.

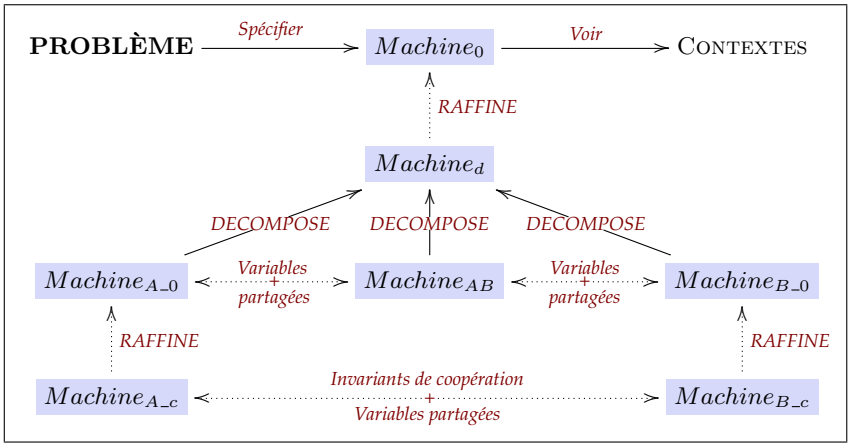


Figure IV.1 – Approche de décomposition

5 Etude de cas

5.1 Présentation

Pour illustrer notre approche, nous avons choisi de combiner les algorithmes de calcul d'arbre recouvrant et de Dijkstra-Scholten. Ce dernier permet de notifier un nœud racine qui va détecter la terminaison globale GTD de l'algorithme de calcul de l'arbre recouvrant. Informellement, l'algorithme peut être décrit de la façon suivante : chaque nœud est initialement étiqueté par son degré dans l'arbre et les arêtes étant non étiquetées. A chaque pas de calcul, une feuille est élaguée (son label est mis à $false$) et le degré de son voisin est décrémenté. Le seul nœud non élagué est alors la racine qui va détecter la terminaison globale. Nous précisons que les nœuds élagués [resp. le nœud racine] restent définitivement dans cet état. La figure Fig.IV.3 [resp. Fig.IV.2] schématise la seule règle de ré-étiquetage de l'algorithme Dijkstra-Scholten [resp. calcul d'arbre recouvrant]. Nous présentons dans la figure Fig.IV.5 un scénario d'exécution de l'algorithme Dijkstra-Scholten.

Figure IV.2 – Règle de ré-étiquetage de l'algorithme de calcul d'arbre recouvrant

Figure IV.3 – Règle de ré-étiquetage de l'algorithme Dijkstra-Scholten

Formellement, ces algorithmes ont été déjà prouvés avec la méthode *B événementiel* dans [15] et dans la section 6.1 du chapitre III. De ce fait, nous n'allons pas se concentrer

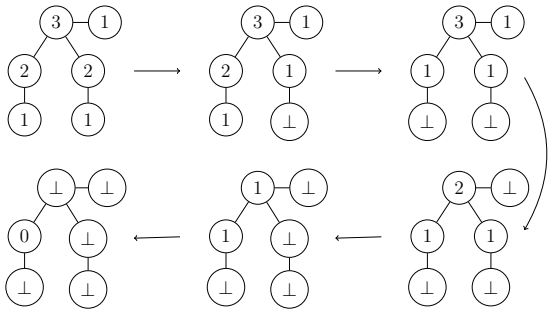


Figure IV.4 – Un scénario d'exécution de l'algorithme Dijkstra-Scholten

sur leurs preuves de correction respectives. Par ailleurs, nous allons mettre l'accent, sur la mise en œuvre de la décomposition et du développement séparé des deux algorithmes.

5.2 Développement formel

La spécification présentée ci-dessous comporte deux contextes et quatorze machines différents. Les deux contextes correspondent respectivement à la définition du graphe (déjà présenté dans la section 6.1 du chapitre II) et de l'arbre recouvrant (auparavant spécifié dans la section 6.2 du chapitre II). Rappelons que ND est l'ensemble des nœuds dans le graphe. g désigne l'ensemble des arêtes qui constituent le graphe et *trees* l'ensemble des arbres recouvrant sur le graphe g. La première machine correspond à l'union des algorithmes ci-dessus cités. Nous raffinons ensuite cette machine, jusqu'à l'obtention d'une autre plus concrète vérifiant les conditions de la décomposition. Nous décomposons le modèle, suivant l'approche que nous avons proposée. Enfin, nous raffinons les machines, issues de la décomposition, jusqu'à l'obtention de deux spécifications concrètes des deux algorithmes. Ces deux spécifications assurent une exécution simultanée des deux algorithmes. L'arbre de raffinement peut être présenté comme suit :

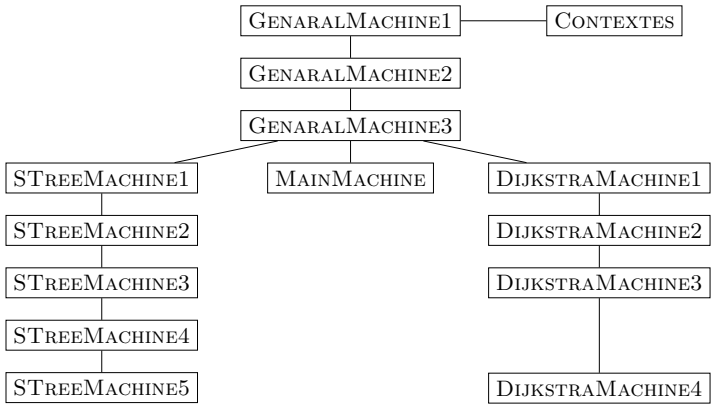


Figure IV.5 – Arbre de raffinement des machines

5.2.1 Modèle initial : GenaralMachine1

La première machine spécifie le résultat de l'algorithme en une seule étape. Noter à ce niveau, que l'algorithme considéré est constitué par l'union de deux tâches : construction de l'arbre recouvrant et la notification d'un nœud "racine". Cette machine comprend un seul événement de type Oneshot [73], défini comme suit :

EVENT MainOneshot
ANY le, tr
WHERE
$grd1 : le \in ND \wedge tr \in trees$
$grd2 : result1 = \varnothing \wedge result2 = \varnothing$
THEN
$act1 : result1 := tr$
$act2 : result2 := \{le\}$
END

5.2.2 1^{er}raffinement : GenaralMachine2

La deuxième machine, qui résulte de ce raffinement, introduit deux nouveaux événements de type *Oneshot* appelés $Oneshot1$ et $Oneshot2$. Ces événements calculent séparément les résultats de $MainOneshot$ et correspondent respectivement aux spécifications abstraites des algorithmes de calcul d'arbre recouvrant et de Dijkstra-Scholten. Ainsi dans cette machine, nous retrouvons aisément la propriété donnée par la définition Def.IV.1. En fait, $Oneshot2$ ne peut être activé que si $Oneshot1$ a été déjà activé. Autrement dit, le résultat de l'algorithme de Dijkstra-Scholten ne peut être calculé qu'après le calcul du celui de l'algorithme de calcul d'arbre recouvrant. Donc, le résultat du premier algorithme est un état initial du deuxième. Ainsi, d'un point de vue abstrait (et aussi global), l'exécution est séquentielle. Mais, localement et comme nous allons le montrer dans les raffinements suivants, le calcul du deuxième algorithme peut démarrer même avant la fin du premier.

EVENT Oneshot1
ANY sp
WHERE
$grd1 : sp \in trees$
$grd2 : Solution\_algo1 = \varnothing$
THEN
$act1 : Solution\_algo1 := sp$
END

EVENT Oneshot2
ANY n
WHERE
$grd1 : n \in ND$
$grd2 : Solution\_algo1 \neq \varnothing$
$grd3 : Solution\_algo2 = \varnothing$
THEN
$act1 : Solution\_algo2 := \{n\}$
END

EVENT MainOneshot
WHERE
$grd1 : resultat\_1 = \varnothing \wedge resultat\_2 = \varnothing$
$grd2 : Solution\_algo1 \neq \varnothing \wedge Solution\_algo2 \neq \varnothing$
WITH
$tr : tr = Solution\_algo1$
$le : \{le\} = Solution\_algo2$
THEN
$act1 : resultat\_1 := Solution\_algo1$
$act2 : resultat\_2 := Solution\_algo2$
END

5.2.3 2èmeraffinement : GeneralMachine3

La troisième machine permet d'observer la terminaison locale de l'exécution du premier algorithme. Elle introduit à cet effet, une nouvelle variable booléenne end_algo1 qui accorde à chaque nœud du graphe son état par rapport à l'exécution du premier algorithme. Initialement, end_algo1 est égale $FALSE$ pour tous les nœuds du graphe. Cette variable est mise à jour par un nouvel événement très abstrait appelé DetectTermination. Il modifie l'état d'un nœud de $FALSE$ à $TRUE$. Plus tard quand la spécification devient plus concrète cette mise à jour ne peut être faite que si le nœud détecte la terminaison local de l'algorithme de calcul d'arbre recouvrant.

L'ajout de cette variable à ce niveau de raffinement est primordial car il prépare la décomposition de la spécification. En effet, il autorise l'exécution du deuxième algorithme sur des nœuds qui ont détecté la terminaison locale du premier algorithme. Intuitivement, la variable end_algo1 sera définie comme une variable partagée entre les algorithmes décomposés (voir Section 5.2.4).

L'invariant ($inv1$) définit formellement la variable end_algo1. L'invariant ($inv2$) spécifie que le résultat de calcul du premier algorithme ne peut être calculé que si tous les nœuds ont détecté la terminaison locale. Plus tard, nous montrons que cette variable permet de supprimer tout référencement à des variables globales.

$inv1 : end\_algo1 \in ND \rightarrow BOOL$
$inv2 : Solution\_algo1 \neq \varnothing \Rightarrow end\_algo1[ND] = \{TRUE\}$

5.2.4 Décomposition

Nous décomposons la machine GeneralMachine3 en trois sous-machines : STree-Machine1, MainMachine et DijkstraMachine1. STreeMachine1 spécifie l'algorithme de calcul d'arbre recouvrant. Elle inclut deux événements internes appelés

DetectTerminaison et *Oneshot*1, et sans aucun événement externe. Également, elle inclut seulement les variables partagées *end_algo*1 et *Solution_algo*1. DijkstraMachine1 correspond à la spécification de l'algorithme de Dijkstra-Scholten. Elle comprend un seul événement interne, *Oneshot*2, et deux événements externes *DetectTerminaison* et *Oneshot_algo*1. Les variables de cette machines sont tous partagées : *Solution_algo*2, *end_algo*1 et *Solution_algo*1. La machine MainMachine combine les résultats des deux autres machines. Elle regroupe tous les variables partagées et les événements externes des autres machines. Cette machine demeure très abstraite et ne sera jamais raffinée. Techniquement, la décomposition est réalisée grâce à un plug-in qui a été rajouté à l'outil RODIN. Le plug-in de décomposition est téléchargeable depuis le site web suivant [1].

5.2.5 1^{er}raffinement : StreeMachine2

Le premier raffinement de StreeMachine1 permet de calculer l'arbre recouvrant d'une manière progressive. A cet effet, il introduit une nouvelle variable appelée *new_tree* pour spécifier l'arbre en cours de construction. Initialement, *new_tree* est vide et grâce à un nouvel événement *progress_algo*1, nous ajoutons les arêtes une par une. A ce niveau, les nœuds sont partagés en deux sous ensembles : les nœuds inclus dans l'arbre *tree_nodes* et les nœuds qui ne le sont pas encore *remaining_node* (pour plus de détail, voir la section 6 du chapitre II). L'événement *DetectTerminaison* est raffiné. Désormais, les nœuds qui détectent la terminaison locale de l'algorithme de calcul d'arbre recouvrant sont ceux qui figurent avec leurs voisins dans l'arbre en cours de construction. La plupart des preuves liées à ce raffinement ont été récupérées à partir des anciens développements.

EVENT DetectTerminaison
ANY x
WHERE
$grd1 : end\_algo1(x) = FALSE$
$grd2 : x \in tree\_nodes$
$grd3 : g[\{x\}] \subseteq tree\_nodes$
THEN
$act1 : end\_algo1(x) := TRUE$
END

5.2.6 $2^{ème}$raffinement : StreeMachine3

L'objectif de ce raffinement est de localiser toutes les données globales. En occurrence, nous remplaçons toutes les variables déclarées dans la machine précédente (*tree_nodes*, *remaining_nodes* et *new_tree*) par les nouvelles variables locales (*n_label* et *e_label*). *n_label* [resp. *e_label*] est une fonction d'étiquetage des nœuds [resp. arêtes]. Les nouvelles variables sont définies par les invariants suivants :

1. wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide

$inv1 : n\_label \in ND \rightarrow node\_label$
$inv2 : e\_label \in g \rightarrow edge\_label$
$inv3 : n\_label^{\sim}[\{N\}] = remaining\_nodes$
$inv4 : n\_label[\{r\}] = \{A\}$
$inv5 : tree\_nodes = n\_label^{\sim}[\{A\}]$
$inv6 : \forall x, y \cdot x \mapsto y \in g \Rightarrow e\_label(x \mapsto y) = e\_label(y \mapsto x)$
$inv7 : new\_tree = dom(e\_label \rhd \{Marque\})$
$inv8 : \forall x, y \cdot n\_label(x) = N \wedge x \mapsto y \in g \Rightarrow e\_label(x \mapsto y) = Init$
$inv9 : \forall x, y \cdot (n\_label(x) = A \wedge n\_label(y) = A) \wedge x \mapsto y \in g$
$\quad \wedge x \mapsto y \notin new\_tree \wedge e\_label(x \mapsto y) \neq Init \Rightarrow e\_label(x \mapsto y) = N\_Marque$
$inv10 : \forall x, y \cdot (n\_label(x) = A \wedge n\_label(y) = A) \wedge x \mapsto y \in g$
$\quad \wedge x \mapsto y \notin new\_tree \wedge e\_label(x \mapsto y) \neq N\_Marque \Rightarrow e\_label(x \mapsto y) = Init$
$inv11 : \forall x \cdot end\_algo1(x) = TRUE \Rightarrow (\forall a \cdot a \mapsto x \in g \Rightarrow e\_label(a \mapsto x) \neq Init)$

L'invariant (inv1) [resp. inv2] donne une définition formelle de la fonction d'étiquetage des nœuds [resp. arêtes]. Le premier invariant spécifie n_label par une fonction totale de ND vers l'ensemble $node\_label$. Cet ensemble correspond aux états possibles d'un nœud : Actif A ou Neutre N ($partition(node\_label, A, N)$). Le deuxième invariant spécifie e_label par une fonction totale de g vers l'ensemble $edge\_label$. L'ensemble $edge\_label$ énumère les états possibles d'une arête : $Marque$ si l'arête est incluse dans l'arbre, N_Marque si l'arête n'est pas incluse dans l'arbre sachant que ses extrémités sont toutes les deux $Actif$, et $Init$ si au moins une seule extrémité de l'arête est $Neutre$ (voir inv8)($partition(edge\_label, Marque, N\_Marque, Init)$). L'état $Init$ peut, aussi, correspondre à une arête ayant des extrémités étiquetées A, mais qui ne figure pas dans l'arbre (voir inv10). Dans ce cas, la différence entre $Init$ et N_Marque est que l'état N_Marque est final.

Les invariants (inv3), (inv5) et (inv7) permettent de remplacer les variables abstraites ajoutées par la machine précédente. Désormais, l'ensemble des nœuds $tree\_nodes$ [resp. $remaining\_nodes$] est désigné par les nœuds étiquetés A [resp. N]. De même, l'arbre en cours de construction new_tree est remplacé par l'ensemble des arêtes marquées $Marque$. L'invariant (inv4) précise que le nœud racine r de l'arbre est étiqueté A. L'invariant (inv6) montre que l'état d'une arête $x \mapsto y$ est égal à l'état de $y \mapsto x$ et ceci est dû au fait que les arêtes ne sont pas orientées. L'invariant (inv11) vérifie qu'un nœud peut détecter la terminaison locale si, et seulement si, toutes les arêtes qui sont issues ne sont pas marquées $Init$.

Toutefois, l'introduction de l'état $Init$ des arêtes, va nous permettre de calculer les degrés des nœuds dans l'arbre. En effet, le degré est une information indispensable pour que le deuxième algorithme puisse faire des pas de calcul complètement locaux. Initialement, toutes les arêtes sont étiquetées $Init$. Une fois les extrémités d'une arête sont marquées A, alors son état peut passer à $Marque$, par l'événement $progress\_algo1$, ou à

N_Marque par un nouvel événement appelé $mark\_edge$. Ci-dessous, nous présentons le code de l'événement $mark\_edge$:

EVENT mark_edge
ANY x, y
WHERE
$grd1 : n\_label(x) = A \land n\_label(y) = A$
$grd2 : x \mapsto y \in g$
$grd3 : e\_label(x \mapsto y) \neq Marque$
$grd4 : e\_label(x \mapsto y) = Init$
THEN
$act1 : e\_label := e\_label \lhd\!\!\!\!- \{x \mapsto y \mapsto N\_Marque, y \mapsto x \mapsto N\_Marque\}$
END

5.2.7 3èmeraffinement : STreeMachine4

A ce niveau de raffinement, les événements spécifiant les règles de ré-étiquetage sont tous localisés. Autrement dit, ils se basent strictement sur les connaissances locales des nœuds. Cependant, l'événement $DetectTerminaison$ demeure toujours non localisé. De ce fait, nous allons raffiner la machine STreeMachine3 pour concrétiser encore une fois la détection de la terminaison locale, et pour calculer les degrés des nœuds dans l'arbre. Cet objectif, s'il est achevé dans une seule étape, va rendre les obligations de preuves très complexes à démontrer. Alors, nous avons décidé de le réaliser progressivement sur deux raffinements.

Dans ce raffinement, nous associons à chaque nœud deux ensembles différents : $marqued$ et $not\_marqued$. Pour un nœud x donné, $marqued(x)$ [resp. $not\_marqued(x)$] correspond à l'ensemble de tous ses voisins dans l'arbre [resp. dans le graphe, mais pas dans l'arbre]. Le raffinement suivant utilise ces deux ensembles pour calculer le degré dans l'arbre. Les invariants présentés ci-dessous, spécifient les deux nouvelles variables $marqued$ et $not\_marqued$.

$inv1 : marqued \in ND \to \mathbb{P}(ND)$
$inv2 : not\_marqued \in ND \to \mathbb{P}(ND)$
$inv3 : \forall x \cdot x \in ND \Rightarrow marqued(x) \subseteq g[\{x\}]$
$inv4 : \forall x \cdot x \in ND \Rightarrow not\_marqued(x) \subseteq g[\{x\}]$
$inv5 : \forall x, y \cdot y \in marqued(x) \Rightarrow e\_label(x \mapsto y) = Marque$
$inv6 : \forall x, y \cdot y \in not\_marqued(x) \Rightarrow e\_label(x \mapsto y) = N\_Marque$
$inv7 : \forall x \cdot x \in ND \Rightarrow marqued(x) \cap not\_marqued(x) = \varnothing$
$inv8 : \forall x, y \cdot y \in g[\{x\}] \land y \notin marqued(x) \Rightarrow \neg e\_label(x \mapsto y) = Marque$
$inv9 : \forall x, y \cdot y \in g[\{x\}] \land y \notin not\_marqued(x) \Rightarrow \neg e\_label(x \mapsto y) = N\_Marque$

Les invariants (inv1) et (inv3) [resp. (inv2) et (inv4)] définissent la variable *marqued* [resp. *not_marqued*] en associant à chaque nœud du graphe un sous-ensemble de ses voisins. Les invariants (inv5) et (inv8) [resp. (inv6) et (inv9)] précisent que les arêtes qui relient un nœud donné à ses voisins inclus dans *marqued* [resp. *not_marqued*], sont toutes étiquetées $Marque$ [resp. N_Marque]. L'invariant (inv7) montre que, pour un nœud donné, aucun voisin ne peut figurer à la fois dans *marqued* et *not_marqued*.

Une conséquence directe de l'ajout de ces deux variables est le raffinement de tous les événements de la machine précédente. La garde de l'événement $DetectTerminaison$ est renforcée par une nouvelle condition (grd3) qui vérifie la non existence d'une arête issue de x et étiquetée $Init$. Cela garantie que le nœud x ainsi que tous ses voisins sont inclus dans l'arbre. De ce fait, une fois le nœud x ainsi que tous ses voisins sont étiqueté A alors x est en mesure de détecter la terminaison locale de l'algorithme de calcul d'arbre recouvrant. Formellement l'évènement $DetectTerminaison$ est présenté comme suit :

EVENT DetectTerminaison
ANY x
WHERE
$grd1 : end\_algo1(x) = FALSE$
$grd2 : n\_label(x) = A$
$grd3 : marqued(x) \cup not\_marqued(x) = g[\{x\}]$
THEN
$act1 : end\_algo1(x) := TRUE$
END

5.2.8 4èmeraffinement : STreeMachine5

Ce dernier raffinement est une mise en œuvre de l'algorithme de calcul de l'arbre recouvrant avec le calcul de degré. Il va permettre de calculer les degrés des nœuds dans l'arbre. Également, il va rendre les pas de calculs totalement localisés. Concrètement, nous ajoutons, pour chaque nœud du graphe, deux compteurs locaux : *card_marqued* et *card_not_marqued*. Pour un nœud donné, le premier [resp. deuxième] compteur correspond à son degré dans l'arbre [resp. aux nombres d'arêtes qui lui sont incidentes et qui ne sont pas dans l'arbre]. Formellement, ces deux compteurs, définis par deux variables internes, correspondent respectivement aux cardinalités des ensembles *marqued* et *not_marqued*. En plus de ces deux variables, nous définissons une autre variable appelée *degree*. Cette dernière est une variable partagée et égale à *card_marqued*. Les invariants, présentés ci-dessous, donnent une description formelle des nouvelles variables :

$inv1 : card\_marqued \in ND \rightarrow \mathbb{N}$
$inv2 : card\_not\_marqued \in ND \rightarrow \mathbb{N}$
$inv3 : degree \in ND \rightarrow \mathbb{N}$
$inv4 : \forall x \cdot x \in ND \Rightarrow card\_marqued(x) = card(marqued(x))$
$inv5 : \forall x \cdot x \in ND \Rightarrow card\_not\_marqued(x) = card(not\_marqued(x))$
$inv6 : \forall x \cdot x \in ND \Rightarrow degree(x) = card(marqued(x))$

Dans ce raffinement, nous sommes en mesure de prouver que si dans une boule aucune arête n'est marquée $Init$, alors la somme des deux compteurs du nœud centre est égale à son cardinalité dans le graphe (voir $Thm1$). Cela, nous ramène à démontrer qu'un nœud peut détecter la terminaison locale de l'algorithme de calcul d'arbre recouvrant quand la somme de ses compteurs est égale à son degré dans le graphe (voir $Thm2$).

$Thm1 : \forall x \cdot e\_label[g \rhd \{x\}] \cap \{Init\} = \varnothing \Rightarrow$
$\quad card\_not\_marqued(x) + card\_marqued(x) = card(g[\{x\}])$
$Thm2 : \forall x \cdot card\_not\_marqued(x) + card\_marqued(x) = card(g[\{x\}]) \Rightarrow$
$\quad e\_label[g \rhd \{x\}] \cap \{Init\} = \varnothing$

Toutefois, aucun nouvel événement n'a été ajouté dans ce raffinement. Tous les événements de l'ancienne machine sont raffinés. En occurrence, la garde de l'événement $DetectTerminaison$ est renforcée (ajout de $grd3$) pour que la détection de terminaison soit totalement localisée.

