Preuves d'algorithmes distribués par composition et raffinement.

Maha Bousabbah

THÈSE EN COTUTELLE PRESENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ET DE L'UNIVERSITÉ DE SFAX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE
ÉCOLE DOCTORALE DES SCIENCES ÉCONOMIQUES, GESTION ET INFORMATIQUE, FACULTÉ DES SCIENCES ÉCONOMIQUES ET DE GESTION

SPATIALITÉ : INFORMATIQUE

Par Maha BOUSSABBEH

Preuves d'Algorithmes Distribués par Composition et Raffinement

---

Sous la direction de : Mohamed MOSBAH et Ahmed HADJ KACEM

Soutenue le 08/12/2017

Membres du jury :

M. Khalil DRIRA	Directeur de Recherche, LAAS-CNRS	Examinateur
M. Dominique MÉRY	Professeur, Université de Lorraine	Rapporteur
Mme. Leila JEMNI BEN AYED	Maître de Conférences, Université de Manouba	Rapporteure
M. Akka ZEMMARI	Maître de Conférences, Université de Bordeaux	Examinateur
Mme. Imen JEMILI	Maître de Conférences, Université de Carthage	Examinatrice
M. Ahmed HADJ KACEM	Professeur, Université de Sfax	Directeur
M. Mohamed Mosbah	Professeur, Bordeaux, INP	Directeur
M. Mohamed TOUNSI	Maître Assistant, Université de Sfax	Co-encadrant

Résumé Dans cette thèse, nous présentons des approches formelles permettant de simplifier la modélisation et la preuve du calcul distribué. Un système distribué est défini par une collection d'entités de calcul autonomes, qui communiquent ensemble pour accomplir une tâche commune. Chaque entité exécute localement son calcul et ne peut interagir qu'avec ses voisins. Le développement et la preuve du calcul distribué est un défi qui nécessite l'utilisation de méthodes et outils avancés. Dans nos travaux de thèse, nous étudions quelques problèmes fondamentaux du distribués, en utilisant Event-B, et nous proposons des schémas de preuve basés sur une approche "correct-par-construction". Nous considérons un système distribué défini par réseau fiable, de processus anonymes et avec un modèle de communication basé sur l'échange de messages. Dans certains cas, nous faisons abstraction du modèle de communications en utilisant le modèle des calculs locaux. Nous nous focalisons d'abord sur le problème de détection de terminaison du calcul distribué. Nous proposons un patron formel permettant de transformer des algorithmes "avec détection de terminaison locale" en des algorithmes "avec détection de terminaison globale". Ensuite, nous explicitons les preuves de correction d'un algorithme d'énumération. Nous proposons un développement formel qui servirait de point de départ aux calculs qui nécessitent l'hypothèse d'identification unique des processus. Enfin, nous étudions le problème du snapshot et du calcul d'état global. Nous proposons une solution basée sur une composition d'algorithmes existants.

Mots-Clés Algorithmes Distribués, Calculs Locaux, Détection de Terminaison, énumération, Snapshots, Composition, Raffinement, Méthodes Formelles, Event-B.

Abstract In this work, we propose formal approaches for modeling and proving distributed algorithms. Such computations are designed to run on interconnected autonomous computing entities for achieving a common task : each entity executes asynchronously the same code and interacts locally with its immediate neighbors. Correctness of distributed algorithms is a difficult task and requires advancing methods and tools. In this thesis, we focus on some basic problems of distributed computing, and we propose Event-B solutions based on the "correct-by-construction" approach. We consider reliable systems. We also assume that the network is anonymous and processes communicate with asynchronous messages. In some cases, we refer to local computations model to provide an abstraction of the distributed computations. We propose a formal framework enhancing the termination detection property of distributed algorithms. By relying on refinement and composition, we show that an algorithm specified with "local termination detection", can be reused in order to compute the same algorithm with "global termination detection". We then focus on the enumeration problem : we start with an abstract initial specification of the problem, and we enrich it gradually by a progressive and incremental refinement. The computed result constitutes basic initial steps of others distributed algorithms which assume that processes have unique identifiers. We therefore focus on snapshot problems, and we propose to investigate how existing algorithms can be composed, with refinement, in order to compute a global state in an anonymous network.

Keywords Distributed algorithms, Local computations, Termination Detection, Enumeration, Snapshots, Composition, Refinement, Formal methods, Event-B

Remerciements

Ce travail n'aurait pu aboutir sans l'aide et le soutien d'un certains nombre de personnes auxquelles j'ai le plaisir d'adresser mes sincères remerciements.

Je tiens d'abord à exprimer ma reconnaissance et mes plus vifs remerciements à mes deux directeurs de thèse M. Ahmed HADJ KACEM et M. Mohamed MOSBAH, pour avoir assuré le bon déroulement de la thèse. Je tiens à leur exprimer mon profond respect pour leurs qualités humaines et scientifiques.

Je remercie M. Ahmed HADJ KACEM pour la confiance qu'il m'a accordée en acceptant la direction scientifique de mes travaux depuis que j'étais étudiante en mastère. Je lui suis reconnaissante pour ses encouragements et ses conseils. Je le remercie aussi de m'avoir mis en contact avec M. Mohamed MOSBAH pour faire cette thèse en cotutelle.

Je remercie M. Mohamed MOSBAH de m'avoir accueilli chaleureusement dans son équipe. Je lui suis reconnaissante pour l'intérêt qu'il a porté à cette thèse. Son soutien, ses conseils, ses suggestions de lecture et ses corrections m'ont aidé à mener à bien ce travail.

Je tiens également à exprimer ma gratitude pour M. Mohamed TOUNSI. Je le remercie d'avoir assuré le co-encadrement de mes travaux de recherche depuis que j'étais étudiante en mastère. Merci pour ses remarques et ses corrections.

Je tiens à remercier tout particulièrement M. Dominique MÉRY et Mme. Leila JEMNI BEN AYED, d'avoir accepté de rapporter ce manuscrit. Un grand merci aussi à M. Khalil DRIRA, M. Akka ZEMMARI et Mme. Imen JEMILI, qui m'ont fait l'honneur d'évaluer mon travail en participant à ce jury.

Je tiens également à exprimer ma profonde reconnaissance et ma gratitude pour M. Yves MÉTIVIER, de m'avoir manifesté son intérêt tout au long de mes recherches. Je le remercie aussi d'avoir assuré un cours au LaBRI autour des algorithmes distribués. Ses explications pertinentes m'ont aidé à assimiler les notions de base de l'algorithmique distribuée et la nature des problèmes qui y sont liés. Je n'oublie pas les membres du laboratoire ReDCAD, les membres du LaBRI, et les agents administratifs.

Ces remerciements ne peuvent s'achever sans une pensée à ma famille et mon mari, qui m'ont toujours soutenu dans mes projets, et plus particulièrement mes parents qui m'ont guidé sur le chemin des études. Enfin, un grand merci à tous mes amis qui m'ont encouragé et ont partagé avec moi tous les souvenirs aussi bien les meilleurs que les pires.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Les systèmes distribués offrent un cadre absolu pour réunir des entités de calcul, et les faire interagir afin d'accomplir un objectif commun. Le fonctionnement d'un système distribué doit préserver l'autonomie des entités, et assurer une transparence totale de leur répartition ainsi que de leurs interactions. Quelque soit sa nature, i.e., un réseau physique de machines, un logiciel avec plusieurs processus, etc, le système doit apparaître à l'utilisateur comme étant une entité unique et cohérente. Compte tenu de leur intérêt, tant qu'au niveau logiciel que matériel, les systèmes distribués ne cessent d'évoluer et d'intégrer divers domaines d'application, à savoir la gestion d'informations, le contrôle d'activités en temps réel, etc. La mise en œuvre d'un tel système permettrait une réalisation d'applications à grande capacité d'évolution, un meilleur partage de ressources, une optimisation de la puissance de calcul et de stockage, etc.

Cependant, un environnement distribué requiert beaucoup d'efforts pour mettre en œuvre un système correct et cohérent. La correction du système est généralement ramenée à la correction du calcul distribué, appelé aussi algorithme distribué. Par définition [59], un algorithme distribué est la suite de transitions locales à effectuer sur chaque entité du système. La difficulté de la preuve de correction de ces algorithmes résulte des caractéristiques et

du mode de fonctionnement du distribué. En effet, chaque entité exécute localement son calcul et ne peut interagir qu'avec ses proches voisins. Ainsi, les éléments du réseau n'ont aucune connaissance de l'état global du système. De plus, il n'y a ni horloge globale, ni temps universel. D'autre part, il n'y a pas de modèle standard pour le calcul distribué, la communication est généralement asynchrone, le calcul est non-déterministe, etc. Toutes ces propriétés donnent naissance à de nombreux défis et problèmes fondamentaux de l'algorithmique distribuée. Par conséquent l'utilisation de méthodes et outils avancés est nécessaire pour faire face à la complexité du développement et de preuve des algorithmes distribués. Dans ce contexte, il est de plus en plus approuvé que la précision et le haut niveau d'abstraction des méthodes formelles permettent de dévoiler toute inconsistance et ambiguïté du calcul.

Nos travaux de thèse consistent à proposer des approches formelles permettant de simplifier la modélisation et la preuve des algorithmes potentiellement complexes. Plus particulièrement, nous nous intéressons à résoudre certains problèmes du distribué. Nous considérons un système distribué défini par un réseau fiable, de processus anonymes et avec un modèle de communication basé sur l'échange de messages. Dans certains cas, nous faisons abstraction du modèle de communications en utilisant le modèle des calculs locaux [52]. Dans ce modèle, un réseau est défini par un graphe simple, connexe et non-orienté. Les sommets du graphe représentent l'ensemble des processus et les arêtes caractérisent les canaux de communication. Les états des processus et des canaux sont définis par des étiquettes associées, respectivement, aux sommets et aux arêtes. La modification d'états des sommets et des arêtes s'effectue selon des règles de réécriture de graphe. Une règle implique deux sommets voisins, ou un sommet avec tous ses voisins. Dans tous les cas, elle nécessite une phase de synchronisation pour qu'elle puisse

être exécutée. Le modèle des calculs locaux se caractérise par sa simplicité de modélisation et par un méta-raisonnement permettant de distinguer entre le calcul et la topologie du réseau sur lequel l'algorithme sera exécuté.

Vérification Formelle des Algorithmes Distribués

Dans la littérature, nous distinguons deux types de vérification formelle à savoir une vérification de modèle (*Model Checking*) et une preuve par théorème (*Theorem Proving*). Le premier type de vérification est dédié aux systèmes à états finis. Il permet de modéliser une abstraction du comportement du système et vérifier si une formule de propriétés peut se reproduire. Bien qu'il soit largement utilisé, le (*Model Checking*) soulève le problème d'explosion combinatoire. Ce type de problèmes se révèle quand le système est formé d'un nombre important d'états. Le deuxième type de vérification consiste à spécifier le système et ses propriétés avec une certaine logique mathématique. La preuve de correction du système est formalisée par un ensemble d'invariants et théorèmes justifiés par une application de preuves intermédiaires.

Chaque type de vérification pourrait être appliqué dans le cadre d'une certaine approche formelle traduisant une logique de raisonnement approprié. L'une des approches les plus prometteuses consiste à raisonner sur la preuve du système selon différents niveaux d'abstraction. Il s'agit d'une approche *correct-par-construction* [41] qui permet de raisonner progressivement sur la correction du calcul. Cette progression est le résultat d'une succession de raffinements transformant un modèle abstrait M_0 en un modèle plus concret M_n (FIGURE 1). L'avantage majeur de cette approche consiste à partager la complexité du développement sur des niveaux de moins en moins abstraits. Chaque niveau représente une spécification d'un modèle correct vis-à-vis le

modèle M_0 du niveau initial. De plus, un raffinement d'un modèle M_i en un nouveau modèle M_{i+1} doit préserver les propriétés de correction de M_i.

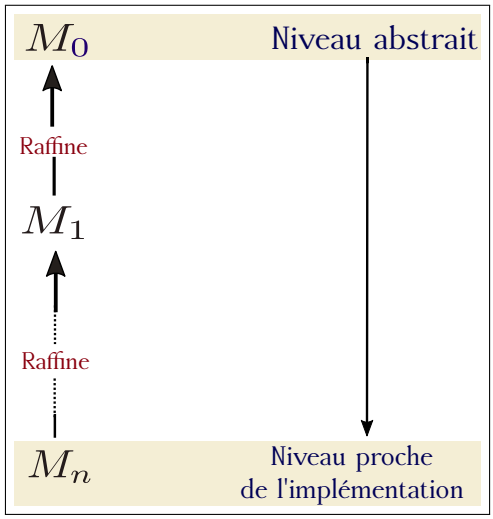

Figure 1 – Raffinement de modèles

L'approche *correct-par-construction*, le raffinement et la preuve par théorème sont largement utilisés dans la littérature, notamment pour définir des patrons de conceptions prouvés [61, 7, 46]. Ils constituent également le fondement de base de nos travaux de thèse. La méthode Event-B [2] et son assistant de preuve Rodin [3] forment l'environnement formel que nous avons choisi pour représenter et analyser nos spécifications.

Objectifs de la thèse

Nos travaux de thèse consistent à proposer un cadre formel pour le développement et la preuve des algorithmes distribués. Nos principaux objectifs se résument comme suit :

– fournir des patrons de conception prouvés, destinés à résoudre certains problèmes du distribué, notamment la détection de terminaison, l'énu-

mération, et le calcul d'état global,

- caractériser de façon formelle et incrémentale des liaisons de compositions entre certains algorithmes existants,
- prouver que cette composition pourrait servir comme solution pour des algorithmes plus complexes,
- réduire l'effort de preuve et de modélisation des problèmes étudiés en proposant un schéma de preuve guidé par des raffinements successifs.

Contributions et Structure de la thèse

Notre travail a fait l'objet de cinq chapitres répartis comme suit :

Dans le premier chapitre nous introduisons le cadre théorique de la thèse. Nous présentons un aperçu sur les principaux modèles et logiques utilisés pour le calcul distribué. Nous détaillons particulièrement les systèmes de réécriture de graphe et les modèles des calculs locaux.

Dans le deuxième chapitre, nous dressons le cadre formel de nos travaux. Nous montrons que les approches par *modularisation*, *composition* et *correct-par-construction* serviront à atteindre nos objectifs de preuve et de réutilisation. Nous montrons également que la méthode Event-B pourrait bien supporter notre travail.

Dans le troisième chapitre, nous présentons notre première contribution. Elle consiste à proposer une transformation d'algorithmes afin de permettre aux processus de détecter une configuration finale du système. Cette transformation permet de passer d'un calcul "avec une détection de terminaison locale",

vers un calcul “avec une détection de terminaison globale”. Nous faisons une abstraction du modèle de communication en utilisant les calculs locaux. Nous optons pour une *modularisation* d'un algorithme de détection de terminaison : l'algorithme est spécifié par un composant prouvé indépendamment, appelé *module*. Nous montrons que la composition de ce *module* avec un algorithme A qui respecte certaines hypothèses, et qui est spécifié selon l'approche *correct-par-construction*, permet de préserver la correction de A et réutiliser la preuve du composant *module*.

Dans le quatrième chapitre, nous traitons le problème d'énumération qui consiste à attribuer des identifiants uniques aux processus. Nous considérons un algorithme basé sur les calculs locaux et défini par des relations de réécriture localement engendrées sur des étoiles (un processus et ses voisins). L'algorithme permet aux processus de calculer une cartographie du réseau et converger, sous certaines hypothèses, vers une énumération. Nous proposons un schéma de preuve basé sur différents niveaux d'abstractions. Notre démarche commence par la vérification globale de l'algorithme jusqu'à prouver, progressivement, sa correction de point de vue locale.

Dans le dernier chapitre, nous étudions le problème de calcul d'état global. Nous proposons un développement incrémental permettant aux processus de calculer d'abord leurs snapshots locaux, détecter ensuite la terminaison global du calcul, et procéder enfin à construire une cartographie du réseau. Un processus peut calculer cette cartographie à partir des informations reçues lors de l'application des étapes d'énumération du chapitre précédent. Nous montrons qu'une solution à ce problème peut réutiliser des algorithmes et preuves existantes. Une liaison de composition est ainsi mise en œuvre entre

le calcul du snapshot, la déetection de terminaison et la construction d'une cartographie du réseau. Nous montrons que cette liaison pourrait être spécifiée et prouvée par un développement *correct-par-construction* qui met en superposition différents algorithmes.

Chapitre

1 Modélisation des Algorithmes Distribués par les Calculs Locaux

Sommaire

1.1 Introduction

Un système distribué fait interagir et collaborer des entités de calculs autonomes (processus, machines, etc.) afin de réaliser un objectif commun [12] : un calcul, un service, une application, etc. Ces entités, inter-connectés

via un réseau, fonctionnent localement selon un algorithme particulier, appelé algorithme distribué, et suivant divers modes de communication.

Les particularités d'un système distribué renforcent le degré de difficulté de sa modélisation ainsi que de la preuve de l'algorithme mis en œuvre : cela est dû notamment au non- déterminisme lors de l'exécution, à l'absence d'une horloge globale, aux connaissances limitées des entités, etc. L'intérêt des travaux de recherche dans ce contexte, consiste à trouver des modèles et logiques pertinents, permettant d'expliciter les propriétés du système et spécifier rigoureusement l'algorithme distribué qui lui correspond.

Ce chapitre constitue une présentation d'un aperçu sur les principaux modèles utilisés dans la littérature particulièrement les calculs locaux et les systèmes de réécritures de graphe. Nous introduisons également les concepts de base et quelques définitions de la théorie des graphes qui sont utilisés tout au long de la thèse.

1.2 Algorithmique distribuée

Un algorithme distribué A, sur un système distribué S, est la suite de transitions locales à effectuer sur chaque entité de S. Les algorithmes distribués se différencient par le mode de communication selon lequel interagissent les différents éléments distribués (mémoire partagée, échange de messages...), par le modèle temporel suivi (synchrone, asynchrone) et par la nature du réseau implémenté (fiable, non fiable, anonyme...). Toutes les propriétés précédemment citées sont détaillés par les définitions suivantes :

Définition 1.1 (Communication par mémoire partagée)**.** *Le mode communication par mémoire partagée considère une zone mémoire vive auquel plusieurs processus peuvent y accéder. Les opérations d'écriture s'effectuent géné-*

ralement d'une manière exclusive. Toutefois, l'accès à la mémoire en lecture peut être concurrent.

Définition 1.2 (Communication par échange de messages)**.** *Dans un mode de communication par échange de messages une interaction s'effectue entre un émetteur de message et un récepteur. Les émetteurs et les récepteurs sont reliés par des canaux de communication unidirectionnels ou bidirectionnels.*

Définition 1.3 (Communication synchrone)**.** *Une communication synchrone est une transmission instantanée. L'émetteur et le récepteur doivent se synchroniser avant de se communiquer.*

Définition 1.4 (Communication asynchrone)**.** *Une communication asynchrone note une absence d'horloge globale. Un message envoyé par un émetteur arrive au récepteur en un temps arbitraire.*

Définition 1.5 (Réseau fiable)**.** *Un réseau est dit fiable si le temps de délivrance d'un message est fini et s'il n'y a ni perte ni duplication de messages.*

Définition 1.6 (Réseau anonyme)**.** *Un réseau est dit anonyme si toutes les entités du réseau ne possèdent pas d'identifiant unique.*

Dans notre travail, nous adoptons un réseau fiable, asynchrone, de processus anonymes, avec un modèle de communication basé sur l'échange de messages. Les Chapitres 3 et 4 font abstraction du modèle de communication en utilisant le modèle des calculs locaux proposé par Y. Métivier et al. [52]. Dans la section suivante, nous présentons un aperçu sur les différents modèles et logiques proposés dans la littérature pour l'encodage des algorithmes distribués. Ensuite, nous définissons les concepts de base et les notions nécessaires du modèle considéré.

1.3 Modélisation des Algorithmes Distribués

Plusieurs modèles et langages formels ont été proposés pour modéliser et vérifier les algorithmes distribués. Nous citons principalement les automates d'entrée-sortie, la logique TLA, la logique PlusCal et les calculs locaux.

1.3.1 Quelques Modèles

I/O automates L'I/O automate (Input/Output), appelé aussi automate d'entrée-sortie, a été développé par N. Lynch et al. [47] pour modéliser les composants d'un système asynchrone et plus précisément pour les systèmes réactifs ou événementiels. Dans un tel modèle, chaque composant est défini par un I/O automate qui peut être représenté comme un système de transitions distinguant entre le calcul inter-processeur et le calcul dont le résultat est sous le contrôle de l'environnement. Ceci est dû aux trois types de transitions que peut implémenter un automate : des transitions d'entrée représentant les signaux reçus par un processeur, des transitions internes permettant de modéliser le calcul effectué, et des transitions de sortie décrivant son signal émis.

Logique TLA+ TLA+, proposée par Leslie Lamport [44] , est une logique de spécification formelle, basée sur la combinaison de TLA (Temporal Logic of Action) [43] et de la théorie des ensembles. Elle est conçue pour la vérification des grands systèmes complexes tels que les réseaux de communication. Le principe de la logique TLA est de représenter les différentes fonctionnalités d'un système sous formes de prédicats initiaux et d'actions. Un prédicat initial permet de spécifier l'état initial du système. Une action précise les transitions d'états possibles. Chaque action permet de spécifier les opérations devant être exécutées, et met à jour l'état du système.

PlusCal (+Cal) PlusCal (+Cal) [45] a été proposé par par Leslie Lamport pour améliorer la logique TLA+. En effet, afin de rendre TLA+ accessible aux concepteurs d'algorithmes, PlusCal permet de décrire un calcul distribué par une syntaxe traduite automatiquement en une spécification TLA+, vérifiable par le model-checker TLC [65].

Calculs Locaux Le modèle des calculs locaux représente un réseau par un graphe étiqueté : les sommets du graphe représentent l'ensemble des processus participants au réseau. Les arêtes traduisent les liens de communication. L'état de chaque processus [respectivement lien de communication] est représenté sous forme d'une étiquette associée au sommet [respectivement au lien de communication]). Un modèle des calculs locaux est défini par des séquences de réécriture caractérisées sur des boules du graphe. La modification des sommets et des arêtes s'effectue selon des règles de réécriture appliquées dans un contexte local, après avoir exécuté un algorithme de synchronisation. L'implémentation d'algorithmes de synchronisation est une solution pour échapper au problème du non-déterminisme des algorithmes distribués pour un réseau asynchrone de processus anonymes qui communiquent via un modèle basé sur l'échange de message. Chaque algorithme de synchronisation implémente un type de règle de réécriture. Nous présentons davantage chaque type de règles dans la Section 1.4.2.