EVENT DetectTerminaison
ANY x
WHERE
$grd1 : end\_algo1(x) = FALSE$
$grd2 : n\_label(x) = A$
$grd3 : card\_not\_marqued(x) + card\_marqued(x) = card(g[\{x\}])$
THEN
$act1 : end\_algo1(x) := TRUE$
END

5.2.9 1^{er}raffinement :DijkstraMachine2

Dans ce raffinement, nous introduisons un nouvel événement, $progress\_algo2$ pour élaguer les feuilles de new_tree. Les feuilles concernées par l'événement sont celles qui ont détecté la terminaison locale du premier algorithme. Les nœuds élagués sont ajoutés à l'ensemble $pruned$. Lorsque tous les nœuds du graphe sont élagués, sauf un, l'exécution de l'algorithme s'achève et le nœud restant détecte la terminaison globale de l'algorithme de calcul d'arbre recouvrant. Les invariants définis dans ce raffinement sont comme suit :

$inv1 : pruned \subset ND$
$inv2 : pruned \cap Solution\_algo2 = \varnothing$
$inv3 : Solution\_algo2 \neq \varnothing \Rightarrow$
$\quad \exists a \cdot a \in ND \wedge ND \setminus pruned = \{a\} \wedge Solution\_algo2 = \{a\}$
$inv4 : pruned \subseteq dom(end\_algo1 \subseteq \{TRUE\})$

L'invariant ($inv1$) montre que l'ensemble *pruned* est strictement inclu dans ND. L'invariant ($inv2$) indique que l'ensemble *pruned* et $Solution\_algo2$ sont disjoints. A l'aide de l'invariant ($inv3$), nous précisons que si l'algorithme Dijkstra-Scholten termine le calcul, alors il existe un seul nœud non élagué. Ce dernier est égal au résultat de l'algorithme qui est le nœud racine ratifié. Le dernier invariant ($inv4$) vérifie que les nœuds élagués sont tous inclus dans l'arbre. Formellement, l'événement $progress\_algo2$ est spécifié comme suit :

EVENT progress_algo2
ANY x, y
WHERE
$grd1 : end\_algo1(x) = TRUE$
$grd2 : x \mapsto y \in g$
$grd3 : x \notin pruned \wedge y \notin pruned$
THEN
$act1 : pruned := pruned \cup \{x\}$
END

5.2.10 2èmeraffinement : DijkstraMachine3

La spécification de l'algorithme de Dijkstra-Scholten se concrétise davantage. En effet, ce raffinement remplacera l'ensemble *pruned* par un autre qui localise la connaissance des nœuds. Ainsi le nouvel ensemble, *label_pruned*, est égal à 1 si le nœud est élagué et 0 sinon. Formellement, cet nouvel ensemble est défini par une fonction totale qui associe à chaque nœud du graphe soit 1 soit 0, et ce en fonction de l'état du nœud (élagué ou non élagué) (inv1). L'invariant (inv2) est un invariant de collage, il retrace la relation entre les variables *pruned* et *label_pruned*. Désormais, les nœuds élagués sont ceux qui ont *label_pruned* égal à 1. L'invariant (inv3) indique que si le deuxième algorithme termine son exécution, alors seul le nœud racine est non élagué.

$inv1 : label\_pruned \in ND \rightarrow \{0, 1\}$
$inv2 : pruned = dom(label\_pruned \rhd \{1\})$
$inv3 : Solution\_algo2 \neq \varnothing \Rightarrow label\_pruned[Solution\_algo2] = \{0\}$

Tous les événements de la machine précédente sont raffinés. Dans ce qui suit, nous donnons la spécification de l'événement $progress\_algo2$,

EVENT progress_algo2
ANY x, y
WHERE
$grd1 : x \mapsto y \in ND$
$grd2 : label\_pruned(x) = 0 \wedge label\_pruned(y) = 0$
$grd3 : end\_algo1(x) = TRUE$
THEN
$act1 : label\_pruned(x) := 1$
END

Dans cette dernière machine, nous remarquons qu'une exécution locale de l'algorithme Dijkstra-Scholten peut commencer même avant que le premier algorithme termine l'exécution (l'arbre recouvre complètement le graphe). Plus précisément, l'introduction de l'événement *progress_algo*2 assurera une exécution entrelacée des deux algorithmes. Toutefois, la technique de raffinement préserve la correction de la machine concrète vis-à-vis de la machine abstraite. Autrement dit, le raffinement assure que les exécutions entrelacées des deux algorithmes sont équivalentes à leur exécution séquentielle (la machine GenaralMachine3 à titre d'exemple).

5.2.11 3èmeraffinement : DijkstraMachine4

L'objectif de ce raffinement est de concrétiser encore une fois *progress_algo*2. Désormais, un nœud ne peut être élagué que s'il est une feuille dans l'arbre. Ainsi, dans un pas de calcul, si le degré d'un nœud est égal à 1 ($degree(x) = 1$), alors il sera élagué ($pruned(x) := 1$) et le degré de son voisin sera systématiquement décrémenté.

Formellement, la variable *degree* doit être accessible en lecture et écriture depuis la machine DijkstraMachine4. Néanmoins, *degree* est une variable partagée et ne pourra être écrite que par un seul algorithme, en occurrence ici l'algorithme de calcul d'arbre recouvrant. Pour cette raison, nous proposons de copier la valeur de cette dernière variable dans une autre interne à la machine DijkstraMachine4. Cela, va nous permettre de manipuler le degré des nœuds et par la suite de raffiner l'évènement *progress_algo*2. A cet effet, nous spécifions un nouvel événement nommé *copy* pour copier la valeur de *degree* dans une variable interne appelée *p_degree*. Initialement, *p_degree* de tout nœud du graphe est égal à 0. Le copiage ne peut avoir lieu que si le nœud a détecté la terminaison locale du premier algorithme. Afin d'éviter un copiage multiple d'un degré, nous introduisons une variable booléenne qui encode l'état d'un nœud par rapport à la mise en œuvre du copiage : $TRUE$ si le nœud a copié la valeur du degré et $FALSE$ sinon. Nous présentons ci-dessous, les spécifications des invariants, et des événements *Copy* et *progress_algo*2.

$inv1 : p\_degree \in ND \rightarrow \mathbb{Z}$
$inv2 : \forall x \cdot x \in ND \Rightarrow p\_degree(x) \leq degree(x)$
$inv3 : take\_copie \in ND \rightarrow BOOL$
$inv4 : \forall x \cdot x \in ND \wedge take\_copie(x) = TRUE \Rightarrow end\_algo1(x) = TRUE$
$inv5 : \forall x \cdot x \in ND \wedge p\_degree(x) > 0 \Rightarrow take\_copie(x) = TRUE$
$inv6 : \forall x \cdot x \in ND \wedge end\_algo1(x) = FALSE \Rightarrow take\_copie(x) = FALSE$
$inv7 : \forall x \cdot x \in ND \wedge take\_copie(x) = FALSE \Rightarrow p\_degree(x) \leq 0$
$inv8 : \forall x \cdot x \in ND \wedge label\_pruned(x) = 1 \Rightarrow take\_copie(x) = TRUE$
$inv9 : \forall x \cdot x \in ND \wedge \{x\} = Solution\_algo2$
$\Rightarrow take\_copie(x) = TRUE \wedge p\_degree(x) = 0$

Contrairement à *degree*, pour un nœud quelconque x, la valeur de $p\_degree(x)$ peut être négative (inv1). Car, tant que la copie du *degré* n'est pas encore faite, $p\_degree(x)$ peut être toujours décrémentée (inv7). Par ailleurs, l'invariant (inv2) montre que, dans aucun cas, $p\_degree(x)$ ne soit supérieur à *degree*. Nous appelons cette invariant, un "*invariant de coopération*" car en effet il permet de définir la relation entre les variables partagées et les variables internes. Les invariants (inv4) et (inv6) vérifient qu'un nœud x ne peut prendre une copie qu'après avoir détecté la terminaison locale du premier algorithme. Ainsi, si $p\_degree(x)$ est strictement supérieure à 0, alors cela signifie que x a détecté la terminaison locale (inv5). Un nœud élagué est un nœud qui auparavant a pris une copie (inv8). Une fois l'algorithme Dijkstra-Scholten est terminé, alors le nœud racine qui a détecté la terminaison globale a forcément pris une copie de degré et il est ainsi le seul nœud non élagué (inv9).

EVENT Copy
ANY x
WHERE
$grd1 : end\_algo1(x) = TRUE$
$grd2 : take\_copy(x) = FALSE$
THEN
$act1 : p\_degree(x) := p\_degree(x) + degree(x)$
$act2 : take\_copy(x) := TRUE$
END

EVENT progress_algo2
ANY x, y
WHERE
$grd1 : x \mapsto y \in g$
$grd2 : label\_pruned(x) = 0 \wedge label\_pruned(y) = 0$
$grd3 : p\_degree(x) = 1$
THEN
$act1 : label\_pruned(x) := 1$
$act1 : p\_degree(y) := p\_degree(y) - 1$
END

Dans ce dernier raffinement, nous remarquons que les machines STreeMachine5 et DijkstraMachine4 correspondent respectivement aux algorithmes A et B. D'autant plus, l'évènement *Copy* associé à la machine DijkstraMachine4 remplit les conditions définies par le mécanisme de copiage. Également, seules les variables internes des deux machines qui peuvent être modifiées. Ainsi, nous attestons que l'union des deux dernières machines concrètes DijkstraMachine4 et STreeMachine5 respecte bien la définition Def.IV.4.

A la fin de cette section, nous rappelons que le développement formel, que nous avons présenté, génère un ensemble d'obligations de preuve qui ont été toutes prouvées avec la plateforme Rodin. La preuve été soit interactive, soit automatique soit récupérée depuis des anciens développements.

6 Conclusion

Nous avons présenté dans ce chapitre, une approche générale pour prouver les algorithmes distribués complexes en combinant plusieurs composants déjà vérifiés. Concrètement, l'approche vise à décomposer un algorithme distribué en sous-algorithmes et à développer séparément chacun d'eux. A cet effet, nous avons d'abord décrit les conditions suffisantes pour réussir la décomposition des algorithmes distribués. Ensuite, nous avons présenté la base théorique de l'approche et enfin nous avons donné un aperçu sur leur application par un exemple concret.

Le principal avantage de notre travail est de réduire la complexité du développement formel et d'améliorer la lisibilité de la spécification. Nous travaillons actuellement sur l'élaboration des exemples portant sur d'autres classes de calculs locaux. Aussi, nous comptons automatiser davantage la réutilisation des preuves. Car jusqu'à lors, la réutilisation est limitée à une opération de copier-coller d'une spécification vers une autre. La solution que nous envisageons pour un tel problème est le multi-raffinement de plusieurs machines qui devra éviter la duplication des preuves et permettre une meilleure automatisation de notre approche.

Chapitre V

Vérification d'algorithmes de synchronisation : l'exemple de handshake

1 Introduction

Une preuve de correction d'un algorithme distribué codé par les calculs locaux est assurée quand le mécanisme de synchronisation sous-jacent est lui aussi vérifié et prouvé. Toutefois, la spécification et la preuve formelle des algorithmes de synchronisation demeurent des tâches très complexes. Ceci est principalement dû à l'intégration de l'aspect probabiliste dans le calcul distribué engendré par ces algorithmes.

Par ailleurs, la preuve des algorithmes de synchronisation n'a pas été abordée par le patron de développement formel, présenté dans le chapitre III. Alors, nous étudions dans ce chapitre la preuve formelle d'algorithmes de synchronisation selon l'approche "correct-par-construction". Dans un premier volet de ce chapitre, nous proposons, avec la méthode *B événementiel* et un *model-checker* quantitatif, une approche de spécification formelle qui permet de développer graduellement un algorithme de synchronisation. Cette approche supporte un développement depuis un niveau abstrait jusqu'à un niveau concret probabiliste. Le choix probabiliste des nœuds sera vérifié avec le *model-checker* Prism [1]. Nous illustrons cette approche par deux algorithmes de synchronisation à savoir le handshake et le LC1. Dans ce chapitre, nous allons se limité à la présentation de l'algorithme handshake, tandis que l'algorithme LC1 sera présenté dans l'annexe 2. Notons que l'algorithme de handshake développé ici est un nouvel algorithme que nous avons proposé [74]. Il se différencie aux autres algorithmes de type Handshake par son aspect asynchrone dans le calcul distribué ainsi que par sa capacité de tolérer les pannes.

Dans un deuxième volet de ce chapitre, nous présentons une étude analytique et expérimentale de l'algorithme de synchronisation que nous avons proposé. La principale mesure à laquelle nous nous somme intéressée, dans l'étude analytique, est le nombre de

1. http ://www.prismmodelchecker.org

messages échangés. A travers cette mesure, nous allons essayer de mettre en valeur la propriété de tolérance aux pannes de notre algorithme. L'étude expérimentale va permettre de comparer notre algorithme avec celui de Y. Métvier et al. [75]. Nous utilisons à cet effet Visidia, la plateforme d'implémentation, de visualisation et de preuve des algorithmes distribués codés par les calculs locaux. Nous étudions trois différents aspects : la complexité en communication, l'efficacité de l'algorithme et la tolérance aux pannes.

Ce chapitre va être organisé de la façon suivante : D'abord, nous présentons l'approche de développement formelle des algorithmes de synchronisation. Ensuite, nous illustrons cette approche par l'algorithme de synchronisation handshake que nous avons proposé. Enfin, nous détaillons les différentes études analytique et expérimentale réalisées en vue de mettre en valeur les différentes caractéristiques de notre nouvel algorithme. Nous désignons l'algorithme que nous avons proposé par ATH (Asynchrone Tolerant Handshake) et l'algorithme de Y. Métvier et al. [75] par SNH (Synchrone Non-tolerant Handshake).

2 Approche de développement formelle d'algorithmes de synchronisation

Le développement formel d'un algorithme de synchronisation adopte la stratégie de raffinement suivante :

- La première machine SpecMachine calcule la solution de l'algorithme de synchronisation dans une seule étape. En occurrence, les deux solutions possibles pour un tel algorithme sont la synchronisation de deux nœuds adjacents ou d'une boule de rayon 1.
- La deuxième machine ChoiceMachine raffine la machine SpecMachine. Elle introduit plus de détails, désormais il est possible d'exprimer le choix des nœuds qui est considéré comme une étape préliminaire pour réaliser des synchronisations.
- La troisième machine MessageMachine introduit l'échange de messages entre les nœuds. Avec cette machine il est possible d'observer localement le calcul réalisé par chaque nœud du graphe. Autrement dit, nous allons pouvoir observer comment les nœuds communiquent ensemble pour réaliser des handshakes.
- La quatrième machine (FaultyMachine) raffine la machine précédente. Dans ce niveau, nous changeons le contexte d'exécution de l'algorithme. Nous définissons un nouveau contexte qui permet d'inclure des nœuds défectueux dans le graphe. Nous ajoutons ce modèle pour vérifier et prouver certaines propriétés liées à la tolérance aux pannes de l'algorithme ATH. Cette machine est optionnelle.
- ModelChecking : Nous avons choisi de vérifier le dernier niveau probabiliste avec un *model-checker* quantitatif. Autrement dit, nous allons implémenter et tester le modèle issu de la machine MessageMachine avec un outil de *model-checker* probabiliste. L'outil que nous avons choisi est appelé PRISM [2]. Cet outil permet l'analyse

2. http ://www.prismmodelchecker.org

des systèmes stochastiques et probabilistes. Il supporte trois types de modèle probabiliste : les chaînes de markov à temps discret (DTMC), les chaînes de markov à temps continu (CTMC) et les processus de décision markoviens (MDP). Ce dernier type va être utilisé pour implémenter notre modèle.

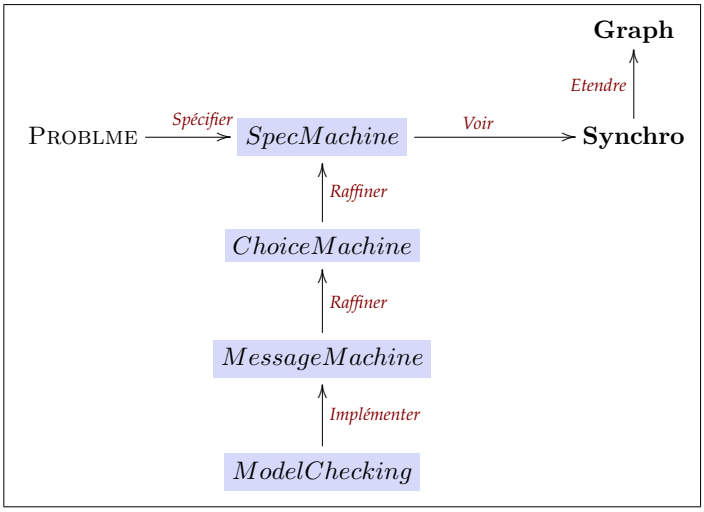


Figure V.1 – Processus de modélisation

3 Preuve de correction de l'algorithme ATH

3.1 Présentation de l'algorithme

Pour illustrer notre approche, nous avons choisi de l'appliquer à notre nouvel algorithme de type handshake ATH. Dans cette section, nous présentons une description informelle de notre algorithme de synchronisation de type handshake (ATH). Nous supposons que chaque nœud v répète infiniment la suite d'instruction suivante : v choisit, d'une manière aléatoire et uniforme, un de ses voisins (nous appelons le nœud choisi $c(v)$). v envoie un message contenant 1 à $c(v)$. v reçoit les messages de ses voisins qui l'ont choisi. À noter qu'il peut y avoir des cas où le nœud v ne reçoit aucun message contenant 1, car il n'a été pas choisi par ses voisins. Une fois le nœud v a envoyé un message contenant 1, trois scénarios sont possibles :

1. Il y a un handshake entre v et $c(v)$ si, et seulement si, v reçoit un message contenant "1" de $c(v)$.
2. Si $c(v)$ a choisi un autre nœud différent de v ; alors, $c(v)$ doit envoyer à v un message contenant 0. Ce message est très important, car il permet à v de se rendre compte qu'il n'a pas été choisi par $c(v)$ et qu'il peut recommencer l'exécution de l'algorithme dès le début.

3. Si le nœud v n'a pas reçu une réponse à sa demande de synchronisation de $c(v)$, alors il devient passif. Nous appelons un nœud passif, un nœud qui reste en attente de la réception d'une réponse à sa demande de synchronisation. Il doit constamment répondre par 0 à toutes les demandes de synchronisation qui viennent des nœuds différents de $c(v)$. Autrement dit, un nœud passif ne peut ni faire un choix, ni envoyer un message contenant 1, mais il peut néanmoins envoyer des messages contenant 0.

Ci-dessous, nous décrivons l'algorithme de synchronisation avec plus de détails.

Algorithm 4 Algorithme de synchronisation de type *Handshake*

1 Chaque nœud v répète infiniment les actions suivantes ;
2 Le nœud v choisit un voisin $c(v)$ au hasard ;
3 Le nœud v envoie 1 à $c(v)$;
4 **if** Le noeud v reçoit 1 de $c(v)$ **then**
5 Réaliser une étape de calcul ;
6 Retour à l'étape n°2 ;
7 **else**
8 **if** Le noeud v reçoit 0 de $c(v)$ **then**
9 Retour à l'étape n°2 ;
10 **else**
11 **if** Le noeud v reçoit 1 d'un voisin $w \neq c(v)$ **then**
12 Le noeud v envoie 0 à w ;
13 Retour à l'étape n°4 ;

3.2 Développement formel

Dans cette section, nous spécifions l'algorithme de synchronisation que nous avons proposé. Notre spécification s'étale sur trois machines différentes : une machine abstraite et une suite de deux raffinements successifs. Un quatrième raffinement à été ajouté pour vérifié quelques propriétés liées à la tolérance aux pannes de notre algorithme. Une fois la correction de la dernière a été vérifiée, nous l'implémentons avec le model-checker *Prism* pour introduire le choix probabiliste des nœuds. En plus des machines, nous avons défini deux contextes : le premier spécifie un graphe (voir la section 4.1 du chapitre II) et le deuxième calcule l'ensemble des "*matching*" qui correspond à l'ensemble de toutes les combinaisons de synchronisation handshake qui peuvent avoir lieu dans le graphe.

3.2.1 Le contexte *synchro*

Le contexte *synchro* est une extension du contexte *graphe*. Il définit toutes les solutions possibles qui peuvent résulter de l'exécution de notre algorithme. Formellement, nous implémentons ces solutions par une constante appelée *all_synchro*. Cette constante

regroupe tous les couplages éventuels sur le graphe. Un couplage sur un graphe $G = (V, E)$ est un sous-ensemble M de E tel que pour toute paire d'arêtes e et e' de M, $e \cap e' = \varnothing$.

Par le biais de $axm2$, nous précisons qu'un couplage est un sous-ensemble d'arêtes. Ensuite, nous vérifions que les arêtes au sein d'un couplage sont disjointes ($axm5$) et non-orientées ($axm3$). Nous ajoutons, l'ensemble vide à *all_synchro* afin d'inclure le cas où les nœuds échouent de produire un handshake après une exécution de l'algorithme ($axm4$). Enfin, nous affirmons que toute solution satisfaisant les axiomes ci-dessus détaillés, doit nécessairement appartenir à *all_synchro* ($axm6$).

$axm1 : all\_synchro \subseteq ND \leftrightarrow ND$
$axm2 : \forall R \cdot R \in all\_synchro \land R \neq \varnothing \Rightarrow R \subseteq g$
$axm3 : \forall R \cdot R \in all\_synchro \Rightarrow R = R^{-1}$
$axm4 : \varnothing \in all\_synchro$
$axm5 : \forall R \cdot R \in all\_synchro \land R \neq \varnothing \Rightarrow (\forall x, y \cdot x \mapsto y \in R \Rightarrow \{x\} \lhd R = \{x \mapsto y\})$
$axm6 : \forall R \cdot R \subseteq g \land R = R^{-1} \land (\forall x, y \cdot x \mapsto y \in R \Rightarrow \{x\} \lhd R = \{x \mapsto y\})$
$\Rightarrow R \in all\_synchro$

3.2.2 La machine `SpecMachine`

La première machine est placée dans un haut niveau d'abstraction ; elle permet simplement d'exprimer l'objectif de notre algorithme : accomplir des handshakes entre les nœuds du graphe. Afin d'atteindre cet objectif, nous avons défini deux nouvelles variables : *actual_state* et *result*. La première variable spécifie le couplage actuel du graphe. Elle est définie comme un ensemble qui regroupe tous les handshakes présents dans le graphe. Formellement, elle est un élément de *all_synchro* ($inv1$). La deuxième variable est introduite pour recevoir le résultat de l'exécution de l'algorithme ($inv2$). Initialement, *result* est vide. Ainsi, nous considérons que l'algorithme termine l'exécution avec succès, si et seulement si, une valeur a été attribuée à la variable *result*.

$inv1 : actual\_state \in all\_synchro$
$inv2 : result \in ND \leftrightarrow ND$

En plus de l'initialisation, cette machine comprend deux autres événements. Le premier événement, appelé *synchronise*, modélise la synchronisation de deux nœuds voisins ($grd1$). Cet événement dévoile le résultat sans donner de détails sur la façon dont il a été calculé. Des explications vont être données plus tard dans les raffinements suivants. Les gardes $grd1$ et $grd2$ de *synchronise* vérifient que les deux nœuds voisins x et y ne sont pas déjà synchronisés. Dna ce cas, alors l'événement déclare l'arête $x \mapsto y$ comme le résultat de l'algorithme et met à jour *actuel_state*. Le deuxième événement que nous pouvons observer à ce niveau est appelé *free_nodes*. Il permet de finir un handshake entre deux nœuds voisins synchronisés. Cet événement met à jour *actual_state*, ce qui génère un nouveau couplage du graphe.

EVENT synchronise
ANY x, y
WHERE
$grd1 : x \mapsto y \in g \wedge result = \varnothing$
$grd2 : x \notin dom(actual\_state)$
$grd3 : y \notin dom(actual\_state)$
$Theorem1 : (actual\_state \cup \{x \mapsto y, y \mapsto x\}) \in all\_synchro$
THEN
$act1 : result := \{x \mapsto y\}$
$act2 : actual\_state := actual\_state \cup \{x \mapsto y, y \mapsto x\}$
END

EVENT free_nodes
ANY x, y
WHERE
$grd1 : x \mapsto y \in actual\_state$
$grd2 : result = \varnothing$
$Theorem2 : (actual\_state \setminus \{x \mapsto y, y \mapsto x\}) \in all\_synchro$
THEN
$act1 : actual\_state := actual\_state \setminus \{x \mapsto y, y \mapsto x\}$
END

Nous prouvons par l'intermédiaire du *Theorem1* que si nous joignons à *actual_state* une nouvelle arête dont les extrémités ne sont pas synchronisées, alors l'ensemble *actual_state* reste toujours correct et représente désormais un nouvel couplage. De la même manière, nous prouvons par *Theorem2* que si nous enlevons un handshake de *actual_state*, nous conservons toujours les propriétés d'exactitude de celui-ci.
Nous montrons par le biais du théorème *deadlock-free* que le blocage ne peut pas avoir lieu lors de l'exécution de l'algorithme.

$$deadlock - free : result = \varnothing \Rightarrow \left(\begin{array}{l} (\exists x, y \cdot x \mapsto y \in g \wedge x \notin dom(actual\_state) \\ \wedge y \notin dom(actual\_state)) \\ \vee (\exists a, b \cdot a \mapsto b \in actual\_state) \end{array} \right)$$

3.2.3 La machine `ChoiceMachine`

Le premier raffinement consiste à préciser qu'une synchronisation est toujours précédée par une opération de choix. Rappelons que le choix d'un nœud est limité à l'ensemble de ses voisins. D'autant plus, ce choix ne peut y avoir que si certaines conditions sont vérifiées (nous les détaillons dans cette machine). Nous spécifions le choix d'un nœud par une variable que nous appelons *choice*. Cette variable est définie par une fonction partielle

qui associe à certain nœud leur voisin choisi. Les invariants, listés ci-dessous, fournissent une définition complète de la nouvelle variable :

(inv1) la variable *choice* est définie par une fonction partielle de ND dans ND ; cela signifie que les nœuds qui n'ont pas fait leurs choix ne figurent pas dans l'ensemble *choice*.

(inv2) un nœud peut choisir un seul nœud parmi ses voisins.

(inv3) tous les nœuds synchronisés ont certainement fait leurs choix auparavant.

(inv4) si deux nœuds voisins sont synchronisés, cela implique que chacun d'eux a choisi l'autre.

$$
\begin{array}{l}
inv1 : choice \in ND \nrightarrow ND \\
inv2 : \forall x \cdot x \in dom(choice) \Rightarrow choice[\{x\}] \subseteq g[\{x\}] \\
inv3 : actual\_state \subseteq choice \\
inv4 : \forall x, y \cdot x \mapsto y \in actual\_state \Rightarrow choice[\{x\}] = \{y\} \wedge choice[\{y\}] = \{x\}
\end{array}
$$

L'initialisation suivante établit les invariants :

$$
\begin{array}{l}
act1 : actual\_state, choice : | \left(\begin{array}{l}
actual\_state' \in all\_synchro \wedge \\
choice' \in ND \nrightarrow ND \wedge \\
(\forall x \cdot x \in dom(choice') \Rightarrow choice'[\{x\}] \subseteq g[\{x\}]) \wedge \\
actual\_state' \subseteq choice'
\end{array} \right) \\
act2 : result := \varnothing
\end{array}
$$

Les événements abstraits du modèle précédent sont aussi présents dans ce modèle ; cependant ils deviennent plus concrets. En fait, la garde de l'événement *synchronise* est renforcée par deux nouvelles conditions qui permettent de vérifier si les nœuds x et y ont déjà fait leurs choix ($x \in dom(choice) \wedge y \in dom(choice)$) et s'il agit d'un choix réciproque ($choice[\{x\}]=\{y\} \wedge choice[\{y\}] = \{x\}$). Pour ce qui concerne l'événement *free_node*, nous ajoutons une nouvelle action qui permet aux nœuds de recommencer un nouveaux choix ($choice := \{x,y\} \lhd choice$). Nous notons que "$\lhd$" est l'opérateur B *événementiel* de soustraction sur le domaine. Soit R et E respectivement une relation et un ensemble ; $E \lhd R$ désigne la soustraction sur le domaine de R à l'ensemble E. C'est l'ensemble des couples ($x \mapsto y$) de R pour lesquels x n'appartient pas à E.

L'introduction de la variable *choice* engendre l'apparition de deux nouveaux événements : *make_choice* et *cannot_synchronise*. Ils raffinent $SKIP$, qui a pour but de modéliser des actions cachées sur les variables qui apparaissent dans ce raffinement. L'événement *make_choice* permet à un nœud de faire un choix. Formellement, cet événement choisit d'une manière non déterministe un nœud y parmi les voisins de x, puis il ajoute l'arête $x \mapsto y$ à l'ensemble *choice*. L'événement *cannot_synchronise* traite le cas où un nœud ne parvient pas à faire une synchronisation, car il n'a pas été choisi par ses voisins.

En conséquence, le nœud est retiré de l'ensemble *choice*. De ce fait, une nouvelle tentative de synchronisation peut démarrer. Formellement, ces deux événements sont donnés comme suit :

EVENT make_choice
ANY x
WHERE
$grd1 : result = \varnothing$
$grd2 : x \notin dom(choice)$
THEN
$act1 : choice :| \exists y \cdot \Big(y \in g[\{x\}] \land choice' = choice \cup \{x \mapsto y\} \Big)$
END

EVENT cannot_synchronise
ANY x
WHERE
$grd1 : x \in dom(choice) \land result = \varnothing$
$grd2 : x \notin dom(actual\_state)$
$grd3 : choice[\{x\}] \cap choice^{-1}[\{x\}] = \varnothing$
THEN
$act1 : choice := \{x\} \lhd choice$
END

3.2.4 La machine `MessageMachine`

La machine `MessageMachine` introduit l'échange des massages entre les nœuds. Bien entendu, cette machine observe de près les communications entre les nœuds. À cette fin, nous introduisons une nouvelle variable, nommée *message*, qui représente l'ensemble de messages non traités et envoyés par les nœuds du graphe. Un message non traité est un message qui a été reçu par le nœud destinataire mais pas encore utilisé dans le calcul de ce dernier. Nous rappelons qu'un message est représenté par 1 ou 0. Dans la suite de ce chapitre nous adoptons l'appellation suivante : messages de type 1 [resp. 0] pour les messages représenté par 1 [resp. 0]. Nous notons que dans notre modèle, les messages envoyés sont tous reçus.

La variable *message* va substituer celle de *choice* qui a été définie dans le modèle précédent. Ce raffinement donnera lieu à un nouvel événement appelé *answer* qui spécifie le cas où un nœud est obligé de répondre à une demande de synchronisation par un message de type 0. Les invariants associés à la nouvelle variable sont spécifiés comme suit :

(inv1) la variable *message* est définie par une fonction partielle de g dans l'ensemble $\{0,1\}$.

(inv2) la variable *choice* est remplacée par le sous-ensemble des messages de type 1. L'invariant $inv2$ est appelé invariant de collage car il relie la variable d'état abstraite avec la variable concrète.

(inv3) nous vérifions avec cet invariant qu'un nœud ne peut pas envoyer un message de type 1 s'il a déjà envoyé un massage de même type qui n'a pas été encore traité.

(inv4) un nœud x est en mesure d'envoyer un message de type 0 à son voisin y si, et seulement si, y a déjà envoyé un message de type 1 à x.

$$
\begin{array}{l}
inv1 : message \in g \pfun \{0,1\} \\
inv2 : dom(message \rhd \{1\}) = choice \\
inv3 : \forall x,y \cdot x \mapsto y \mapsto 1 \in message \Rightarrow dom(message \rhd \{1\})[\{x\}] = \{y\} \\
inv4 : \forall x,y \cdot x \mapsto y \mapsto 0 \in message \Rightarrow y \mapsto x \mapsto 1 \in message
\end{array}
$$

L'événement d'initialisation est raffiné afin d'établir les nouveaux invariants. Ici, la variable *choice* est remplacée par le *witness* : $choice' = dom(message' \rhd \{1\})$.

$$
actual\_state, message :| \left(\begin{array}{l}
actual\_state' \in all\_synchro \land \\
message' \in g \pfun \{0,1\} \land \\
(\forall x,y \cdot x \mapsto y \mapsto 1 \in message' \Rightarrow dom(message' \rhd \{1\})[\{x\}] = \{y\}) \land \\
actual\_state' \subseteq dom(message' \rhd \{1\})
\end{array} \right)
$$

La garde *grd3* de l'événement *synchronise* permet de vérifier si le nœud x a reçu un message de son voisin y et vice versa, tandis que la garde *grd4* précise que le message reçu doit être de type 1. La garde *grd3* de l'événement *make_choice* vérifie que le nœud x n'a pas encore envoyé un message de type 1, ce qui signifie aussi qu'il na pas une demande de synchronisation en cours. Dans ce cas, une tentative de synchronisation peut avoir lieu si, et seulement si, la demande a été traitée (un message de type 1 ou 0 a été reçu par le nœud qui a été sollicité pour réaliser une synchronisation). Nous spécifions avec l'événement *answer* l'opération de réponse à une demande de synchronisation.

EVENT synchronise
ANY x, y
WHERE
$grd1 : x \mapsto y \in g \land result = \varnothing$
$grd2 : x \notin dom(actual\_state) \land y \notin dom(actual\_state)$
$grd3 : x \mapsto y \in dom(message) \land y \mapsto x \in dom(message)$
$grd4 : message(x \mapsto y) = 1 \land message(y \mapsto x) = 1$
THEN
$act1 : result := \{x \mapsto y\}$
$act2 : actual\_state := actual\_state \cup \{x \mapsto y, y \mapsto x\}$
END

EVENT make_choice
ANY x
WHERE
$grd1 : result = \varnothing$
$grd2 : x \notin dom(actual\_state)$
$grd3 : x \notin dom(dom(message \rhd \{1\}))$
$grd4 : \forall z \cdot x \mapsto z \in g \Rightarrow x \mapsto z \notin dom(message)$
THEN
$$act1 : choice :| \exists y \cdot \left(\begin{array}{l} y \in g[\{x\}] \wedge \\ message' = message \cup \{x \mapsto y \mapsto 1\} \end{array} \right)$$
END

L'événement *answer* permet de spécifier la réponse d'un nœud à une demande de synchronisation provenant d'un voisin autre que celui qui a été choisi. Nous supposons que le nœud x a choisi son voisin y pour réaliser un handshake ($grd3$) et ce dernier ne lui a pas encore répondu ($grd4$). Entre temps, si le nœud x reçoit un message de type 1 d'un nœud z différent de y ($grd5$), alors le nœud x doit lui envoyer un message de type 0. Par la garde ($grd6$), nous vérifions que le nœud x envoi une seule réponse à une demande de synchronisation. La spécification formelle de l'événement *answer* est donnée comme suit :

EVENT answer
ANY x, y, z
WHERE
$grd1 : x \notin dom(actual\_state)$
$grd2 : result = \varnothing$
$grd3 : x \mapsto y \in dom(message \rhd \{1\})$
$grd4 : y \mapsto x \notin dom(message)$
$grd5 : y \neq z \wedge z \mapsto x \mapsto 1 \in message$
$grd6 : x \mapsto z \mapsto 0 \notin message$
THEN
$act1 : message := message \cup \{x \mapsto z \mapsto 0\}$
END

3.2.5 La machine `FaultyMachine`

Nous montrons dans ce raffinement comment notre algorithme peut continuer l'exécution dans un environnement disposant d'un ensemble de nœuds défectueux. Pour cela, nous supposons que certains nœuds dans le graphe sont défectueux d'une façon définitive. Ces nœuds ne peuvent ni faire un choix, ni répondre à une demande de synchronisation. Cette nouvelle propriété est introduite dans un nouveau contexte appelé *faulty_context*. Ce nouveau contexte étend le contexte synchro et partage l'ensemble des nœuds *ND* en deux sous-ensembles : les nœuds défectueux *f_ND* et les nœuds non-défectueux *non_f_ND*

($axm1$). L'axiome $axm2$ précise que le graphe n'est pas entièrement composé de nœuds défectueux car dans un tel cas, aucun algorithme ne peut y fonctionner. Par conséquent, l'ensemble des "*matching*" du graphe est redéfini en écartant les nœuds défectueux de toutes les solutions qui peuvent avoir lieu ($axm3$ et $axm4$). Les axiomes du nouveau contexte sont définis comme suit :

$$
\begin{array}{l}
axm1 : partition(ND, f\_ND, non\_f\_ND) \\
axm2 : non\_f\_ND \neq \varnothing \\
axm3 : all\_synchro \subseteq non\_f\_ND \leftrightarrow non\_f\_ND \\
axm4 : \forall R \cdot R \subseteq g \land R = R^{-1} \land \\
(\forall x, y \cdot x \mapsto y \in R \Rightarrow \{x\} \lhd R = \{x \mapsto y\} \land \{x, y\} \subseteq non\_f\_ND) \Rightarrow R \in all\_synchro
\end{array}
$$

Dans la machine `FaultyMachine`, nous introduisons deux nouvelles variables *passif_ND* et *actif_ND* qui représentent respectivement les nœuds passifs et les nœuds actifs. Ces sous-ensembles sont disjoints et forment une partition de *non_f_ND* ($inv1$ et $inv2$). Par le biais de l'invariant $inv3$, nous précisons qu'un nœud défectueux ne peut jamais envoyer un message. De même, nous montrons qu'un nœud passif ne peut pas faire un choix, c-à-d il ne peut pas envoyer un message de type 1 ($inv4$). Enfin, l'invariant $inv5$ précise que l'ensemble des nœuds passifs est inclu dans l'ensemble des voisins de nœuds défectueux.

$$
\begin{array}{l}
inv1 : passif\_ND \cup actif\_ND = non\_f\_ND \\
inv2 : passif\_ND \cap actif\_ND = \varnothing \\
inv3 : f\_ND \cap dom(dom(message)) = \varnothing \\
inv4 : passif\_ND = dom(dom(message \rhd \{1\}) \rhd f\_ND) \\
inv5 : passif\_ND \subseteq g[f\_ND]
\end{array}
$$

Dans ce niveau de raffinement aucun nouvel événement n'a été ajouté, la machine `FaultyMachine` reprend et raffine les événements de la machine précédente.

Toutefois, des nouvelles preuves ont été ajoutées ; parmi lesquelles celles qui permettent de vérifier les propriétés suivantes :

Theorem1 le nombre des nœuds passifs ne peut pas dépasser le nombre de nœuds non défectueux et qui sont des voisins à des nœuds défectueux.

Theorem2 Si un nœud passif envoie un message ; alors ce dernier est nécessairement de type 0, et s'il reçoit des messages alors ils sont de type 1.

$$
\begin{array}{l}
Theorem1 : card(passif\_ND) \leq card(g[f\_ND] \setminus f\_ND) \\
Theorem2 : \forall x \cdot x \in passif\_ND \Rightarrow (\forall y \cdot y \mapsto x \in dom(message) \land x \mapsto y \in dom(message) \\
\quad \Rightarrow message(y \mapsto x) = 1 \land message(x \mapsto y) = 0)
\end{array}
$$

3.2.6 ModelChecking

Dans ce niveau nous implémentons le choix probabiliste des nœuds avec l'outil de model checking probabiliste PRISM. Plus précisément, nous implémentons la machine MessageMachine. Pour vérifier la correction et analyser les performances de notre algorithme, nous l'avons implémenté sur un graphe simple et connexe composé de trois nœuds. Dans ce graphe, les nœuds sont deux à deux connectés. Avec une telle topologie de graphe, la probabilité attribuée à un choix de voisin est égale à 1/2. Le code de l'algorithme comme il a été implémenté avec PRISM est présenté et commenté dans l'annexe 1. Les propriétés que nous avons définies sont les suivantes :

Propriété n°1 (leaders<=1) vérifie si le nombre de nœuds synchronisés dans le graphe est inférieur ou égal à 2. Afin de vérifier cette propriété, nous définissons une variable aléatoire *leaders* qui compte le nombre de nœuds synchronisés. Formellement, nous définissons un $formula$ qui attribue 1 si l'état du nœud est égal à 3, et attribue 0 dans les autres cas.

Propriété n°2 (P>=1 [F "synchro"]) atteste qu'avec une probabilité égale à 1, éventuellement une synchronisation handshake est réalisée. Afin de vérifier cette propriété, nous spécifions une expression logique *synchro* qui est égale à $TRUE$ si une des trois arêtes du graphe est marquée. Formellement, nous définissons un *label* qui renvoie $TRUE$ s'il y a une synchronisation handshake dans le graphe.

4 Analyse de l'algorithme ATH

Dans cette section nous présentons une analyse de l'algorithme ATH. Nous étudions, d'abord, la probabilité d'avoir une synchronisation sur une arrête quelconque dans le graphe. À partir de ce premier résultat, nous interprétons la probabilité d'avoir une synchronisation handshake à partir d'un nœud quelconque du graphe. Ensuite, notre intérêt portera sur la vérification de la possibilité qu'un nœud voisin d'un nœud en panne puisse réaliser des handshakes avant de devenir passif. Pour pouvoir évaluer convenablement cette possibilité, nous calculons le nombre moyen de handshakes que le nœud en question puisse l'avoir. Nous utilisons à cet effet, l'espérance mathématique pour mesurer ce nombre moyen. Enfin, nous nous sommes intéressés à la probabilité de succès de l'algorithme ATH. Cette probabilité est un succès si les nœuds du graphe réussissent à avoir au moins un handshake.

Dans cette section, nous utilisons les notations standards de la théorie des graphes présentées dans le chapitre I. Avant d'entamer la présentation de la suite de cette section, nous rappelons brièvement ces notations standards de la théorie des graphes. Ainsi, nous spécifions un graphe non orienté, simple et connexe (nommé G) par le couple (V, E), où V est l'ensemble des nœuds et E est l'ensemble des arêtes. Nous désignons par n la taille de graphe, définie par le nombre d'éléments de V et par m le nombre d'éléments de E. Soit une arête $e = \{u, v\}$ connectant deux nœuds voisins u et v, et w un nœud ($w \in V$) alors, le degré du nœud w $d(w)$ est défini par le nombre de voisins de w. $N(v)$ désigne

l'ensemble des voisins du nœud v. Nous désignons par $\mathcal{F}$ l'ensemble des nœuds sûrs (qui ne sont pas en pannes). Nous considérons qu'une arête $e = \{e, v\}$ est sûre si, et seulement si, les nœuds u et v sont tous les deux sûrs. Dans le cas inverse, l'arête est dite en panne.

4.1 Probabilité d'avoir un handshake à partir d'un nœud

Lemme V.1. *Dans ce premier lemme, nous supposons que, $n \geq 2$ et $m \geq 1$.*

1. *Nous désignons par $p(e)$ la probabilité d'avoir une synchronisation entre u et v. Alors, $p(e)$ est égale à 0 si au moins un des deux nœuds u et u est en panne et différent de 0 sinon. $p(e)$ est donnée comme suit :*

$$p(e) = \begin{cases} \frac{1}{d(u)d(v)} & \text{if} \quad u \notin \mathcal{F} \text{ and } v \notin \mathcal{F} \\ 0 & \text{Sinon.} \end{cases} \tag{V.1}$$

2. *Nous désignons par $p(w)$ la probabilité d'avoir une synchronisation à partir du nœud w. Cette probabilité est donnée comme suit :*

$$p(w) = \begin{cases} \frac{1}{d(w)} \sum_{x \in N(w) \setminus \mathcal{F}} \frac{1}{d(x)} & \text{if} \quad w \notin \mathcal{F} \\ 0 & \text{Sinon.} \end{cases} \tag{V.2}$$

Démonstration. La preuve du premier lemme est triviale. En effet, chaque nœud v choisit uniformément et aléatoirement un de ses voisins $d(v)$. Ainsi, le nœud v possède la même probabilité $\frac{1}{d(v)}$ d'être choisi. Par conséquent, l'arête e a une probabilité égale à $\frac{1}{d(u)d(v)}$ d'être impliqué dans une synchronisation handshake entre les nœuds v et u. Par définition, il y a un handshake entre u et v si, et seulement si, ils se choisissent mutuellement.
Pour prouver le second lemme, il suffit de signaler que w peut réussir un `handshake` avec un de ses voisins qui ne sont pas en pannes. Par conséquent, nous pouvons déduire la probabilité suivant :

$$\begin{aligned} p(w) &= \sum_{e=\{w,x\} \in E \text{ and } x \notin \mathcal{F}} p(e) \\ &= \sum_{x \in N(w) \setminus \mathcal{F}} \frac{1}{d(w)d(x)}, \end{aligned}$$

Fin de la preuve. □

4.2 Nombre moyen de handshakes

Soit v un nœud sûr et soit $\mathcal{F}(v)$ l'ensemble des nœuds en pannes et voisins à v ($\mathcal{F}(v) = \mathcal{F} \cap N(v)$), le nœud v peut devenir passif s'il a au moins un voisin en panne ($f(v) > 0$) et choisi lors d'un essai de synchronisation. Cet événement se déclenche avec une probabilité égale à $\frac{f(v)}{d(v)}$. Nous désignons par $T_p(v)$ la variable aléatoire (v.a.) qui compte le nombre

de rounds avant que v devienne passif. Ainsi, $T_p(v)$ suit la loi géométrique de paramètre $\frac{f(v)}{d(v)}$ et par conséquent, son espérance est égale à $\mathbb{E}(T_p(v)) = \frac{d(v)}{f(v)}$.

Soit la v.a. $H(v)$ qui permet de compter le nombre de handshakes obtenus par v avant qu'il devienne passif, $H(v)$ est une variable aléatoire de loi binomiale de paramètres t et $p(v)$ conditionnée par $Tp(v) = t$. $\mathbb{P}r(H(v) = k \mid T_p(v) = t)$ est la probabilité que v réussisse k synchronisations conditionnée par $T_p(v) = t$. Cette probabilité est donnée comme suit :

$$\mathbb{P}r(H(v) = k \mid T_p(v) = t) = \binom{t}{k} p(v)^k (1 - p(v))^{t-k},$$

et $\mathbb{E}(H(v) \mid T_p(v) = t) = tp(v)$. Ceci donne :

$$\begin{aligned} \mathbb{E}(H(v)) &= \sum_{t \geq 1} \mathbb{E}(H(v) \mid T_p(v) = t)\, \mathbb{P}r(T_p(v) = t) \\ &= \sum_{t \geq 1} tp(v) \left(1 - \frac{f(v)}{d(v)}\right)^{t-1} \frac{f(v)}{d(v)}. \end{aligned} \tag{V.3}$$

Étant donné que $\forall x$, $\sum_{t \geq 1} t(1-x)^{t-1} = \frac{1}{x^2}$, alors nous pouvons déduire que :

$$\mathbb{E}(H(v)) = \frac{d(v)p(v)}{f(v)}. \tag{V.4}$$

En utilisant (V.2), nous obtenons :

$$\mathbb{E}(H(v)) = \frac{1}{f(v)} \sum_{x \in N(w) \setminus \mathcal{F}} \frac{1}{d(x)}. \tag{V.5}$$

Remarque 4.1. *L'égalité (V.5) confirme qu'un nœud v, ayant un voisin en panne, peut réussir d'avoir des handshakes avant qu'il ne devienne passif. Toutefois, si G est un graphe étoile, et v est le nœud sûr et centre de G, alors (V.5) devient :*

$$\mathbb{E}(H(v)) = \frac{n}{|\mathcal{F}|} - 1. \tag{V.6}$$

Donc, si $|\mathcal{F}|$ est une constante, alors $\mathbb{E}(H(v)) = \Theta(n)$.

4.3 Probabilité de succès

Dans cette section, nous étudions la probabilité d'obtenir au moins un handshake dans le graphe. Pour toute arête e, nous désignons par $\mathcal{HS}(e)$ l'événement : il y a un handshake sur e et par $\overline{\mathcal{HS}(e)}$ l'événement complémentaire.

Proposition 4.1. *Soit $G = (V, E)$ un graphe ayant $n \geq 2$ nœuds et $m_s \geq 1$ arêtes sûres, et soit $s(G)$ la probabilité d'avoir au moins un handshake dans le graphe et soit $f(G) = 1 - s(G)$. La probabilité $s(G)$ est donnée comme suit :*

$$s(G) \geq 1 - \prod_{i=1}^{m} \left(1 - \mathbb{P}r\left(\mathcal{HS}(e_i)\right)\right). \tag{V.7}$$

Démonstration. Il ne peut pas y avoir un handshake dans le graphe G si, et seulement si, pour toute arête $e \in E$, nous avons $\overline{\mathcal{HS}(e)}$.

$$\begin{aligned} f(G) &= \mathbb{P}r\left(\wedge_{i=1}^{m} \overline{\mathcal{HS}}(e_i)\right) \\ &= \prod_{i=1}^{m} \mathbb{P}r\left(\overline{\mathcal{HS}}(e_i) \mid \wedge_{j=1}^{i} \overline{\mathcal{HS}}(e_j)\right) \\ &= \prod_{i=1}^{m} \left(1 - \mathbb{P}r\left(\mathcal{HS}(e_i) \mid \wedge_{j=1}^{i} \overline{\mathcal{HS}}(e_j)\right)\right). \end{aligned} \tag{V.8}$$

D'autre part, nous constatons que pour toutes arêtes e et e' :

$$\mathbb{P}r\left(\mathcal{HS}(e') \mid \overline{\mathcal{HS}(e)}\right) \geq \mathbb{P}r\left(\mathcal{HS}(e')\right)$$

Par conséquent, l'égalité (V.8) implique le suivant :

$$f(G) \leq \prod_{i=1}^{m} \left(1 - \mathbb{P}r\left(\mathcal{HS}(e_i)\right)\right), \tag{V.9}$$

Fin de la preuve. □

5 Expérimentations et résultats

Dans cette section, nous présentons les résultats de l'étude expérimentale que nous avons réalisée. L'object de cette étude est de montrer que notre algorithme de synchronisation est plus performant que celui de [75] (SNH). Les principales mesures auxquelles nous nous somme intéressées dans cette études sont :

La complexité en communication : Le nombre de messages échangés entre les nœuds du graphe. Nous montrons que notre algorithme permet de réduire énormément le nombre de messages échangés entre les nœuds du graphe.