1.3.2 Synthèse

Certes, les premiers modèles cités ci-dessus permettent de modéliser et vérifier la correction des algorithmes distribués. Cependant, nous y distinguons quelques limites que nous pouvons dépasser par le modèle des calculs locaux. En effet, dans les I/O automates, l'alternance entre émission et réception

des signaux ne permettent pas une distinction explicite entre le traitement et la communication, ce qui influence négativement sur le raisonnement abstrait des modèles. Par ailleurs, la modélisation des algorithmes en des actions décrites par des formules logiques n'est pas une tâche facile. Certes, l'amélioration de cette logique par la logique PlusCal permet d'automatiser les spécifications TLA+, cependant, dans la pratique, PlusCal ne peut pas être utilisé sans une bonne expertise du formalisme TLA+.

Les modèles des calculs locaux, quand à eux, se caractérisent par la simplicité de modélisation, le méta-raisonnement, et le haut niveau d'abstraction qu'ils peuvent fournir pour analyser et décrire les algorithmes distribués. La preuve de correction d'un algorithme distribué codé par les calculs locaux est assurée quand le mécanisme de synchronisation est lui aussi vérifié et prouvé. Un tel modèle permet de distinguer entre la topologie du réseau sur lequel l'algorithme est exécuté et le calcul réalisé. Ceci nous permet de raisonner sur les problèmes d'algorithmes distribués en fonction de la nature du réseau implémenté (graphe quelconque, arbre, anneau, etc). Plusieurs efforts sont entrepris dans ce contexte : nous citons par exemple le travail de Y. Métivier et al. [51] , dans lequel les auteurs ont proposé deux algorithmes pour réaliser une élection dans une boule de rayon K. L'analyse de ces algorithmes est basée sur des rounds décrivant le nombre maximal d'élection autorisé par l'algorithme. Une simple caractérisation des familles des réseaux admettant un algorithme d'élection a été proposée par E. Godard et al. [34].
Finalement, les modèles des calculs locaux bénéficient de l'environnement de visualisation et de simulation *ViSiDia* [9] qui permet de fournir les outils nécessaires pour que l'algorithme s'exécute en temps réel. Le concepteur est ramené à implémenter sons propre algorithme et lancer ensuite la simulation. Il est capable ainsi de vérifier et visualiser les changements d'états qui

s'effectuent dans le système.

1.4 Système de Réécriture de Graphe et Calculs Locaux

Les systèmes de réécriture de graphe et plus généralement les calculs locaux constituent un model formel solide pour exprimer, dans un haut niveau d'abstraction, les algorithmes distribués. Ce modèle représente les systèmes distribués en se basant sur des notions fondamentales de la théorie de graphe. Dans ce qui suit, nous allons présenter d'abord, un petit rappel sur ces notions. Ensuite, nous allons expliquer le modèle des calculs locaux, distinguer ses différentes modes de synchronisation, et illustrer un exemple simple implémenté par ce modèle.

1.4.1 Notions Standards des Graphes

Le modèle des calculs locaux nécessite une expertise en matière de graphe. Les définitions que nous présentons dans cette section proviennent essentiellement de [60, 30]. Dans notre travail, nous nous sommes basés sur des graphes simples, connexes et non orientés. Formellement un graphe G est représenté par un couple G=(V, E), où V est un ensemble fini de sommets et E est un ensemble d'arêtes. E={{v,v'}|v,v' ∈ V, v≠v'}.

Définition 1.7 (Voisinage d'un sommet)**.** *Le voisinage d'un sommet v est défini comme l'ensemble des sommets adjacents à v.*

Définition 1.8 (Degré)**.** *Soit v un sommet 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 $N(v)$. Les sommets de degré 1 sont appelés des feuilles.*

Définition 1.9 (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 sommets et d'arêtes, tel que, pour chaque j tel que $1 \leq j \leq i$ alors e_j est une arête incidente aux sommets v_j et v_{j+1}. Si v_1 $=v_i$ alors P est un cycle.*

Définition 1.10 (Distance). *Soit G un graphe connexe. La distance d'un sommet u à un sommet v dans G, est la longueur du plus court chemin de u à v.*

Définition 1.11 (Diamètre). *Le diamètre d'un graphe connexe G est la plus grande distance entre deux sommets de G.*

Définition 1.12 (Graphe connexe). *Un graphe est dit connexe si pour tout couple de sommets distincts (v,w), il existe un chemin reliant v à w.*

Définition 1.13 (Cycle). *Un cycle est un chemin dont les deux extrémités sont égales.*

Définition 1.14 (Arbre). *Un arbre est un graphe non orienté, acyclique et connexe.*

Définition 1.15 (Arbre-recouvrant). *Un arbre recouvrant d'un graphe G est un sous graphe de G contenant tous les sommets de G et étant un arbre.*

Définition 1.16 (Anneau). *Un anneau est un graphe constitué d'un cycle simple.*

Définition 1.17 (Boule). *Soit G un graphe et c un sommet de G. La boule $B_G(c)$ est le sous-graphe de G contenant c, son voisinage, et l'ensemble des arêtes incidentes à c.*

Définition 1.18 (Graphe étiqueté). *Un graphe étiqueté est noté par le couple (G,λ) où G désigne le graphe et λ désigne une fonction d'étiquetage qui associe à chaque sommet et arête de G un label appartenant à un ensemble finis d'alphabets.*

Définition 1.19 (Étiquetage des ports). *Étant donné un graphe simple G de sommets $V(G)$, un étiquetage de ports δ est un ensemble de fonctions $\{\delta_u | u \in V(G)\}$ tel que pour tout sommet u, δ_u est une bijection entre le voisinage de u et l'intervalle $[1..d(u)]$, $d(u)$ étant le degré du sommet u.*

Définition 1.20 (Homomorphisme). *Un homomorphisme d'un graphe G sur un autre graphe G' est une application $\gamma : V \to V'$ telle 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. Dans ce cas, G est isomorphe à G' (noté $G \simeq G'$).*

Définition 1.21 (Graphe Orienté). *Un graphe orienté D est défini par un ensemble de sommets $V(D)$, un ensemble d'arcs $A(D)$ et deux fonctions s et t de $A(D)$ dans $V(D)$. Pour chaque arc $a \in A(D)$, $s_D(a)$ est la source de l'arc a et $t(a)$ est la cible de a : l'arc a est incident à $s(a)$ et à $t_D(a)$. Si $s(a) = t(a)$, l'arc a est une boucle. Pour tous sommets u, v appartenant à $V(D)$, s'il existe un arc $a \in A(D)$ tel que $s(a) = u$ et $t(a) = v$, u est un prédécesseur de v et v est un successeur de u.*

Définition 1.22 (Graphe Orienté Symétrique). *Un graphe orienté symétrique est un graphe orienté D muni d'une involution $Sym : A(D) \to A(D)$ qui à chaque arc $a \in A(D)$ associe son arc symétrique $Sym(a)$ tel que $s(Sym(a)) = t(a)$.*

Définition 1.23 (Graphe Simple non-Orienté)**.** *Un graphe simple G non-orienté (non-orienté) est un graphe défini par un ensemble de sommets $V(G)$, un ensemble d'arêtes $E(G)$ et par une fonction ext qui associe à chaque arête deux éléments distincts de $V(G)$, appelés ses extrémités. G est un graphe sans boucle tel qu'il y ait au plus une arête entre deux sommets, i.e., pour tous sommets u, v appartenant à $V(G)$, $|e \in E(G)|ext(e) = u, v| = 1$. Chaque arête $e \in E(G)$ peut alors être vue comme une paire de sommets distincts qui sont ses extrémités.*

Définition 1.24 (Des Graphes non-orientés aux Graphes-Orientés)**.** *À partir d'un graphe $G = (G, \lambda)$, de sommets $V(G)$ et d'arêtes $E(G)$, nous pouvons construire un graphe orienté symétrique, noté $Dir(G) = (Dir(G), \lambda')$, de sommets $V(Dir(G))$ et d'arcs $A(Dir(G))$: l'ensemble des sommets $V(Dir(G))$ est l'ensemble de $(V(G))$. Pour chaque arête $e \in E(G)$ dont les extrémités sont u et v, il existe deux arcs $a_{e,u,v}$ et $a_{e,v,u}$ dans $A(Dir(G))$ tels que $s(a_{e,u,v}) = t(a_{e,v,u}) = u$, $s(a_{e,v,u}) = t(a_{e,v,u}) = v$ et $Sym(a_{e,u,v}) = a_{e,v,u}$. Pour tout sommet $u.u \in V(Dir(G)), \lambda(u) = \lambda'(u)$, et pour tout arête e dont les extrémités sont u et v, $\lambda'(a_{e,u,v}) = \lambda'(a_{e,v,u}) = \lambda(e)$. Un exemple de construction d'un graphe orienté symétrique à partir d'un graphe simple est illustré dans la Figure 1.1.*

Définition 1.25 (Revêtement)**.** *Un graphe orienté D est un revêtement d'un graphe orienté D' s'il existe un homomorphisme ϕ de D vers D' qui est localement bijectif.*

Définition 1.26 (Revêtement Symétrique)**.** *Un graphe orienté symétrique D est appelé un revêtement symétrique d'un autre graphe orienté symétrique D' via un homomorphisme ϕ si D est un revêtement de D' via ϕ et si pour chaque arc a de D, $\phi(Sym(a)) = Sym(\phi(a))$.*

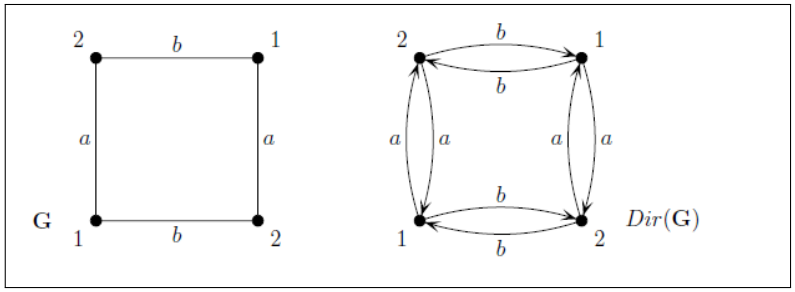

FIGURE 1.1 – Un Graphe Simple G et le Graphe orienté symétrique Correspondant

Définition 1.27 (Règles de réécriture)**.** *Une règle de réécriture est un triplet R=(G, λ, λ') où (G, λ) et (G, λ') sont deux graphes étiquetés représentant respectivement le côté droit et le côté gauche de la règle R. Deux mécanismes peuvent restreindre l'applicabilité de R. Un mécanisme qui introduit une relation de priorité sur l'ensemble de R et un autre qui interdit l'application de R sous certaines conditions.*

Définition 1.28 (Système de réécriture de graphes (GRS))**.** *Un système de réécriture de graphes est un triplet R=(L, I, P) où L désigne l'ensemble des labels, I est l'ensemble des labels de l'état initial du graphe avec $I \subset L$ et P désigne un ensemble finis de règles de réécriture.*

Définition 1.29 (GRS avec priorité)**.** *Un système de réécriture de graphe avec priorité est défini comme un 4-uplet R= (L, I, P,>) où (L, I, P) est un système de réécriture de graphe et '>' est un ordre partiel défini sur l'ensemble P.*

Définition 1.30 (GRS avec contexte interdit)**.** *Un système de réécriture de graphe avec contexte interdit est un système où l'application des règles de réécriture est empêchée quand l'occurrence correspondante est inclue dans des configurations particulières. Ces configurations forment un ensemble de cas de figures appelé un contexte interdit.*

Définition 1.31 (Graphe irréductible)**.** *Un graphe G est dit irréductible si aucune des règles de réécriture R n'est applicable dans G. Le calcul distribué se termine au moment où le graphe devient irréductible.*

1.4.2 Calculs Locaux

Pour un modèle de calculs locaux, le réseau est représenté par un graphe étiqueté. La modification des sommets et des arêtes s'effectue selon des règles de réécriture après avoir exécuté un algorithme de synchronisation. Chaque algorithme implémente un type de règle de réécriture. Dans la littérature, nous distinguons trois types d'algorithmes de synchronisation formant chacun une grande classe des calculs locaux, à savoir la classe LC0 (ou rendez-vous), la classe LC1 et la classe LC2.

Synchronisation de type LC0

Une règle de type LC0 est une règle qui s'applique sur deux sommets adjacents. L'application de la règle n'est effectuée que si les étiquettes des deux sommets et de l'arête satisfont la condition du déclenchement de la règle. Ce type de règle autorise la modification des deux sommets ainsi que l'arête qui les relie. Un système de réécriture est un système de type LC0 si est seulement si toutes les règles qui le décrivent sont de type LC0.

Synchronisation de type LC1

Une règle de type LC1 est une règle qui s'applique sur une boule de rayon 1. L'application de la règle n'est effectuée que si les étiquettes des sommets de la boule et des arêtes issues du sommet centre satisfont à la condition du déclenchement de la règle. Ce type de règle autorise la modification des

étiquettes du centre et des arêtes de la boule, et il empêche une modification des étiquettes des sommets voisins.

Synchronisation de type LC2

Une règle de type LC2 est une règle qui s'applique sur une boule de rayon 1 (Définition 1.4.1) du graphe. De même pour LC1, l'application de la règle n'est effectuée que si les étiquettes des sommets de la boule et des arêtes issues du sommet centre satisfont à la condition du déclenchement de la règle. Par contre, ce type de règle autorise, en plus des modifications sur les étiquettes du centre et des arêtes, une modification sur les sommets voisins. Ainsi tous les sommets de la boule peuvent être mis à jour.

Un système de réécriture est un système de type LC2 si est seulement si toutes les règles qui le décrivent sont de type LC2. De même pour LC1, nous pouvons introduire des mécanismes de contrôle locaux sur l'ensemble des règles qui le décrivent.

Exemples : calcul d'arbre recouvrant

Dans cette section, nous présentons un exemple simple du calcul distribué spécifié avec le modèle des calculs locaux. Le but de l'algorithme est de créer une arborescence hiérarchique sur l'ensemble du réseau : il permet de trouver un chemin acyclique reliant tous les processus. Formellement, il peut être encodé par le système S avec $S = (L, I, P)$ où $L = \{A, N, 0, 1\}$, $I = \{A, N, 0\}$ et $P = \{R1\}$. L désigne l'ensemble des labels des sommets et d'arêtes, I est l'ensemble des étiquettes initiales et P est l'ensemble des règles. Un sommet étiqueté A est un sommet dans un état *actif*. Les sommets dans l'état *neutre* sont étiquetés N. Initialement un seul sommet est étiqueté A, tous les autres

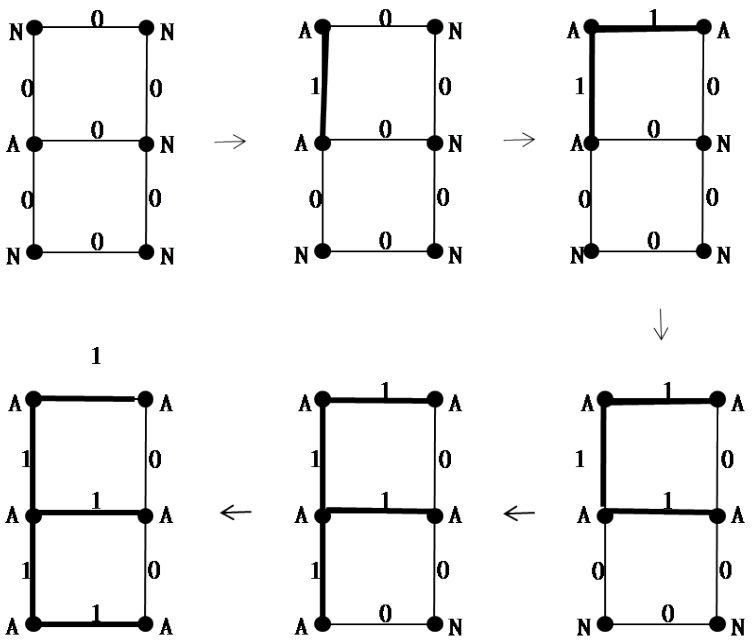

FIGURE 1.2 – Scénario d'exécution du calcul d'arbre recouvrant

sommets sont étiquetés N et toutes les arêtes sont étiquetées 0. Une arête qui change d'étiquette de 0 à 1 est considérée comme arête *marquée*. Par application de la règle *R1*, le sommet A peut activer un de ses voisins N et marquer l'arête correspondante. L'arbre recouvrant calculé est le chemin portant toutes les arêtes marquées. Nous schématisons dans la Figure1.2, un scénario d'exécution de ces règles. Notons que les règles de réécriture peuvent être appliquées en parallèle dans des parties disjointes du graphe.

Dans la suite de ce chapitre, nous présentons un aperçu sur les travaux relatifs à la spécification des calculs locaux ainsi que sur les méthodologies de preuves proposées dans ce contexte.

1.5 Spécifications Formelles sur les Calculs Locaux

Les travaux de recherche menés dans l'étude et la spécification formelle des calculs locaux ont proposé de différentes approches variées entre patron, sémantique et langage.

Spécification d'un patron Le niveau d'abstraction élevé des méthodes formelles permet une spécification concise et générique des patrons et de leurs applications. Les patrons formels appelés aussi patrons de conception prouvés [42], sont construits d'après un objectif de validation des objets conçus. Ils capturent des informations orientées preuve du système en cours de construction. Nous citons le travail de M. Tounsi et al. [61] dans lequel les auteurs proposent une spécification des calculs locaux basée sur l'approche *correct-par-construction*. Cette approche consiste à dériver, à partir des spécifications abstraites, des relations de réécriture localement engendrées. En d'autres termes, la modélisation du système débute par une spécification dans un contexte globale et s'achève par une intégration des relations qui ne dépendent que du contexte locale. Les auteurs ont proposé un patron formel basé sur le modèle des calculs locaux et sur une technique de raffinement. Ce patron est défini comme étant une description générale des solutions possibles à des problèmes d'algorithmes distribués.
Le patron proposé s'articule autour de trois niveaux d'abstraction :

- Niveau 0 : décrit une modélisation d'un passage direct de l'état initiale à l'état finale du système. Il permet ainsi de décrire en un seul pas ce que fait l'algorithme.

– Niveau 1 : incorpore des propriétés qui permettent d'exprimer le calcul du système du point de vue global.

– Niveau 2 : décrit les changements locaux effectués sur les états des sommets et des arêtes. Ces changements sont effectués suite à l'application des différentes règles de réécriture.

Plus récemment, nous avons proposé un patron formel [13], sous forme de spécifications génériques, qui prouve les propriétés de base relatives à l'application des règles de réécriture sur les modèles des calculs locaux. Nous avons suivi une approche modulaire, par laquelle nous avons pu fournir un environnement pour une spécification optimisée, basée sur la notion de réutilisation du modèle et des preuves qui lui sont associées. Nous fournissons plus de détails dans le chapitre 2.

Spécification d'une sémantique P. Castéran et al. [19] suivent une formalisation différente des calculs locaux. Ils ont défini une sémantique relationnelle exprimée dans le calcul des constructions inductives. Ce dernier est un lambda calcul dont le système autorise les familles de types (ou types d'ordre supérieur), le polymorphisme et les types dépendants. La sémantique proposée par les auteurs a été définie dans le cadre d'une bibliothèque formelle structurée en trois couches :

– Couche 1 : regroupe les définitions de la théorie de graphe.

– Couche 2 : décrit les résultats communs à l'ensemble des relations de réécriture de graphe, les modes de détection de terminaison, etc.

– Couche 3 : décrit les différents modes de synchronisations.

Les auteurs se sont basés sur la notion de tâche qui correspond grossièrement à la notion de relation de réécriture. Une tâche est composée d'un ensemble de graphes étiquetés initiaux et finaux, et d'une relation entre un état initial

et un état final. En d'autres termes, il s'agit d'une spécification pour une transformation de graphes étiquetés. Sa définition revient à préciser sur quel ensemble de graphes étiquetés elle doit opérer et quelle relation lie entre les états initiaux des sommets et des arêtes et entre leurs états finaux.
Formellement, les auteurs ont défini la notion de tâche par un prédicat de type :

$$\sum_{G,L_i} \rightarrow \sum_{G,L_o} \rightarrow \mathit{Prop}$$

$\sum_{G,L_i}$ désigne l'ensemble des étiquettes du graphe G traduisant les états initiaux, et $\sum_{G,L_o}$ désigne l'ensemble des étiquettes du graphe G traduisant les états finaux. *Prop* est un type prédéfini en Coq, il désigne une sorte de propositions qui peut énoncer un théorème. La relation engendrée par une règle de réécriture correspond à un plongement de celle-ci dans le graphe G. Ce plongement définit la sémantique du mode de synchronisation.

Spécification d'un langage M. Mosbah et al. [54] ont défini les relations de réécriture en se basant sur la notion de transition gardée. Ils ont développé un langage de programmation nommé *Lidia*, conçu pour la mise en œuvre des algorithmes distribués encodés par les calculs locaux. L'approche proposée par les auteurs repose sur un modèle de système de transition à deux niveaux. Le premier est utilisé pour spécifier le comportement de chaque composant du système, et le second décrit leurs interactions. Il s'agit ainsi d'un langage de transitions gardées où les pré-conditions de chaque transition sont exprimées par la logique ${L^*}_\infty$(infinitary logic with counters). Selon les auteurs, ${L^*}_\infty$ est une extension de la logique du premier ordre. Elle est dotée de nouveaux quantificateurs de comptage et elle a aussi la possibilité de simuler des opérations probabilistes comme le choix d'un élément dans un ensemble donné.

1.6 Conclusion

Tout au long de ce chapitre, nous avons défini les concepts de base liés au cadre théorique de la thèse. Dans un premier volet, nous avons introduit les différents aspects de l'algorithmique distribuée et nous avons expliqué les notions de modélisation de ces algorithmes. Nous avons montré, via une comparaison entre certaines approches qui sont proposées pour l'encodage des algorithmes distribués, que les calculs locaux constituent un modèle puissant caractérisé par sa simplicité et sa distinction explicite entre la topologie du graphe et le calcul effectué. Sa vérification revient à prouver la correction des mécanismes de synchronisations et des règles de réécritures qui lui sont associés. Dans un deuxième volet, nous avons présenté quelques définitions des systèmes de réécriture de graphe et des calculs locaux. Finalement, nous avons donné un aperçu sur quelques travaux menés pour la spécification et la preuve de correction des calculs locaux.

Le chapitre suivant portera sur les approches de preuves suivies dans nos travaux de thèse. La méthode Event-B fera l'objet de ce chapitre.

Chapitre

2 Vérification du Calcul Distribué et Formalisme Event-B

Sommaire

2.1 Introduction

L'utilisation du calcul distribué dans des systèmes sûrs requiert une vérification algorithmique afin de s'assurer de la correction et de l'exactitude du résultat abouti. Formellement, tout graphe irréductible obtenu à la fin d'une chaine de réécriture doit correspondre à une solution valide et correcte par rapport à l'algorithme associé. Pour faire face à la complexité du développement et de la vérification à effectuer sur un calcul distribué, le concepteur a besoin de méthodes et techniques qui le guident vers une spécification précise, dépourvue de toute sorte d'ambiguïté.