L'efficacité de l'algorithme : Le rapport entre le nombre total de handshakes réussis et le nombre total des tentatives de synchronisations effectués par les nœuds du graphe. Nous montrons que notre algorithme réduit le nombre moyen de tentatives de synchronisations nécessaires pour avoir un certain nombre de handshakes dans le graphe.

La tolérance aux pannes : Un aspect de sécurité qui permet de garantir que l'algorithme assure la non-propagation de la panne et qu'il continue à fonctionner de manière réduite. Nous montrons que algorithme est tolérant aux pannes contrairement à celui de [75].

Nous avons utilisé la plateforme Visidia [24–26] pour implémenter, exécuter, visualiser et simuler les algorithmes distribués. Afin de comparer les deux algorithmes, nous avons choisi de les implémenter pour les deux algorithmes distribués : Calcul d'arbre recouvrant et calcul du degré. L'algorithme de calcul d'arbre recouvrant a été présenté et spécifié dans la section 6.1 du chapitre III. L'algorithme de calcul de degré a pour but d'effectuer le calcul des degrés des nœuds d'un graphe non étiqueté. Rappelons que le degré d'un nœud est défini par le nombre de ses voisins et q'un graphe non étiqueté est un graphe dont les nœuds et les arêtes ont tous un étiquetage uniforme (nous supposons qu'ils sont tous étiquetés 0). A n'importe quelle étape du calcul, un nœud x et son voisin y peuvent incrémenter leurs degrés et marquer l'arête correspondante si, et seulement si, x et y sont synchronisés et l'arête commune n'est pas marquée. Le marquage d'une arête est l'action de changer son label de 0 à 1. Le calcul s'arrête quand toutes les arêtes ont été marquées. La seule règle de ré-étiquetage qui encode l'algorithme de calcul de degré est présentée comme suit :

Figure V.2 – Règle de ré-étiquetage

Les algorithmes distribués (calcul d'arbre recouvrant et calcul de degré) ne permettent pas la détection locale de la terminaison. En d'autres termes, les nœuds poursuivent l'exécution de l'algorithme sans arrêt même si la solution est calculée. Ceci est dû à l'inaptitude des nœuds de savoir l'état global du graphe. Toutefois, cette propriété (la non détection locale de la terminaison globale) rend impossible l'exécution d'une étude expérimentale correcte ; car elle ne permet pas de déterminer les valeurs exactes des mesures quand la solution de l'algorithme est calculée. Pour faire face à cette lacune, nous ajoutons un superviseur global sur le graphe qui a pour rôle d'arrêter l'exécution de tous les nœuds quand la solution est calculée. D'autre part, afin d'effectuer les simulations, nous avons implémenté les algorithmes sur une grille de taille 10*10, et nous avons exécuté chacun d'entre eux 100 fois avec les mêmes paramètres de Visidia et sous la même machine. Pour ces simulations, nous supposons que les communications ne prennent pas de temps et que le temps nécessaire pour effectuer une tentative de synchronisation ou réaliser une étape de calcul est quasiment le même.

5.1 Complexité en communication

Dans cette section, la comparaison expérimentale des algorithmes ATH et SNH va se focaliser sur la complexité en communication. Concrètement, nous mesurons le nombre de

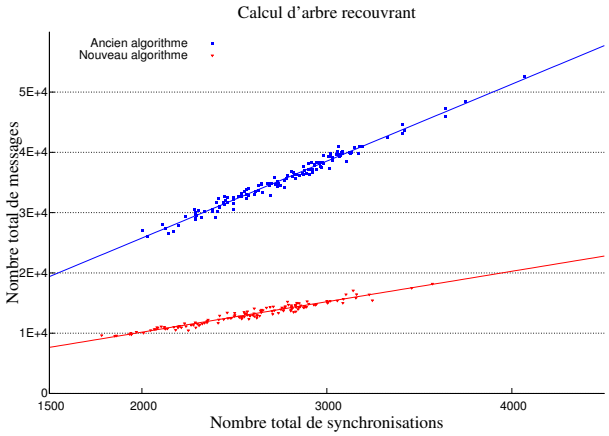


Figure V.3 – Complexité en communication (1)

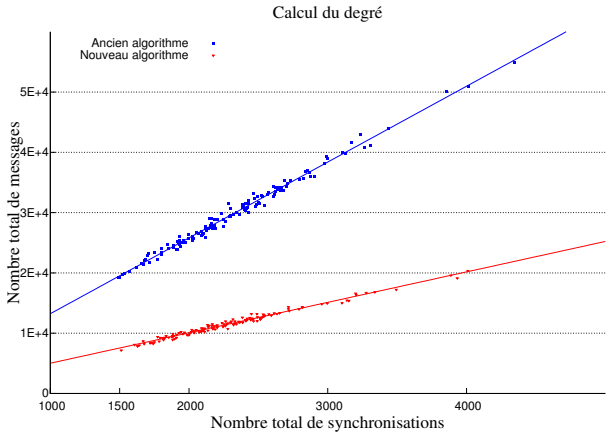


Figure V.4 – Complexité en communication (2)

messages de synchronisations échangés et le nombre de handshakes qui ont eu lieu, et ce pour chaque simulation et quand l'algorithme distribué atteint son objectif. Ces mesures permettent, par la suite, de calculer :

1. Le rapport entre le nombre total de handshakes et le nombre de messages de synchronisation,
2. Le nombre moyen de messages de synchronisation.

Dans la figure Fig.V.3 [resp. Fig.V.4], nous traçons deux allures des courbes de complexité en communication relatives aux deux algorithmes de synchronisation. Cette figure [resp. Fig.V.4] correspond à l'algorithme de calcul d'un arbre recouvrant [resp. calcul de degré]. Chaque point de l'allure exprime le résultat de l'algorithme distribué en terme de rapport entre le nombre total de handshakes et le nombre de messages de synchronisation. A titre d'exemple, quand une simulation calcule un arbre recouvrant avec 2500 handshakes cela nécessite un transfert de 32167 messages de synchronisation avec l'algorithme SNH, et 12696 messages de synchronisation avec l'algorithme ATH. Ainsi, nous pouvons remarquer que notre algorithme réduit considérablement le nombre de messages de synchronisation transmis par les nœuds du graphe. Ceci est dû au fait que la quantité d'informations transférées sur le graphe est extrêmement réduite. Cette amélioration est le résultat de l'élimination de l'approche multi-cast dans l'envoi des messages. Plus précisément, quand un nœud (x) choisit un nœud voisin (y) pour faire une synchronisation ; (x) envoie un seul message à (y) et puis (y) doit le répondre par un message de type "1" ou "0". Au contraire, dans l'approche multi-cast, (x) envoie un "1" à (y) et en plus il envoie "0" à tous les autres.

Les résultats de l'étude expérimentale sont présentés dans le tableau Tab.V.1. Ils sont classés selon deux différentes mesures. La première mesure est le nombre moyen de messages de synchronisation nécessaires au calcul d'un arbre recouvrant (ou des degrés des nœuds). La deuxième mesure est le nombre moyen de messages de synchronisation requis pour avoir un seul handshake (messages/synchro).

	Calcul d'arbre recouvrant		Calcul de degré	
	messages	messages/synchro	messages	messages/synchro
Algorithme SNH	36 012	12.78	30 520	12.86
Algorithme ATH	12 365	5.08	11 671	5.05

Tableau V.1 – Complexité en communication

Toutefois, le nombre de messages de synchronisation échangés dépend essentiellement de la topologie du réseau. En fait, la complexité en communication est égale à $2m$ pour l'algorithme SNH et $2n$ pour notre algorithme ATH. Il convient de noter que la complexité en communication est égale au nombre total de messages de synchronisation échangés. Néanmoins, ces mesures sont exactes si, et seulement si, le graphe ne présente aucun nœud en pannes. Dans le cas contraire, la complexité en communication de l'ancien algorithme est égale à 2m-$\sum_i^K(d(i))$. Nous désignons par $d(i)$ le degré du nœud i et par K est l'ensemble des nœuds en panne. Cette mesure n'est valide que pour une seule exécution, car comme nous le verrons dans la suite de ce chapitre (Section 5.3), l'algorithme SNH n'est pas tolérant aux pannes. En ce qui concerne l'algorithme ATH, sa complexité est égale à 2n - 2$|K|$.

5.2 Etude de performance

Le but de cette section est de comparer l'efficacité des deux algorithmes de synchronisation. Comme nous l'avons déjà précisé, l'efficacité est exprimée par le rapport entre le nombre de tentatives de synchronisation et le nombre de handshakes qui ont eu lieu. Plus ce rapport est faible plus l'algorithme est efficace. En d'autres termes, nous vérifions si notre algorithme peut avoir des handshakes avec un nombre de tentatives minimal par rapport à celui de SNH. Le rapport d'efficacité est calculé si, et seulement si, tous les nœuds du graphe finissent l'exécution de l'algorithme distribué. Nous illustrons dans les figures présentées ci dessous les résultats des simulations. La figure Fig.V.5 [resp. Fig.V.6] représente les résultats de simulation de l'algorithme de calcul d'arbre recouvrant [resp. calcul de degré] par deux allures. Chaque allure correspond à un algorithme de synchronisation. Nous remarquons que les deux allures des deux algorithmes de synchronisation sont très proches, et que notre algorithme présente un avantage visible par rapport à SNH. A titre d'exemple, quand une simulation calcule un arbre recouvrant avec 2500 handshakes cela nécessite 8944 tentatives de synchronisation avec l'algorithme SNH, et 7613 tentatives avec notre algorithme.

Nous estimons que cet avantage est dû à l'aspect asynchrone de notre algorithme. Afin de développer cette idée, nous présentons deux scénarios d'exécution des deux algorithmes de synchronisation. Nous implémentons les algorithmes sur un graphe qui contient seulement quatre nœuds étiquetés de 0 à 3. Dans la figure Fig.V.7, nous reproduisons un scénario de synchronisation avec l'algorithme SNH. Rappelons à ce niveau que cet algorithme est synchrone. Cette synchronisation est rythmée par des rounds qui sont gérés par l'envoi et la réception des messages de synchronisation. Nous divisons un round en deux

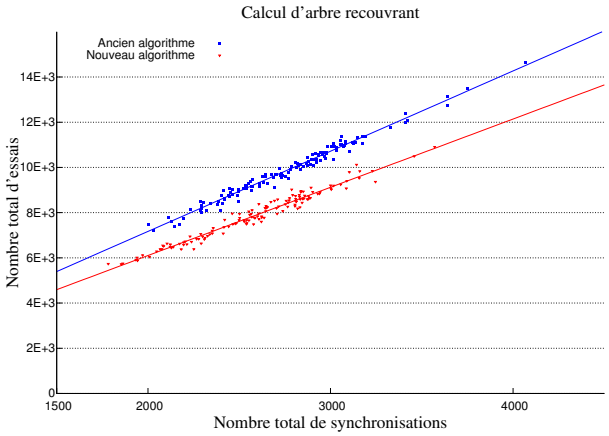


Figure V.5 – Etude de performance (1)

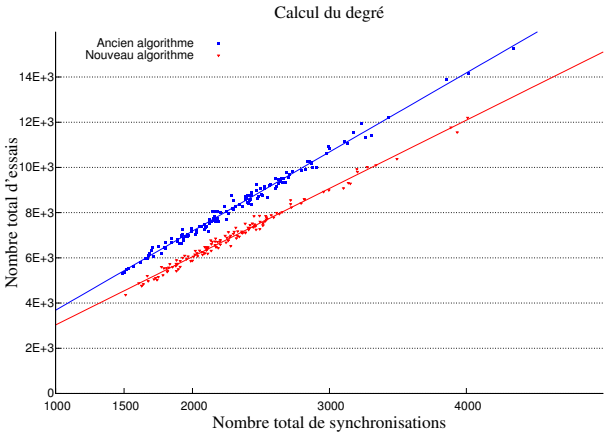


Figure V.6 – Etude de performance (2)

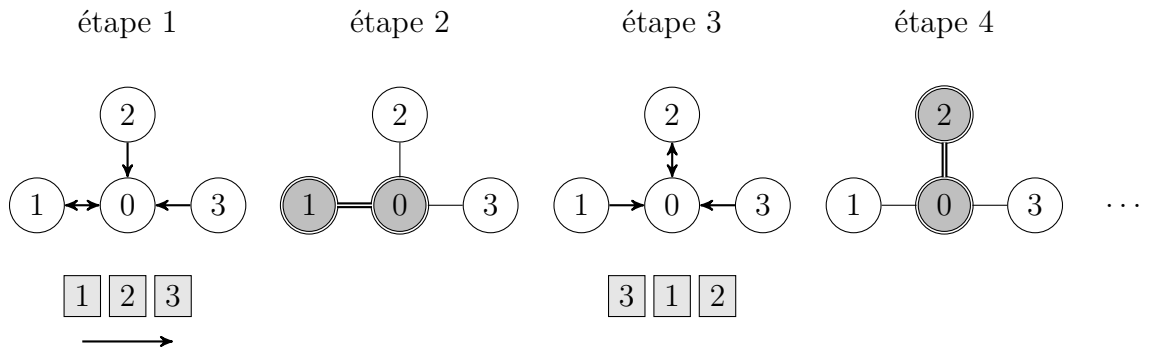


Figure V.7 – Un scénario d'exécution de l'algorithme SNH

étapes ; la première comprend l'envoi et la réception des messages de synchronisation, et la deuxième correspond à l'exécution d'un étape de calcul. Le premier [resp. deuxième] round est composé par l'étape 1 et 2 [resp. 3 et 4]. Cette façon de procéder nous permet de rendre facile l'analyse et la compréhension de l'exemple proposé. Dans la première [resp. troisième] étape, les processus envoient et reçoivent les messages de synchronisation. Dans la deuxième [resp. quatrième] étape, les nœuds 1 et 0 [resp. 0 et 2] réussissent à avoir un handshake. Dans ce scénario, les nœuds réussissent à avoir deux handshakes après huit essais (dans les étapes 1 et 3).

Dans la même figure, nous schématisons par des carreaux l'ordre d'envoi des messages de synchronisation (Bien entendu l'ordre de réception des messages par le nœud 0). Par exemple, pour l'étape 1, le nœud 1 est le premier qui a envoyé un message au nœud 0, ensuite ce sont les nœuds 2 et 3 qui se suivent. Toutefois, l'ordre de l'envoi et de la réception des messages n'a aucune importance pour le nœud 0 car l'algorithme est synchrone. Dans un tel algorithme, chaque nœud doit attendre la réception de tous les messages de ses voisins avant de passer à l'étape de calcul.

Dans la figure Fig.V.8, nous reproduisons un scénario de synchronisation avec notre

algorithme. Nous supposons que dans l'étape 1, les nœuds 1, 2 et 3 envoient consécutivement une demande au nœud 0 et juste après, le nœud 0 envoi une demande au nœud 1. En conséquence, les nœuds 1 et 0 réussissent à avoir un handshake dans la deuxième étape. Cependant, les autres demandes de synchronisation reçues par le nœud 0 sont conservées ; elles restent toujours en attente d'une réponse. Quand les nœuds 1 et 0 finissent leur synchronisation, ils vont refaire des nouvelles tentatives de synchronisation : le nœud 0 choisi le nœud 2 et le nœud 1 choisi le nœud 0 (étape 3). Cela permet de produire un handshake entre le nœud 0 et le nœud 2 dans la quatrième étape.

Ainsi, nous déduisons que dans ce scénario, les nœuds ont réussi à reproduire deux handshakes après seulement six tentatives de synchronisation (4 tentatives à la première étape et 2 tentatives à la troisième étape).

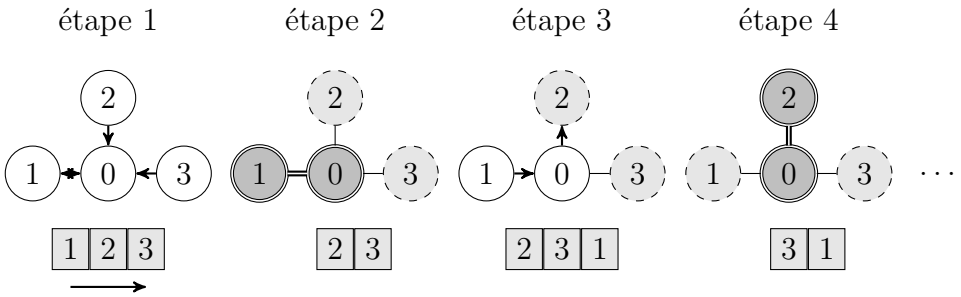


Figure V.8 – Un scénario d'exécution de l'algorithme ATH

5.3 Tolérance aux pannes

La tolérance aux pannes est un critère important de sécurité pour les algorithmes distribués, et en particulier pour les algorithmes de synchronisation. En effet, la tolérance aux pannes tire son importance du fait qu'elle peut garantir une exécution continue même si quelques défaillances sont survenues dans certains nœuds du graphe [76]. En partie, cela est assuré par la "*fault confinement*" qui est considéré comme une technique de tolérance aux pannes [76]. Elle implique une propagation limitée de la panne dans le graphe. Nous désignons par faible fault-containment, la propagation observable limitées à seulement les voisins immédiats d'un nœud en panne avec une grande probabilité [77].

La dernière expérience accomplie dans le cadre de cette étude expérimentale permet de montrer que notre algorithme est tolérant aux pannes. Concrètement, nous examinons l'effet de l'insertion de quelques nœuds défaillants sur les autres nœuds sûrs du graphe. Dans cette expérience, le graphe qui va servir comme un support pour la simulation des algorithmes de synchronisation est représenté dans la figure Fig.V.9.

Nous avons choisi de suivre l'activité de trois différents nœuds dans le graphe (les nœuds colorés) au cours de l'exécution de l'algorithme de handshake. Dans la première expérience, nous avons essayé d'insérer un nœud en panne d'une façon permanente (le nœud noir), et nous avons suivi l'activité des autres nœuds. Un nœud en panne d'une façon

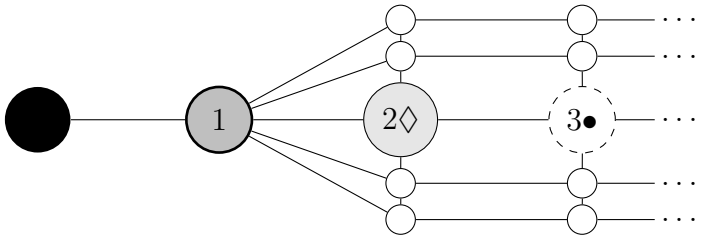


Figure V.9 – Graphe de simulation

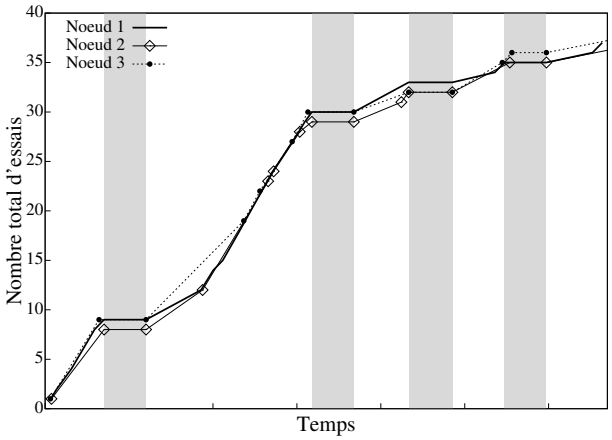


Figure V.10 – Tolérance aux pannes (1)

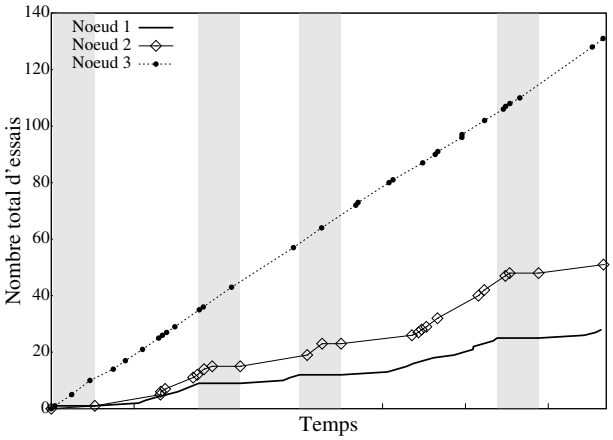


Figure V.11 – Tolérance aux pannes (2)

permanente signifie qu'il est en arrêt d'activité définitivement : il ne peut ni faire un choix, ni répondre aux requêtes des nœuds voisins. Ainsi, l'algorithme SNH est totalement bloqué et notre algorithme continue à fonctionner tout en assurant un faible "*fault-containment*". Cependant, et afin de permettre une bonne comparaison entre les deux algorithmes de synchronisation, nous avons changé, dans la deuxième expérience, le type de la panne pour qu'elle soit non permanente. Le nœud en panne devient très lent quand il est synchronisé. Ainsi, nous allons pouvoir observer l'impact de la défaillance sur l'activité des nœuds avec l'algorithme SNH. Le résultat de cette expérimentation est illustré par la figure Fig.V.10 (l'algorithme SNH) et par la figure Fig.V.11 (l'algorithme ATH).

Chaque figure comprend trois courbes, et chaque courbe retrace l'activité d'un des nœuds choisis. L'activité d'un nœud est interprétée par le nombre de tentatives de synchronisation durant l'exécution. Dans les deux figures, les zones grises correspondent aux périodes où le nœud défaillant est en synchronisation avec le nœud 1. La première figure montre bien qu'un nœud en panne peut paralyser tous les autres nœuds du graphe. Cependant, avec notre algorithme, il est clair que seuls les nœuds 1 et 2 qui sont partiellement affectés par la défiance du nœud en panne ; les autres nœuds peuvent continuer normalement l'exécution de l'algorithme. Toutefois, le blocage partiel des nœuds 1 et 2 diminue leurs disponibilités, mais il va permettre aux autres nœuds de poursuivre le calcul distribué.

6 Conclusion

Dans un premier volet de ce chapitre, nous avons proposé une approche formelle pour développer des algorithmes de synchronisation. Ces algorithmes adoptent des hypothèses probabilistes qui sont pratiquement constantes. L'approche est conçue par une modélisation formelle incrémentale qui utilise la méthode *B événementiel*. Toutefois, l'implémentation des probabilités est réalisée avec le model-checker probabiliste **PRISM**. L'approche est illustrée par un algorithme de synchronisation de type handshake que nous avons proposé.
Le résumé des obligations de preuve de la spécification de l'algorithm handshake et qui sont déchargées soit automatiquement, soit d'une manière interactive est ci-dessous présenté. Le tableau représente une mesure de la complexité du développement. A partir de ce tableau, nous pouvons signaler que le pourcentage des obligations de preuve déchargées automatiquement est faible et le pourcentage des obligations de preuve interactivement déchargées est égal à 61,25 %.

Les Machines Handshake	Nombre de obligations de preuve	Automatiquement déchargées	Interactivement déchargées
SpecMachine	10	6 (60%)	4 (40%)
ChoiceMachine	23	12 (52%)	11 (48%)
MessageMachine	43	13 (30%)	30 (70%)
FaultyMachine	4	0 (0%)	4 (100%)
Totale	80	31 (38.75%)	49 (61.25%)

Dans les prochains travaux, nous envisageons d'inclure la gestion des hypothèses probabilistes directement avec la méthode *B événementiel*. Aussi, nous proposons d'élargir l'application de ce modèle à une classe d'algorithme distribué plus grande que celle présenté dans ce chapitre.

Dans le deuxième volet de ce chapitre, nous avons présenté une étude analytique et expérimentale de l'algorithme ATH. Ainsi, l'étude analytique que nous avons élaborée, nous a permis de dégager les points suivants :

- Si le graphe ne contient aucune panne, alors les nœuds ont la même chance d'avoir des handshakes que l'algorithme SNH.
- Dans le cas où le graphe contient des pannes, nous avons montré que les nœuds, malgré la présence de la défaillance, réussissent à avoir des handshakes. Certes la probabilité d'en avoir est faible, mais elle est bel et bien différente de 0 dans la majorité des cas

L'étude expérimentale a permis de comparer notre algorithme à celui de SNH. Cette étude a montré que notre algorithme est plus performant en terme de réduction du nombre de messages échangés sur le graphe, de réduction du nombre de tentatives de synchronisation pour avoir des handshakes, et de la tolérance aux pannes. Comme future travail et afin d'améliorer les performances de notre algorithme, nous envisagerons de briser le choix probabiliste d'un nœud en ajoutant une procédure de choix intelligente. Autrement dit, le nœud ne choisit que les voisins qui sont prêts pour un handshake. Les premières études expérimentales ont montré que cette amélioration apporte de très bons résultats

en terme de performance. Cependant une étude analytique et une preuve formelle doivent être réalisées pour prouver la correction de l'algorithme.

Chapitre VI

Impléméntation des calculs locaux et génération du code

1 Introduction

La correction est un objectif capital dans le développement d'algorithmes distribués. Ces algorithmes doivent être rigoureusement spécifiés et prouvés par des méthodes formelles vus les enjeux qu'ils abritent. Cependant, étant donné la complexité du développement des algorithmes distribués d'une part, et de l'aspect mathématique des méthodes formelles d'autre part, la preuve de correction devient une tâche très ardue. En particulier, cette complexité prend une dimension considérable avec des utilisateurs non expérimentés.

Dans ce contexte, la solution que nous proposons consiste à conforter un développement formel par la visualisation et la simulation. Dans ce chapitre, nous proposons une nouvelle approche, appelée B2Visidia, qui permet de visualiser une spécification *B événementiel* d'un algorithme distribué. Cette approche est implémentée par un outil appelé aussi B2Visidia. Il permet de générer automatiquement un code Java à partir d'une spécification *B événementiel* d'un algorithme distribué codé par les calculs locaux. Le code généré est destiné à être exécuté seulement sous la plateforme Visidia. Ainsi, ce travail a trois principaux buts : premièrement, la visualisation d'algorithmes permettra à des concepteurs utilisant la méthode *B événementiel* d'avoir une meilleure idée sur les problèmes éventuels de leurs codes. Deuxièmement, la preuve d'algorithmes permettra d'obtenir une bibliothèque d'algorithmes prouvés sous Visidia. Troisièmement, la visualisation d'une spécification formelle pourra être considérée comme un assistant pour l'enseignement.

La suite de ce chapitre va être organisée de la façon suivante : d'abord, nous présentons un état de l'art sur la génération automatique du code à partir d'une spécification formelle. Dans cette première section, nous insistons sur la méthode *B événementiel*. Ensuite, nous présentons le cadre général dans lequel nous inscrivons B2Visidia. Ce cadre est illustré par une démarche constructive pour l'implémentation des algorithmes distribués codés par les calculs locaux. Par la suite, nous développons l'architecture globale de l'outil B2Visidia.

Enfin, nous essayons, dans la dernière partie de ce chapitre, de répondre à la question : comment B2Visidia traduit une spécification *B événementiel* ? Les éléments qui seront développés dans cette partie sont le langage, l'approche et l'outil B2Visidia.

2 État de l'art

La génération automatique de code est une technique standard de génie logiciel. Les théories et les outils en la matière sont nombreux et divergents. En effet, nous distinguons deux classes d'outils. La première classe renferme l'ensemble des outils qui génèrent un code à partir d'un autre mais tout en gardant le même niveau d'abstraction (par exemple, générer un code C à partir d'un code Java [78]). La deuxième classe regroupe les outils qui construisent à partir des modélisations abstraites, comme UML [79], des implémentations de bas niveau d'abstraction. Dans cette deuxième classe sont inscrits les outils de génération de code à partir des spécifications formelles. Toutefois, les travaux de génération de code à partir d'une spécification formelle sont relativement récents et ne sont pas assez nombreux. Parmi ces travaux nous exposons ceux qui supportent les langages formels VHDL, AADL, B classique et *B événementiel*.

Langages VHDL et AADL Dans [80], P. Christopher et al. ont proposé l'outil vMAGIC qui permet de générer automatiquement un code pour VHDL. Selon les auteurs, cet outil peut accélérer le processus de conception chaque fois où les tâches peuvent être automatisées et réutilisées plusieurs fois. Cependant, cet outil est considéré fiable bien que d'autres opérations sémantiques soient nécessaires pour le renforcer. Le travail [81] a introduit, **OCARINA**, un outil développé en Ada qui permet de générer automatiquement un code à partir des modèles AADL (Architecture Analysis and Design Language). Ces modèles ont été utilisés pour concevoir et valider l'efficience des systèmes DRE (Distributed Real-time Embedded).

Langage B classique La problématique de la génération automatique de code à partir du langage B classique a été bien développée par D. Bert, et al. dans [82]. Toutefois, la génération de code est conditionnée par la définition d'une machine dite d'implémentation (B0). Cette machine suppose que les instructions des opérations soient déterministes et les valeurs des variables ne soient ni des entiers bornés, ni des littéraux de types énumérés, ou des booléens ou des tableaux.

Les outils de la méthode B classique comme AtelierB et B4free intègrent directement des traducteurs comme ComenC [1] et C_4B [2] qui permettent d'obtenir un code C, C++ ou ADA à partir d'un modèle B0.

Langage *B événementiel* Dans [83], les auteurs introduisent un langage de spécification intermédiaire, appelé **OCB** (Orientée objet, Concurrent-B), pour relier les modèles *B*

1. http ://www.comenc.eu
2. http ://www.tools.clearsy.com

événementiel et les implémentations orientées objet. Le langage OCB est désormais utilisé pour spécifier des détails d'implémentation portant sur des aspects de la programmation concurrente dans un développement *B événementiel*. Comme suite à ce travail, les auteurs ont proposé dans [84], un outil qui supporte leur approche. En effet, cet outil est défini comme un "plug-in" intégré dans Rodin permettant la traduction d'une spécification OCB à une spécification *B événementiel* et à un code Java.

À notre connaissance, le premier travail intéressant qui permet de générer un code à partir du langage *B événementiel*, a été proposé par S. Wright [85]. Il a définit une approche qui permet de générer un code C à partir d'une spécification *B événementiel* en plusieurs étapes. Cette approche exige que la machine *B événementiel* soit d'abord raffinée jusqu'à un niveau dit aisément traduisible, spécifié avec un sous-ensemble du langage *B événementiel* et avec une syntaxe assez réduite. Ensuite, une fois la machine est bien spécifiée, le code source C qui lui correspond est automatiquement généré. Enfin, la dernière étape consiste à compiler le code source pour générer un fichier exécutable. L'auteur a implémenté son approche par un outil appelé B2C qui a été développé comme un "plug-in" pour Rodin.

Le travail de D. Mery et al. [86] est considéré comme une évolution de [85]. En effet, les auteurs ont proposé un outil qui peut générer automatiquement jusqu'à quatre codes à partir d'une spécification *B événementiel*. Le code généré dépend du langage de programmation choisi par l'utilisateur qui peut être C ou C++ ou Java ou C#. Ce travail ajoute des techniques de vérification du code généré qui permettent d'examiner les comportements souhaités du système spécifié. Ces techniques utilisent des méta-preuves et des outils de model cheker comme BLAST [87]. L'outil de génération de code est défini comme un "plug-in" de Rodin.

Notre proposition Du point de vu méthodologie, nous concevons notre approche de génération de code d'une façon très similaire aux travaux [85, 86, 88]. Nous définissons un sous langage de *B événementiel* qui satisfait les exigences de spécification des systèmes de ré-étiquetage de graphe, et qui est adapté à une traduction aisée. Nous définissons une traduction de la spécification sous plusieurs phases incluant la préparation du code, le filtrage, l'analyse et la génération de code (l'approche de traduction est expliquée plus tard dans la section 4).

Du point de vu objectifs, B2Visidia est différent des travaux liés méthodologie, puisque il se focalise seulement sur le domaine algorithmique distribuée. Plus précisément, il ne traite, comme programme source, que des spécifications d'algorithmes codés par le modèle des calculs locaux. Autant, le code Java généré par B2Visidia ne peut être exécuté que sur la plateforme Visidia. Enfin, l'outil que nous avons construit est indépendant de toutes autres plateformes ce qui présente un avantage en soi. Il est important de signaler qu'au moment de la rédaction de ce rapport, nous n'avons pas eu de connaissance sur d'autres travaux semblables permettant la génération de code à partir des spécifications codées par les calculs locaux.

3 Démarche constructive pour l'implémentation des calculs locaux

L'objectif de cette section est de décrire la démarche constructive que nous avons proposé pour l'impléméntation des calculs locaux. Au delà de l'implémentation, B2Visidia est peut être considéré comme un outil d'aide à la preuve de correction des calculs locaux. B2Visidia permet aussi de vérifier la cohérence d'un algorithme distribué avec son cahier des charges. Il est important de rappeler qu'un cahier des charges est la définition des besoins à satisfaire. Cependant une spécification est une description détaillée qui explique comment ces besoins doivent être réaliseés.

Dans cette démarche, l'outil B2Visidia a un rôle axial, il s'inscrit dans toutes les étapes qui sont en relation avec la preuve des algorithmes distribués à savoir la spécification, l'implémentation et la visualisation. Il constitue une passerelle entre l'étape de la spécification formelle et la visualisation de l'algorithme distribué en question. Ainsi, la visualisation d'une spécification formelle d'un algorithme distribué, permet de s'assurer de sa correction et d'examiner sa conformité au cahier des charges. Rappelons ici, que B2Visidia est un outil qui génère un code Java exécutable sous Visidia à partir d'une spécification formelle en *B événementiel* d'un algorithme distribué.

Notre démarche constructive comporte 3 différentes étapes : la spécification, la génération du code et la visualisation. Nous considérons le concepteur comme l'acteur principal dans cette démarche, il doit intervenir dans toutes les étapes. Au départ, le concepteur est invité à fournir une spécification de l'algorithme distribué. Pour cela, il peut soit utiliser la plateforme Rodin, soit utiliser B2Visidia en y éditant directement sa spécification. Toutefois, l'utilisation de Rodin garantit la correction de la syntaxe et des obligations de preuves à l'opposé de B2Visidia qui n'assure que la correction de la syntaxe.

La seconde étape consiste à traduire la spécification avec l'outil B2Visidia. Néanmoins, si l'utilisateur choisit Rodin dans la première étape de la démarche alors il doit importer le fichier source de la spécification à B2Visidia pour qu'il soit traduit. Cependant, le fichier source ne peut pas être directement traduit pour des raisons que nous allons expliquer dans la section 4. Alors, B2Visidia, via son système de filtrage et de réécriture, récupère et transforme le fichier source de Rodin pour qu'il soit prêt pour être traduit. Une fois le fichier source est filtré et réécrit (ou bien la spécification est directement éditée et validée syntaxiquement sous B2visidia), l'utilisateur peut traduire la spécification et générer le code Java approprié pour Visidia. La dernière étape consiste à lancer Visidia et à exécuter le code Java généré pour visualiser et tester la spécification de l'algorithme ; ici deux possibilités sont envisageables :

1. Si le test est réussi, l'utilisateur peut alors conclure que la spécification est une représentation correcte de l'algorithme.
2. À défaut, l'utilisateur localise le problème potentiel, le corrige, et recommence la démarche dès le départ. Cette opération peut être répétée autant de fois que nécessaire jusqu'à l'obtention du résultat souhaité. L'utilisateur ne s'arrête que lorsque le résultat de la simulation est jugé conforme à ses attentes (ou aussi au cahier des

charges).

Dans la figure Fig.VI.1, nous représentons une vue synoptique de la démarche ainsi que ses différentes étapes.

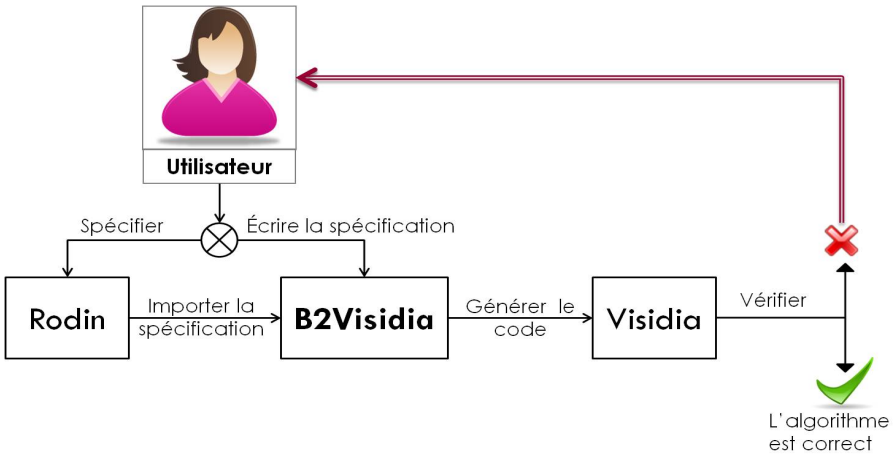


Figure VI.1 – Démarche constructive pour l'impléméntation des calculs locaux

4 Architecture globale

Traduire une spécification *B événementiel* dans un langage concret (comme le langage Java) n'est pas possible en une seule étape. Ceci est principalement dû au degré d'abstraction assez élevé des spécifications élaborées avec un tel langage. Bien entendu, cette abstraction ne tient pas compte de l'implémentation de l'algorithme et elle peut omettre beaucoup de détails. A titre d'exemple, nous pouvons citer le type de synchronisations appliqué par l'algorithme (LC0 ou LC1 ou LC2) et le paradigme de communication adopté par les nœuds du graphe. Aussi, une spécification B événementiel d'un algorithme distribué peut inclure des conditions globales, ce qui complique par la suite l'extraction des données locales de chaque nœud du graphe. L'autre difficulté que nous avons rencontrée est l'extraction des variables indispensables pour l'implémentation ainsi que leurs types respectifs. En effet, une spécification peut inclure aussi des variables qui ne sont définis que pour être utilisés dans le processus de preuve.

B2Visidia peut traduire une spécification *B événementiel*, argumentée par des annotations utiles, en un code Java suivant une approche bien déterminée. Comme indiqué dans la figure Fig.VI.2, notre approche comprend trois étapes principales : la préparation du fichier source, l'analyse du fichier interprétable et la génération du code. La première étape consiste à préparer le fichier source pour qu'il soit aisément analysé (le fichier source est produit par la plateforme Rodin sous la forme XML). L'objectif de cette étape est de générer un fichier simple, interprétable et ne contenant que les données utiles pour la traduction. Pour atteindre cet objectif, nous avons choisi d'utiliser TOM [89] qui est un

langage et un environnement logiciel adapté pour la programmation de diverses transformations sur les arbres/termes. Le langage TOM est une extension des langages impératifs existants (comme Java et C). Il ajoute des constructions de filtrage inspirées par la réécriture de signatures algébriques qui permet de connaître des motifs dans une structure de données. Après la reconnaissance du motif, TOM peut soit filtrer vers la structure, soit exécuter une action écrite en Java. Pour notre contexte, TOM peut parfaitement être utilisé pour réécrire des documents XML.

Dans la deuxième étape de l'approche, nous entamons une analyse lexicale et syntaxique du fichier interprétable. L'analyse lexicale est accomplie par un outil logiciel appelé JFLEX [3]. Cet outil permet de générer des analyseurs lexicaux. Il lit les caractères du fichier source un à un et les sépare sous forme d'unités lexicales. Ensuite, et afin de reconnaître la structure de la spécification, nous exécutons une analyse syntaxique. Cette analyse permet de connaître les relations de dépendance entre les unités lexicales et de construire par la suite une représentation syntaxique du fichier source. Cette représentation est codée par le biais d'un Arbre Abstrait Syntaxique (AST). La génération de l'AST est une affirmation au sujet de la conformité de la spécification de l'algorithme distribué au langage B2Visidia. Ce dernier est un sous langage de *B événementiel* dédié pour les calculs locaux (nous allons le présenter dans la section 5.2). Autrement dit, le fichier interprétable satisfait bien les règles grammaticales imposées par la syntaxe du langage B2Visidia. Pour la réalisation de l'analyse syntaxique, nous utilisons l'outil CUP [4]. Cet outil est écrit en Java et génère des analyseurs écrits avec le même langage.

Dans la dernière étape, nous utilisons des règles appropriées pour effectuer la traduction des nœuds de l'AST et pour générer le code Java dédié pour Visidia.

5 Comment B2Visidia traduit une spécification *B événementiel* ?

Dans le chapitre III, nous avons présenté un patron formel pour le développement des algorithmes codés par les calculs locaux. Nous avons montré, qu'avec un développement basé essentiellement sur le raffinement et les techniques de preuves des calculs locaux, nous pouvons produire des spécifications d'algorithmes correctes par construction. Cependant, les spécifications produites sont correctes et concrètes, mais elles ne sont pas adaptées à une traduction aisée. Ceci est dû principalement à la richesse du langage *B événementiel* qui offre une multitude de possibilités dans la description d'une spécification. Dans le projet B2Visidia, nous proposons d'ajouter une étape de pré-traitement sur la base du dernier modèle afin d'en produire un autre constamment correct et concret, et aussi traduisible par notre outil. L'étape de pré-traitement consiste à introduire un niveau de raffinement décrit strictement par le langage B2Visidia. Cette phase de raffinement fournit des définitions déterministes des constantes, des variables et des événements tout en assurant la correction et l'adaptabilité à la traduction. Dans la suite de cette section,

3. http://jflex.de
4. http://www2.cs.tum.edu/projects/cup

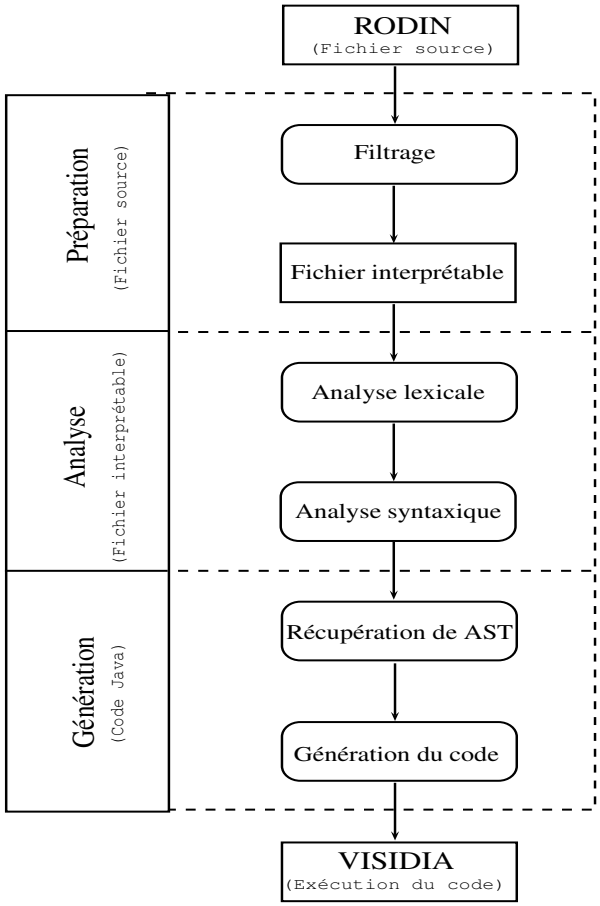


Figure VI.2 – Approche B2Visidia

nous rappelons d'abord la méthodologie de développement formel qui permet de générer des modèles pour B2Visidia. Ensuite, nous détaillons le langage B2Visidia. Enfin, nous expliquons comment notre outil génère un code Java pour Visidia.

5.1 Méthodologie de développement formel pour B2Visidia

Le diagramme suivant se base sur le patron présenté et détaillé dans le chapitre III. Il introduit en plus un niveau de raffinement. La caractéristique primordiale de ce niveau est de permettre la spécification d'une dernière machine décrite avec le langage B2Visidia et traduisible par notre outil. La figure Fig.VI.3 schématise les différentes étapes du diagramme :

- Les contextes déclarent les propriétés des graphes et du problème mathématique à résoudre.
- Le niveau $Machine_0$ spécifie le problème à résoudre par le biais des événements qui modélisent la relation entre l'état initial et l'état final. Ce niveau demeure très

abstrait.
- Le deuxième niveau, appelé $Machine_1$, est obtenu après des raffinements du précédent. $Machine_1$ peut introduire les propriétés inductives qui permettent d'exprimer le calcul dans un modèle de calculs locaux.
- Le troisième niveau, appelé $Machine_2$, découle des raffinements de celui d'avant. Ce niveau produit une description concrète de l'algorithme distribué spécifié. Il permet de spécifier toutes les règles de ré-étiquetage de l'algorithme.
- Le quatrième niveau, appelé $Machine_V$, est le résultat d'un raffinement de $Machine_2$. Concrètement, ce niveau est défini par une seule machine. Cette dernière est spécifiée par le langage B2Visidia et répond à toutes les exigences imposées par notre outil.
- À partir de la machine $Machine_V$, nous générons le code Java qui peut être exécuté sous Visidia. Ce code est l'implémentation du problème sous la plateforme Visidia.

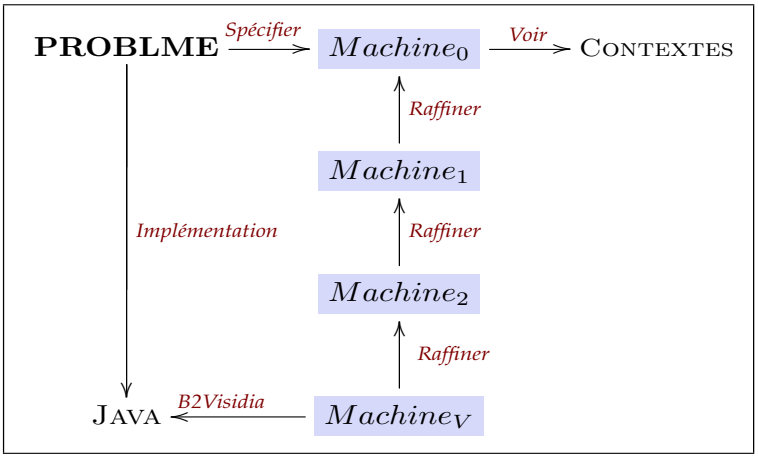


Figure VI.3 – Diagramme de développement formel pour B2Visidia

5.2 Langage B2Visidia

Le langage *B événementiel* offre une ample possibilité dans la spécification d'un algorithme distribué. En effet, syntaxiquement une même spécification peut être réécrite sous plusieurs différentes formes. Ceci complique énormément l'analyse de la spécification et rend laborieux sa traduction. Pour pallier à ce problème, nous proposons un nouveau langage appelé B2Visidia. Ce dernier est un sous langage de *B événementiel* permettant la spécification d'une large classe d'algorithmes codés par les calculs locaux. Sa syntaxe est largement simplifiée pour garantir une traduction non-ambiguë. Dans la définition de ce langage, nous gardons les mêmes définitions d'associativité et de priorité des opérateurs qui sont adoptés par *B événementiel*. Dans la suite de cette section, nous présentons la syntaxe du langage B2Visidia et nous détaillons les différentes conditions à entretenir afin d'assurer une traduction correcte de la spécification.

5.2.1 Contexte

Notre approche traduit uniquement les machines. Afin d'éviter l'analyse du contexte, nous avons imposé quelques noms conventionnels pour les ensembles et les constantes : ND pour encoder les nœuds, g pour encoder les arêtes, ID pour encoder l'identificateur unique d'un nœud dans le graphe, et *card* pour encoder le degré d'un nœud. En plus, pour ce qui concerne le typage des variables nous avons mis en place une solution qui permet de connaître les types sans faire recours aux contextes. Nous expliquons cette solution plus tard dans la section 5.2.2.