Les études menées dans ce contexte ont prouvé que les méthodes formelles permettent de palier aux faiblesses des méthodes traditionnelles et de révéler tout inconsistance ou dysfonctionnement du système. Les descriptions informelles, habituellement utilisées, manquent de rigueur et de précision. Elles consistent généralement à tester le comportement du système en traitant, d'une manière exhaustive, toutes les situations attendues. Dans la littérature, nous distinguons différentes approches et techniques de vérification formelle supportées par divers langages et outils.

Event-B [2], est l'un des formalismes les plus efficaces et les plus utilisés pour alléger le degré de difficulté de la modélisation et de la preuve du calcul. Il s'agit à la fois d'une méthode et d'un langage qui est développé par Jean-Raymond Abrial comme évolution de la méthode B classique [1]. Event-B est basé sur la logique du premier ordre et la théorie des ensembles. Il est caractérisée par sa démarche progressive basée sur la technique du raffinement. Cette démarche permet de modéliser le système à différents niveaux d'abstraction et de vérifier la cohérence entre ces niveaux par des preuves

mathématiques.

Dans ce chapitre, nous dressons le cadre formel de nos travaux. D'abord, nous présentons un aperçu sur les principaux techniques et approches formelles présentées dans la littérature. Ensuite, nous expliquons davantage le formalisme adopté.

2.2 Techniques et Approches de Vérification Formelle

Lors de la preuve de correction du calcul distribué, le concepteur se trouve face à deux types de vérification formelle : une vérification de modèle, connue sous son nom anglais *Model Checking*, et une preuve par théorème ou *Theorem Proving*. Pour chaque type, le concepteur peut réfléchir soigneusement à une approche particulière qui pourrait mieux satisfaire à son objectif. Cette approche pourrait faire l'objet d'une mise en oeuvre de nouvelles tactiques à savoir une composition de spécifications, une décomposition, un raffinement, etc. Dans cette section, nous expliquons les principaux types de vérification formelle et nous donnons un aperçu sur les différentes techniques et approches utilisées dans la littérature. Finalement, nous clôturons cette section par présenter le choix adopté pour nos travaux de thèse.

2.2.1 Model Checking/Theorem Proving

Le Model Checking est une technique de vérification conçue pour les systèmes à états finis. Il s'agit de modéliser une abstraction du comportement du système et fixer une formule de propriétés spécifiée en logique temporelle. Généralement, le formalisme utilisé pour la modélisation est un système de

transition. Différents langages ont été proposés pour spécifier un tel système, à savoir Promela [55], TLA+ [44], etc. Chaque langage dispose de son propre *model checker*. Il s'agit d'un outil qui renvoie un contre exemple si l'abstraction ne satisfait pas la formule de propriété. Sinon, le système est considéré comme un modèle valide de la formule. En d'autres termes, cette formule de propriété sera certainement valide dans toutes les situations possibles.

La technique du *Model Checking* est utilisée dans divers cas d'étude : Konur et al. [39] utilisent le model-checking probabiliste pour la vérification des comportements des essaims de robots fourrageurs. Mercaldo et al. [49] proposent un processus de vérification sur *Android*. L'approche consiste à dériver, à partir du *Bitcode* de l'application à analyser, un modèle formel basé sur l'algèbre de processus, puis détecter un logiciel malveillant et identifier ses caractéristiques. Le logiciel concerné est un *ransomware* qui, une fois lancé, chiffre les fichiers personnels de l'utilisateur et lui réclame une rançon. Le *model checker* utilisé est CWB-NC [26]. Il génère *true* si l'application appartient à la famille caractérisée d'un *ransomware*, et *false* sinon. Par ailleurs, Tobias et al. [38] proposent une approche de vérification des systèmes de transformation de graph (GTS), qui sont utilisés pour décrire les systèmes dynamiques. L'approche proposée consiste à générer, à partir d'une instance du GTS, une formule SMT (Satisfability Modulo Theories) décrivant les chemins d'un graph interdit. Cette formule est passée à un *model checker* pour calculer sa satisfaction. Si la formule est satisfaite, alors le modèle possède une instance d'un graphe interdit, ce qui engendre une erreur de modélisation.

Le Theorem Proving consiste à spécifier le système et ses propriétés avec une certaine logique mathématique définie par un ensemble d'axiomes, d'inva-

riants et de règles fondamentales de raisonnement appelées règles d'inférence. Une preuve de correction du système est formalisée par des théorèmes justifiées par application de ces règles et de preuves intermédiaires. Généralement, ces preuves sont réalisées à l'aide d'un assistant qui génère les propositions qui doivent être vérifiées pour valider la spécification. Ces propositions, appelées des obligations de preuve, peuvent être vérifiées soit automatiquement par l'assistant de preuve, ou bien interactivement en suivant une certaine logique.

Le *Theorem Proving* a donné naissance à plusieurs outils comme *Coq* [10], *Isabelle* [56], *PVS* [57], *Rodin* [62, 3], etc. Ces outils sont utilisés dans divers domaines pour traiter une multitude de problèmes : Achim et al. [17] proposent une sémantique formelle des concepts liés aux politiques de sécurité. Les auteurs utilisent les tactiques d'*Isabelle* pour prouver un cadre formel qui peut servir comme un méta-modèle pour la modélisation des politiques de contrôles d'accès. Par ailleurs, Fontaine et al. [32] ont raisonné en *Coq* sur les algorithmes distribués probabilistes qui font des choix aléatoires basés sur une certaine distribution de probabilité. Ce raisonnement, qui est fondé sur le calcul des constructions inductives, a permis aux auteurs de définir et prouver des classes d'algorithmes. En effet, les auteurs ont traité le problème du *rendez-vous* qui permet de réaliser des communications exclusives entre des paires de sommets voisins. Ils ont prouvé que ce type de problème ne peut jamais être résolu en supposant que le calcul est un algorithme déterministe s'effectuant sur n'importe quel graphe ayant n'importe quelle numérotation de ports. D'abord, les auteurs ont supposé l'existence d'un tel algorithme, ensuite ils ont montré que, pour un graphe particulier et une numérotation particulière, cet algorithme ne produit aucun rendez-vous. Finalement, ils ont prouvé qu'il existe un algorithme probabiliste qui résout ce problème sur

un graphe contenant au moins une arête. Ce type de résultat, connu sous le nom de preuve d'impossibilité, a été aussi démontré dans une bibliothèque en Coq proposée par Castéran et Filou [19]

Ces deux types de vérification sont généralement appliqués dans le cadre d'une certaine approche formelle traduisant une logique de raisonnement approprié. Dans la section suivante, nous introduisons les principaux approches présentées dans la littérature.

2.2.2 Correct-Par-Construction

Correct-Par-Construction est une approche qui permet de raisonner progressivement sur le développement formel du système en veillant à ce que les fonctionnalités et les propriétés de celui ci soient vérifiées, et qu'au moment ou le développement est terminé, celui ci soit déjà correct. Il s'agit d'une approche descendante fortement liée à la technique du raffinement : en effet, une approche ascendante permet de raisonner sur la preuve à partir d'une vision locale des processus pour finalement vérifier la correction globale du calcul. En revanche, dans une approche *Correct-Par-Construction*, nous commençons par prouver l'objectif global de l'algorithme pour ensuite vérifier des spécifications plus élaborées traduisant le comportement local des processus. Ce type de raisonnement permet de mener le développement sur des nivaux de moins en moins abstraits. Cette abstraction est progressivement relevée par différentes étapes de raffinement qui permettent de transformer un modèle de spécifications abstraites SP_i en un modèle mathématique plus concret SP_{i+1}. En d'autres termes, une construction progressive à partir de SP_1 aboutit à un modèle SP_n qui réduit l'indéterminisme et dont ses détails de spécifications sont plus proches de l'implémentation ($SP_1 \rightarrow SP_2 \rightarrow \ldots \rightarrow SP_n$). À chaque étape i, SP_{i+1} préserve les propriétés de SP_i.

Dans la littérature, le raffinement est largement utilisé pour prouver des systèmes corrects par construction. Nous citons principalement le patron formel des calculs locaux proposé par Tounsi et al. [61] que nous avons déjà présenté dans la section 1.5. Plus récemment, Andriamiarina et al. [6] ont proposé une méthodologie de développement formel des algorithmes distribués pour spécifier les comportements et les traces du système. Les auteurs se sont basés sur le raffinement de modèles fourni par la méthode Event-B [2], et sur des formules temporelles de vivacité exprimées en logique TLA [43]. Le principe consiste d'abord, à spécifier d'une manière abstraite les différents services offerts par l'algorithme à spécifier. Chaque service "s" est exprimé par une propriété de vivacité notée $P \rightsquigarrow Q$ où P définit la condition d'invocation de s et Q représente la condition vérifiée quand s termine. Par définition, $P \rightsquigarrow Q$ signifie qu'à chaque fois que P est vérifiée, Q l'est aussi ou elle le sera certainement dans le futur. Ensuite, pour procéder aux raffinements et enrichir la spécification, il suffit de décomposer chaque propriété $P \rightsquigarrow Q$ à l'aide des règles d'inférences relatives aux propriétés de vivacité.

2.2.3 Composition/Décomposition

La spécification et la preuve de correction des algorithmes complexes peuvent être simplifiées en optant pour une approche efficace fondée sur une composition et/ou décomposition de modèles. En effet, certains algorithmes, existants ou nouveaux, peuvent être composés pour fournir une solution à un problème plus complexe. Par contre, la décomposition vise à diviser un problème en plusieurs parties développées indépendamment. Cette division s'arrête à un niveau auquel la preuve est suffisamment simple pour pouvoir la démontrer. Une résolution satisfaisante des sous problèmes ainsi qu'une bonne étude des conditions nécessaires et suffisantes pour effectuer une composition

ou décomposition, permettraient de maîtriser convenablement la preuve de correction du résultat souhaité.

La composition et la décomposition d'algorithmes distribués sont largement étudiées dans divers champs d'application. Leurs définitions ainsi que leurs principes diffèrent d'une étude à l'autre. Filou et al. [31] ont proposé une approche de composition de deux classes d'algorithme de type LC0. L'approche est basée sur un mécanisme de transfert de résultat du premier algorithme vers le second. Pour la même classe d'algorithme, les auteurs ont également défini des propriétés suffisantes pour effectuer une décomposition d'un algorithme C en deux sous algorithmes A et B. Ces propriétés se résument en trois points : d'abord, il faut vérifier que les connaissances acquises lors de l'exécution de A soient suffisantes pour résoudre l'algorithme B. Ensuite, il faut s'assurer que toute exécution entrelacée de A et B sur un même graphe soit équivalente à une exécution séquentielle de A suivie par une exécution de B. En d'autres termes, il s'agit de la commutativité de A et B. Finalement, il faut garantir que la forme normale de A soit stable et non altérée par l'exécution de B.

Récemment, Altisen et al. [5] ont traité la composition d'algorithmes autostabilisants qui terminent. Les auteurs ont défini un opérateur de composition *op* qui garantit la correction de l'algorithme composé à l'aide du théorème suivant : si A_1 et A_2 sont deux algorithmes autostabilisants alors $A_1 \quad op \quad A_2$ préserve l'autostabilisation par conjonction de spécifications de A_1 et A_2. Dans ce cas, la composition est effectuée sous la condition que toute configuration terminale de A_1 vérifie les pré-conditions de A_2.

2.2.4 Modularisation

Les approches de spécification modulaire sont généralement utilisées pour réduire l'effort de développement et de preuve des systèmes complexes et améliorer l'extensibilité de modélisation. En effet, la modularisation consiste à décomposer le système en des modules qui peuvent être développés indépendamment. La particularité de ces derniers réside dans leur aspect plus ou moins générique favorisant leur réutilisation dans d'autres spécifications, ce qui mène à réfléchir sur la spécification de patrons formels. Iliasov et al. [37] attestent que l'effet des patrons peut être amplifié par l'utilisation d'une approche modulaire.

Récemment [13], nous avons suivi une approche modulaire afin de proposer un patron formel dédié à la preuve de correction de l'application des règles de réécriture des calculs locaux, lors d'une synchronisation de type LC1. Le patron dispose principalement de trois modules fortement liés. Dans un premier module, nous avons spécifié les propriétés de base du champ d'application d'un algorithme distribué défini selon le modèle des calculs locaux. Ce module comporte la spécification formelle d'un graphe simple, connexe et non orientée. Dans un deuxième module, nous avons défini des formules génériques pour une spécification abstraite d'une règle de type LC1 qui s'applique sur une boule du graphe. Nous avons constaté que toute règle de ce type peut être écrite sous la forme d'une relation entre l'étiquette de la boule avant et après réécriture du graphe. Finalement, dans un troisième module, nous avons prouvé un comportement dynamique correct d'une règle de réécriture, tout en précisant les conditions générales de son application ainsi que ses effets sur le graphe. Le principe d'utilisation de notre approche consiste à (1) améliorer, en premier lieu, le premier module en ajoutant d'autres propriétés du réseau, (2) définir, en deuxième lieu les différentes règles spécifiées à l'algo-

rithme par instanciation et utilisation des descriptions génériques présentées dans le deuxième module, (3) et instancier, en dernier lieu, le troisième module pour appliquer les règles de réécritures de l'algorithme. L'instanciation de ce module peut être effectuée une seule fois pour appliquer plus qu'une règle de réécriture.

2.2.5 Approches et Technique Utilisées

Les approches et techniques formelles habituellement utilisées dans la littérature sont variées et disposent de différentes particularités qui peuvent servir à atteindre l'objectif du concepteur. L'étude formelle de certains problèmes du distribué constitue l'objectif principal de nos travaux de thèse. Pour faire face à la complexité du distribué, nous effectuons un raisonnement progressif qui permet de valider les propriétés du système à différents niveaux d'abstraction, avant d'entamer tout détails de conception. Nous optons ainsi pour une approche *Correct-par-Construction* utilisée tout au long de la thèse.

En outre, la composition d'algorithmes est digne d'intérêt en ce quelle a pour effet d'utiliser des algorithmes existants pour résoudre certains problèmes qui nous intéressent, i.e., la détection de terminaison et le calcul de l'état global. La composition que nous réalisons est une composition séquentielle effectuée localement sur l'ensemble des processus : considérons deux algorithmes A et B. Après avoir prouvé les propriétés de A sur différents niveaux d'abstraction, le calcul de B se lance sur un niveau local et sur chaque processus ayant terminé le calcul de A. Dans certains cas, des informations locales seront obligatoirement fournies à l'ensemble des processus pour pouvoir démarrer le calcul de B. Le calcul final disposerait de nouvelles fonctionnalités qui peuvent être prouvées à partir des propriétés de correction de A et B.

Ces fonctionnalités satisferaient pleinement aux solutions des problèmes étudiés. De plus, nous avons constaté que la composition avec un algorithme B, utilisé pour résoudre le problème de terminaison, pourrait être effectuée sur tout algorithme sous certaines hypothèses. Par conséquent, nous avons opté pour une approche modulaire et nous avons spécifié un module comportant la preuve de correction des propriétés de B.
La combinaison de ces approches permet de tirer profit de l'efficacité de chacune et fournir un système correct tout en réutilisant la preuve de certains algorithmes. À nos connaissances, Event-B [2] est le formalise le plus approprié qui pourrait supporter ces combinaisons et réduire le degré de difficulté de la preuve de correction du système.

Bien que Coq [10] est utilisé dans différents travaux de recherche pour la preuve de correction des problèmes du distribué, cet environnement de preuve manque de méthodologie pour aborder un processus de développement formel. D'abord, il est important de signaler que Coq est fondé sur une logique suffisamment expressive pour définir des algorithmes comme étant des objets de premier ordre. De plus, certaines contributions [28, 11, 19, 30] sont basées sur Coq pour prouver des méta-théorèmes pour les calculs locaux, la théorie des graphes, etc. Filou et al. [19] ont proposé une bibliothèque formelle pour étudier l'expressivité des calculs locaux et l'impossibilité de réaliser certaines spécifications. Ils ont proposé une sémantique relationnelle pour chaque mode de synchronisation (voir Chapitre 1, section 1.5).

Toutefois, la formalisation d'algorithmes complexes nécessite une cohérence de logiques. Contrairement à Event-B, cette cohérence est difficile à traiter en Coq. La modélisation avec Event-B est très attirantes grâce au raisonnement progressif qui peut alléger la complexité du développement. Event-B, basé sur une vérification par *Theorem Proving*, fournit un cadre

formel pour la modélisation et la preuve des systèmes distribués. Il dispose aussi d'une possibilité de traduction des spécifications formelles des modèles en du code exécutable. Le reste du chapitre constitue une présentation détaillée de ce formalisme et des concepts clés utilisés.

2.3 Formalisme Event-B

Event-B [2] utilise la logique du premier ordre et la théorie des ensembles pour modéliser et prouver une variété de systèmes à évènements discrets. Une spécification formelle écrite en Event-B démontre, de façon fondamentale, qu'un système est correct par construction. Elle est principalement constituée de deux types de composants fortement liés : *Contexte* et *Machine*. Dans cette section, nous détaillons d'abord ces composants ; nous expliquons ensuite le concept du raffinement qui constitue l'un des points clés d'Event-B ; finalement nous présentons l'outil qui supporte cette méthode et que nous utilisons comme environnement de développement formel pour nos travaux.

2.3.1 Composant "Context"

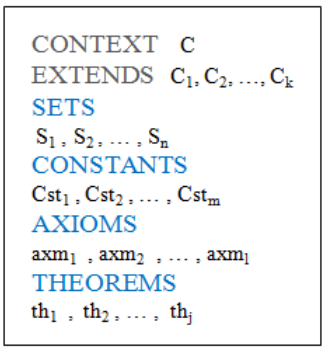

FIGURE 2.1 – Structure d'un "Context" Event-B

Un contexte Event-B fournit la déclaration statique et les propriétés des structures de données du système. Il comporte des ensembles, des constantes, des axiomes et des théorèmes. Ces éléments sont décrits respectivement par les clauses suivantes : SETS, CONSTANTS, AXIOMS et THEOREMS. Les axiomes sont spécifiés sous formes de prédicats définissant les propriétés relatives aux constantes et aux ensembles. Certains axiomes peuvent être prouvés à l'aide des axiomes déjà définis. Dans ce cas, ils sont marqués comme des théorèmes. Un contexte peut étendre d'autres contextes en utilisant leurs éléments et en ajoutant d'autres propriétés. La structure d'un contexte est présentée dans la FIGURE 2.1.

Obligations de Preuve pour la Bonne Définition d'un Contexte

Deux types d'obligations de preuve peuvent être figurés dans une spécification d'un contexte Event-B : le premier type noté WD (*Well Definedness*) permet de vérifier les contraintes d'une bonne définition d'un prédicat ou d'une affection. Par exemple, vérifier qu'un ensemble E est fini dans l'expression $card(E)$ qui permet de compter les éléments de E. Le deuxième type d'obligations de preuve est noté THM ($THeoreM$)). Elle assure qu'un axiome marqué en tant que théorème sera effectivement prouvé à partir d'autres axiomes déjà définis.

2.3.2 Composant "Machine"

Une *Machine* Event-B décrit l'état et le comportement dynamique du système. Ce comportement est spécifié par des évènements, de variants et d'invariants appliqués sur des variables d'états. Un évènement d'initialisation est obligatoire pour déclencher le scénario de déroulement du système. Les invariants permettent de typer l'ensemble des variables et de spécifier

les propriétés qui restent vraies pour tous les événements, y compris l'événement initial. De même pour les axiomes, les invariants peuvent être spécifiés en tant que théorèmes. Un variant est une expression numérique ou un ensemble fini spécifié pour prouver les évènements convergents du système, en montrant sa décroissance après chaque déclenchement de l'un de ces évènements. La structure d'une machine est présentée dans la FIGURE 2.2. Une

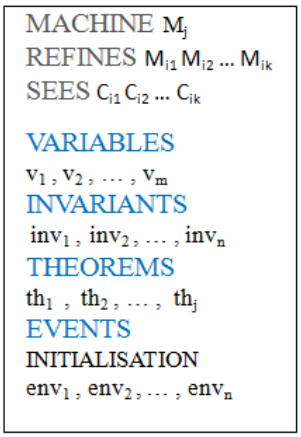

FIGURE 2.2 – Structure d'une "Machine" Event-B

machine fait recours à un ou plusieurs contextes pour utiliser les propriétés qui caractérisent le système. Cette relation est spécifiée par la clause SEES. La clause REFINES pointe sur le nom de la machine à raffiner. En effet, une machine peut raffiner une autre machine, et cette relation de raffinement est transitive. Elle s'effectue entre les évènements des machines. Un évènement $e1$ peut être raffiné par plusieurs évènements d'une autre machine (par exemple $e2$ *refines* $e1$, et $e3$ *refines* $e1$). Inversement, un évènement peut raffiner plusieurs autres évènement en appliquant une fusion de ces évènements ($e1$ *refines* $e2$ et $e3$).

Un évènement décrit les changements d'états du système en effectuant des mises à jour sur les variables. À un instant donné, il n'y en a qu'un seul qui

Évènement e		$Grd(e)(v)$	$BA(e)(v, v')$
Non déterministe	$ANY \quad p$ $WHEN$ $\quad G(p, v)$ $THEN$ $\quad v :\|A(p, v, v')$	$\exists p \cdot G(p, v)$	$\exists p \cdot G(p, v) \wedge A(p, v, v')$
Gardé	$WHEN$ $\quad G(v)$ $THEN$ $\quad v :\|A(v, v')$	$G(v)$	$G(v) \wedge A(v, v')$
Simple	$BEGIN$ $\quad v :\|A(v, v')$ END	$TRUE$	$A(v, v')$

TABLE 2.1 – Différents types d'évènements

se déclenche. Un évènement prend l'une des trois formes situées dans TABLE 2.1 : la forme générale est celle d'un évènement non-déterministe comportant trois parties : (1) un ensemble de paramètres p ; (2) une garde $G(v, p)$ spécifiant les conditions nécessaires au déclenchement de l'évènement en fonction des variables v de la machines et des paramètres p de l'évènement ; et (3) une action $A(p, v, v')$ qui décrit les mises à jours à effectuer sur les valeurs des variables concernées par le déclenchement de cet évènement. v' désigne les nouvelles valeurs de v selon un prédicat avant-après le déclenchement de l'évènement noté $BA(Before - After)$. Les autres variables gardent leurs valeurs inchangées. Contrairement à un évènement non déterministe, un évènement gardé ne dépend que des variables de la machine, il ne possède aucun paramètre. Finalement, un évènement est dit simple s'il ne possède aucune garde, autrement dit, sa garde est toujours vraie. Ce type d'évènement peut toujours se déclencher. Notons que l'évènement d'initialisation est un évènement simple qui se déclenche qu'une seule fois.