5.2.2 Machine

Une machine peut contenir des sections qui servent seulement à prouver la correction de la spécification. À titre d'exemple, nous citons les sections THEOREMS et VARIANT. En effet, elles sont nécessaires pour faire la preuve, mais elles ne sont pas appropriées pour exprimer l'aspect fonctionnel de l'algorithme distribué. Par conséquent, les sections THEOREMS et VARIANT ainsi que les clauses de structuration de machines (SEES, REFINES,...) seront supprimées du fichier source, une fois il est filtré. L'outil B2Visidia considère quatre sections nécessaires pour la traduction d'une machine : un nom de machine, un type de synchronisation, un ensemble de variables, des invariants et des événements.

Les types de synchronisation Le type de synchronisation ne peut pas être ajouté dans une spécification d'un algorithme distribué car il est considéré comme un détail d'implémentation. Afin de combler cette lacune, nous utilisons des annotations pour guider l'outil B2Visidia dans la translation et la génération du code. Les annotations seront ajoutées au code *B événementiel* sous forme de commentaires pour guider le processus de traduction. Ainsi, les annotations seront une partie intégrante du langage B2Visidia défini. Elles se différencient des autres commentaires par le caractère #. L'annotation de synchronisation est ajoutée dans l'espace du commentaire en face du nom de la machine. Nous distinguons trois différents types d'annotation de synchronisation : #LC0, #LC1 et #LC2.

Les variables Dans un système de ré-étiquetage de graphe (GRS), les nœuds et les arêtes sont tous étiquetés par des labels qui appartiennent à un ensemble d'alphabets finis. Concrètement, l'état d'un nœud ou d'une arête peut être encodé par un ou plusieurs labels à la fois. Ceci dépend essentiellement du nombre de propriétés qui définissent cet état. Toutefois, afin de spécifier un algorithme distribué codé par les calculs locaux, la dernière machine concrète doit déclarer une fonction pour chaque propriété. La fonction permet d'attribuer un label par propriété pour chaque nœud ou arête. Les noms de ces fonctions doivent être déclarés dans la clause VARIABLE. Dans la syntaxe du langage B2Visidia, nous attribuons le nom "*visidia_variable*" aux variables qui déclarent ces fonctions.

Les invariants Les "*visidia_variable*" sont définis et typés dans la clause INVARIANTS de la façon suivante [5] :

$$<visidia\_invariant> ::= <visidia\_variable> \text{ '}\in\text{' } <label\_type>$$

$$\begin{aligned} <label\_type> ::= & \text{ 'ND' } <relational\_set\_op> <set\_expression> \# <type> \\ & \mid \text{ 'g' } <relational\_set\_op> <set\_expression> \end{aligned}$$

<*relation_set_op*> est l'ensemble des opérateurs qui servent à construire des relations ou des fonctions.

$$<\text{relational\_set\_op}> ::= \leftrightarrow \mid \nleftrightarrow \mid \twoheadleftarrow \mid \nleftrightarrow \mid \rightarrow \mid \rightarrow \mid \rightarrowtail \mid \rightarrowtail \mid \twoheadrightarrow \mid \twoheadrightarrow \mid \twoheadrightarrow$$

<*label_type*> permet de préciser si la fonction spécifie un label d'un nœud (la première forme) ou d'une arête (la deuxième forme). Également, <*label_type*> permet aussi de fixer le type de labels qui correspond à chaque fonction. Comme nous l'avons précisé précédemment, nous n'allons pas faire recours au contexte pour déterminer les types, mais nous allons les récupérer depuis une annotation spéciale. Cette annotation est ajoutée par l'utilisateur, dans la zone des commentaires, après chaque définition de fonction. Étant donné que le type est indiqué directement par l'utilisateur, alors l'expression <*set_expression*> ne sera pas analysée par notre outil. Nous distinguons quatre différentes annotations de types (les types supportés par B2Visidia) :

$$\begin{aligned} <type> ::= & \text{ 'int' //un nombre entier} \\ & \mid \text{ 'Bool' //un booléen} \\ & \mid \text{ 'String' //une chaîne de caractères} \\ & \mid \text{ 'float' //un nombre à virgule flottante} \end{aligned}$$

Cependant, l'outil Visidia connaît une limite au niveau de l'étiquetage des arêtes. En fait, le marquage est la seule action possible sur un état d'une arête. Cela signifie qu'une arête ne peut avoir que deux états possibles : marqué ou non-marqué. Ainsi, le type de la fonction qui spécifie l'état d'une arête est inévitablement booléen : le label " true " pour représenter l'état d'une arête marquée et le label " false " dans le cas contraire. L'annotation de type pour une telle fonction est désormais sans importance et pour cette raison elle n'a pas été introduite dans le langage B2Visidia.

Les événements Dans le chapitre II, nous avons montré que plusieurs types d'événements peuvent être utilisés dans une spécification formelle *B événementiel*. Parmi ces types, nous attestons que l'événement indéterministe est le mieux adopté pour décrire une règle de ré-étiquetage. Dans sa clause ANY, il permet de déclarer les nœuds et les arêtes associés à l'exécution d'une règle de ré-étiquetage. La garde et les actions de cet événement correspondent respectivement aux conditions et aux actions de la règle. Ainsi, cet

5. Nous utilisons dans ce chapitre la notation de forme étendue Backus-Naur (Extended Backus-Naur Form - EBNF) pour décrire la syntaxe. Dans cette notation, les non-terminaux sont entourés par des crochets angulaires et les terminaux sont entourés par deux apostrophes.

événement fera une partie intégrante de notre langage B2Visidia. D'autre part, les clauses REFINES et WITH servent principalement à prouver la correction de l'événement. Pour cette raison, ces clauses ne seront pas inclues au langage B2Visidia et elles sont supprimées après le filtrage de la spécification. La syntaxe d'un événement dans B2Visidia est présentée comme suit :

<event_name>
ANY
<variable_name>
.....
WHERE
<label> : <visidia_condition>
.....
THEN
<label> : <visidia_action>
.....
END

L'événement d'initialisation Un événement d'initialisation est un événement spécial qui permet de définir les valeurs initiales de toutes les variables d'état du modèle. Dans notre contexte, cet événement permet d'initialiser les états de tous les nœuds et les arêtes du graphe. Cependant, une vue plus rapprochée révèle que l'initialisation peut être soit globale, soit locale. L'initialisation globale affecte une même valeur à tous les nœuds du graphe. L'initialisation locale distingue quelques nœuds et les initialise d'une manière différente de tous les autres. Toutefois, l'initialisation locale représente un vrai obstacle pour la traduction de cet événement. Parce que, aucune primitive de la bibliothèque d'algorithmes distribués, de Visidia, ne permet une telle initialisation. La seule solution de ce problème est alors de faire l'initialisation manuellement via l'interface de Visidia. Afin de simplifier la traduction de cet événement, B2Visidia permettra via son assistant de génération du code, une initialisation globale plus aisée (voir Section 6.2). Néanmoins, pour une initialisation globale associée à une initialisation locale, le concepteur peut différencier manuellement les nœuds particuliers, en utilisant l'interface de Visidia.

La clause ANY La clause ANY d'un événement permet de déclarer les noms des nœuds et des arêtes nécessaires pour la réalisation de la règle de ré-étiquetage. Nous signalons, que dans le but de simplifier la détection du nœud centre de la boule, nous avons fixé sa nomination. Le nom que nous avons attribué est "c". La distinction de ce nœud spécial permet de simplifier énormément le processus de translation.

La clause WHERE Une condition peut être exprimée de différentes formes syntaxiques. A titre d'exemple, la condition $lab(x)=A$ peut être écrite $lab[\{x\}] =A$ ou $x \mapsto A \in lab$,etc. (x est un nœud quelconque, A est un label et lab est un *visidia_variable*). Afin de simplifier le processus de translation et de génération automatique de code, nous

avons choisi d'adopter deux possibles formes pour écrire une condition sous le langage B2Visidia. La première forme est simple ; elle est définie par un entête suivi par un opérateur de comparaison, puis par une expression. La deuxième forme est plus complexe ; elle est définie par une proposition mathématique qui utilise un quantificateur universel. Cette deuxième forme est utilisée, uniquement par les algorithmes de types LC1 ou lC2, pour que le nœud centre de la boule puisse parcourir l'ensemble de ses voisins.

L'entête de la première forme de la condition est représenté par le nom de la fonction <visidia_variable> suivi par une variable qui peut être soit un nœud soit une arête. Dans le langage B2Visidia, nous avons choisi d'appeler cet entête <visidia_fonction>. Syntaxiquement, <visidia_fonction> est défini comme suit :

$$\begin{aligned} &<\textit{visidia\_fonction}> \; ::= \; <\textit{visidia\_fonction}> \; \textit{'p\_a\_c'} \; \textit{'ident'} \; \textit{'p\_a\_c'} \\ &\quad \mid \; <\textit{visidia\_fonction}> \; \textit{'p\_a\_c'} \; \textit{'ident'} \; \textit{'}\mapsto\textit{'} \; \textit{'ident'} \; \textit{'p\_a\_c'} \end{aligned}$$

Nous désignons par 'p_a_c' une série de crochets, accolades ou parenthèses et par <ident> une chaîne de caractères. La première forme syntaxique de <visidia_fonction> correspond à une condition qui porte sur un label d'un nœud. La deuxième forme syntaxique est relative à une condition qui porte sur un label d'une arête. Toutefois, le concepteur peut toujours utiliser la première forme pour spécifier, s'il le veut, des conditions qui portent sur un label d'une arête. L'ensemble des opérateurs de comparaison concrets, déterministes et traduisibles sont :

$$<\textit{relop}> \; ::= \; \textit{'}>\textit{'} \mid \textit{'}<\textit{'} \mid \textit{'}\leq\textit{'} \mid \textit{'}\geq\textit{'} \mid \textit{'}=\textit{'} \mid \textit{'}\neq\textit{'}$$

Afin de développer la syntaxe d'une expression, nous nous sommes largement inspirés du langage *B événementiel*. La syntaxe d'une expression comme elle a été définie par B2Visidia est présentée comme suit :

$$\begin{aligned} &<\textit{expression}> \; ::= \; <\textit{term}> \\ &\quad \mid \; \textit{'-'} \; <\textit{term}> \\ &\quad \mid \; \textit{'p\_a\_c'} \; <\textit{expression}> \; \textit{'p\_a\_c'} \\ &<\textit{term}> \; ::= \; <\textit{term1}> \\ &\quad \mid \; <\textit{term1}> \; \textit{'+'} \; <\textit{term}> \\ &\quad \mid \; <\textit{term1}> \; \textit{'-'} \; <\textit{term}> \\ &<\textit{term1}> \; ::= \; <\textit{factor}> \\ &\quad \mid \; <\textit{factor}> \; \textit{'*'} \; <\textit{term1}> \\ &\quad \mid \; <\textit{factor}> \; \textit{'/'} \; <\textit{term1}> \\ &\quad \mid \; <\textit{factor}> \; \textit{'mod'} \; <\textit{term1}> \\ &<\textit{factor}> \; ::= \; \textit{'}\top\textit{'} \\ &\quad \mid \; \textit{'}\bot\textit{'} \\ &\quad \mid \; \textit{'number'} \\ &\quad \mid \; <\textit{commun}> \\ &<\textit{commun}> \; ::= \textit{'ident'} \\ &\quad \mid \; <\textit{visidia\_fonction}> \\ &\quad \mid \; \textit{'card'} \; \textit{'p\_a\_c'} \; \textit{'g'} \; \textit{'p\_a\_c'} \; \textit{'ident'} \; \textit{'p\_a\_c'} \\ &\quad \mid \; \textit{'ID'} \; \textit{'p\_a\_c'} \; \textit{'ident'} \; \textit{'p\_a\_c'} \\ &\quad \mid \; \textit{'g'} \; \textit{'p\_a\_c'} \; \textit{'ident'} \; \textit{'p\_a\_c'} \end{aligned}$$

Pour ce qui concerne la deuxième forme de la condition, nous avons choisi d'utiliser uniquement le quantificateur universel ∀. Ceci permet de simplifier la spécification formelle et la génération de code. Nous avons choisi d'éliminer l'opérateur existentiel du fait qu'il soit équivalent à l'opérateur universel. Selon la logique classique, nous avons les deux théorèmes suivants :

$$(\neg \forall \text{ x } \text{P(x)}) \Rightarrow (\exists \text{ x } \neg \text{ P(x)})$$
$$(\neg \exists \text{ x } \text{P(x)}) \Rightarrow (\forall \text{ x } \neg \text{ P(x)})$$

Nous signalons aussi, que cette deuxième forme de condition permet de déclarer des nouvelles variables qui portent des nouvelles propriétés sur un ensemble de voisins du nœud centre de la boule. Ces nouvelles variables sont par la suite utilisées pour permettre un ré-étiquetage d'un nombre non définis des nœuds voisins. Dans le langage B2Visidia, nous appelons ce ré-étiquetage " modification multiple non énumérée " que nous allons le présenter dans la section suivante. La syntaxe d'une condition dans le langage B2Visidia est présentée comme suit :

<visidia_condition> : := <literal>
| <literal> ∧ <visidia_condition>
| <literal> ∨ <visidia_condition>
| <predicat>
<literal> : := <atomic_predicat>
| '¬' <atomic_predicat>
<atomic_predicat> : := 'p_a_c' <visidia_condition> 'p_a_c'
| <expression1> <relop> <expression>
<expression1> : := <visidia_fonction>
| 'card' 'p_a_c' 'g' 'p_a_c' 'ident' 'p_a_c'
| 'ID' 'p_a_c' 'ident' 'p_a_c'
<predicat> : := '∀' 'ident_list_comma' '·' <ident_dec> '∧' <visidia_condition> '⇒' <visidia_condition>
| '∀' 'ident_list_comma' '·' <ident_dec> '⇒' <visidia_condition>
<ident_dec> : := <expression> '∈' <expression>
| <expression> '∈' <expression> '∧' <ident_dec>
<ident_list_comma> : := 'ident'
| 'ident' ',' <ident_list_comma>

La clause THEN Parmi les trois substitutions généralisées présentées dans la Section 3.2.2 du chapitre II ; seule la substitution déterministe est traduisible par l'outil B2Visidia. D'une façon générale, cette substitution permet de mettre à jour les valeurs des variables qui encodent les états des nœuds ou des arêtes (<visidia_variable>). Ainsi, nous distinguons trois possibles formes d'une substitutions : simple ou multiple énumérée ou multiple non énumérée. Autrement dit, elles peuvent modifier à la fois soit un seul label

d'un seul nœud (ou arête) soit des labels de plusieurs nœuds (ou arêtes). Syntaxiquement, le langage B2Visidia définit une substitution généralisée de la façon suivante :

<visidia_action> : := <visidia_fonction> ' :=' <expression>
| <visidia_variable> ' :=' <visidia_variable> '⩤ ' 'p_a_c' <nouvel_etat> 'p_a_c'
| <visidia_variable> ' :=' <visidia_variable> '⩤ ' 'ident'
<nouvel_etat> : := 'ident' '↦' <expression>
| 'ident' '↦' <expression> ',' <nouvel_etat>

La première forme de <visidia_action> correspond à une substitution simple, la deuxième correspond à une substitution multiple énumérée, et la troisième correspond à une substitution multiple non énumérée. Afin de bien illustrer la différence entre ces trois formes, nous présentons les exemples suivants (*Lab* est une *<visidia_fonction>*, x et y sont deux nœuds différents, et A et B sont deux labels) :

une modification simple *Lab(x) := A* :le nouveau label du nœud x est A.

une modification multiple énumérée $Lab := Lab \Leftarrow \{ x \mapsto A, y \mapsto B\}$: les nouveaux labels de x et y sont respectivement A et B.

une modification multiple non énumérée *Lab := Lab ⩤ nouveau_Lab* : dans cette substitution nous ne connaissons pas exactement le nombre de nœuds qui vont ré-étiqueter leurs états. La nouvelle variables *nouveau_Lab* va permettre d'emporter les nouveaux labels des nœuds qui vont ré-étiqueter leurs états. Cette variable est déclarée comme une variable locale de l'événement dans clause ANY, ensuite elle est définie dans la clause WHERE.

Nous rappelons que l'opérateur $\Leftarrow$ désigne la surcharge. R1 $\Leftarrow$ R2 correspond à une relation constituée des éléments de R2 et des éléments de R1 dont le premier élément n'appartient pas au domaine de R2.

5.3 Génération de code

Le but de cette section est de montrer les principes de base pour traduire une spécification *B événementiel* d'un algorithme distribué vers un code Java. Le fichier Java généré possède le même nom que la machine *B événementiel* source. Il est compilé et inséré dans un répertoire choisi par l'utilisateur. Par la suite, Visidia implémente ce code sur chaque nœud du graphe pour simuler l'exécution de l'algorithme.

Nous allons utiliser les formalismes dirigés par la syntaxe pour traduire la spécification *B événementiel*. L'approche la plus générale pour une traduction dirigée par la syntaxe consiste à construire un arbre syntaxique abstrait (AST), puis à calculer la valeur des attributs aux nœuds de cet arbre [90]. Souvent, une analyse sémantique est nécessaire pour réussir la génération du code. Une fois l'AST est généré, nous allons le parcourir d'une façon linéaire. Tout d'abord, nous commençons par extraire le nom de la machine, puis l'annotation de synchronisation, ensuite nous calculons les *variable_visidia* et enfin nous traduisons les événements un par un.

5.3.1 Annotation de synchronisation

L'annotation de synchronisation joue un rôle primordial dans la définition de la structure du code Java produit par B2Visidia. En effet, nous distinguons trois différents patrons du code Java sous Visidia (un patron pour chaque type de synchronisation). Pour les algorithmes de type LC0, les règles de ré-étiquetage sont introduites juste après l'échange de labels entre les deux nœuds synchronisés. Le patron du code Java pour les algorithmes de type LC0 est illustré par la figure FigVI.4. Pour les algorithmes de type LC1, nous ajoutons une structure de contrôle afin de distinguer le nœud centre de la boule. En effet, dans le cas où le nœud qui exécute l'algorithme est le centre de la boule alors les règles de ré-étiquetage sont introduites juste après la réception de tous les messages des nœuds voisins. Dans le cas où le nœud qui exécute l'algorithme est une feuille, alors il doit uniquement envoyer son message au nœud centre de la boule. Dans la figure Fig.VI.5, nous présentons le patron du code Java pour les algorithmes de type LC1. Les algorithmes de types LC2 sont similaires à ceux des LC1. Par contre, les règles de ré-étiquetage peuvent être appliquées à la fois par les nœuds centre et feuilles de la boule. Ce dernier patron est décrit par la figure Fig.VI.6. Les codes des algorithmes de synchronisations sont récupérés depuis l'API Visidia. Cet API fournit de nombreuses méthodes qui aident l'utilisateur à implémenter son algorithme distribué sous Visidia.

```
while (run)
{
  envoyer_label();
  reçevoir_label();
  /**règles de ré-étiquetage**/
  breakSynchro();

}
```

Figure VI.4 – Patron Java pour les algorithmes LC0

```
while (run){
  if (noeud == Centre_boule){
  for (parcours des noeuds voisins){
    reçevoir_label();
     /**règles de ré-étiquetage**/
     breakSynchro();}}
    else
    envoyer_label();}
```

Figure VI.5 – Patron Java pour les algorithmes LC1

5.3.2 Invariants

Visidia implémente un nœud par une classe Java qui contient un ensemble de propriétés éditables et un tableau de bord. Les propriétés d'un nœud sont son identifiant, son état interne, son degré et éventuellement la taille de graphe. Le tableau de bord permet de sauvegarder les messages reçus. Les messages sont programmés par une classe qui contient toutes les informations indispensables, parmi les méthodes les plus importantes nous citons :

- *sendTo(voisin, message)* : envoyer un message à un voisin particulier.
- *receiveFrom(int)* : recevoir un message d'un voisin particulier.

Dans un sens, la traduction des invariants permet de déclarer l'état interne des nœuds. Dans un autre sens, elle permet de définir le tableau de bord de chaque nœud. Ainsi, pour

```
while (run){
  if (noeud == Centre_boule){
    for (parcours des noeuds voisins){
      reçevoir_label();
      /**règles de ré-étiquetage**/
      breakSynchro();}
   }
    else {
      envoyer_label();
      reçevoir_label();
       /**règles de ré-étiquetage**/
   }}
```

Figure VI.6 – Patron Java pour les algorithmes LC2

un algorithme de type LC0, le tableau de bord est implémenté par une simple variable dans le cas où l'état interne des nœuds est encodé par un seul label. Autrement, l'état interne est encodé par un vecteur Java. Pour un algorithme de type LC1 ou LC2, et dans le cas où l'état interne des nœuds est encodé par un seul label, le tableau de bord est encodé par un tableau ayant comme taille le nombre de voisins du nœud. Dans l'autre cas, le tableau de bord est encodé par une matrice. Celle ci ayant le même nombre de lignes que le nombre de voisins d'un nœud, et dont le nombre de colonnes est le nombre de labels qui encodent l'état interne des nœuds.

5.3.3 Événements

Précédemment, nous avons précisé que, dans une machine concrète, chaque événement correspond à une règle de ré-étiquetage de l'algorithme distribué. En fonction du type de l'algorithme spécifié, les événements traduits seront insérés dans le code Java généré selon les patrons présentés dans la Section 5.3.1. Un événement est peut être traduit par un ou deux structures de contrôle *if (condition) then (action)*. En effet, il est traduit par deux structures de contrôle dans ces deux cas :

1. quand il spécifie un ré-étiquetage à la fois des deux nœuds synchronisés pour un algorithme distribué de type LC0,
2. quand il spécifie un ré-étiquetage des nœuds centre et feuilles de la boule pour un algorithme distribué de type LC2.

Dans ces deux cas, chaque structure de contrôle permet d'implémenter l'événement selon la perception du nœud qui va l'exécuter. Afin de bien illustrer ce point, nous présentons un exemple d'un événement qui appartient à un algorithme de type LC0. L'exemple est présenté par la figure VI.7. Cet événement permet de ré-étiqueter les labels des deux nœuds synchronisés $n1$ et $n2$. Dans cet exemple, nous supposons que l'état d'un nœud est défini par un seul label, et *Lab* est la fonction qui l'encode. Nous signalons que les

Spécification *B événementiel*

EVENT exemple
ANY n_1, n_2
WHERE
$grd1 : Lab(n_1) = 10$
$grd2 : Lab(n_2) = 20$
THEN
$act1 : Lab := Lab \Leftarrow \{n_1 \mapsto 20, n_2 \mapsto 10\}$
END

$\Rightarrow$

Code Java

```
.....................
/* Structure de contrôle 1 */
if (mon_label==10 && voisin_label == 20)
{ mon_label = 20; }
/* structure de contrôle 2 */
if (mon_label==20 && voisin_label == 10)
{ mon_label = 10; }
.....................
```

Figure VI.7 – Exemple de traduction d'un événement LC0

	Un nœud	Une arête	Commentaires
Récupération de son propre label	*this.getproperty()*	×	Sous Visidia V1.3 la récupération de l'état d'une arête est impossible
Récupération d'un label voisin	*this.receivefrom()*	×	
Modification de son propre label	*this.PutProperty()*	*this.setDoorState()*	
Modification d'un label voisin	*this.sendTo()*	×	Cette modification est possible qu'avec un agorithme LC2 : le nœud centre de la boule calcule et envoi le nouveau label et ensuite la feuille fait le ré-étiquetage

Tableau VI.1 – Principales méthodes de manipulation des labels sous Visidia

nœuds effectuent leurs calculs distribués sur la base des labels reçus avant l'application de l'événement.

La traduction des gardes et des actions d'un événement est réalisée à l'aide d'un ensemble de règles de réécriture de code. Ces règles permettent de traduire la syntaxe *B événementiel* en code Java, et aussi d'implémenter les *fonction_visidia*. Une *fonction_visidia* est implémentée par une méthode de l'API Visidia qui permet de récupérer l'état d'un nœud ou d'une arête. Cependant, elle peut être aussi implémentée par une méthode de ré-étiquetage si elle est placée dans la partie gauche d'une substitution généralisée. Dans le tableau VI.1 nous récapitulons l'ensemble des méthodes de l'API Visidia qui peuvent implémenter une fonction_visidia dans les deux cas de figures qui sont la récupération et la ré-étiquetage d'un label. Nous présentons dans le tableau VI.2 la correspondance entre la syntaxe du langage *B événementiel* et du langage Java. Dans ce tableau, nous supposons que *Lab* est une fonction qui encode l'état d'un nœud.