Le déclenchement d'un événement d'une machine est ainsi conditionné par une observation correcte d'un ensemble de gardes. Une fois cette condition est

satisfaite, les actions associées à cet événement seront réalisées. Sinon, l'événement reste en attente. Chaque action présente une substitution généralisée qui définit un changement sur une variable. Cette substitution peut se présenter comme une affectation, une substitution non déterministe ensembliste, une substitution non déterministe avec un prédicat, ou une substitution vide :

- une affectation sous la forme de $x := Exp(V, p)$ est une substitution déterministe qui remplace la valeur de la variable x par la valeur de l'expression $Exp(V, p)$. Par exemple $x := x + y$ est une affectation.

- une substitution non déterministe ensembliste est définie sous la forme $x :\in E(V, p)$. Une valeur de l'ensemble $E(V, p)$ sera affectée arbitrairement à x. Par exemple, la substitution $x :\in \{y, z, t\}$ affecte à x une valeur parmi l'ensemble $\{y, z, t\}$.

- une substitution non déterministe avec un prédicat est définie sous la forme $x : |P(V, p, x')$. Elle modifie la valeur de x en x' tout en préservant le prédicat P. Cette forme est une forme générale qui peut exprimer les autres formes de substituons : par exemple, l'affectation $x := x + y$ peut être écrite sous la forme de $x : |x' = x + y$. La substitution $x :\in \{y, z, t\}$ peut également être spécifiée de la manière suivante : $x : |x' \in \{y, z, t\}$. L'utilisation de ce genre de substituions non déterministe avec un prédicat est parfois obligatoire pour décrire des substituons parallèles. Il s'agit d'une action qui modifie plusieurs valeurs de variables telle que la valeur d'une de ces variables dépend de la nouvelle valeur d'une autre variable.

- une substitution vide est définie par le mot clé *skip*. Elle n'effectue aucun changement d'état.

Obligations de Preuve pour la Consistance d'une Machine

La consistance d'une machine en Event-B est garantie par la vérification de deux propriétés fondamentales, la préservation d'invariants et la faisabilité des actions :

- la préservation d'invariants, notée INV, assure que les propriétés invariantes du système ne sont jamais violées quelque soit l'évolution du comportement du système. Considérons un invariant $I(v)$ d'une machine M. Cette propriété est spécifiée pour chaque évènement e dans M. Elle est définie par l'implication suivante : $I(v) \wedge BA(e)(v, v') \Rightarrow I(v')$

- la faisabilité des actions notée FIS, assure qu'à chaque déclenchement d'un évènement e qui effectue une substitution généralisée $v : |P(v, v')$, il y aurait une valeur v' qui pourrait être attribuée à v et qui satisferait bien le prédicat P. Formellement, cette propriété est spécifiée par une implication conditionnée par la conjonction d'invariants relatifs à v et des gardes de l'évènement : $I(v) \wedge G(p, v) \Rightarrow \exists v' \cdot P(v, v')$

Dans certains cas, il est indispensable de prouver le non-blocage d'une machine. En d'autres termes, il faut s'assurer qu'il existe au moins un évènement déclenchable. Soit M une machine disposant d'un ensemble de variables v et de n évènements $e_1, e_2, \cdots, e_n$. Les gardes de ces évènements sont notées respectivement $G(e_1)(v, p_1), G(e_2)(v, p_2), \cdots, G(e_n)(v, p_n)$, où p_i représente l'ensemble des paramètres de l'évènement e_i. Plus précisément, cette obligation de non blocage doit vérifier que la disjonction des gardes des évènements est toujours vraie sous la condition de préservation d'invariants $I(v)$ de la machine. Formellement, est est définie comme suit : $I(v) \Rightarrow G(e_1)(v, p_1) \vee G(e_2)(v, p_2) \vee \cdots \vee G(e_n)(v, p_n)$

2.3.3 Raffinement en Event-B

Le concept du raffinement permet de modéliser un système à partir des spécifications abstraites et construire progressivement des niveaux de moins en moins abstraits tout en an ajoutant des détails de spécifications. Chaque niveau préserve les propriétés de corrections vis-à-vis au niveau précédent.

Soit M_0 une machine abstraite. Un niveau de raffinement en Event-B est effectué par la spécification d'une nouvelle machine M_1 qui porte un raffinement sur les données et/ou sur les évènements de M_0. M_1 est une machine résultat du raffinement de M_0. Les variables, paramètres, gardes et évènements de M_0 sont définis en tant qu'éléments abstraits. Ceux de M_1 sont considérés comme éléments concrets par rapport à la machine abstraite M_0. Un raffinement sur les donnés consiste à ajouter de nouvelles variables d'états, que nous notons v_c et/ou remplacer des variables abstraites notées v_a. Un invariant $J(v_a, v_c)$, appelé *invariant de collage* ou *gluing invariant* doit figurer dans M_1 pour exprimer le lien entre les variables abstraites v_a et les variables concrètes v_c. Un raffinement sur les évènements consiste à ajouter de nouveaux évènements et/ou raffiner des évènements abstraits :

- l'ajout d'un nouvel évènement consiste à raffiner l'évènement *Skip*. Il s'agit d'un évènement qui n'effectue aucun changement d'état.
- le raffinement d'un évènement abstrait peut être effectué par un ou plusieurs évènements concrets. Il s'agit de raffiner les gardes et/ou actions et/ou paramètres de l'évènement abstrait :
  - ◇ renforcer les gardes de l'évènement abstrait en ajoutant d'autres conditions qui garantiraient le déclenchement de l'évènement abstrait.
  - ◇ réduire le non déterminisme des actions abstraites

◇ remplacer les paramètres d'un évènement abstrait. Dans ce cas, un *témoin* doit figurer dans l'évènement concret via la clause $WITNESS$ pour exprimer la relation entre les paramètres abstraites et les paramètres concrètes.

Obligations de Preuve pour la Correction du Raffinement

Il est indispensable de prouver que la machine concrète raffine correctement la machine abstraite. Soit M_a une machine abstraite disposant d'un ensemble de variables et invariants abstraits notés respectivement v_a et $I_a(v)$. Soit M_c une machine qui raffine M_a et contenant des variables concrets notés v_c. Notons $J(v_a, v_c)$ l'invariant de collage reliant M_c à M_a. Soit un événement e_c dans M_c qui raffine un événement e_a de la machine M_a. Les paramètres, gardes, et prédicats avant-après de l'événement e_a sont notés respectivement p_a, $G(e_a)(p_a, v_a)$ et $BA(e_a)(v_a, v'_a)$. Ceux de l'évènement concret e_c sont notés respectivement par p_c, $G(e_c)(p_c, v_c)$ et $BA(e_c)(v_c, v'_c)$. Les valeurs de v'_a et v'_c désignent toujours les nouvelles valeurs attribuées aux variables v_a et v_c. La correction du raffinement est vérifiée par les obligations de preuves suivantes :

- le renforcement des gardes, noté GRD, assure qu'un évènement concret ne doit être déclenché que lorsque l'évènement abstrait l'est aussi. Cette obligation de preuve doit satisfaire l'implication suivante :
  $I(v_a) \wedge J(v_a, v_c) \wedge G(e_c)(p_c, v_c) \Rightarrow G(e_a)(p_a, v_a)$
- la simulation des actions, notée SIM, assure que le comportement d'un évènement concret se synchronise correctement avec celui de l'évènement abstrait qui lui correspond. Formellement, cette obligation de preuve est traduite par l'implication suivante :
  $I(v_a) \wedge J(v_a, v_c) \wedge BA(e_c)(v_c, v'_c) \Rightarrow \exists v'_a \cdot (BA(e_a)(v_a, v'_a) \wedge J(v'_a, v'_c))$

– l'égalité des variables préservées, notée EQL, assure que les variables abstraites gardent leurs valeurs inchangées. Généralement, cette obligation de preuve est vérifiée lorsqu'un évènement concret attribue la valeur de x' à une variable x déclarée dans la machine abstraite. Formellement, cette obligation doit satisfaire l'implication suivante :
$I(v_a) \wedge J(v_a, v_c) \wedge BA(e_c)(v_c, v'_c) \Rightarrow x' = x$

En outre, si la machine concrète dispose d'un variant $V(v_c)$, il sera indispensable de vérifier certaines propriétés : si le variant est un ensemble, une obligation de preuve, notée FIN, doit vérifier que $V(v_c)$ est un ensemble fini. Formellement, elle est formalisée comme suit : $I(v_a) \wedge J(v_a, v_c) \Rightarrow Finite(V(v_c))$
Si le variant est une expression, alors une obligation de preuve, notée NAT, doit vérifier que cette expression est entière : $I(v_a) \wedge J(v_a, v_c)) \Rightarrow V(v_c) \in \mathbb{N}$
Dans tous les cas, la définition d'un variant suppose la spécification d'un évènement convergeant qui fait décroitre ce variant. Par conséquent, une autre obligation de preuve, notée VAR, est définie pour vérifier qu'à chaque déclenchement d'un évènement convergeant e_c, alors le variant décroit en fonction des nouvelles valeurs attribuées à v_c. Formellement, cette obligation est formalisée comme suit : $I(v_a) \wedge J(v_a, v_c) \wedge G(e_c)(p_c, v_c) \Rightarrow V(v'_c) < V(v_c)$

2.4 Exemple de spécifications Event-B : calcul d'un arbre recouvrant

Dans cette section, nous nous référons au travail d'Abrial et al. [4] et de Cansell al. [18] pour présenter un aperçu sur la spécification formelle du calcul d'arbre recouvrant présenté dans le chapitre précédent (Section 1.4.2). La partie statique de l'algorithme réside dans la spécification des propriétés du graphe. Formellement, nous spécifions l'ensemble des sommets du graphe

par un ensemble abstrait nommé ND. Les arêtes sont définies par la constante g. Un ensemble d'axiomes est appliqué sur ND et g pour spécifier toutes les propriétés du graphe. L'axiome $axm1$ assure que ND est un ensemble fini, l'axiome $axm2$ définit la spécification formelle d'arêtes du graphe. Notons aussi qu'un calcul de l'arbre s'effectue sur un graphe simple, connexe, et non orienté. Ces trois propriétés doivent figurer également dans le contexte. Elles ne sont pas détaillées dans cet exemple, mais elles sont spécifiées dans la Section 3.4.1 du chapitre suivant. Outre ces propriétés, nous avons besoin de définir l'ensemble des arbres qui pourraient recouvrir les sommets du graph. Formellement, nous spécifions par r la racine de l'arbre ($r \in ND$), et par $trees$ l'ensemble des solutions possibles. Un élément de $trees$ doit construire un sous-graphe acyclique qui passe par tous les sommets. r est considéré comme racine si pour chaque autre sommet s, il existe un chemin issu de r à s. La construction finale des arbres recouvrants est spécifiée par l'axiome $axm4$.

$$
\begin{array}{|l|}
\hline
axm1 : finite(ND) \\
axm2 : g \subseteq ND \times ND \\
axm3 : r \in ND \\
axm4 : trees = \{t | \in ND \setminus \{r\} \rightarrow ND \wedge t \subseteq g \\
\qquad \wedge (\forall q \cdot q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q)\} \\
\hline
\end{array}
$$

Le calcul d'arbre recouvrant est effectué selon la règle de réécriture R en fonction d'étiquettes attribuées aux éléments du graphe. Ainsi, nous définissons deux types d'ensembles LN et LE désignant respectivement les étiquettes des sommets et des arêtes. Ces ensembles sont spécifiés respectivement par l'axiome $axm5$ et $axm6$. Notons que l'opérateur $partition$ est utilisé pour exprimer le faite que A et N sont distincts et constitues les partitions de LN.

$axm5 : partition(LN, A, N)$
$axm6 : LE = \{0, 1\}$

Une première machine M_0 Ce niveau peut être défini par une première machine abstraite M_0 spécifiant le résultat dans une seule étape sans détails de spécification. Cette machine calcule à travers l'évènement abstrait *Spaning_ Tree* une construction correcte de l'arbre. L'objectif du déclenchement de l'évènement est de générer, à partir d'un ensemble vide St, un arbre recouvrant. Le résultat doit satisfaire une des solutions déclarées dans l'ensemble $trees$.

EVENT	$Spaning\_Tree$
any	t
where	$grd1 : t \in trees \quad \wedge \quad st = \varnothing$
then	$act1 : st := t$

Une deuxième machine M_1 Un deuxième niveau est spécifié pour introduire une deuxième machine M_1 qui raffine M_0. Cette nouvelle machine permet de calculer d'une façon progressive le résultat de l'algorithme. De nouvelles variables sont introduites dans cette machine : les sommets qui sont inclus dans l'arbre en cours de construction et ceux qui restent ailleurs sont spécifiés respectivement par les variables tr_nodes ($inv1$) et $remaining\_nodes$ ($inv2$). Ces variables sont respectivement initialisées par $act1$ et $act2$ ($act1 : tr\_nodes := \{r\}$ et $act2 : remaining\_nodes := ND \setminus \{r\}$) . Les invariants $inv3$ et $inv4$ vérifient les propriétés d'intersection et d'union de ces deux ensembles. L'arbre en cours de construction est défini par la nouvelle variable new_tree ($inv5$ et $inv6$) que nous initialisons à un ensemble vide ($act3 : new\_tree := \varnothing$).

$inv1 : tr\_nodes \subseteq ND$

$inv2 : remaining\_nodes \subseteq ND$

$inv3 : remaining\_nodes \cap tr\_nodes = \varnothing$

$inv4 : remaining\_nodes \cup tr\_nodes = ND$

$inv5 : new\_tree \subseteq tr\_nodes \times tr\_nodes \quad \wedge \quad new\_tree = new\_tree^{-1}$

$inv6 : \forall q \cdot r \in q \wedge new\_tree^{-1}[q] \subseteq q \Rightarrow tr\_nodes \subseteq q$

L'évènement abstrait $Spaning\_Tree$ est raffiné dans M_1 par un renforcement d'une garde $grd2$ appliquée sur la variable $remaining\_nodes$. En effet, l'arbre n'est totalement construit qu'après vérification que $remaining\_nodes$ est un ensemble vide ($grd2$).

EVENT $Spaning\_Tree$ **refines** $Spaning\_Tree$

any $\cdots$

where $\cdots$

$\oplus grd2 : remaining\_nodes = \varnothing$

then $\cdots$

De plus, le raffinement est effectué par un ajout d'un nouvel évènement $Progress$ spécifiant le calcul progressif de l'arbre : une étape de calcul s'effectue entre deux sommets $s1$ et $s2$ appartenant respectivement à tr_nodes ($grd1$)et $remaining\_nodes$ ($grd2$). À chaque étape du calcul, $s2$ est ajouté à tr_nodes ($act1$) et supprimé de $remaining\_nodes$ ($act2$). L'arête correspondante à ces deux sommets est éventuellement ajoutée à new_tree ($act3$).

EVENT $Progress$

any $s1, s2$

where

$grd1 : s1 \in tr\_nodes$

$grd2 : s2 \in remaining\_nodes$

$grd3 : s1 \mapsto s2 \in g$

then

$act1 : tr\_nodes := tr\_nodes \cup \{s2\}$

$act2 : remaining\_nodes := remaining\_nodes \setminus \{s2\}$

$act3 : new\_tree := new\_tree \cup \{s2 \mapsto s1, s1 \mapsto s2\}$

Une troisième machine M_3 Un dernier niveau est introduit pour spécifier les interactions locales entre les sommets selon la règle de réécriture R (FIGURE 1.2). Ce niveau comporte une machine M_3 qui raffine M_2 et introduit deux nouvelles fonctions remplaçant certaines variables abstraites :

$inv1 : Lab\_ND \in ND \rightarrow LN$

$inv2 : Lab\_E \in g \rightarrow LE$

$inv3 : tr\_nodes = Lab\_ND^{-1}[\{A\}]$

$inv4 : inv2 : remaining\_nodes = Lab\_ND^{-1}[\{N\}]$

Lab_ND et Lab_E sont définis pour attribuer les étiquettes aux sommets et aux arêtes respectivement. Des invariants de collages sont introduits pour lier les variables abstraites tr_nodes et $remaining\_nodes$ aux variables concrètes Lab_ND et Lab_E : les éléments de tr_nodes constituent les sommets étiquetés A ($inv3$), et ceux de $remaining\_nodes$ représentent les sommets étiquetés N ($inv4$). L'évènement $Progress$ est raffiné par un évènement $SpTree\_Rule$ qui spécifie en détails la construction de l'arbre de point de vue locale. Cet évènement introduit les changements locales effectués sur

les étiquettes des sommets et d'arêtes suite à l'application de la règle.

EVENT $SpTree\_Rule$

refines $Progress$

any

$s1, s2$

where

$grd1 : s1 \mapsto s2 \in g$

$grd2 : lab\_ND(s1) = A$

$grd3 : lab\_ND(s2) = N$

$grd4 : lab\_E(s1 \mapsto s2) = 0$

then

$act1 : lab\_ND(s1) := A$

$act2 : lab\_E(s1 \mapsto s2) := 1$

$act3 : new\_tree := new\_tree \cup \{s2 \mapsto s1, s1 \mapsto s2\}$

2.5 Outil pour Event-B : Rodin

L'*Atelier B* [29] est le premier outil qui a été proposé pour une utilisation opérationnelle de la méthode B classique. Cet outil, développé par la société ClearSy, dispose d'un ensemble de prouveurs qui aident à démontrer des obligations de preuve relatives aux modèles. L'évolution du B classique en Event-B fait appraître un nouvel outil appelé *Rodin* [62]. Ce dernier dispose d'éléments et d'interfaces graphiques nécessaires pour spécifier un modèle en Event-B. Plus précisément, Rodin est une extension de l'environnement de développement *éclipse*[1] et s'appuie sur les mêmes prouveurs qui proviennent de l'*Atelier B* : c'est aussi un assistant qui génère les obligations de preuve indispensables pour la vérification formelle du modèle, à savoir la bonne défini-

1. www.eclipse.org

tion des contextes, la consistance des machines, la correction du raffinement, etc.

Le processus de vérification avec “Rodin” est illustré dans la FIGURE 2.3 : soit un modèle en Event-B spécifié par des contextes, des machines, des relations de raffinements “refines”, d’extensions entre contextes “extends”, etc. Le principe de vérification est effectué en passant par deux étapes : la première consiste à vérifier syntaxiquement la spécification du modèle introduit. En cas d’erreurs ($1'$), la spécification doit être corrigée ($1', 2$). Si tout est vérifié (1), *Rodin* passe à la deuxième étape qui consiste à générer les obligations de preuves nécessaires (WD, INV, THM, etc). Certaines de ces obligations peuvent être prouvées par des moteurs d’inférences introduits à *Rodin* avec de puissances variées. Ces moteurs, appelés *prouveurs automatiques*, permettent de déduire l’implication associée à l’obligation de preuve. Dans ce cas, la preuve est dite preuve automatique ($1.1'$). Cependant, les *prouveurs* de *Rodin* se trouvent parfois incapables de déduire cette implication (1.1). Dans ce cas, cette preuve est dite preuve interactive qui doit faire interagir les compétences du concepteur (1.2). Ce dernier pourra ainsi guider le processus de preuve en rappelant d’hypothèses déjà introduites ou en utilisant d’autres propriétés vérifiées auparavant (1.3). Dans certains cas, l’erreur de preuve provient de la logique de spécifications, ce qui impose le concepteur à revoir son modèle et le modifier le cas échéant ($1.3'$).

Rodin constitue l’environnement de développement formel utilisé dans nos travaux de thèse. Il est largement utilisé dans divers cas d’étude comme application de la méthode Event-B. Certains travaux de la littérature ont proposé d’étendre les fonctionnalités de cette plateforme à travers de plugins intégrés. Parmi ces travaux, nous nous sommes intéressés au plugin de liasov et al. [37] qui a été proposé comme extension à *Rodin* pour intégrer des

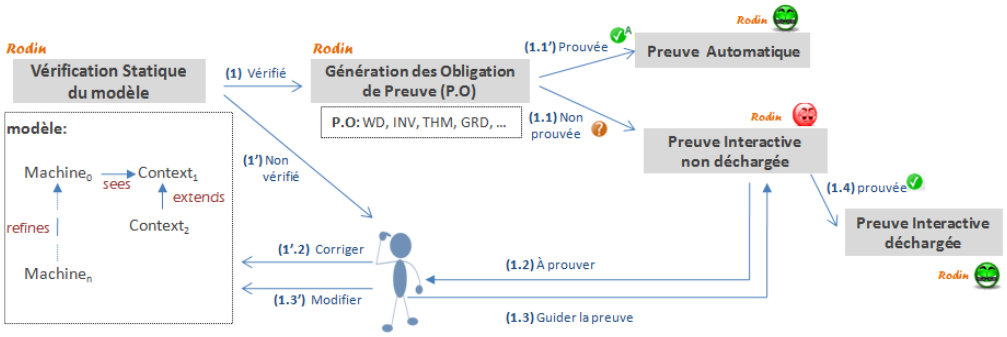

FIGURE 2.3 – Processus de vérification avec Rodin

techniques de spécifications supportant une approche modulaire réutilisable. Le principe de fonctionnement de ce plugin, nommé "modularisation" est détaillé dans la section suivante.

2.5.1 Plugin utilisé : "Modularisation"

Le plugin "modularisation" [37] intègre à la plateforme *Rodin* un autre type de composant, appelé "module interface" ou "interface" tout court. Ce composant dispose de variables, invariants, et d'un ensemble d'opérations qui peuvent encapsuler de fonctionnalités génériques. La structure d'une interface est présentée dans la FIGURE 2.4. La consistance d'une interface est vérifiée par certaines obligations de preuves semblables à celles générées pour une machine à savoir, la faisabilité des actions et la préservation d'invariants. La particularité d'un tel composant est qu'il peut être spécifié séparément, puis composé avec le système principal au cours de son développement, tout en préservant ses preuves de correction. Cette composition est réalisée par une relation machine/interface qui s'ajoute au développement formel via la clause *uses* : une interface I peut être importée dans une machine M pour que M puisse utiliser et réaliser toutes les fonctionnalités offertes par I (M *uses* I). En d'autres termes, une instance de I est crée pour M. Cette instance peut

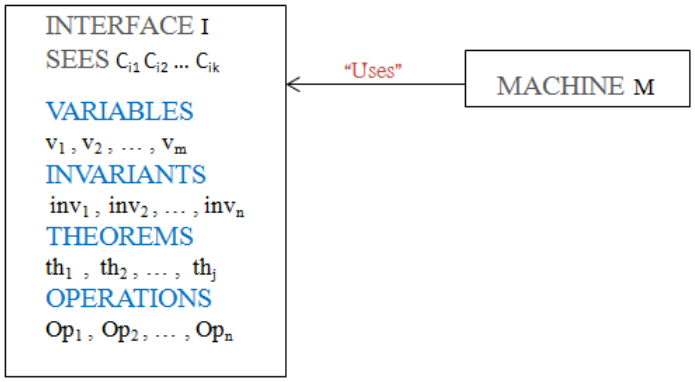

FIGURE 2.4 – Structure d'une "Interface"

être nommée par un préfixe MI précisé par l'utilisateur. Dans ce cas, les noms de tous les éléments de I, variables et opérations, sont préfixés dans M par MI.