5.4 Étude de cas

Dans cette section, nous illustrons à travers un exemple simple le principe de génération automatique de code. L'exemple choisi est l'algorithme de calcul d'arbre recouvrant

Langage *B événementiel*	Langage Java	Commentaires
x = y	x == y ou x.equal(y)	Condition
x ≠ y	x != y	Condition
x > y	x > y	Condition
x < y	x < y	Condition
x ≤ y	x <= y	Condition
x ≥ y	x >= y	Condition
¬ (Condition)	!(Condition)	Condition
(Condition) ∨ (Condition)	(Condition) || (Condition)	Condition
∀ variables · declaration	for (parcours voisins)	Condition
⇒ Condition	{Condition}	
(Condition) ∧ (Condition)	(Condition) && (Condition)	Condition
Lab(x) := y	this.PutProperty(y) ;	Affectation
Lab := Lab ⩤ { x ↦y }	this.PutProperty(y) ;	Affectation
Lab(x) := y + z	this.PutProperty(y + z) ;	Affectation
Lab(x) := y - z	this.PutProperty(y - z) ;	Affectation
Lab(x) := y * z	this.PutProperty(y * z) ;	Affectation
Lab(x) := y ÷ z	this.PutProperty(y / z) ;	Affectation
ID(x)	*Integer getId()*	Méthode Visidia
card(g(x))	*int getArity()*	Méthode Visidia

Tableau VI.2 – Traduction de la syntaxe *B événementiel* vers Java

spécifié dans la Section 6.1 du chapitre III . Sur la base de cette spécification, nous raffinons encore le modèle afin d'obtenir une machine traduisible par B2Visidia. En effet, nous ajoutons l'annotation de synchronisation correspondante au type de l'algorithme (à savoir #LC0). Étant donné que les invariants et le choix de la terminologie sont bien spécifiés (ils respectent le langage B2Visidia), donc il suffit d'ajouter les annotations de type correspondantes. Effectivement, nous ajoutons l'annotation #String à l'invariant inv2 qui déclare <visidia_variable> lab. Quant aux événements, aucune modification ne va leur être apportée. En effet l'événement qui spécifie la seule règle de ré-étiquetage est considéré traduisible par notre outil. Les autres événements ne seront pas analysés car ils seront supprimés après l'étape de filtrage.

L'opération de filtrage va supprimer *inv2* car il ne spécifie pas une <visidia_variable>. De même, elle va garder seulement l'événement Rule car c'est le seul qui spécifie une règle de ré-étiquetage. L'outil TOM reconnaît un événement par son type (déterministe et non-déterministe) et par l'ensemble des <visidia_variable> qu'il abrite. L'analyse syntaxique et lexicale produit un arbre abstrait syntaxique qui nous permet d'extraire les deux <visidia_variables> de la spécification ainsi que leurs types : **lab** est une fonction qui encode l'état d'un nœud qui est de type String, et **mark** est une fonction pour les arêtes, elle est de type Bool.

La génération de code Java pour Visidia commence par la création d'une nouvelle classe et d'un nouveau fichier ayant le même nom que celui du modèle *B événementiel*. La classe Java générée étend la classe *Algorithm* définis dans l'API Visidia. Cette dernière incorpore toutes les primitives nécessaires pour assurer la communication entre des processus non-mobiles. Ensuite, nous déclarons les variables nécessaires pour l'affichage et le déroulement de la visualisation. Par la suite, nous ajoutons la méthode init() qui va inclure la traduction de l'événement Rule selon le pattern LC0 et suivant des règles de génération de code présenté ci-dessus. Enfin, nous importons les différentes méthodes indispensables pour la synchronisation des nœuds. Un aperçu simplifié du code java produit par B2Visidia est présenté comme suit :

```
public class Spanning_Tree_RDV extends Algorithm {

/* déclaration des variables liées à l'affichages de la simulations */
static MessageType synchronization = new MessageType("synchronization",
  false, java.awt.Color.blue);
static MessageType labels = new MessageType("labels", true);
    public Collection getListTypes(){...}

/* la cette méthode implemente le pattern LC0 */
    public void init(){
        int synchro;
        boolean run=true;
        String neighbourValue;

        while(run){ synchro=synchronization();

/* échange des labels via l'envoi et la réception des messages */
    sendTo(synchro,new StringMessage((String) getProperty("label"),labels));
```

```
    neighbourValue=((StringMessage) receiveFrom(synchro)).data();

/* implémentation des gardes grd1 et grd2 de l'événement Rule */
    if ((neighbourValue.compareTo("A")==0)
        && (((String) getProperty("label")).compareTo("N")==0))

/* implémentation des actions act2 et act3 de l'événement Rule */
      { putProperty("label",new String("A"));
        setDoorState(new MarkedState(true),synchro); }}}

/* importation des méthodes nécessaires pour la synchronisation des noeuds */
    public int synchronization(){...}
    private int trySynchronize(){...}
    public void breakSynchro() {...}

    public Object clone() { return new Spanning_Tree_RDV(); }
}
```

6 Outil B2Visidia

B2Visidia est un outil graphique qui a été développé en langage Java. Il peut être exécuté soit localement, soit en ligne comme une applet Java. La version locale peut être exécutée sur différentes plateformes comme Linux, Windows, etc. La source de l'outil ainsi que sa documentation sont visibles depuis le site web `visidia.labri.fr`. Dans cette section, nous allons présenter l'interface ainsi que les différentes fonctionnalités offertes par l'outil B2Visdidia.

6.1 Interface graphique

L'interface graphique de B2Visidia comprend quatre parties. Dans la Figure.VI.8, nous présentons une vue globale de l'interface utilisateur.

1. La première partie englobe deux différentes zones. La première zone (celle qui est en haut) permet d'éditer une spécification *B événementiel* et d'afficher le code Java qui la correspond. Cette zone est structurée en plusieurs onglets. Chaque onglet correspond à une page. Perpétuellement, la première page est une page d'édition, tandis que les autres sont des pages d'affichage. Nous signalons que les lignes de la page d'édition sont automatiquement numérotées, ceci est dans le but de faciliter la localisation des erreurs. La deuxième zone (celle qui est en bas) est une console qui permet d'afficher les messages d'erreur et de compilation.
2. La deuxième zone contient une palette d'outils. Parmi ces outils, nous distinguons une bibliothèque des anciennes spécifications réalisées avec B2Visidia, un vérificateur de la syntaxe de la spécification, un générateur du code et un lanceur de la plateforme Visidia. Le vérificateur de la syntaxe peut détecter si la spécification formelle est syntaxiquement incorrect, et fournit à cet effet un message afin que l'utilisateur puisse la corriger.

3. La troisième zone comprend une liste d'outils. Parmi ces outils, nous citons l'importation des fichiers depuis Rodin, le filtrage des spécifications importées, la sauvegarde de la spécification filtrée ou éditée, et la création d'une nouvelle spécification.
4. La quatrième zone est un clavier visuel qui incorpore la plupart des symboles du langage *B événementiel*. Ce clavier permet d'aider l'utilisateur à éditer sa spécification.

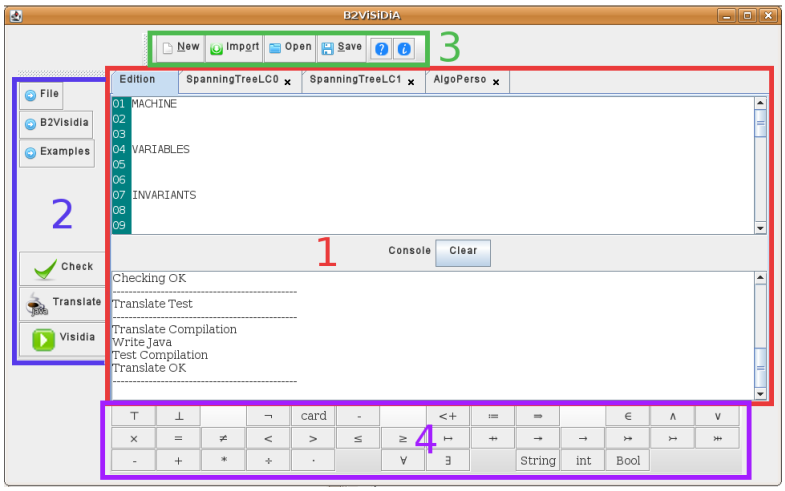


Figure VI.8 – Interface de l'outil B2Visidia

6.2 Assistant de génération du code

L'outil B2Visidia offre une assistance complète dans la génération du code Java. Il permet de guider l'utilisateur dans le processus de la génération de code. Afin de simplifier ce processus, nous avons choisi de le décomposer en trois étapes. Ces étapes ne peuvent démarrer qu'après une validation de la spécification de l'algorithme. La première étape (illustrée par la Figure.VI.9) consiste à fixer le type de synchronisation mis en vigueur par l'algorithme distribué (optionnel), à briser la symétrie, et à affirmer le choix d'utiliser certains paramètres. Briser la symétrie sert à différencier deux nœuds synchronisés ayant les mêmes étiquettes. Toutefois elle n'est valable que dans le cas d'un algorithme distribué adoptant une synchronisation de type LC0. Les autres paramètres concernent le choix d'utiliser des variables comme l'identité et les cardinalités des nœuds. La deuxième étape (illustrée par la Figure.VI.10) consiste à instancier les variables qui encodent les différentes étiquettes des nœuds. Une instantiation par défaut est définie si l'utilisateur saute cette étape. La troisième étape (illustrée par la Figure.VI.11) consiste à choisir l'emplacement du fichier généré et à donner une description informelle de l'algorithme spécifié. Il est important de noter qu'après cette étape l'outil B2Visidia génère deux différents fichiers : le code source (**.java*), et le fichier exécutable (**.class*) généré après la compilation du premier.

Figure VI.9 – Génération du code : étape n°1

Figure VI.10 – Génération du code : étape n°2

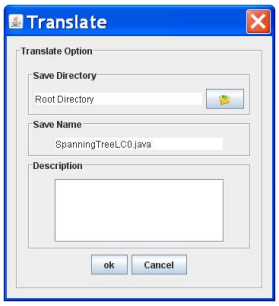


Figure VI.11 – Génération du code : étape n°3

7 Conclusion

Dans ce chapitre, nous avons présenté B2Visidia qui est un cadre général pour produire automatiquement un code Java à partir d'une spécification *B événementiel* d'un algorithme distribué. Nous avons décrit son fondement théorique et nous avons donné un aperçu sur son implémentation. A travers ce travail, nous avons montré que l'approche et l'outil B2Visidia peuvent être utilisés pour une large classe d'algorithmes distribués codés par les calculs locaux. De nombreux exemples ont été étudiés et rassemblés dans une bibliothèque de programmes d'algorithme distribués prouvés. Nous travaillons actuellement sur l'intégration de l'outil B2Visidia à la plate-forme Visidia.

Conclusion générale

Prouver la correction d'un algorithme distribué peut être un travail long, contenant de nombreux calculs, et pouvant être entaché d'erreurs ou d'omissions s'il est fait à la main. Par ailleurs, nous avons constaté que les travaux qui s'appuient sur des assistants automatiques de preuve pour vérifier et valider de tels algorithmes ne sont pas assez nombreux. En particulier, ceci concerne le système de calculs locaux qui nous intéresse dans cette thèse.

Dans ce cadre s'inscrivent nos travaux de recherches. Nous nous sommes concentrés essentiellement sur le problème de preuve de correction des algorithmes distribués codés par les calculs locaux. Dans le chapitre I de cette thèse, nous avons montré que ce modèle constitue une abstraction réaliste d'un système informatique réparti. Il comprend également les éléments et les outils essentiels qui permettent de raisonner et de comprendre d'une façon rigoureuse un système distribué. Nous avons montré que deux récents travaux de recherches ont proposé des solutions pour la preuve de correction des calculs locaux par des outils formels en occurrence Coq et *B événementiel* [14, 15].

Ainsi, l'approche fondamentale que nous avons choisie pour aborder ce problème se base sur l'approche *correct-par-construction* et utilise la méthode *B événementiel*. Cette méthode s'appuie principalement sur la technique de raffinement pour diffuser la complexité des preuves et valider l'intégration des invariants du système. Aussi, elle permet de traiter la spécification de différents niveaux d'abstraction. Cette méthode a fait l'objet du chapitre II.

Il est connu que la plupart des applications d'un système distribué sont centré autour d'un ensemble de sous-problèmes communs. Si nous pouvons résoudre ces sous-problèmes d'une manière satisfaisante, alors cela va nous permettre de maîtriser convenablement la conception du système. Il y a quelques exemples de sous-problèmes communs : élection d'un nœud, la synchronisation, la construction d'un arbre recouvrant. En particulier, nous avons traité le problème de calcul d'arbre recouvrant [65] pour proposer une première esquisse d'un patron de développement formelle. Par la suite, nous avons élargie le défi pour traiter d'autre classes d'algorithmes et nous avons ainsi proposé, dans le chapitre III, un patron plus générique. Ce patron peut être instancié et/ou réutilisé en partie pour spécifier un algorithme distribué correct par construction.

Toutefois, nous avons remarqué que notre patron est partiellement inopérant face à des problèmes de preuve de grandes tailles. Ces problèmes incorporent souvent plusieurs sous-problèmes complexes qui peuvent être traités séparément. Ainsi, dans le chapitre IV, nous avons présenté une approche pour spécifier des algorithmes distribués en combinant des composants déjà prouvés. Nous avons utilisé le raffinement et la décomposition pour surmonter la complexité de la preuve d'un algorithme distribué.

Une preuve de correction d'un algorithme distribué codé par les calculs locaux est assurée quand le mécanisme de synchronisation sous-jacent est lui aussi vérifié et prouvé. Dans cette optique, nous avons proposé, dans le chapitre V, une approche formelle pour développer des algorithmes de synchronisation. L'approche utilise la méthode *B événementiel* pour spécifier l'algorithme jusqu'à un niveau concret non-probabiliste. Par la suite, l'implémentation des probabilités est réalisée avec le model-checker probabiliste PRISM.

Enfin, dans le dernier chapitre de cette thèse nous avons proposé une approche et un outil appelés $B2Visidia$ pour implémenter une spécification formelle d'un algorithme codé par les calculs locaux. A travers $B2Visidia$ nous avons essayé de conforter un développement formel par la visualisation et la simulation. Nous générons automatiquement un code Java à partir d'une spécification *B événementiel* d'un algorithme distribué qui est exécuté par la suite sous la plate-forme Visidia.

Comme perspectives à nos travaux de thèse, nous pensons que trois principales idées peuvent être réalisées : La première idée consiste à développer un plugin Rodin de spécification formelle des algorithmes distribués codés par les calculs locaux. Ce plugin se base sur le patron développé dans le chapitre III. L'objectif étant de systématiser davantage la spécification et la preuve des algorithmes. Ainsi, des obligations de preuves destinées à vérifier les propriétés des différents niveaux du patron peuvent être définies.
La deuxième idée consiste à unifier les résultats de nos travaux de recherches avec celles élaborées dans le cadre du projet LOCO [14] pour proposer un langage commun servant de base à différents outils pour les calculs locaux. Le nouveau langage peut être utilisé pour générer une spécification Coq ou un code Java pour Visidia à partir d'une spécification B événementiel. Ce langage pourra avoir certaines ressemblances au niveau syntaxique avec le langage LIDIA [91].
La troisième idée consiste à étudier l'aspect probabiliste des algorithmes distribué modélisés par les calculs locaux. Pour réaliser ce but, deux voies de recherches sont possibles : La première permet de concevoir et d'analyser les algorithmes probabilistes en étudiant leurs complexités (temps, espace, communication). La deuxième voie consiste à prouver la correction de ces types d'algorithmes par des méthodes formelles. Cette voie a été peu explorée dans la littérature. Notamment, ceci s'applique au système des calculs locaux. Le travail le plus intéressant est celui de la thèse d'Allyx Fontaine (membre de notre équipe de recherche) qui vient de démarrer et qui vise à utiliser COQ pour prouver la correction des calculs locaux probabilistes. Jusqu'à lors, les résultats trouvés nous laissent penser que l'étude de l'aspect probabiliste des algorithmes distribué avec l'approche *correcte-par-*

construction est possible. Ceci pourra être réalisé en introduisant un choix probabiliste quantitatif [92] pour chaque distribution probabiliste connue à chaque choix particulier.

Annexe 1

Spécification formelle de l'algorithme de synchronisation ATH avec $PRISM$

```
// Algorithme de synchronisation de type Handshake.
// Cette spécification est implémentée sur un graphe
// de 3 noeuds numérotés 0, 1 et 2 et qui sont deux à deux reliés.

mdp //processus de décision markovien

const NV= 2; // nombre de noeuds voisins

module noeud0

//---------------Déclaration des variables------------

// Compteur
compteur0 : [0..2];

// États du noeud n°0
```

Annexe 1

```
etat0 : [0..4] init 0;
// 0  le noeud (inactive) fait son choix
// 1  le noeud attend la reception des messages
// 2  le noeud est entrain de recevoir les messages
// 3  le noeud est synchronisé
// 4  le noeud est non synchronisé


// état de la porte01: l'arête qui relie le noeud 0 et le noeud 1
porte01 : [0..2] init 2;
// 0: le noeud 0 a attribué 0 à son voisin 1
// 1: le noeud 0 a attribué 1 à son voisin 1
// 2: le noeud 0 n'a rien attribué à son voisin 1


// état de la porte02: l'arête qui relie le noeud 0 et le noeud 2
porte02 : [0..2] init 2;
// 0: le noeud 0 a attribué 0 à son voisin 2
// 1: le noeud 0 a attribué 1 à son voisin 2
// 2: le noeud 0 n'a rien attribué à son voisin 2



recevoir01:[0..1];
//0: Le noeud 0 n'a rien reçu de son voisin 1
//1: Le noeud 0 a reçu un message de son voisin 1
```

```
recevoir02:[0..1];
//0: le noeud 0 n'a rien reçu de son voisin 2
//1: le noeud 0 a reçu un message de son voisin 2


envoyer0 : [0..1];
// 0: Le noeud 0 n'a rien envoyé à ses voisins
// 1: Le noeud 0 a envoyé un message à un de ses voisins

//---------------Fin de déclaration des variables-----




// faire un choix: le noeud 0 est inactive alors il peut choisir soit
    // le noeud 2 avec une probabilité qui est égale à 0.5, soit
    // le noeud 1 avec la même probabilité (0.5).
[] (etat0=0) ->
        0.5:(etat0'=1) & (porte01'=0) & (porte02'=1) +
        0.5:(porte01'=1) & (porte02'=0) &(etat0'=1);



// envoyer mon choix: Le noeud 0 envoi son choix et met à jour son état
```

```
  [] (etat0=1) & (envoyer0=0) -> (envoyer0'=1) & (etat0'=2);



//recevoir les preferences: Le noeud 0 reçoit les messages des ses voisins
[receive] (envoyer1=1) & (etat0=2)  & (recevoir01=0) & (compteur0<NV)
            -> (compteur0'= compteur0 +1) & (recevoir01'=1);

[receive] (envoyer2=1) & (etat0=2) & (recevoir02=0)& (compteur0<NV)
            -> (compteur0'= compteur0 +1) & (recevoir02'=1);



// test les preferences
[test0] (compteur0=NV) & (porte02=1) & (porte20=1) & (etat0=2)
            -> (etat0'=3) ;
[test0] (compteur0=NV) & (porte01=1) & (porte10=1) & (etat0=2)
            -> (etat0'=3) ;
[test0] (compteur0=NV) & (porte01=0 | porte10=0) & (porte02=0 | porte20=0)
            & (etat0=2) -> (etat0'=4) ;

// done + add loop
[done] (etat0=3) -> (etat0'= etat0);

// add loop for processes who are inactive
[done] (etat0=4) & (etat1=3 | etat2=3) -> (etat0'=etat0);
```

```
//reset
[reset] (etat0=4) & (etat1=4 | etat2=4) ->
            (etat0'=0)&(envoyer0'=0)&(recevoir02'=0)&(recevoir01'=0)
            &(porte02'=2)&(porte01'=2)&(compteur0'=0);

endmodule




// construire les autres noeuds par copie
    module
    noeud1=noeud0[etat0=etat1, etat1=etat0, envoyer0=envoyer1,
        recevoir02=recvevoir12, porte02=porte12, porte01=porte10,
        compteur0=compteur1, recevoir01=recevoir10, test0=test1,
        porte10=porte01, porte20=porte21]
    endmodule




    module
    noeud2=noeud0[etat0=etat2, etat2=etat0, envoyer0=envoyer2,
        recevoir02=recvevoir20, porte02=porte20, porte01=porte21,
        compteur0=compteur2, recevoir01=recevoir21, test0=test,
```

```
        porte10=porte12, porte20=porte02]
    endmodule



//Ces formules servent à dégager des mesures utiles à la preuve.
    formula leaders =
        (etat0=3?1:0) + (etat1=3?1:0) + (etat2=3?1:0);


    label "synchro" =
        etat0=3 & etat1=3 | etat0=3 & etat2=3 | etat1=3 & etat2=3;
```

Annexe 2

Spécification formelle de l'algorithme LC1

Le contexte Synchro

Le contexte *synchro* est une extension du contexte *graphe*. Il permet de spécifier l'ensemble de toutes les combinaisons de synchronisation qui peuvent résulter d'une exécution de l'algorithme $LC1$. Formellement, la variable $all\_synchro$ est définie par l'ensemble des boules disjointes de rayon 1 qui peuvent cœxister dans le graphe g. L'axiome $axm2$, définit une boule par un nœud centre associé à tous ses voisins (feuilles). Deux boules sont disjointes si, et seulement si, les feuilles et les nœuds centres des deux boules sont disjoints ($axm3$ et $axm4$). Nous ajoutons, l'ensemble vide à $all\_synchro$ pour inclure le cas où les nœuds échouent de produire une seule synchronisation ($axm5$). Enfin, nous affirmons que toutes les solutions qui satisfont les axiomes ci-dessus détaillés, doivent nécessairement appartenir à $all\_synchro$ ($axm6$).

$axm1 : all\_synchro \subseteq ND \leftrightarrow \mathbb{P}_1(ND)$
$axm2 : \forall R \cdot R \in all\_synchro \land R \neq \varnothing \Rightarrow (\forall x, y \cdot x \mapsto y \in R \Rightarrow g[\{x\}] = y)$
$axm3 : \forall R \cdot R \in all\_synchro \land R \neq \varnothing \Rightarrow (\forall c1, s \cdot c1 \in dom(R) \land s \in ran(R) \Rightarrow c1 \notin s)$
$axm4 : \forall R \cdot R \in all\_synchro \land R \neq \varnothing \Rightarrow (\forall c1, c2 \cdot c1 \in dom(R) \land c2 \in dom(R) \land c1 \neq c2$
$\quad \Rightarrow g[\{c1\}] \cap g[\{c2\}] = \varnothing)$
$axm5 : \varnothing \in all\_synchro$
$axm6 : \forall R \cdot R \in ND \leftrightarrow \mathbb{P}_1(ND) \land (\forall c1, c2 \cdot c1 \in dom(R) \land c2 \in dom(R) \land c1 \neq c2$
$\quad \Rightarrow g[\{c1\}] \cap g[\{c2\}] = \varnothing) \land R \neq \varnothing \land (\forall c \cdot c \in dom(R) \Rightarrow c \mapsto g[\{c\}] \in R) \land$
$\quad (\forall c3, s \cdot c3 \in dom(R) \land s \in ran(R) \Rightarrow c3 \notin s) \Rightarrow R \in all\_synchro$

La machine SpecMachine

L'objectif de cette machine est de produire, dans une seule étape, une synchronisation de type LC1. Toutefois, la plupart des variables définies dans la spécification de l'algorithme handshake seront aussi redéfinies dans cette spécification. En effet, la variable

actual_state exprime l'état actuel du graphe, autrement dit, elle englobe les synchronisations LC1 qui existaient dans le graphe. Formellement, *actual_state* est défini comme un élément de l'ensemble *all_synchro*. En plus de l'événement d'initialisation, cette machine comprend deux autres événements en occurrence *synchronize* et $free\_nodes$. Le premier événement modélise la réalisation d'une synchronisation de type LC1. Les gardes de cet événement sont définies comme suit : les nœuds voisins du nœud x ne sont pas synchronisés, c.-à-d. ils ne sont ni centres ni feuilles d'autres boules synchronisées ($grd1$ et $grd4$). Également, le nœud x, qui est supposé être le centre de la boule, n'est pas synchronisé ($grd3$ et $grd2$). Nous prouvons ensuite que l'ajout de la boule de centre x à *actual_state* n'altère pas la définition de cette dernière et désormais *actual_state* représente un nouvel état correct du graphe. Dans la clause *action* de cet événement, nous mettons à jour la variable *actual_state* ($act2$).