L'importation de I dans M mène à une instanction directe des opérations de I via une *invocation* effectuée par un ou plusieurs évènements spécifié(s) dans M (TABLE 2.2) : une opération est caractérisée par des pré- et post-conditions. Les pré-conditions sont définies par une liste de prédicats appliquée sur des paramètres locales (p_o) à l'opération, et sur une liste de variables de I (nommée w). Elles spécifient les conditions de déclenchement de l'opération. Les post-conditions définissent les nouvelles valeurs attribuées à w, suite à cette opération. Outre ces mises à jour effectuées sur les variables, l'opération pourra aussi retourner un résultat *result* introduit dans la clause *return* et spécifié dans la clause *post*.

Comme montré dans la TABLE 2.2, l'instanciation de l'opération Op, d'une interface I, est spécifiée par un évènement $Calling\_Op$ d'une machine qui importe I : les gardes de l'évènement doivent satisfaire les conditions de déclen-

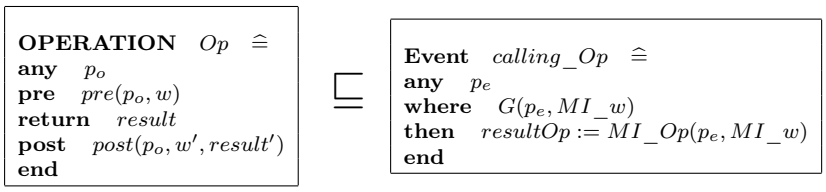

TABLE 2.2 – Instanciation d'une opération

chement de l'opération. Ces gardes $G(p_e, MI\_w)$ sont appliquées sur les variables w et les paramètres p_e de l'évènement. Ces derniers, paramètres et variables, doivent être correctement passés à l'opération. L'invocation se fait par une action de substitution qui remplace la valeur d'une variable $result\_Op$ par le retour d'appel de l'opération ($result\_Op := MI\_Op(p_e, MI\_w)$). Notons que $result\_Op$ est une variable qui doit être définie dans M et qui doit prendre le même type que $result$ qui a été défini dans I. Cet appel constitue une instanciation de l'opération, contrôlée par une certaine liste d'obligations de preuve. Une instanction correcte résulte une mise à jour directe sur les variables w, observée par l'évènement $Calling\_Op$.

2.6 Conclusion

La vérification du calcul distribué peut être assurée en utilisant de différentes techniques et approches qui ont fait preuves d'une grande efficacité dans la littérature. L'approche *Correct-par-Construction* permet de partager la complexité du développement et de raisonner progressivement sur les propriétés de correction du modèle. Ce type de raisonnement est contrôlé par les exigences et les principes de correction du raffinement. L'approche par composition et/ou décomposition apporte(ent) une certaine réutilisation de composants logiciels prouvés et une mise en oeuvre d'algorithmes existants. Une décomposition d'un problème en sous problèmes peut faire l'objet

d'une composition d'algorithmes traités auparavant pour apporter une solution correcte au problème initial. La réutilisation peut être amplifiée par la proposition de modules prouvés et facilement utilisés lors du développement sans avoir besoin d'en refaire la preuve.

Dans un premier volet de ce chapitre, nous avons donné un aperçu sur ces différentes approches et nous avons fixé celles que nous utilisons tout au long de nos travaux de thèse. L'approche Correct-par-construction constitue la base de tous nos contributions. Nous lui associons une composition locale d'algorithmes existants pour résoudre certains problèmes traités dans cette thèse. De plus, nous adoptant une approche modulaire pour généraliser la composition proposée pour résoudre le problème de détection de terminaison. Pour mettre en oeuvre ces approches, nous avons opté pour une vérification par *Theorem Proving* que nous avons introduit également dans ce chapitre.

Dans un deuxième volet, nous avons présenté l'environnement formel que nous utilisons pour supporter nos contributions. Il s'agit de la méthode Event-B et son assistant de preuve *Rodin*. Nous avons introduit les concepts clés d'un développement Event-B et nous avons traité l'exemple du chapitre précédent. Finalement, nous avons expliqué le processus de vérification et de preuve avec *Rodin*. Le chapitre suivant fait l'objet de notre première contribution, à travers laquelle nous traitons le problème de détection de terminaison du calcul distribué.

Chapitre

3 Approche de Composition : Détection de Terminaison Globale

Sommaire

3.1 Introduction

La terminaison du calcul est une propriété fondamentale qui garantit l'absence d'une suite d'exécution infinie. Néanmoins, il n'est pas facile de détecter cette propriété dans un contexte distribué. En effet, les algorithmes

distribués obéissent à un principe de fonctionnement en vertu duquel chaque entité fait son calcul avec une connaissance locale limitée à son voisinage et à quelques propriétés du réseau. En revanche, détecter une telle propriété est parfois nécessaire pour pouvoir exploiter un résultat obtenu et, le cas échéant, enchaîner sur un nouveau calcul.

Godard et al. [36] et Chalopin et al. [20] on étudié le la problème de la terminaison pour les systèmes distribués. Plus précisément, ils expliquent le principe de terminaison d'un système de ré-étiquetage et quels sont les processus qui pourraient la détecter. Dans ce cadre, les auteurs introduisent deux modes extrêmes de détection de terminaison : Détection de Terminaison Locale (LTD) et Détection de Terminaison Globale (GTD). Par définition, un algorithme distribué est dit terminé, si au bout d'un temps fini, tous les processus du réseau sont passifs et aucun message n'est en transit. Cette terminaison est dite implicite, si aucun processus ne peut la déduire à partir de son état local. Parfois, un processus peut détecter que le résultat global a été obtenu. Dans ce cas, la terminaison est dite explicite. Le mode LTD décrit une terminaison implicite et signifie que, dans une configuration finale, tout processus ne peut détecter que sa terminaison locale (i.e., son résultat est définitif). Le mode GTD fournit plus d'information que celui du LTD : dans ce mode, au moins un processus est averti de la terminaison globale du calcul.

Dans ce chapitre, nous proposons un patron formel qui permet de renforcer la propriété de terminaison des algorithmes définis en mode LTD. Plus précisément, nous visons à transformer ces algorithmes afin de permettre aux processus de détecter une configuration finale du système. L'objectif principal est de certifier cette transformation tout en gardant l'exactitude et la correction de l'algorithme en question.

Dans la suite de ce chapitre, nous présentons d'abord un aperçu sur les principes de détection de terminaison globale basés sur certains algorithmes de la littérature. Ensuite, nous détaillons le principe de notre approche. Enfin, nous enchaînons avec l'exemple du calcul d'arbre recouvrant.

3.2 Principes de Détection de Terminaison

La structuration des processus est l'une des solutions envisageables pour détecter la terminaison globale du calcul distribué. Il s'agit de parcourir les processus selon une topologie particulière tout en vérifiant la propriété de terminaison. D'autres algorithmes ont été aussi proposés comme solution pour une détection globale de terminaison à savoir Dijkstra-Scholten [27], Misra [53] et SSP (Szymanski, Shy, et Prywes) [58].

3.2.1 Dijkstra-Scholten/Misra

Dijkstra-Scholten [27] est une solution reconnue pour la détection de terminaison d'un calcul diffusant qui démarre à partir d'un unique sommet. Il s'agit de maintenir un arbre dynamique contenant tous les sommets actifs et leurs descendants. Un processus dans l'arbre expédie tout message reçu vers ses fils, et attend à ce que chaque fils lui ait envoyé une réponse. Chaque processus émet un message lorsqu'il en a reçu un. Ce message constitue une sorte d'acquittement. Un processus qui devient à la fois passif et feuille quitte l'arbre. À la fin du calcul, l'arbre sera réduite à un unique processus p. Quand p devient passif, il peut déduire la terminaison globale de l'algorithme. Malgré son efficacité pour une détection globale de terminaison, l'algorithme de Dijkstra-Scholten présente une contrainte forte qui consiste à lancer le calcul sur un unique sommet n'ayant pas de prédécesseurs dans le graph.

Certains travaux présentent des algorithmes en mode GTD à la manière de Dijkstra-Scholten. Le travail proposé par V. Filou et al. [31] semble celui qui se rapproche le plus de notre objectif. Les auteurs ont présenté une formalisation en Event-B qui permet de composer le calcul d'arbre recouvrant avec l'algorithme de Dijkstra-Scholten afin de progresser vers un mode GTD. La démarche suivie par les auteurs se résume dans l'application de la règle suivante : chaque sommet de l'arbre recouvrant est étiqueté par son degré d. À chaque pas de calcul, une feuille de l'arbre est élaguée, et le degré de son voisin est décrémenté. Le seule sommet non élagué détecte la terminaison globale du calcul de l'arbre recouvrant.

L'algorithme de Misra [53] exige une hypothèse sur le séquencement des messages selon la méthode FIFO. Son principe consiste à faire visiter les processus par un jeton qui les colorie selon leurs états actif ou passif. Le jeton véhicule une valeur j permettant de mémoriser le nombre de traversées dans lesquelles les processus visités sont restés passifs en permanence. Ainsi, le jeton détectera la terminaison lorsque cette valeur atteint la taille du réseau.

Un algorithme plus simple a été proposé par Szymanski, Shy, et Prywes [58], nommé SSP tout court. Aucune connaissance de la topologie du réseau n'est requise. En revanche, l'algorithme suppose que le réseau est fortement connexe, et une borne supérieure sur le diamètre est connue par chacun des processus. E. Godard et al. [35] ont proposé une approche de composition entre deux systèmes de réécritures de graphes R_A et R_T. Le premier système est un système quelconque qui est défini en mode LTD et que nous souhaitons le transformer en mode GDT. R_T est un système de réécriture de graphes traduisant le fonctionnement d'un algorithme de terminaison tel que celui de SSP. Le principe de cette composition consiste à appliquer le produit cartésien $R_A \times R_T$, tout en ajoutant un ensemble de contextes interdits

précisant la condition de terminaison locale d'un sommet. Il est clair que cette transformation autorise la modification des règles de réécritures de R_A. Par conséquent, elle ne préserve pas la preuve de correction de l'algorithme dont nous souhaitons détecter sa terminaison globale, ce qui n'est pas le cas dans notre travail.

Il est bien approuvé que SSP peut être appliqué sur divers algorithmes tout en respectant l'hypothèse de la connaissance initiale du diamètre et de la connexité du réseau [21]. L'algorithme présente un ensemble de règles qui pourraient facilement être appliquées aux modèles des calculs locaux. Ci dessous nous expliquons le principe de fonctionnement de cet algorithme, qui constitue le fondement de base de notre approche.

3.2.2 Le principe de SSP

SSP [58] est conçu pour détecter la terminaison globale d'un autre algorithme distribué, dans lequel chaque processus est capable de savoir s'il a terminé sa tâche. Nous considérons un algorithme distribué qui termine à partir du moment où chaque processus a atteint sa condition locale de terminaison. Soit A un algorithme qui se déroule sur un graph G. À chaque sommet v de G est associé :

- un prédicat $P(v)$, initialisé à $False$, et qui précise si le sommet a atteint sa terminaison locale.
- un entier $a(v)$, initialisé à -1 et qui correspond au rayon de confiance du sommet v. Plus précisément, il défini la distance, par rapport à v, et jusqu'à laquelle tous les processus ont terminé leurs tâches locales.

Le principe de SSP est le suivant : si v atteint sa condition locale de terminaison par rapport à l'algorithme A, alors $P(v)$ devient $True$. Nous supposons que cette valeur reste $True$ jusqu'à la fin de l'exécution de l'algorithme. Soit

$\{v_1, v_2, ..., v_d\}$ l'ensemble des voisins de v ; d étant le degré de v. La modification de $a(v)$ dépend des rayons de confiances associés aux voisins de v. Cette modification est traduite par les règles suivantes :

- Si $P(v) = False$ alors $a(v) = -1$
- Si $P(v) = True$ alors $a(v) = 1 + Min\{a(v_k)|0 \leq k \leq d\}$.

Il suffit que la valeur $a(v)$ atteigne le diamètre du réseau pour conclure, localement, que le prédicat est satisfait sur tous les processus. Par conséquent, la terminaison globale est détectée. Dans ce qui suit, nous présentons un exemple d'application de SSP sur le calcul d'arbre recouvrant.

Nous introduisons dans la section suivante le principe de notre approche, qui consiste à développer un patron formel spécifiant le fonctionnement de SSP. Ce patron pourrait être composé avec tout algorithme détectant la terminaison locale, et ayant besoin de satisfaire la propriété de détection de terminaison globale. La particularité de notre patron réside dans son incorporation dans un développement basé sur l'approche *correct-par-construction*, tout en préservant son exactitude et sa correction.

3.3 Composition SSP pour une Transformation d'Algorithmes

Notre idée de base consiste à spécifier formellement les règles SSP pour qu'elles puissent être composées avec tout algorithme A qui respecte les hypothèses suivantes : (i) A est correcte, (ii) A est défini en mode LTD, et (iii) A s'applique sur un graphe connexe, simple et non orienté. Admettons que l'algorithme A peut être spécifié par une chaîne de raffinement selon l'approche *correct-par-construction*. Nous supposons que chacun des processus ait une connaissance locale du diamètre du réseau. Une composition des

règles SSP avec l'algorithme A le transformera du mode LTD vers le mode GTD. Dans la FIGURE 3.1, nous présentons le formalisme de transformation : notre approche se base essentiellement sur deux éléments, le patron et son incorporation au sein d'un processus de Raffinement. Nous schématisons le patron par un module que nous appelons $SSP\_Interface$ pour voir où il se situe par rapport à l'algorithme initial, et comment nous pouvons tirer profit de la correction de SSP ainsi que du calcul de A :

- ***Le patron : SSP_ Interface*** : il contient une spécification modulaire permettant d'encapsuler les règles et les preuves de SSP. La particularité de cette unité logique est qu'elle est spécifiée et prouvée sous forme d'un patron dédié à être utilisé dans un contexte local associé à un algorithme A. Cela permet de simplifier le calcul et contribuer à la preuve de sa correction.
- ***La chaîne $Machine_0$...$Machine_n$:*** elle représente un développement incrémental de l'algorithme A. Cette chaîne est guidée par la technique du raffinement qui constitue le fondement de base de l'approche *correct-par-construction*
- ***$Machine_n$*** : la preuve de correction de A est préservée jusqu'au dernier niveau spécifié par la $Machine_n$. Toutes les règles de réécritures sont introduites dans ce niveau afin de décrire les changements locaux des états des sommets dans le graphe.
- ***$Machine_{n+1}$*** : elle raffine la $Machine_n$ et *utilise* le $SSP\_Interface$, ce qui signifie qu'elle préserve la correction de l'algorithme A et réutilise les règles et les preuves associées au $SSP\_Interface$.
- ***Graph* et *Graph'*** : $Graph$ définit nos hypothèses ainsi que les propriétés de base du réseau. Dans notre cas, nous considérons un réseau représenté par graphe simple et connexe. $Graph$ est un contexte de base

utilisé pour la spécification de notre patron. En revanche, d'autres propriétés peuvent être ajoutées dans $Graph'$ pour introduire plus de spécificités au contexte de l'algorithme A. Ces propriétés pourront réduire la topologie du réseau et définir d'autres hypothèses complémentaires aux hypothèses principales.

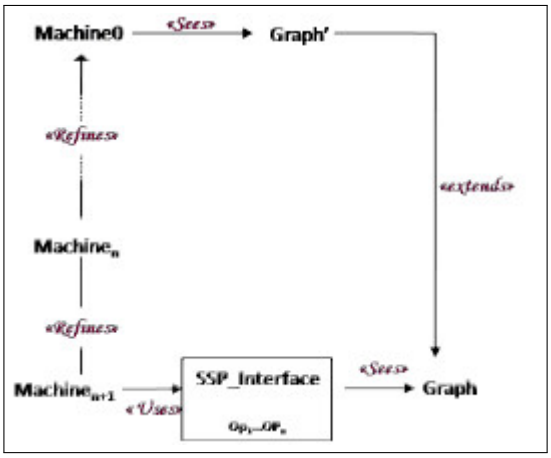

FIGURE 3.1 – Approche de Composition SSP pour une Transformation GTD

3.4 Spécification et Preuves de la Transformation

3.4.1 Le Contexte *Graph*

Dans ce contexte, nous présentons une spécification formelle des propriétés de base d'un réseau sur lequel les algorithmes doivent fonctionner. Nous précisons que cette spécification est présentée par Jean-Raymond Abrial et al. [4]. Formellement, les sommets du graphe désignent l'ensemble des processus, nommé ND. Les arêtes correspondent aux liens de communications, défini par g. Un réseau est modélisé par un graphe connexe ($axm6$), sans nœuds

isolés ($axm3$), non-orienté ($axm4$), et simple sans arêtes multiples ni boucles ($axm2, axm5$). Un graphe est connexe, si et seulement si, pour chaque paire de sommets, il existe un ensemble d'arêtes qui les relient ($axm6$)

$axm1 : finite(ND)$
$axm2 : g \subseteq ND \times ND$
$axm3 : dom(g) = ND$
$axm4 : g = g^{-1}$
$axm5 : ND \lhd id \cap g = \varnothing$
$axm6 : \forall s \cdot s \subseteq ND \wedge s \neq \varnothing \wedge g[s] \subseteq s \Rightarrow ND \subseteq s$

Soit *chains* l'ensemble des chaines possibles dans le graphe. Nous désignons par *diameter* le nombre de sommets qui passent par la chaine la plus longue du graphe, nommé *Longuestch*.

$axm7 : diameter \in \mathbb{N}$
$axm8 : Chains = \{A, p1, p2 \cdot A \subseteq C \wedge p1 \neq p2 \wedge p1 \in dom(A) \wedge$
$p2 \in ran(A) \wedge (\forall a \cdot a \in A \wedge prj2(a) \neq p2 \Rightarrow prj2(a) \in dom(A)) | A\}$
$axm9 : Nodes \in Chains \rightarrow \mathbb{P}(P)$
$axm10 : \forall ch \cdot ch \in Chains \Rightarrow Nodes(ch) = ran(ch) \cup dom(ch)$
$axm11 : Longestch \in Chains$
$axm12 : \forall ch2 \cdot ch2 \subseteq Chains \Rightarrow card(ch2) \leq card(Longestch)$
$axm13 : diameter = card(Longestch)$

3.4.2 Le Module SSP_Interface

$SSP\_Interface$ est un patron formel basé sur le haut niveau d'abstraction de Event-B ainsi que de SSP. Nous supposons que ND représente l'ensemble des sommets sur lesquels s'exécutera un algorithme avec une détection de terminaison locale. Les sommets doivent disposer de nouvelles étiquettes qui représentent des nouvelles variables externes :

– un prédicat $LocalTD \in ND \rightarrow BOOL$: il permet de caractériser la convergence du sommet par rapport à l'algorithme A. Soit n un sommet, $LocalTD(n) = TRUE$ signifie que n a localement terminé et a détecté cette terminaison. Initialement, aucun sommet ne détecte sa terminaison locale.

– un rayon de confiance $counter \in ND \rightarrow \mathbb{P}(\mathbb{N} \times \mathbb{Z})$: pour un sommet n, $counter(n)$ permet de stoker le nombre de voisins qui ont convergé par rapport à l'algorithme A durant les étapes de calculs sur n. Soit i une étape de calcul, $(i \mapsto j) \in counter(n)$ signifie qu'à l'étape de calcul i, tous les sommets, qui sont distants de n d'une valeur inférieur ou égale à j, ont localement terminé. Initialement, $counter(n) = \{0 \mapsto -1\}$. Après i étapes de calculs, $counter(n) = \{0 \mapsto -1, 1 \mapsto 0, ..., i \mapsto j\}$.

– un prédicat $GlobalTD \in ND \rightarrow BOOL$: il permet de vérifier si un sommet a pris connaissance de la détection de terminaison globale de l'algorithme A.

Grâce au haut niveau d'abstraction de la technique de modularisation que fournit la méthode Event-B, les nouvelles étiquettes et règles définies sont spécifiées indépendamment de tout algorithme, et peuvent être composées avec l'algorithme auquel nous souhaitons ajouter la propriété de détection de terminaison globale. Notre patron peut être appliqué dès qu'une boule de centre n atteint une configuration finale par rapport à l'algorithme introduit en paramètre. Rappelons qu'une boule est définie par un sommet centre, ses voisins et les arêtes qui leur sont associées. Le processus de transformation est décrit par l'algorithme suivant. Notons $compteur_i(n)$ le compteur du sommet n à l'étape de calcul i. Notre objectif est de fournir une spécification à un niveau d'abstraction élevé qui pourrait être réutilisée et composée avec d'autres calculs distribués. Pour ce faire, nous proposons de spécifier les différentes

Algorithme 1 : Transformation du mode LTD vers un mode GTD

1 Soit B_G(n) une boule du graphe de centre n.
2 Si configuration de B_G(n) est finale dans A et $LocalTD(n) = False$
3 alors $LocalTD(n) = True$ et $counter_{i+1}(n) = 0$
4 Sinon Si $LocalTD(n) = True$ et $counter_i(n) < diameter$
5 $counter_{i+1}(n) = 1 + min\{counter_i(n_k) | k \in [0; k]\}$
6 Sinon Si $counter_i(n)$=diamètre
7 Alors $GlobalTD(n) = TRUE$
8 Si $GlobalTD(n) = FALSE$ et $\exists v$ dans B_G(n) tel que $GlobalTD(v) = True$
9 Alors $GlobalTD(n) = True$

étapes de l'algorithme ci-dessus par un ensemble d'opérations abstraites qui agissent directement sur les étiquettes. Ces opérations pourront être invoquées lors du développement d'une approche $correct - par - construction$.

Interface $SSP\_Interface$

SEES $Graph$

Variables

$LocalTD$

$counter$

$GlobalTD$

...

OPERATION $UpdateTermination(...)$

...

OPERATION $ToGlobalTermination(...)$

...

OPERATION $GlobalTermination(...)$

...