EVENT synchronise
ANY x
WHERE
$grd1 : (\forall c \cdot c \in dom(actual\_state) \Rightarrow g[\{x\}] \cap g[\{c\}] = \varnothing)$
$grd2 : (\forall s \cdot s \in ran(actual\_state) \Rightarrow x \notin s)$
$grd3 : x \notin dom(actual\_state)$
$grd4 : \forall c \cdot c \in dom(actual\_state) \Rightarrow c \notin g[\{x\}]$
$grd5 : result = \varnothing$
$thm : (actual\_state \cup \{x \mapsto g[\{x\}]\}) \in all\_synchro$
THEN
$act1 : result := \{x \mapsto g[\{x\}]\}$
$act2 : actual\_state := actual\_state \cup \{x \mapsto g[\{x\}]\}$
END

EVENT free_nodes
ANY x
WHERE
$grd1 : x \in dom(actual\_state)$
$grd2 : result = \varnothing$
$thm : (actual\_state \setminus \{x \mapsto g[\{x\}]\}) \in all\_synchro$
THEN
$act1 : actual\_state := actual\_state \setminus \{x \mapsto g[\{x\}]\}$
END

La machine ChoiceMachine

Dans cette machine nous spécifions l'opération de choix non-déterministe d'un entier positif quelconque faite par les nœuds du graphe. Cette opération précède l'étape de la synchronisation. Formellement, nous définissons la variable *choice* comme un ensemble

assignant à chaque nœud son entier naturel choisi. Les invariants présentés ci-dessous décrivent formellement la variable *choice* :

(inv1) L'ensemble *choice* est spécifié par une fonction partielle assemblant d'une part les nœuds ND et d'autre part l'ensemble $\{1 \,..\, C\}$, où C est une constante strictement supérieur à 1. Cela signifie que les nœuds qui n'ont pas fait leurs choix ne figurent pas dans l'ensemble *choice*.

(inv2 & inv3) Les nœuds synchronisés sont les nœuds qui ont fait déjà leurs choix.

(inv4) Si un nœud x est un centre d'une boule synchronisée, alors le choix de x est strictement supérieur au choix de chaque voisin de x.

$$
\begin{array}{l}
inv1 : choice \in ND \pfun 1\,..\,const \\
inv2 : dom(actual\_state) \subseteq dom(choice) \\
inv3 : \forall a \cdot a \in ran(actual\_state) \limp a \subseteq dom(choice) \\
inv4 : \forall x, y \cdot x \mapsto y \in actual\_state \limp (\forall a \cdot a \in y \limp choice(x) > choice(a))
\end{array}
$$

L'événement *synchronize* est raffiné ; son garde est renforcé par trois nouvelles conditions. Les gardes $grd6$ et $grd7$ vérifient si le nœud x et l'ensemble de tous ses voisins ont auparavant faits leurs choix (grd6 : $x \in dom(choice)$ et grd7 : $g[\{x\}] \subseteq dom(choice)$). La garde $grd8$ examine le choix du nœud x et vérifie s'il est bien supérieur aux choix des ses voisins (grd8 : $\forall y \cdot y \in g[\{x\}] \limp choice(x) > choice(y)$). Quant à l'événement $free\_nodes$, il est raffiné par l'ajout d'une nouvelle substitution généralisée dans la clause action. Cette substitution permet de supprimer le choix d'un nœud synchronisé ($choice := (\{x\}) \domsub choice$). Les deux événements *make_choice* et *cannot_synchronize*, spécifiés dans la Section 3.2.3 du chapitre V, sont aussi ajoutés à ce raffinement. L'événement *make_choice* permet à un nœud de choisir d'une façon aléatoire un entier positif entre 1 et la constante C. L'événement *cannot_synchronize* modélise une tentative de synchronisation manquée. Cet événement suppose que tous les nœuds de la boule ont fait leurs choix, et le choix du nœud centre n'est pas forcément supérieur à ceux de ses voisins. Dans un tel cas, le choix du nœud est annulé, et ainsi une nouvelle tentative de synchronisation est sensée démarrer. Les événements *make_choice* et *cannot_synchronize* sont ci-dessous présentés :

EVENT make_choice
ANY x
WHERE
$grd1 : result = \varnothing$
$grd2 : x \notin dom(actual\_state)$
$grd3 : x \notin dom(choice)$
$grd4 : \forall p \cdot p \in ran(actual\_state) \limp x \notin p$
THEN
$$act1 : choice :\mid \exists v \cdot \left(\begin{array}{l} v \in 1\,..\,const \land \\ choice' = choice \cup \{x \mapsto v\} \end{array} \right)$$
END

EVENT cannot_synchronise
ANY x
WHERE
$grd1 : result = \varnothing$
$grd2 : x \in dom(choice) \wedge g[\{x\}] \subseteq dom(choice)$
$grd3 : x \notin dom(actual\_state)$
$grd4 : \neg(\forall y \cdot y \in g[\{x\}] \Rightarrow choice(x) > choice(y))$
$grd5 : \forall a \cdot a \in ran(actual\_state) \Rightarrow x \notin a$
THEN
$act1 : choice := \{x\} \lhd choice$
END

La machine MessageMachine

L'algorithme de handshake, présenté ci-dessus, est un algorithme asynchrone qui n'exige aucun ordre précis dans l'envoi et la réception des messages. En d'autres termes, il n'y a pas des rounds qui rythment l'exécution de l'algorithme, par conséquent seul le contexte local d'un nœud qui organise l'envoi des message. A l'opposé, LC1 est un algorithme synchrone, sa synchronisation est dépend directement de l'envoi et de la réception des messages. Plus précisément, la réception des messages permet d'ordonner l'exécution des nœuds du graphe. En effet, tous les nœuds envoient les messages et restent bloqués jusqu'à la réception de tous les messages de leurs voisins (ceci est équivalent à un round).
Formellement, nous déclarons une variable, appelée *receive*, pour associer à chaque nœud l'ensemble de messages reçus. Par exemple si x , y et z sont trois nœuds et $x \mapsto \{y, z\}$, alors cela signifie que x a reçu deux messages de ses deux voisins y et z ($inv1$). L'invariant ($inv2$) précise qu'un nœud ne peut recevoir des messages que de ses voisins. L'invariant ($inv3$) est un invariant de collage, il exige que la réception d'un message par un nœud x de son voisins de y ne peut se faire que si le nœud y a déjà fait son choix. L'invariant ($inv4$) montre qu'une composition valide est formée par des nœuds qui ont tous reçus les messages de leurs voisins. Cet invariant instaure le concept de round.

$inv1 : receive \in ND \nrightarrow \mathbb{P}(ND)$
$inv2 : \forall x \cdot x \in dom(receive) \Rightarrow receive(x) \subseteq g[\{x\}]$
$inv3 : \forall a \cdot a \in ran(receive) \Rightarrow a \subseteq dom(choice)$
$inv4 : \forall a, b \cdot a \mapsto b \in actual\_state \Rightarrow a \mapsto b \in receive \wedge (\forall z \cdot z \in b \Rightarrow z \mapsto g[\{z\}] \in receive)$

L'événement *synchronize* est raffiné. Il peut être déclenché que si le nœud x a reçu tous les messages de ses voisins ($grd13 : x \mapsto g[\{x\}] \in receive$). De même, les voisins de x doivent à leurs tours recevoir les messages de tous leurs voisins ($grd14 : \forall x1 \cdot x1 \in g[\{x\}] \Rightarrow x1 \mapsto g[\{x1\}] \in receive$). Un nouveau événement est ajouté *receive_message*, il permet d'alimenter l'ensemble *receive*. Cet événement est déclenché si le nœud x à

au moins reçu un message. Autrement, un autre événement est spécifié pour initialiser $receive(x)$. L'ensemble $receive$ est réinitialisé avec l'événement $free\_nodes$.

EVENT receive_message
ANY x, y
WHERE
$grd1 : x \mapsto y \in g$
$grd2 : y \in dom(choice) \wedge x \in dom(receive)$
$grd3 : y \notin receive(x)$
$grd4 : x \notin dom(actual\_state)$
$grd4 : \forall s \cdot s \in ran(actual\_state) \Rightarrow x \notin s$
THEN
$act1 : receive(x) := receive(x) \cup \{y\}$
END

Références bibliographiques

[1] Gerard Tel. *Introduction to distributed algorithms.* Cambridge University Press (2000). 1

[2] Sukumar Ghosh. *Distributed systems : an algorithmic approach.* Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC (2006). 1

[3] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002). 1, 2

[4] Tanja Ernestina Jozefina Vos. *UNITY in Diversity, a stratified approach to the verification of distributed algorithms.* Thèse de Doctorat, Utrecht University january (2000). 1

[5] Igor Litovsky, Yves Métivier, and Éric Sopena. *Graph relabelling systems and distributed algorithms* pages 1–56. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999). 1, 8

[6] Orna Grumberg Edmund M. Clarke and Doron A. Peled. *Model checking.* MIT Press (1999). 2

[7] Gerard J. Holzmann. *The model checker spin.* IEEE Trans. Softw. Eng. **23**, 279–295 May (1997). 2

[8] Hubert Garavel and Laurent Mounier. *Specification and verification of various distributed leader election algorithms for unidirectional ring networks.* Sci. Comput. Program. **29**, 171–197 July (1997). 2

[9] Kenneth J. Turner. *Using Formal Description Techniques : An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition (1993). 2

[10] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/-HOL : a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg (2002). 3

[11] Ching-Tsun Chou. Mechanical verification of distributed algorithms in higher-order logic. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 158–176, London, UK (1994). Springer-Verlag. 3

[12] The Coq Development Team. *Reference Manual*. IRIA (Institut de Recherche en Informatique et en Automatique), (2010). 3, 20, 61

[13] Sam Owre, Natarajan Shankar, John Rushby, and David W.J. Stringer-Calvert. Pvs language reference. Miscellaneous Version 2.4 SRI International, Computer Science Laboratory, Menlo Park CA 94025 November (2001). 3

[14] Pierre Castéran and Vincent Filou. Tâches, types et tactiques pour les systèmes de calculs locaux. In *Journées Francophones des Langages Applicatifs* (2010). 3, 20, 61, 125, 126

[15] Dominique Cansell and Dominique Méry. The event-b modelling method : Concepts and case studies. In Dines Bjoerner and Martin C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science. An EATCS Series, pages 47–152. Springer Berlin Heidelberg (2008). 3, 38, 46, 65, 125

[16] Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems : Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA USA (2005). 7

[17] Jérémie Chalopin. *Algorithmique Distribuée, Calculs Locaux et Homomorphismes de Graphes*. Thèse de Doctorat, Université Bordeaux 1, Cours de la Libération 33405 Talence novembre (2006). 8

[18] Nancy A. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 1–28, New York, NY, USA (1989). ACM. 12

[19] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 82–93, New York, NY, USA (1980). ACM. 12

[20] Christian Lavault. *Evaluation des algorithmes distribués – Analyse, complexité, méthodes*. collection Informatique. Lavoisier 2000-2010 (1995). 12

[21] Yves Métivier, Nasser Saheb, and Akka Zemmari. *Randomized local elections*. Inf. Process. Lett. **82**, 313–320 June (2002). 12, 14, 15

[22] Yves Métivier, Nasser Saheb, and Akka Zemmari. *Analysis of a randomized rendezvous algorithm.* Inf. Comput. **184**(1), 109–128 (2003). 12, 13

[23] Abdelaziz ElHibaoui, Yves Métivier, John-Michael Robson, Nasser Saheb-Djahromi, and Akka Zemmari. *Analysis of a randomized dynamic timetable handshake algorithm.* Pure Mathematics and Applications (PuMA) (2008). 13

[24] Michel Bauderon, Stefan Gruner, and Mohamed Mosbah. *A new tool for the simulation and visualization of distributed algorithms.* MFI'01 **1**, 165–177 mai (2001). Toulouse, France. 17, 94

[25] MMichel Bauderon, Stefan Gruner, Yves Mtivier, Mohamed Mosbah, and Afif Sellami. *Visualization of distributed algorithms based on labeled rewriting systems.* Second International Workshop on Graph Transformation and Visual Modeling Techniques, ENTCS **50**(3), 229–239 juillet (2001). Crete, Greece. 17, 94

[26] Michel Bauderon and Mohamed Mosbah. *A unified framework for designing, implementing and visualizing distributed algorithms.* Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation) **72**(3), 13–24 (2003). 17, 94

[27] Bilel Derbel and Mohamed Mosbah. Distributing the execution of a distributed algorithm over a network. In *Proceedings of the Seventh International Conference on Information Visualization*, pages 485–, Washington, DC, USA (2003). IEEE Computer Society. 18

[28] Yoram Moses, Zvi Polunsky, Ayellet Tal, and Leonid Ulitsky. Algorithm visualization for distributed environments. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 71–78, Washington, DC, USA (1998). IEEE Computer Society. 18

[29] John T. Stasko. The parade environment for visualizing parallel program executions : A progress report. Technical report Georgia Institute of Technology, Atlanta, GA (1995). 18

[30] Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. *Lydian : An extensible educational animation environment for distributed algorithms.* J. Educ. Resour. Comput. **6** June (2006). 18

[31] Jérémie Chalopin, Yves Métivier, and Wieslaw Zielonka. *Local computations in graphs : The case of cellular edge local computations.* Fundam. Inf. **74**, 85–114 October (2006). 20

[32] Igor Litovsky, Yves Métivier, and Eric Sopena. *Different local controls for graph relabeling systems.* Mathematical Systems Theory **28**(1), 41–65 (1995). 20

[33] Igor Litovsky and Yves Métivier. *Computing with graph rewriting systems with priorities*. Theoretical Computer Science **115**, 191–224 (1993). 20

[34] Igor Litovsky, Yves Métivier, and Wieslaw Zielonka. The power and the limitations of local computations on graphs. In *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '92, pages 333–345, London, UK (1993). Springer-Verlag. 20

[35] David Gries and Fred B. Schneider. *A logical approach to discrete math*. Springer-Verlag New York, Inc., New York, NY, USA (1993). 23

[36] Dominique Cansell and Dominique Méry. Tutorial on the event-based b method. Technical report INRIA a CCSD electronic archive server based on P.A.O.L [http ://hal.inria.fr/oai/oai.php] (France) (2006). 24, 33, 44

[37] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press (2010). 24, 37, 38, 42

[38] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA (1996). 24, 28

[39] ClearSy System Engineering. *Atelier B, Manuel Utilisateur*. 13857 AIX EN PROVENCE CEDEX 3 FRANCE, (2001). 24

[40] Laurent Voisin. Public versions of basic tools and platform. Public Document Project IST-511599 ETH Zurich 29 October (2007). 24

[41] Christophe Metayer, Jean-Raymond Abrial, and Laurent Voisin. Event-b language. Rodin deliverable 3.2 ClearSy and ETH Zürich May (2005). 27

[42] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall (1976). 28

[43] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990). 28

[44] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus : A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition (1998). 28

[45] Frédéric Gervais, Marc Frappier, and Régine Laleau. Vous avez dit raffinement ? Technical Report CEDRIC-05-829 CEDRIC laboratory, CNAM-Paris, France Mars (2005). 28

[46] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In *ZB*, pages 457–476 (2003). 33, 38, 44

[47] Thai Son Hoang, Hironobu Kuruma, David Basin, and Jean-Raymond Abrial. *Developing topology discovery in event-b.* Sci. Comput. Program. **74**, 879–899 November (2009). 38

[48] Dominique Cansell and Dominique Méry. *Formal and incremental construction of distributed algorithms : On the distributed reference counting algorithm.* Theor. Comput. Sci. **364**(3), 318–337 (2006). 38

[49] Luc Moreau and Jean Duprat. *A construction of distributed reference counting.* Acta Inf. **37**, 563–595 May (2001). 38

[50] Dominique Cansell and Dominique Méry. Tutorial on the event-based b method : Concepts and case studies. In Dines Bjoerner and Martin Henson, editors, *Logics of Formal Software Specification Languages - LFSL'2004*, The High Tatras, Slovakia (2004). Colloque avec actes et comité de lecture. nationale. 38

[51] Projet ANR-RIMEL. Developpement d'algorithmes répartis : Livrable 1. Technical report MOSEL - INRIA Lorraine - LORIA - INRIA - CNRS : UMR7503 - Université Henri Poincaré - Nancy I - Université Nancy II - Institut National Polytechnique de Lorraine février (2008). 38, 46

[52] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. *A mechanically proved and incremental development of ieee 1394 tree identify protocol.* Formal Aspects of Computing **14**(3), 215–227 (2003). 38

[53] Stefan Hallerstede and Thai Son Hoang. Qualitative probabilistic modelling in event-b. In *Proceedings of the 6th international conference on Integrated formal methods*, IFM'07, pages 293–312, Berlin, Heidelberg (2007). Springer-Verlag. 39

[54] Vincent Couturier. Des patterns pour la coopération de systèmes d'information : application à l'architecture coopérative acsis. In *Journée du travail bi-thématique du GDR-PRC* decembre (2001). 41

[55] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language : towns, buildings, construction.* Center for Environmental Structure series. Oxford University Press (1977). 42

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995). 42

[57] Rafael Marcano Kamenoff, Nicole Lévy, and Francisca Losavio. Spécification et spécialisation de patterns en UML et B. In *Langages et Modèles à Objets - LMO'2000*, page 16 p, Montreal, Canada (2000). Editions Hermes. Colloque avec actes et comité de lecture. nationale. 42

[58] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Une démarche outillée pour spécifier formellement des patrons de conception réutilisables. In *Workshop Objets, Composants et Modèles dans l'ingénierie des SI (OCM-SI)*, pages 5–9, Nancy, France 3 June (2003). INFORSID. 42

[59] Jean-Raymond Abrial and Stefan Hallerstede. *Refinement, decomposition, and instantiation of discrete models : Application to event-b.* Fundam. Inf. **77**, 1–28 January (2007). 42, 60, 61

[60] Dominique Cansell and Dominique Méry. *Incremental parametric development of greedy algorithms.* Electron. Notes Theor. Comput. Sci. **185**, 47–62 July (2007). 42

[61] Thierry Lecomte, Dominique Méry, and Dominique Cansell. *Patrons de conception prouvés.* Génie Logiciel - Magazine de l'ingéniérie du logiciel et des systèmes **81**, 14–18 (2007). 42

[62] Jean-Raymond Abrial and Thai Son Hoang. Using design patterns in formal methods : An event-b approach. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 1–2, Berlin, Heidelberg (2008). Springer-Verlag. 42, 52

[63] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-b patterns and their tool support. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 210–219, Washington, DC, USA (2009). IEEE Computer Society. 42

[64] Joris Rehm. *Gestion du temps par le raffinement.* Thèse de Doctorat, Université Henri Poincaré, Nancy 1 France decembre (2009). 43

[65] Mohamed Tounsi, Ahmed Hadj Kacem, Mohamed Mosbah, and Dominique Méry. A refinement approach for proving distributed algorithms : Examples of spanning tree problems. In *Integration of Model-based Formal Methods and Tools - IM_FMT'2009 - in IFM'2009*, Düsseldorf Allemagne 02 (2009). 46, 125

[66] Martín Abadi and Leslie Lamport. *Conjoining specifications.* ACM Trans. Program. Lang. Syst. **17**, 507–535 May (1995). 60

[67] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332 (1983). 60

[68] Jean-Raymond Abrial. Event model decomposition. Technical report ETH Zurich April (2009). 60, 61

[69] Kriangsak Damchoom and Michael Butler. *Applying event and machine decomposition to a flash-based filestore in event-b.* Formal Methods : Foundations and Applications **5902**, 134–152 (2009). 60

[70] Elisabeth Ball and Michael Butler. Using decomposition to model multi-agent interaction protocols in event-b. In *FM'06 Doctoral Symposium*. Springer (2006). 60

[71] Thai Son Hoang and Jean-Raymond Abrial. Event-b decomposition for parallel programs. In *Proceedings of the Second international conference on Abstract State Machines, Alloy, B and Z*, ABZ'10, pages 319–333, Berlin, Heidelberg (2010). Springer-Verlag. 60

[72] Jérémie Chalopin, Emmanuel Godard, and Yves Métivier. Local terminations and distributed computability in anonymous networks. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 47–62, Berlin, Heidelberg (2008). Springer-Verlag. 62

[73] Tounsi Mohamed, Mosbah Mohamed, and Mery Dominique. *Proving distributed algorithms by combining refinement and local computations*. Automated Verification of Critical Systems **35** (2010). 67

[74] Dominique Méry, Mohamed Mosbah, and Mohamed Tounsi. Refinement-based verification of local synchronization algorithms. , **6664**, pages 338–352, Limerick Irlande 06 (2011). Springer Berlin - Heidelberg. 79

[75] Yves Métivier, Nasser Saheb, and Akka Zemmari. *Analysis of a randomized rendezvous algorithm*. Inf. Comput. **184**(1), 109–128 (2003). 80, 93, 94

[76] Victor P. Nelson. *Fault-tolerant computing : Fundamental concepts*. Computer **23**, 19–25 July (1990). 98

[77] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In *Proceedings of the 9h international conference on Stabilization, safety, and security of distributed systems*, SSS'07, pages 189–203, Berlin, Heidelberg (2007). Springer-Verlag. 98

[78] Youngsun Han, Shinyoung Kim, Hokwon Kim, Seok Joong Hwang, and Seon Wook Kim. Code generation and optimization for java-to-c compilers. In *Proceedings of the 2006 international conference on Emerging Directions in Embedded and Ubiquitous Computing*, EUC'06, pages 785–794, Berlin, Heidelberg (2006). Springer-Verlag. 104

[79] Aristos Stavrou and George A. Papadopoulos. Automatic generation of executable code from software architecture models. In Chris Barry, Michael Lang, Wita Wojtkowski, Kieran Conboy, and Gregory Wojtkowski, editors, *Information Systems Development*, pages 1047–1058. Springer US (2009). 104

[80] Pohl Christopher, Paiz Carlos, and Porrmann Mario. *vmagic-automatic code generation for vhdl.* International Journal of Reconfigurable Computing **2009** (2009). 104

[81] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '09, pages 237–250, Berlin, Heidelberg (2009). Springer-Verlag. 104

[82] Didier Bert, Sylvain Boulmé, Marie-Laure Potet, Antoine Requet, and Laurent Voisin. Adaptable translator of b specifications to embedded c programs. In *FME 2003 : Formal Methods*, Lecture Notes in Computer Science, pages 94–113. Springer Berlin - Heidelberg (2003). 104

[83] Andrew Edmunds and Michael Butler. *Linking event-b and concurrent object-oriented programs.* Electron. Notes Theor. Comput. Sci. **214**, 159–182 June (2008). 104

[84] Andrew Edmunds and Michael Butler. *Tool support for event-b code generation.* Workshop on Tool Building in Formal Methods February (2010). 105

[85] Steve Wright. Automatic generation of c from event-b. In *Workshop on Integration of Model-based Formal Methods and Tools.* http ://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/im_fmt2009_proceedings.html February (2009). 105

[86] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA (2011). ACM. 105

[87] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. *The software model checker blast : Applications to software engineering.* Int. J. Softw. Tools Technol. Transf. **9**, 505–525 October (2007). 105

[88] EB2ALL. Automatic code generation from event-b to many programming languages. http ://eb2all.loria.fr, (2011). 105

[89] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom manual. Technical report INRIA CNRS (2008). 107

[90] Aho Alfred and Jeffrey Ullman Monica Lam, Ravi Sethi. *Compilateurs : principes, techniques et outils : Avec plus de 200 exercices.* Pearson Education novembre (2007). 116

[91] Mohamed Mosbah and Rodrigue Ossamy. A programming language for local computations in graphs : Computational completeness. In *Proceedings of the Fifth Mexican International Conference in Computer Science*, ENC '04, pages 12–19, Washington, DC, USA (2004). IEEE Computer Society. 126

[92] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Towards probabilistic modelling in `Event-B`. , **6396**, pages 275–289, Nancy France 10 (2010). INRIA Nancy Grand Est, Springer Berlin / Heidelberg. 127