OPERATION $Diffusion(...)$

Une Première Opération : UpdateTermination Un sommet n étant dans un état terminal, met à jour son étiquette $LocalTD(n)$ et agit sur son compteur de la manière suivante : nous considérons que $counter(n)$ est la valeur calculée par n lors de sa dernière étape de calcul i. Formellement, nous retrouvons la valeur de i par $max(dom(counter(n)))$. La nouvelle valeur du $counter(n)$ est mémorisée à l'étape de calcul $i+1$ et définie par $counter'(n)$. Les conditions de déclenchement de l'opération sont définies pas des pré-conditions désignés par le mot clé *pre*. Il faut noter que le déclenchement de l'opération devrait être bloqué dès lors que n satisfait sont prédicat. Par ailleurs, nous exigeons le fait que $LocalTD(n)$ soit égale à $False$. Ainsi, $counter_i(n)$ devrait avoir la valeur -1. Dans notre cas, nous nous sommes intéressés par la mise à jour directe des variables externes ($counter'$ et $LocalTD'$ sans retour de résultats particuliers ($result' = True$).

OPERATION	$Update\_Termination \;\widehat{=}$
any	n, i
pre	$pre1 : LocalTD(n) = FALSE$
	$pre2 : i = max(dom(counter(n)))$
	$pre3 : max(ran(counter(n))) = -1$
return	$result1$
post	$post1 : LocalTD'(n) = TRUE$
	$post2 : counter'(n) = counter(n) \cup \{(i+1) \mapsto 0\}$
	$post3 : result1' = TRUE$

Une Deuxième Opération : ToGlobalTermination Une fois qu'un sommet n satisfait son prédicat $LocalTD(n)$, il calcule une nouvelle valeur de $counter(n)$: soit i la dernière étape de calcul dans laquelle n a modifié son compteur. La nouvelle valeur du $counter(n)$ est toujours définie à l'étape de calcul $i+1$, et dépend du dernier compteur calculé par les voi-

sins de n. Notons $Ng(n)$ l'ensemble des voisins de n. $Ng(n) = \{ng_1, ..., ng_d\}$. Soit $\{counter_j(ng_1), ..., counter_k(ng_d)\}$ l'ensemble des dernières valeurs mémorisées par chaque sommet figurant dans $Ng(n)$. Nous désignons par C l'ensemble des dernières valeurs calculées par les sommets de $(Ng(n) \cup \{n\})$. $C = \{counter_j(ng_1), ..., counter_k(ng_d), counter_i(n)\}$. La nouvelle valeur de $counter(n)$, notée par $counter'(n)$, est égale à l'ancienne valeur de $counter(n)$ que nous lui associons le couple $(i + 1) \mapsto (1 + min(C))$. Afin d'éviter un accroissement infini de $counter(n)$, le déclenchement de l'opération devrait être bloqué dès lors que cette valeur atteint le diamètre du graphe. En outre, notons que le calcul de $counter(n)$ sera inutile si l'un des voisins de n a déjà détecté la terminaison globale du calcul.

OPERATION $ToGlobalTermination \;\widehat{=}$

any n, i, Ng, C

pre

$pre1 : LocalTD(n) = TRUE$

$pre2 : max(ran(counter(n))) < diameter$

$pre3 : i = max(dom(counter(n)))$

$pre4 : Ng = \{vi \cdot vi \mapsto n \in g | vi\}$

$pre5 : C = \{a1, n1 \cdot n1 \in Ng \cup \{n\} \wedge a1 = max(ran(counter(n1))) | a1\}$

$pre6 : \forall vi \cdot vi \in Ng \cup \{n\} \Rightarrow GlobalTD(vi) = FALSE$

return $result2$

post $post1 : counter'(n) = counter(n) \cup \{(i + 1) \mapsto (1 + min(C)\}$

$post2 : result2' = TRUE$

Une Troisième Opération : GlobalTermination Au bout d'un temps fini, $counter(n)$ peut atteindre le diamètre du graphe. Dans ce cas, nous pouvons déduire que tous les sommets du graphe ont localement terminé. Par conséquent n peut détecter que le calcul global est complètement réalisé.

Ainsi, n peut satisfaire son prédicat $GlobalTD$ et utiliser cette information pour en informer ses voisins. Afin d'éviter des déclenchements infinis de cette opération, nous exigeons que $LocalTD(n)$ devrait être égale à $False$.

OPERATION	$GlobalTermination \ \widehat{=}$
any	n
pre	$pre1 : max(ran(counter(n))) = diameter$
	$pre2 : GlobalTD(n) = FALSE$
return	$result3$
post	$post1 : GlobalTD'(n) = TRUE$
	$post2 : result3' = TRUE$

Une quatrième Opération : Diffusion Quand un sommet n détecte la terminaison globale du calcul, il transmettra cette information à ses voisins en agissant sur leurs attributs $GlobalTD$. Formellement, nous spécifions la mise à jour des étiquettes des voisins de n par une relation *override* entre l'ensemble des étiquettes associées à tous les sommets du graphe $GlobalTD$ et les nouvelles étiquettes des voisins de n ($\{v \cdot v \in g[\{n\}] || v \mapsto TRUE\}$). En effet, une relation d'*override* entre deux ensembles r et s est définie par : $r \leftarrowtail s = s \cup \{x \mapsto y | x \mapsto y \in r \wedge x \notin dom(s)\}$. Chaque voisin pourrait, à son tour, transmettre les mêmes informations. Le déclenchement de cette opération doit être bloqué lorsque tous les voisins de n détectent la terminaison globale. Par conséquent, avant de déclencher la présente opération, il faut vérifier que n ait au moins un voisin ne satisfaisant pas son prédicat $GlobalTD$.

OPERATION $Diffusion \ \widehat{=}$

any n, Ng

pre $pre1 : GlobalTD(n) = TRUE$

$pre2 : Ng = \{v \cdot v \in g[\{n\}]|v\}$

$pre3 : \exists v \cdot v \in Ng \wedge GlobalTD(v) = FALSE$

return $result4$

post $post1 : GlobalTD' = GlobalTD \leftarrow \{v \cdot v \in g[\{n\}]|v \mapsto TRUE\}$

$post2 : result4' = TRUE$

3.4.3 Preuves de Détection de Terminaison Globale

L'objectif de notre travail est de fournir un patron formel qui permet de transformer des algorithmes d'un mode LTD en un mode GTD, en utilisant SSP. Cette transformation est assurée par la composition de notre patron et l'utilisation directe de ses opérations abstraites. Pour assurer le bon fonctionnement de l'algorithme résultat, nous devons montrer que ces opérations doivent certainement prouver la détection de terminaison globale. Dans ce qui suit, nous étudions les invariants du modèle et nous prouvons les propriétés suivantes :

(P1) : l'algorithme issu de la transformation doit maintenir la propriété LTD de l'algorithme initial ($Thorme1$).

Théorème 3.1. $\forall n \cdot GlobalTD(n) = TRUE \Rightarrow LocalTD(n) = TRUE$

Il est bien évident que, pour chaque sommet n, $counter(n)$ accroît d'une étape de calcul à une autre ($Invariant1$). Ainsi, quand la valeur de $counter(n)$ devient égale à zéro, elle ne décroît plus et reste positive tout au long des prochains calculs sur n. Par conséquent, n détecte toujours sa terminaison locale.

Invariant 3.1. $\forall n, i, a, i', a' \cdot i' < i \wedge (i \mapsto a) \in counter(n) \wedge i' \mapsto a' \in counter(n) \Rightarrow a' < a$

Invariant 3.2. $\forall n \cdot GlobalTD(n) = TRUE \Rightarrow (max(ran(counter(n))) \geq 0)$

De plus, il est clair que, pour chaque sommet n qui détecte la terminaison globale, le dernier compteur mémorisé par n est une valeur positive, i.e., $(max(ran(counter(n)))$ (Invariant 3.2)). Rappelons aussi que si n détecte localement sa terminaison, l'opération $UpdateTermination$ se déclenche et met à zéro la valeur de $counter(n)$. Par conséquent, nous prouvons facilement qu'un sommet détecte sa terminaison locale si et seulement si la valeur du dernier compteur calculé par ce sommet est positive (Invariant 3.3).

Invariant 3.3. $\forall n \cdot LocalTD(n) = TRUE \Leftrightarrow (max(ran(counter(n))) \geq 0)$

(P2) Si un sommet n satisfait son prédicat $GlobalTD(n)$, alors la terminaison locale devrait être vérifiée sur tous les sommets du graphe (Théorème 3.2).

Théorème 3.2. $(\forall n \cdot GlobalTD(n) = TRUE) \Rightarrow (\forall v \cdot v \in ND \Rightarrow LocalTD(v) = TRUE)$

En effet, un sommet qui détecte sa terminaison locale engendre le déclenchement de l'opération $ToGlobalTermination$ et agit directement sur son compteur. La nouvelle valeur calculée dépend des valeurs associées aux voisins. Nous prouvons que la différence entre le maximum et le minimum du dernier compteur calculé par deux voisins ne dépasse pas 1 (Invariant 3.4).

Invariant 3.4. $\forall n, v, a, b \cdot v \in g[\{n\}] \wedge a = max(ran(counter(n))) \wedge b = max(ran(counter(v))) \Rightarrow (max(\{a, b\}) - min(\{a, b\}) \leq 1)$

En effet, soit $Chains$ l'ensemble des chaînes possibles spécifié dans la Section 3.4.1. $Nodes(ch)$ désigne l'ensemble des sommets concernés dans une chaîne ch. Nous désignons par E, l'ensemble des derniers compteurs calculés par les sommets de $Nodes(ch)$. $E = \{n{\cdot}n \in Nodes(ch) | max(ran(counter(n)))\}$. Nous notons par $Max - Min$, la différence entre le maximum et minimum des valeurs de E. Nous prouvons, par induction sur la taille de ch, que $Max - Min \leq card(ch) - 1$ (Invariant 3.5).

Invariant 3.5. $\forall ch, N{\cdot}ch \in chains \wedge N = Nodes(ch) \Rightarrow max(\{n{\cdot}n \in N |$
$max(ran(counter(n)))\}) - min(\{n{\cdot}n \in N | max(ran(counter(n)))\})$
$\leq card(N) - 1$

En outre, nous rappelons que le diamètre du graph correspond à la plus grande distance entre deux sommets. Ainsi, $card(ch) \leq card(diameter)$, d'où $Max - Min \leq card(diameter)$. De plus, nous rappelons que le dernier compteur calculé par un sommet, détectant la terminaison globale, atteint certainement la taille du diamètre. Dans ce cas, la valeur maximale de l'ensemble E prendra la valeur de $card(diameter)$. Par conséquent, nous prouvons que dans ce cas, $Min \geq 0$. En d'autre terme, si un sommet détecte la terminaison globale, alors tous les autres sommets ont localement terminé (Theorem 3.2).

3.5 Exemple d'application

Nous reprenons dans cette section l'exemple du calcul d'arbre recouvrant [14] que nous avons présenté au premier chapitre (1.4.2). L'objectif principal est d'expliquer comment le patron $SSP\_Interface$ pourrait être composé avec le calcul initial afin d'ajouter la propriété de détection de terminaison globale. Nous rappelons qu'une étape de calcul est définie par une application de la règle de réécriture $R1$: un sommet étiqueté A peut activer l'un de ses

voisins qui sont à l'état *neutre* et il marque l'arête qui lui est associée. L'arbre recouvrant calculé est le chemin portant toutes les arêtes marquées.

Comme nous l'avons expliqué au chapitre 2, l'algorithme est spécifié sur différents niveaux d'abstraction. Après avoir effectué une suite de raffinements successives (M_0, M_1, et M_2), nous obtenons un niveau concret dans lequel nous spécifions la règles de réécriture $R1$. Ce niveau correspond à la $Machine_n$ de la FIGURE 3.1.

Le principe de composition de notre patron est le suivant : en première étape, nous spécifions une nouvelle machine M_3 qui *raffine* M_2 et *utilise* $SSP\_Interface$. Cette relation entre machine et Interface permet d'utiliser les variables externes du patron et déclencher ses *opérations*.

Soit $LabN$ la fonction introduite en M_2 pour spécifier les étiquettes possibles des sommets : $LabN \in ND \rightarrow LN$ avec $LN = \{A, N\}$. Dans la machine M_3, nous spécifions la condition de terminaison locale d'un sommet pour pouvoir enchainer avec les opérations abstraites du patron. Soit v un sommet de l'arbre. v termine localement quand il est étiqueté A, et aucun de ses voisins n'est dans un état *neutre* ; son voisinage porte également l'étiquète A. Formellement, nous devons ajouter l'invariant suivant :

Invariant 3.6. $\forall n \cdot n \in ND \wedge LabN[\{n\}] = \{A\} \wedge g[\{n\}] \not\subseteq LabN^{-1}[\{N\}] \Leftrightarrow LocalTD(n) = TRUE$

En deuxième étape, Le calcul de $SSP\_Interface$ est spécifié par des événements introduits à la machine M_3. Ces événements expriment à travers leurs gardes les pré-conditions de déclenchement des opérations et à travers leurs actions l'appel de l'opération concernée. Notons que le déclenchement de l'opération $UpdateTermination$ doit être spécifié à notre algorithme. Nous devons certainement ajouter un nouveau garde (grd4) pour spécifier la condition sous laquelle un sommet satisfait son prédicat $LocalTD$.

EVENT $Calling\_Op1$
any
n, i
where
$grd1 : LocalTD(n) = FALSE$
$grd2 : i = max(dom(counter(n)))$
$grd3 : max(ran(counter(n))) = -1$
grd4 : $Lab\_N[\{n\}] = \{A\}$
$\wedge g[\{n\}] \not\subseteq Lab\_N^{-1}[\{N\}]$
then
act1 : $res\_OP1 :=$
$Update\_Termination(n \mapsto i)$

EVENT $Calling\_Op2$
any
n, Ng, C, i
where
$grd1 : LocalTD(n) = TRUE$
$grd2 : max(ran(counter(n))) < card(ND)$
$grd3 : i = max(dom(counter(n)))$
$grd4 : Ng = \{vi \cdot vi \mapsto n \in g | vi\}$
$grd5 : C = \{a1, n1 \cdot n1 \in Ng \cup \{n\}$
$\wedge a1 = max(ran$
$(counter(n1)))|a1\}$
$grd6 : \forall vi \cdot vi \in Ng \cup \{n\}$
$\Rightarrow GlobalTD(vi) = FALSE$
then
act1 : $res\_OP2 :=$
$To\_Global\_Termination\_Detection$
$(n \mapsto V \mapsto C \mapsto i)$

EVENT $Calling\_Op3$
any n
where
$grd1 : max(ran(counter(n)))$
$= card(ND)$
$grd2 : GlobalTD(n) = FALSE$
then
$act1 : res\_OP3 :=$
$Global\_Termination\_Detection(n)$

EVENT $Calling\_Op4$
any n, Ng
where
$grd1 : GlobalTD(n) = TRUE$
$grd2 : Ng = \{v \cdot v \in g[\{n\}]|v\}$
$grd3 : \exists v \cdot v \in Ng \wedge GlobalTD(v)$
$= FALSE$
then
$act1 : res\_OP4 :=$
$Diffusion(n \mapsto V)$

3.6 Conclusion

Dans ce chapitre, nous avons traité le problème de détection de terminaison des algorithmes distribués. Basé sur le haut niveau d'abstraction de SSP, nous avons proposé un patron formel qui pourrait être composé avec d'autres algorithmes LTD afin de les transformer en des algorithmes GTD [16]. Bien qu'une telle transformation est mise en œuvre avec l'assistant de preuve Coq [33], le principal avantage de notre travail consiste à préserver,

d'une part, la preuve de correction de l'algorithme initiale et de réutiliser d'autre part, la preuve de la transformation. Nous avons illustré l'exemple du calcul d'arbre recouvrant pour montrer le principe de composition de notre patron [14]. L'idée de base consiste à spécifier la technique de transformation par des opérations abstraites prouvées dans le cadre d'une approche modulaire, et les incorporer dans une approche $correct - par - construction$. Cette combinaison nous garantit l'exactitude de l'algorithme et la réutilisation des preuves associées à la détection de terminaison globale.

Dans le chapitre suivant, nous traitons un autre problème du distribué, qui consiste à énumérer les sommets dans un graphe. Nous nous référons à l'algorithme de Mazurkiewicz [48] pour fournir une preuve formelle détaillée par une succession de raffinements. L'étude de ces deux problèmes : l'énumération et la détection de terminaison globale, nous sera utile pour traiter un autre problème du distribué présenté dans le dernier chapitre de la thèse. Il s'agit du calcul des snapshots locaux et de l'état global dans un réseau anonyme.

Chapitre

4 Algorithme d'Énumération : Preuves par Raffinements

Sommaire

4.1 Introduction

L'identification des entités dans un système distribué est parfois une condition indispensable pour un fonctionnement correct de divers algorithmes et applications. Si de tels identifiants ne sont pas définis, le réseau est dit anonyme. La mise en oeuvre d'un système de *nommage* est l'un des problèmes

fondamentaux du distribué : il s'agit d'attribuer des identifiants uniques aux processus du système. Toutefois, il est difficile de prouver l'unicité d'identifiants dans un système distribué sans horloge commune ni connaissance de l'état global. La façon dont cette attribution d'identifiants est effectuée joue un rôle important dans l'efficacité du système.

La mise en oeuvre d'un système de *nommage* permettrait un meilleur routage d'informations, une meilleure gestion de ressources et de performances, etc. Par exemple, la diffusion de l'information pourrait être effectuée à partir d'un sommet distingué, tout en minimisant le nombre de messages utilisés. Dans certains modèles, une solution du problème de *nommage* pourrait servir à résoudre le problème d'élection. En effet, Le but d'un algorithme d'élection est de distinguer un sommet qui est dans un état *Èlu* alors que tous les autres sommets sont dans un état *non-Èlu*. L'identification des sommets nous permettrait d'effectuer une élection directe sur le sommet qui a le plus petit/grand identifiant.

Les systèmes admettant une solution pour le problème de *nommage* sont largement étudiés pour divers modèles ayant différentes caractéristiques : la topologie du réseau, le type de calcul, la connaissance initiale requise par les processus, etc. Si le problème est solvable pour certains modèles, il est indispensable de parler de la correction et de l'exactitude des solutions. Par définition [63], un unique identifiant doit disposer des caractéristiques suivantes : un identifiant se réfère à au plus un même processus, et chaque processus est désigné par au plus un identifiant. L'énumération est une variante du problème de *nommage* : admettons un graphe G, le but d'un algorithme d'énumération est d'attribuer un entier à chaque sommet de G. Dans la configuration finale, l'ensemble des numéros des sommets de G doit correspondre exactement à l'intervalle $1..|V(G)|$, où $|V(G)|$ désigne le nombre de sommets

dans G. En revanche, il est difficile de garantir que deux processus distincts ne choisissent pas le même numéro.

Dans ce chapitre, nous présentons une preuve détaillée d'un algorithme d'énumération. Plus précisément, nous sommes intéressés par le modèle de *Mazurkiewicz* [48], en ce qu'il a pour effet d'appliquer des relations de réétiquetage localement engendrées sur les étoiles (boules), et d'utiliser uniquement les connaissances locales de chaque sommet. Notre objectif est de fournir un schéma de preuve guidé par des raffinements successifs. Nous pensons aussi qu'un raisonnement progressif d'un calcul distribué aussi complexe, aiderait à mieux comprendre le comportement théorique de l'algorithme.

Dans la suite de ce chapitre, nous expliquons d'abord, le principe de calcul de l'énumération proposée par *Mazurkiewicz*. Ensuite, nous proposons un processus de développement incrémental composé de différents niveaux d'abstractions. Nous nous Enfin, nous détaillons progressivement la spécification et la preuve de chaque niveau : nous commençons la vérification globale de l'algorithme jusqu'à prouver sa correction d'un point de vue locale.

4.2 Algorithme d'énumération

Un algorithme d'énumération impose au système d'atteindre une configuration finale dépourvue de toute *similarité locale* entre les processus : admettons A et B deux réseaux, tel que B dispose d'un nombre inférieur de processus. A est dit *symétrique* s'il est schématiquement constitué de plusieurs copies entrelacées de B, et aucun processus ne peut savoir localement s'il se trouve au sein du réseau A ou du réseau B.

Formellement, l'existence de symétrie locale pourrait être capturée par la notion de revêtement utilisée dans le travail d'Angluin [8]. Par définition, un

graphe G est un revêtement d'un autre graphe G' s'il y a un homomorphisme surjectif de G vers G' qui est localement bijectif. Cet outil mathématique est par la suite utilisé dans certains travaux de la littérature lors de la définition des caractérisations de graphes pour divers modèles. Chaque modèle dispose d'un mode différent de synchronisation et applique une certaine forme de ré-étiquetage de graphes [64, 22, 23]. Ces caractérisations décrivent les graphes pour lesquels nous pouvons résoudre les problèmes qui nécessitent une brise de symétrie, à savoir l'énumération ou l'élection. Dans certains travaux tel que celui de *Chalopin et al.* [23], la résolution du problème d'énumération et d'élection est équivalente. Le modèle proposé par les auteurs est défini par l'utilisation de calculs locaux sur les arêtes étiquetées.

Le travail de *Mazurkiewicz* [48] considère des calculs locaux sur des étoiles qui sont définies comme des sous-graphes constitués d'un sommet s et de ses voisins. La modification de l'étiquette de s est effectuée dans l'étoile, suivant des règles dépendant de cette étoile uniquement. La caractérisation donnée par *Mazurkiewicz* pour le problème d'énumération est définie par des graphes qui sont minimaux pour les revêtements. En d'autres termes, il s'agit des graphes qui ne sont un revêtement que d'eux même.

Énumération Mazurkiewicz Le processus d'énumération est décrit par des règles qui expliquent comment un sommet choisit un numéro, et comment cette information se propage sur les autres sommets. Ces étapes de calcul sont définies par les règles de renommage et de diffusion expliquées ci-dessous : chaque sommet x essaie d'obtenir une identité unique. Dès qu'il choisit un numéro $n(x)$, il l'envoi à chaque voisin y avec le port de sortie p par lequel $n(x)$ est envoyé. Quand y reçoit le message, il mémorise $n(x), p$ et le port d'entrée q, par lequel y a reçu le message. À partir des informations

reçues de ses voisins, chaque sommet x construit sa vue locale $Lv(x)$. Il s'agit des numéros des voisins de x avec les numéros de ports correspondants. L'algorithme repose sur un ordre total défini sur les vues locales des sommets. En effet, si x s'aperçoit qu'il existe un autre sommet y avec le même numéro alors il compare sa vue locale $Lv(x)$ avec celle de y. Si $Lv(x)$ est plus faible ($<$), x change de numéro et le diffuse accompagné de sa vue locale. À partir des vues locales reçues, chaque sommet y peut construire une mémoire $M(y)$. Elle contient les numéros des voisins de y et leurs vue locales. Suite à chaque changement de numéro, des mises à jour doivent être effectuées sur certaines vues locales et mémoires. Initialement, pour tout sommet x, $n(x) = 0$, $Lv(x) = \varnothing$, et $M(x) = \varnothing$. Les valeurs de $n'(x)$, $M'(x)$ et $Lv'(x)$ désignent respectivement les nouvelles valeurs attribuées à $n(x)$, $M(x)$ et $Lv(x)$.

– Règle de renommage (R_1) : si x n'a pas modifié son numéro, ou bien il s'aperçoit qu'il a dans sa mémoire un couple $(n(x), N)$ tel que sa vue locale $Lv(x)$ est plus faible que N ($Lv(x) < N$), alors il décide de changer son numéro. Cette règle de renommage doit être déclenchée sous la condition que tous les sommets de la boule de centre x disposent de la même mémoire. En d'autres termes, toutes les mémoires sont mises à jour. Plus formellement, cette règle est définie par les pré et post-conditions suivantes :

  Pré-conditions :

  $\forall y \in B(x), M(y) = M(x)$

  $(n(x) = 0) \vee (n(x) > 0 \wedge \exists (n(x), Lv') \in M(x) | Lv(x) < Lv')$

  Post-conditions :

  $n'(x) = 1 + max\{n | (n, Lv) \in M(x)\}$

  $\forall y \in B(x) \setminus \{x\}, Lv'(y) = (Lv(y) \setminus \{n(x)\}) \cup \{n'(x)\}$

$\forall y \in B(x), M'(y) = M(y) \cup \{n'(w), Lv'(w))|w \in B(x)\}$

– Règle de diffusion (R_2) : une fois qu'un sommet d'une boule $B(x)$ change de mémoire, une mise à jour doit être effectuée sur $B(x)$: la mémoire de chaque sommet de la boule prend l'union de tous les mémoires des sommets de $B(x)$. La règle est formalisée comme suit :

Pré-conditions : $\exists y \in B(x), M(y) \neq M(x)$

Post-conditions : $\forall y \in B(x), M'(y) = \bigcup_{w \in B(x)} M(w)$

4.3 Un Développement Incrémental

Dans cette section, nous expliquons le processus de raffinement que nous avons suivi pour résoudre le problème d'énumération (FIGURE 4.1). Nous nous inspirons du travail effectué lors du projet RIMEL[1]. D'abord, nous spécifions les propriétés du graphe sur lequel nous souhaitons calculer une énumération. Nous supposons que chaque sommet ait une connaissance de la taille du réseau. De plus, étant donné que l'algorithme de *Mazurkiewicz* [48] termine sur des graphes minimaux pour les revêtements, il est indispensable de définir une hypothèse sur la minimalité du graphe.

Ensuite, nous spécifions un résultat que nous pourrons observer globalement, sans aucune connaissance des détails de calculs. Il s'agit d'une numérotation définie par une fonction bijective entre les les sommets $V(G)$ d'un graphe G et l'intervalle $1..|V(G)|$. Ce résultat global est spécifié, en un premier niveau de la spécification, par une machine abstraite $GLOBAL\_ENUM$.

En deuxième niveau, nous suivons progressivement la surjection sur l'intervalle $1..|V(G)|$. Nous montrons qu'à chaque étape de calcul, il y a un in-

1. rimel.loria.fr

tervalle d'entiers qui couvre tous les numéros de $V(G)$. Nous observons une extension progressive de cette plage de numéros jusqu'à atteindre l'intervalle résultat. Ainsi, la machine $GLOBAL\_PROGRESS\_ENUM$, introduite au deuxième niveau, dévoile la preuve de la surjectivité de l'énumération. Toutefois, l'injectivité est prouvée progressivement au fur et à mesure des raffinements qui suivent.

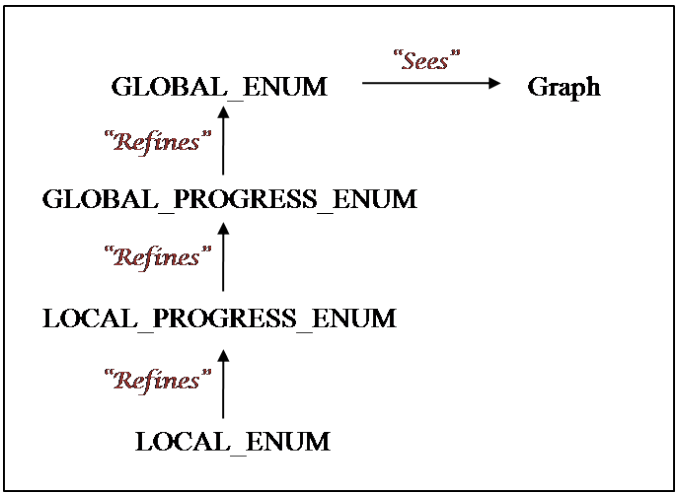

FIGURE 4.1 – Processus de Raffinement

En effet, dans un troisième niveau, nous introduisons une nouvelle machine $LOCAL\_PROGRESS\_ENUM$ qui raffine la machine du niveau précédent et introduit d'autres détails pour mieux observer l'injectivité de la numérotation. Dans ce niveau, nous montrons la preuve d'une numérotation unique par rapport aux visions locales des sommets. Ces visions sont limitées aux boules auxquelles les sommets appartiennent. La preuve de l'unicité des numéros par rapport à tous les sommets du graphe est repoussée au niveau suivant. Nous introduisons les raffinements nécessaires pour construire les vues locales des sommets et définir une abstraction de la règle de renommage.

Dans un dernier niveau, nous raffinons $LOCAL\_PROGRESS\_ENUM$

par une nouvelle machine $LOCAL\_ENUM$ pour vérifier une énumération correcte à travers des étapes de calculs effectuées localement sur les sommets. L'unicité des numéros est vérifiée grâce à la notion de mémoires qui fournit aux sommets une vision plus large. Ainsi, nous ajoutons les raffinements et les étapes de calculs nécessaires pour construire les mémoires et donner une spécification détaillée des règles de *renommage* et de *diffusion*. Nous prouvons les propriétés invariantes du modèle et celle qui doivent être satisfaites pour un étiquetage final.

4.4 Spécifications Formelles Détaillées

4.4.1 Spécification du Graphe

Nous gardons les mêmes propriétés définies dans le contexte *Graph* du chapitre précédent (Section 3.4.1) : un graphe g de sommets ND, connexe, simple, et non orienté. Toutefois, nous supposons que chaque sommet distingue ses voisins. Ainsi, nous ajoutons un étiquetage de ports sur le graphe (voir Définition 1.19). Formellement, nous utilisons d'autres constantes et propriétés ajoutées au contexte :

- $degré \in ND \to \mathbb{N}_1$. Pour tout sommet x, $degré(x)$ désigne le nombre de voisins de x. La spécification du $degré(x)$ est la suivante : $degré(x) = card(\{e | e \in g \wedge prj1(e) = x\})$.
- $ports \in ND \to \mathbb{P}(ND \times \mathbb{N}_1)$. Pour tout sommet x, $ports(x)$ appartient à un ensemble de fonctions bijectives f calculées entre le voisinage de x, noté V, et l'intervalle $1..degr(x)$ ($f \in V \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1..degré(x)$). Un exemple est illustré dans la FIGURE 4.2. Formellement, $ports(x)$ est spécifié comme suit : $ports(x) \in \{V, f \cdot V = \{y | x \mapsto y \in g\} \wedge f \in V \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1 \,..\, degré(x) | f\}$.

– $LE \in g \rightarrow \mathbb{N} \times \mathbb{N}$. Pour toute arête $x \mapsto y$, nous définissons un étiquetage, noté $LE(x \mapsto y)$, en fonction des numéros de ports i et j, à travers lesquels x et y sont reliés. Formellement, nous spécifions $LE(x \mapsto y)$ par les deux implications suivantes. Un exemple est illustré dans la FIGURE 4.2.

$\forall i, j \cdot x \mapsto i \in ports(y) \wedge y \mapsto j \in ports(x) \Rightarrow LE(x \mapsto y) = (i \mapsto j)$.

$\forall i, j \cdot LE(x \mapsto y) = (i \mapsto j) \Rightarrow y \mapsto j \in ports(x) \wedge x \mapsto i \in ports(y)$.

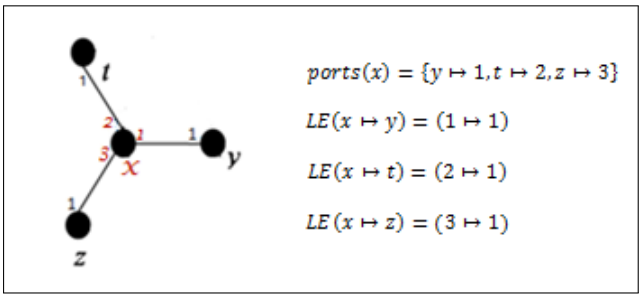

FIGURE 4.2 – Un graphe avec un étiquetage de ports

De plus, l'algorithme d'énumération termine si est seulement si le graphe dirigé correspondant à g est minimal pour les relations de revêtements. Théoriquement, pour toute arête $x \mapsto y$ appartenant à g, deux arêtes symétriques $x \mapsto y$ et $y \mapsto x$ sont construites pour le graphe dirigé de g (voir Définition 1.4.1). Étant donné que (1) pour tous sommets voisins x et y, les couples $x \mapsto y$ et $y \mapsto x$ figurent dans g, et (2) si $LE(x \mapsto y) = p \mapsto q$ alors $LE(y \mapsto x) = q \mapsto p$; nous faisons l'hypothèse de minimalité sur g.

Nous ajoutons l'axiome suivant pour vérifier que pour tout graphe D tel que g est un revêtement symétrique de D, g et D sont isomorphes. Nous rappelons que g est un revêtement de D s'il existe un homomorphisme surjectif $(\twoheadrightarrow)$ ϕ de ND vers les sommets de D et qui est localement bijectif : D est défini sur un ensemble de sommets inclus dans $\mathbb{N}$. La symétrie de D est représentée par une fonction Sym_D. La fonction LE_D désigne l'étiquetage de ports

sur les arcs de D. D est construit via ϕ comme suit : pour toute arête $x \mapsto y$ appartenant à g, $\phi(x) \mapsto \phi(y) \in D$ et $LE(x \mapsto y) = LE_D(\phi(x) \mapsto \phi(y)))$. Par conséquent, $LE(y \mapsto x) = LE_D(\phi(y) \mapsto \phi(x)))$. L'homomorphisme ϕ est localement bijectif signifie que pour tout sommet x, $\phi[g[\{x\}]] = D[\phi[\{x\}]]$ et pour tous sommets distincts $(y,\ z)$ appartenant à la boule de centre x, $\phi(z) \neq \phi(y)$. Le revêtement est symétrique si pour tout arête $x \mapsto y$ appartenant à g, $\phi(y) \mapsto \phi(x) = Sym_D(\phi(x) \mapsto \phi(y))$ (voir Définition 1.4.1). L'hypothèse de minimalité consiste à supposer que pour tout graphe D et tout homomorphisme ϕ qui satisferaient les propriétés ci-dessus, ϕ est bijective ($\rightarrowtail\!\!\!\!\!\twoheadrightarrow$).

$$
\begin{array}{l}
\forall \phi, D, N, LE_D, Sym_D \cdot \\
\left(\begin{array}{l}
N \subseteq \mathbb{N} \land D \subseteq N \times N \land LE_D \in D \rightarrow \mathbb{N} \times \mathbb{N} \land \phi \in ND \twoheadrightarrow N \land \\
Sym_D \in D \rightarrow D \land \\
(\forall n, m \cdot n \mapsto m \in D \Rightarrow Sym_D(n \mapsto m) = m \mapsto n) \land \\
(\forall x, y \cdot x \mapsto y \in g \Rightarrow \phi(x) \mapsto \phi(y) \in D \land \\
LE(x \mapsto y) = LE_D(\phi(x) \mapsto \phi(y))) \land \\
(\forall x \cdot \phi[g[\{x\}]] = D[\phi[\{x\}]]) \land \\
(\forall x, y, z \cdot \{y, z\} \subseteq g[\{x\}] \cup \{x\} \land y \neq z \Rightarrow \phi(z) \neq \phi(y)) \land \\
(\forall x, y \cdot x \mapsto y \in g \Rightarrow \phi(y) \mapsto \phi(x) = Sym_D(\phi(x) \mapsto \phi(y)))
\end{array}\right) \\
\Rightarrow \phi \in ND \rightarrowtail\!\!\!\!\!\twoheadrightarrow N
\end{array}
$$

4.4.2 Niveau 0 : Résultat Global en "One Shot"

Le premier niveau du modèle est spécifié par une machine abstraite qui décrit le résultat à travers un seul évènement *Enum*. L'objectif de cet évènement abstrait est de générer une *bijection* entre les sommets ND et l'intervalle $1..card(ND)$, à partir d'une numérotation zéro sur les sommets. Nous disposons de deux variables *enumeration* et *enumPossibles* désignant res-

pectivement l'énumération souhaitée et l'ensemble des solutions possibles. Ces variables sont initialisées d'une façon indéterministe comme suit :

INITIALISATION
$act1 : enumeration :\in ND \rightarrow \{0\}$
$act2 : enumPossibles := \{e \mid e \in ND \rightarrowtail\!\!\!\!\!\rightarrow 1 \,..\, card(ND)\}$

L'évènement *Enum* associe d'une manière abstraite un élément des solutions possibles à la fonction résultat *enumeration* ($act1$).

EVNT	*Enum*
any	e
where	$e \in enumPossibles$
then	$act1 : enumeration := e$

4.4.3 Niveau 1 : Calcul Global Progressif

Nous introduisons un deuxième niveau pour spécifier un raffinement de la machine $GLOBAL\_ENUM$ par $GLOBAL\_PROGRESS\_ENUM$. Dans ce niveau de raffinement, nous décrivons d'une façon progressive et globale la propagation de la numérotation jusqu'à une énumération . Deux nouvelles variables sont introduites :

- $maxId \in 0 \,..\, card(ND)$. Cet entier est initialisé à 0. Il désigne le maximum des numéros alloués.
- $enum \in ND \rightarrow 0 \,..\, maxId$. C'est une fonction qui associe à chaque sommet un numéro. Initialement, pour tout sommet x, $enum(x) = 0$.

Le principe est de garantir une surjection entre ND et l'intervalle $0..maxId$, ou $1..maxId$, selon le cas. En effet, si tous les sommets ont changé de numéros ($enum \rhd \{0\} = \varnothing$), cela signifie que le minimum de numérotation est passé de 0 à 1 ($inv2$). Si ce n'est pas le cas alors il existe encore des sommets portant

le numéro zéro ($enum \rhd \{0\} \neq \varnothing$). Dans ce cas, le minimum reste zéro et la surjection doit être vérifiée sur $0..maxId$. Formellement, ces propriétés sont spécifiées par les invariants et Théorème suivants :

Invariant 4.1. $enum \rhd \{0\} = \varnothing \Rightarrow enum \in ND \twoheadrightarrow 1 \,..\, maxId$

Invariant 4.2. $enum \rhd \{0\} \neq \varnothing \Rightarrow enum \in ND \twoheadrightarrow 0 \,..\, maxId$

Théorème 4.1. $\forall x, y \cdot \{x, y\} \subseteq ND \wedge x \neq y \wedge enum(x) = enum(y) \Rightarrow maxId < card(ND)$

Démonstration. Pour vérifier le Théorème 4.1, nous admettons le cas contraire, i.e., $maxId \geq card(ND)$. Dans ce cas, la valeur de $maxId$ devrait être égale à $card(ND)$ car nous avons déjà $maxId \leq card(ND)$. Toutefois, une surjection entre ND et $0..card(ND)$ ne pourra pas être vérifiée vue que $card(0..card(ND)) > card(ND)$. Donc, la seule surjection qui pourra être assurée est celle définie par l'invariant 2 ($inv2$). De plus, étant donné que $card(dom(enum)) = card(ran(enum))$, nous pouvons montrer que $enum$ est bijective. Par conséquent, nous déduisons une fausse hypothèse ($\bot$) : $x \neq y \wedge enum(x) = enum(y)$

L'évènement $Enum$ de la machine abstraite est raffiné par un autre évènement $Enum'$ afin de faire disparaître le paramètre abstrait e. Ce paramètre est remplacé par la variable $enum$ de nouvelle la machine ($withe : e = enum$). Toutefois, un renforcement de garde doit être ajouté pour garantir que le déclenchement de l'évènement $Enum'$ induit à une énumération des sommets, i.e., $card(ND) \in enum$ ($grd1$).

EVNT	$Enum'$
refines	$Enum$
where	
$grd1$: $card(ND) \in enum$	
with e : $e = enum$	
then	$act1$: $enumeration := enum$

En outre, un autre événement est ajouté dans la nouvelle machine afin de spécifier la progression de la numérotation et l'incrémentation progressive de la valeur de $maxId$. Cet événement, présenté ci-dessous, est appelé $EnumProgress$. Il dévoile la condition globale pour effectuer une nouvelle numérotation locale sur un sommet x ($grd2$). En effet, x doit changer de numéro sous l'une des conditions suivantes :

- $enum(x) = 0$, en d'autres termes, x n'a pas encore choisi un numéro, ou bien

- il existe un autre sommet y qui a le même numéro que x. Par conséquence directe du Théorème 4.1, nous déduisons que $maxId < card(ND)$ ($Th1$ de l'événement).

Le déclenchement de $EnumProgress$ permet à x de choisir une nouvelle valeur n ($act1$) : c'est un numéro distinct de $enum(x)$ et appartenant à l'intervalle $enum(x) + 1..maxId + 1$ ($grd3$). Suite au changement de numéro de x, le maximum des numéros alloués pourra être modifié. Ainsi, le déclenchement de l'évènement devrait aussi mettre à jour la valeur de $maxId$. Cette dernière prendra l'élément maximale de $\{maxId, n\}$ ($act2$).

EVNT $ProgressEnum$

any x, n

where

$grd1 : x \in ND$

$grd2 : (enum(x) = 0) \lor (\exists y \cdot y \in ND \setminus \{x\} \land enum(x) = enum(y))$

$Th1 : maxId < card(ND)$

$grd3 : n \in enum(x) + 1 \,..\, maxId + 1$

then

$act1 : enum(x) := n$

$act2 : maxId := max(\{maxId, n\})$

De plus, nous prouvons le Théorème 4.2 pour s'assurer du non blocage de la machine. En d'autres termes, nous vérifiions qu'il existe au moins un évènement qui se déclenche. Formellement, il s'agit de prouver que la disjonction des gardes des deux évènements est toujours vraie.

Théorème 4.2. $(ran(enum) = 1 \,..\, card(ND))$

$$\lor \left(\begin{array}{l} \forall x, n \cdot (x \in ND) \land \\ ((enum(x) = 0) \lor (\exists y \cdot y \in ND \setminus \{x\} \land enum(x) = enum(y))) \land \\ (n \in enum(x) + 1 \,..\, maxId + 1) \end{array} \right)$$

4.4.4 Niveau 2 : Calcul Local Progressif

La machine $GLOBAL\_PROGRESS\_ENUM$ est raffinée par une autre machine $LOCAL\_PROGRESS\_ENUM$. Dans cette nouvelle machine, nous vérifions la progression de l'énumération dans un niveau local. Plus précisément, nous surveillons le choix de numéros de chaque sommet x, sous une vision limitée à la boule de centre x. Nous montrons comment une injection est assurée, localement, sur chaque boule. Toutefois, dans ce niveau de raffinement, les étapes de calculs qui aboutissent à une injection sur l'ensemble

des sommets du graphe, manqueraient encore de précisions.

Nous introduisons une variable Lv dans la nouvelle machine, afin de construire une vue locale pour chaque sommet du graphe (voir Section 4.2) :

- $Lv \in ND \rightarrow \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1))$. La vue locale d'un sommet x, notée $Lv(x)$, est initialisée à l'ensemble vide. Elle constitue l'ensemble des numéros des voisins de x qui sont différents de la valeur zéro. Il s'agit des numéros des voisins qui ont changé leurs numérotations initiales. Plus précisément, pour tout x qui a un voisin y tel que $LE(x \mapsto y) = (i \mapsto j)$, alors le triplet $n \mapsto (i \mapsto j)$ est un élément de $Lv(x)$ si le dernier message que x a reçu de y indique que $enum(y) = n$. Si nous considérons l'exemple du sommet x de la FIGURE 4.2, la vue locale de x sera définie par l'ensemble suivant : $Lv(x) = \{y \mapsto (1 \mapsto 1), t \mapsto (2 \mapsto 1), z \mapsto (3 \mapsto 1)\}$. Formellement, pour tout sommet x, $Lv(x)$ est spécifié comme suit :
$$Lv(x) = \left\{ \begin{array}{l} n, y \cdot y \in g[\{x\}] \wedge enum(y) \neq 0 \wedge n = enum(y) \\ |n \mapsto LE(x \mapsto y) \end{array} \right\}.$$

Le Théorème 4.3 est une conséquence directe de la définition de Lv. Il s'agit de vérifier que pour tout triplet $n \mapsto (i \mapsto j)$ appartenant à $Lv(x)$, il existe un voisin y numéroté n et $LE(x \mapsto y) = i \mapsto j$.

Théorème 4.3. $\forall x, n, i, j \cdot x \in ND \wedge n \mapsto (i \mapsto j) \in Lv(x)$
$\Rightarrow (\exists y \cdot x \mapsto y \in g \wedge enum(y) \neq 0 \wedge n = enum(y) \wedge LE(x \mapsto y) = i \mapsto j)$

Dans ce niveau de raffinement, l'évènement $EnumProgress$ de la machine abstraite $GLOBAL\_PROGRESS\_ENUM$ est raffiné par un évènement $EnumProgress'$. La spécification de ce dernier est présentée à la fin de la section. Dans cet évènement, la comparaison de $enum(x)$ avec un numéro d'un autre sommet y s'effectue localement, plutôt que globalement

$(\ominus grd2, \oplus grd2)$. En effet, si x s'aperçoit qu'il existe un autre sommet y qui est distant de x d'une valeur ≤ 2 et qui a le même numéro que x, alors x change de numéro. En d'autre terme, x compare son numéro avec les numéros de ses voisins, i.e., qui appartiennent à $g[x]$, et les numéros des voisins de voisins, i.e., qui appartiennent à $g[g[x]]$. En effet, pour garantir une numérotation injective sur les boules, deux sommets d'une même boule ne doivent pas choisir le même numéro. Par exemple, si nous considérons les boules $B(x)$ et $B(y)$ de la FIGURE 4.3, nous devons vérifier que les numéros de $B(x)$ i.e., $\{enum(x), enum(y), enum(z), enum(t)\}$, ainsi que les numéros de $B(y)$ i.e., $\{enum(y), enum(x), enum(k), enum(w)\}$, sont deux à deux distincts. Par conséquent, le numéro de x doit être différent des numéros de $g[x]$ et des numéros de $g[g[x]]$.

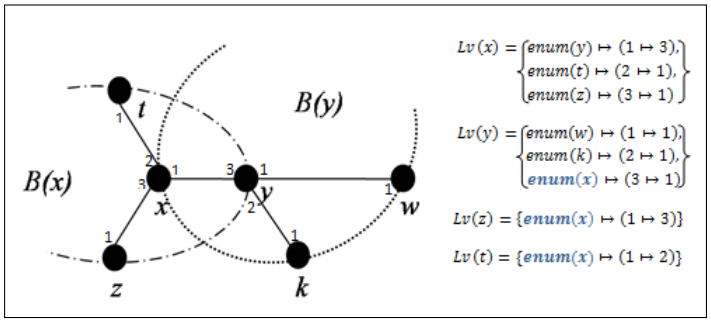

FIGURE 4.3 – Vues Locales construites par des sommets d'une boule

Le paramètre abstrait n est remplacé par un autre paramètre plus concret $n1$. Ce dernier doit être différent des numéros de la boule de centre x $(\ominus grd3, \oplus grd3)$, et ne doit pas appartenir à l'ensemble des numéros des voisins de voisins de x. Par définition de Lv, il suffit de vérifier que $n1$ n'appartient pas aux vues locales des voisins de x $(\oplus grd4)$. Ainsi, nous pouvons facilement garantir l'Invariant 4.3 : il s'agit de maintenir une numérotation injective sur les boules.

Invariant 4.3. $\forall x, B \cdot x \in ND \land B = g[\{x\}] \cup \{x\} \land 0 \notin ran(B \lhd enum)$
$\Rightarrow B \lhd enum \in B \rightarrowtail 1 \cdot \cdot maxId$

EVNT $EnumProgress'$

refines $EnumProgress$

any $x, n1, Lv1$

where

$\ldots$

$\ominus grd2 : (enum(x) = 0) \lor (\exists y \cdot y \in ND \setminus \{x\} \land enum(x) = enum(y))$

$$\oplus grd2 : (enum(x) = 0) \lor \left(\begin{array}{l} \exists y \cdot y \in (g[\{x\}] \cup g[g[\{x\}]]) \land \\ enum(x) = enum(y)) \end{array} \right)$$

$\ominus grd3 : n \in enum(x) + 1 \,..\, maxId + 1$

$\oplus grd3 : n1 \notin dom(Lv(x))$

$\oplus grd4 : \forall y \cdot x \mapsto y \in g \Rightarrow n1 \notin dom(Lv(y))$

$\oplus grd5 : Lv1 \in g[\{x\}] \rightarrow \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1))$

$\oplus grd6 : \forall y \cdot y \in g[\{x\}] \Rightarrow$

$$Lv1(y) = \left(\begin{array}{l} (Lv(y) \setminus \{enum(x) \mapsto LE(y \mapsto x)\}) \\ \cup \{n1 \mapsto LE(y \mapsto x)\} \end{array} \right)$$

$\oplus Th1 : \forall y \cdot y \in g[\{x\}] \Rightarrow ordre(Lv1(y) \mapsto Lv(y)) = 1$

with $n : n = n1$

then

$\ldots$

$\oplus act3 : Lv := Lv \lhd\!\!\!- Lv1$

En outre, si x change de numéro alors les vues locales, dans lesquelles figure $enum(x)$, doivent être mises à jour. Il s'agit des vues locales des voisins de x (voir l'exemple de la Figure 4.3). Par conséquent, nous ajoutons $act3$ pour définir les nouvelles valeurs de Lv calculées sur $g[x]$: pour simplifier cette action de substitution, nous utilisons une fonction intermédiaire $Lv1$ passée en paramètre de l'événement et définie par $grd5$ et $grd6$. En effet, pour

tout sommet y voisin de x, $Lv1(y)$ remplace dans $Lv(y)$ le couple $enum(x) \mapsto LE(y \mapsto x)$ par le couple $n1 \mapsto LE(y \mapsto x)$. La garde $Th1$ est une conséquence directe du Théorème 4.7.

4.4.5 Niveau 3 : Calcul Local Déterminé

Un troisième niveau est introduit pour spécifier un raffinement de la machine $LOCAL\_PROGRESS\_ENUM$ par une machine $LOCAL\_ENUM$. Dans la machine précédente, nous avons spécifié les pas de calculs nécessaires pour qu'un sommet x change de numéro, tout en garantissant une injection de $enum$ sur la boule de centre x. À travers ce nouveau raffinement, nous expliquons (i) comment x évite des changements infinis sur sa numérotation et (ii) quels pas de calculs à ajouter pour garantir une injection de $enum$ sur tout le graphe. Il s'agit d'une application directe des règles R_1 et R_2 présentées dans la Section 4.2.

Une nouvelle variable M est introduite pour construire les mémoires des sommets : $M \in ND \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)))$. Pour tout sommet x, et tout sommet y voisin de x, $M(x)$ est construite à partir des numéros ainsi que des vues locales de y qui sont définis au cours d'une exécution. En d'autres termes, pour tout déclenchement d'évènements, $M(x)$ garde trace du couple $enum(y) \mapsto Lv(y)$ (Invairant 4.4). De plus, étant donné que $Lv(x)$ contient les numéros des voisins de x, nous déduisons l'inclusion suivante : $dom(Lv(x)) \subseteq dom(M(x))$ (Théorème 4.4).

Invariant 4.4. $\forall x, y \cdot x \mapsto y \in g \wedge enum(y) \neq 0 \Rightarrow enum(y) \mapsto Lv(y) \in M(x)$

Théorème 4.4. $\forall x \cdot x \in ND \Rightarrow dom(Lv(x)) \subseteq dom(M(x))$

Événement Renam Raffine EnumProgress'

Un nouvel événement *Renam* (TABLE 4.1) raffine $EnumProgress'$. Il introduit les conditions nécessaires pour le déclenchement et l'application de la règle de renommage. Par définition de cette règle, un ordre total doit être défini sur l'ensemble des vues locales. Par conséquent, nous ajoutons quelques constantes et propriétés au contexte *Graph* comme suit :

- $DifSym \in \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \times \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \to \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big)$. Pour tout ensemble $L1$ et $L2$ appartenant chacun à $\mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big)$, $DifSym(L1 \mapsto L2)$ désigne la différence symétrique de $L1$ et $L2$. Formellement, cette différence est définie comme suit :
  $DifSym(L1 \mapsto L2) = (L1 \setminus L2) \cup (L2 \setminus L1)$.
- $maxDifSym \in \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \times \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \to \big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big)$. Le maximum de la différence symétrique de deux ensembles $L1$ et $L2$ est un élément de $L1$ ou $L2$. $(maxDifSym(L1 \mapsto L2) \in L1 \cup L2)$. Il s'agit du triplet le plus élevé de $DifSym(L1 \mapsto L2)$, vérifié par l'implication suivante :
  $\forall n, p, q, n', p', q' \cdot maxDifSym(L1 \mapsto L2) = n \mapsto (p \mapsto q) \wedge$
  $n' \mapsto (p' \mapsto q') \in DifSym(L1 \mapsto L2) \Rightarrow \begin{pmatrix} n > n' \vee \\ (n = n' \wedge p > p') \vee \\ (n = n' \wedge p = p' \wedge q \geq q') \end{pmatrix}$.
- $ordre \in \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \times \mathbb{P}\big(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)\big) \to \{-1, 0, 1\}$. Nous définissons une fonction *ordre* pour comparer tout ensemble $L1$ et $L2$ en fonction du maximum de leur différence symétrique :
  - $ordre(L1 \mapsto L2) = -1 \Leftrightarrow maxDifSym(L1 \mapsto L2) \in L2$, i.e., $L1$ est plus faible que $L2$ $(L1 < L2)$,
  - $ordre(L1 \mapsto L2) = 1 \Leftrightarrow maxDifSym(L1 \mapsto L2) \in L1$, i.e., $L1$ est plus forte que $L2$ $(L1 > L2)$,

$\diamond$ $ordre(L1 \mapsto L2) = 0 \Leftrightarrow L1 = L2$.

Nous utilisons la fonction *ordre* pour comparer les vues locales des sommets. De plus, nous définissons quelques théorèmes qui nous seront utiles pour la preuve des invariants de la machine $LOCAL\_ENUM$:

Théorème 4.5. $\forall L1, L2 \cdot \{L1, L2\} \subseteq \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)) \wedge$
$ordre(L1 \mapsto L2) = -1 \Rightarrow ordre(L2 \mapsto L1) = 1$

Pour tout ensemble $L1$ et $L2$, si $L1$ est plus faible que $L2$ alors $L2$ est plus forte que $L1$ (Théorème 4.5).

Théorème 4.6. $\forall L1, L2, L3 \cdot \{L1, L2, L3\} \subseteq \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)) \wedge$
$\left(\begin{array}{l} ordre(L1 \mapsto L2) = 1 \wedge \\ ordre(L2 \mapsto L3) = 1 \end{array} \right) \Rightarrow ordre(L1 \mapsto L3) = 1.$

De plus, si $L1$ et plus forte qu'un ensemble $L2$, et $L2$ est plus forte qu'un ensemble $L3$, alors $L1$ est plus forte que $L3$ (Théorème 4.6).

Théorème 4.7. $\forall L1, L2, n1, n2, e \cdot \{L1, L2\} \subseteq \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)) \wedge$
$\left(\begin{array}{l} n1 \mapsto e \in L1 \wedge \\ n2 > n1 \wedge \\ L2 = (L1 \setminus \{n1 \mapsto e\}) \cup \{n2 \mapsto e\} \end{array} \right) \Rightarrow ordre(L2 \mapsto L1) = 1.$

Nous spécifions, par le raffinement de $EnumProgress'$, qu'une nouvelle numérotation s'effectue après comparaisons des numéros des sommets d'une part, et de leurs vues locales, d'autre part. En effet, un sommet x doit choisir un nouveau numéro $n1$, s'il se rend compte qu'il existe un couple $enum(x) \mapsto L$ dans sa mémoire, telle que sa vue locale est plus faible que L, i.e., $ordre(Lv(x) \mapsto L) = -1$ ($\ominus grd2$, $\oplus grd2$). Dans ce raffinement, le choix de $n1$ s'effectue d'une manière déterministe : $n1 = 1 + max(dom(M(x)) \cup \{0\})$ ($\ominus grd3, \ominus grd4$ et $\oplus grd3$). En revanche, la décision d'une nouvelle numérotation ne peut pas s'effectuer en se basant sur des mémoires qui ne sont pas encore mises à jour. Ainsi, une nouvelle garde ($\oplus grd4$) est ajoutée pour vérifier

EVNT $Renam$
refines $EnumProgress'$
any $x, n1, Lv1, M1, M2$
where
...
$\ominus grd2 : (enum(x) = 0) \lor \left(\begin{array}{l} \exists y \cdot y \in (g[\{x\}] \cup g[g[\{x\}]]) \land \\ enum(x) = enum(y)) \end{array} \right)$
$\oplus grd2 : enum(x) = 0 \lor \left(\begin{array}{l} \exists L \cdot enum(x) \mapsto L \in M(x) \land \\ ordre(Lv(x) \mapsto L) = -1 \end{array} \right)$
$\ominus grd3 : n1 \notin dom(Lv(x))$
$\ominus grd4 : \forall y \cdot x \mapsto y \in g \Rightarrow n1 \notin dom(Lv(y))$
$\oplus grd3 : n1 = 1 + max(dom(M(x)) \cup \{0\})$
$\oplus grd4 : \forall y \cdot y \in g[\{x\}] \cup \{x\} \Rightarrow M(x) = M(y)$
$\oplus grd7 : M1 \in g[\{x\}] \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)))$
$\oplus grd8 : \forall y \cdot y \in g[\{x\}] \Rightarrow M1(y) = M(y) \cup \{n1 \mapsto Lv(x)\}$
$\oplus grd9 : M2 \in g[g[\{x\}]] \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)))$
$\oplus grd10 : \forall y, w \cdot y \in g[\{x\}] \land w \in g[\{y\}]$
$\Rightarrow M2(w) = M(w) \cup \{enum(y) \mapsto Lv1(y)\}$
then
...
$\ominus act2 : maxId := max(\{maxId, n\})$
$\oplus act4 : M := (M \lhd\!\!\!+ M1) \lhd\!\!\!+ M2$

TABLE 4.1 – Spécification de l'événement Renam

que tous les sommets de la boule de centre x disposent de la même mémoire. Notons que tout remplacement de gardes est contrôlé par des obligations de preuves déchargées. Ainsi, la suppression des gardes $grd3$ et $grd4$ de l'évènement abstrait n'implique pas leur contradiction avec les gardes ajoutées.

Démonstration. Si $n1 = 1 + max(dom(M(x)) \cup \{0\})$ signifie que $n1 \notin dom(M(x))$. De plus, par application du Théorème 4.4, nous déduisons que $n1 \notin dom(Lv(x))$. Ainsi, $grd3$ de l'évènement abstrait est vérifiée. En outre, d'une part, pour tout sommet y, $1 + max(dom(M(y)) \cup \{0\}) \notin dom(M(y))$. D'autre part, si y est un voisin de x alors $M(x) = M(y)$ $(\oplus grd4)$. Ainsi, $1 + max(dom(M(x)) \cup \{0\}) \notin dom(M(y))$. D'où $n1 \notin dom(M(y))$. Par conséquent, $n1 \notin dom(Lv(y))$ (Théorème 4.4). D'où $grd4$ de l'évènement abstrait est vérifiée.

Par ailleurs, nous rappelons que toute modification de $enum(x)$ engendre une modification des vues locales du voisinage de x ($grd5$, $grd6$ et $act3$ de $EnumProgress'$). Toutefois, d'après la définition de M, d'autres mises à jour doivent être vérifiées :

1. une mise à jour de $enum(x)$ produit une mise à jour des vues locales ainsi que des mémoires du voisinage de x, i.e., $g[\{x\}] \triangleleft Lv$ et $g[\{x\}] \triangleleft M$,
2. une mise à jour de $Lv(x)$ produit une mise à jour des mémoires du voisinage de x.

En d'autre termes, étant donné un sommet x, si x change de numéro alors $g[\{x\}] \triangleleft Lv$, $g[\{x\}] \triangleleft M$ et $g[g[\{x\}]] \triangleleft M$ doivent être modifiées. Ces mises à jours sont effectuées à travers l'action de substitution $act4$. Pour des raisons de simplification, nous utilisons des fonctions intermédiaires $M1$ et $M2$ pour calculer respectivement les nouvelles mémoires de $g[\{x\}]$ et celles de $g[g[\{x\}]]$. Ces fonctions sont définies par de nouvelles gardes $grd7$ et $grd8$ pour $M1$; et $grd9$ et $grd10$ pour $M2$. Notons que les gardes $grd1$, $grd5$ et

$grd6$ de l'évènement précédent sont préservés dans $Renam$. Toutefois, l'action de substitution de $maxId$ par $act3$ est considérée comme une action globale ($\ominus act3$). Elle est supprimé, vue que $n1$ prendra toujours la valeur de $1 + max(dom(M(x)) \cup \{0\})$.

Événement Diffusion

EVNT $Diffusion$
$\mathbf{x}, B, MB$
where
$grd1 : x \in ND$
$grd2 : enum(x) \neq 0$
$grd3 : B = \{x\} \cup g[\{x\}]$
$grd4 : \exists y \cdot y \in B \wedge M(y) \neq M(x)$
$grd5 : MB = \{union(\{y \cdot y \in B \mid M(y)\})\}$
then
$quadact1 : M := M \lhd\!\!+ (B \times MB)$

La nouvelle machine $Enum\_LOCAL$ dispose d'un nouvel évènement nommé $Diffusion$. Il s'agit d'une application de la règle R_2 (voir Section 4.2). En effet, pour tout sommet x appartenant à une boule B, $M(x)$ doit être construite par union de toutes les mémoires de B. L'évènement $Diffusion$ se déclenche quand un sommet de B change de mémoire ($grd4$). Dans ce cas, l'union des mémoires de la boule est recalculée (MB) et attribuée à tous les sommets de B ($act1$).

Événement Enum" Raffine Enum'

EVNT $Enum''$

refines $Enum'$

where

$\ominus grd1 : card(ND) \in enum$

$\oplus grd2 : \forall z, x \cdot z \in g[\{x\}] \Rightarrow enum(z) \neq 0$

$\oplus grd3 : \forall x, L \cdot enum(x) \mapsto L \in M(x) \wedge L \neq Lv(x)$

$\Rightarrow ordre(Lv(x) \mapsto L) = 1$

$\oplus grd4 : \forall x, y \cdot y \in g[\{x\}] \Rightarrow M(y) = M(x)$

$\oplus Th1 : \forall x, y \cdot \{x, y\} \subseteq ND \Rightarrow M(x) = M(y)$

$\oplus Th2 : \forall x, y . \{x, y\} \subseteq ND \Rightarrow enum(x) \mapsto Lv(x) \in M(y)$

$\oplus Th3 : \{x \cdot x \in ND | enum(x)\} = 1 \,..\, card(ND)$

then

...

L'évènement $Enum'$ est raffiné dans $LOCAL\_ENUM$ par $Enum''$: ce raffinement spécifie les propriétés d'une configuration finale. $Enum''$ n'est plus un "one shot". Il doit être observé que dans un seul cas : les évènement $Renam$ et $Diffusion$ ne sont plus déclenchables. Cette vérification est effectuée par négation de gardes de ces deux évènements. Ainsi, nous ajoutons $grd2$ et $grd3$ pour vérifier $\neg Rnam$; et $grd4$ pour vérifier $\neg Diffusion$. Toutefois, $grd1$ de l'évènement abstrait est supprimée ($\ominus grd1$). Les gardes $Th1$, $Th1$ et $Th3$ sont des conséquences directes de la spécification. En effet, g étant un graphe connexe, l'égalité qui est vérifiée entre les mémoires des boules de g, sera aussi vérifiée entre tous les sommets de g ($Th1$). Par conséquent de cette égalité, nous montrons que pour tout sommet x et y le couple $enum(x) \mapsto Lv(x)$ est un élément de $M(y)$ ($Th2$). Par ailleurs, vue la surjection et l'injection assurée pour $enum$, nous pouvons déduire que l'ensemble $\{x \cdot x \in ND | enum(x)\}$

désigne l'intervalle $1..card(ND)$ $(Th3)$. Par conséquent, il est bien clair que $card(ND) \in ran(enum)$ $(\ominus grd1)$.

4.4.6 Autres Propriétés Invariantes et Théorèmes

La propriété essentielle de l'algorithme consiste à vérifier que pour tout sommet x qui a dans sa mémoire un couple $enum(x) \mapsto L$; et que L est plus forte que $Lv(x)$ alors il y a un autre sommet y ayant le même numéro que x (Théorème 4.8). En effet, ce Théorème est une conséquence de l'invariant suivant :

Invariant 4.5.

$$\forall x, n, L \cdot n \mapsto L \in M(x) \Rightarrow \left(\begin{array}{l} \exists y \cdot y \in ND \wedge enum(y) = n \wedge \\ (L \neq Lv(y) \Rightarrow ordre(Lv(y) \mapsto L) = 1) \end{array} \right).$$

Démonstration. Si un sommet x déclenche l'évènement $Renam$, $enum(x)$ prend une nouvelle valeur $n1$, les mémoires et les vues locales calculées par $g[\{x\}]$ sont modifiées et prennent respectivement les valeurs de $M1$ et $Lv1$. Les mémoires calculées par $g[g[\{x\}]]$ sont également modifiées et prennent la valeur de $M2$. Pour prouver la préservation de l'Invariant 4.5 par $Renam$, il suffit de vérifier l'implication suivante :

$$\forall x0, n, L \cdot n \mapsto L \in (M2(x0) \cup M1(x0))$$
$$\Rightarrow \left(\begin{array}{l} \exists y \cdot ((\{x\} \lhd enum) \cup \{x \mapsto n1\})(y) = n \wedge \\ (y \in g[\{x\}] \wedge L \neq Lv1(y)) \Rightarrow ordre(Lv1(y) \mapsto L) = 1 \end{array} \right).$$

- si $n \mapsto L \in M2(x0)$ alors, $x0 \in g[g[\{x\}]]$, et pour tout sommet $x1$ tel que $x1 \in (g[\{x\}] \cap g[\{x0\}])$, $M2(x0) = M(x0) \cup \{enum(x1) \mapsto Lv1(x1)\}$. Ainsi, $n \mapsto L \in M(x0) \cup \{enum(x1) \mapsto Lv1(x1)\}$

  (a) si $n \mapsto L \in M(x0)$, alors en remplaçant x par $x0$ dans l'Invariant 4.5, nous prouvons l'implication suivante : $\exists y.enum(y) = n \wedge (L \neq Lv(y) \Rightarrow ordre(Lv(y) \mapsto L) = 1)$. Ainsi, il suffit de montrer (i)

$y \neq x$ et (ii) $L \neq Lv1(y) \Rightarrow ordre((Lv \cup Lv1)(y) \mapsto L) = 1$. En effet, si $x = y$ alors l'implication suivante est vraie : $\forall L.enum(x) \mapsto L \in M(x) \wedge L \neq Lv(x) \Rightarrow ordre(Lv(x) \mapsto L) = 1$. Toutefois, selon l'événement *Renam*, il existe une vue locale $L0$ qui vérifie la propriété suivante : $enum(x) \mapsto L0 \in M(x) \wedge ordre(Lv(x) \mapsto L) = -1$. Par conséquent (i) est vérifié. De plus, étant donné que $ordre(Lv(y) \mapsto L) = 1 \wedge ordre(Lv1(y) \mapsto Lv(y) = 1$, nous montrons, d'après le Théorème 4.6, que $ordre(Lv1(y) \mapsto L) = 1$, i.e., (ii) est vérifiée.

(b) si $n \mapsto L = enum(x1) \mapsto Lv1(x1)$ i.e., $enum(x1) = n$ et $L = Lv1(x1)$, alors il suffit de montrer que $x1 \neq x$. En effet, $x1 \in g[\{x\}]$ et g est un graphe sans boucle.

– si $n \mapsto L \in M1(x0)$ alors $x0 \in g[\{x\}]$ et $M1(x0) = M(x0) \cup \{n1 \mapsto Lv(x)\}$. Si $n \mapsto L \in M(x0)$ alors l'invariant est vérifié en suivant le même raisonnement présenté dans (a). Si $n \mapsto L = n1 \mapsto Lv(x)$, alors $n = n1$. D'où, s'il existe un sommet y tel que $((\{x\} \lhd enum) \cup \{x \mapsto n1\})(y) = n$, alors $y = x \wedge y \notin g[\{x\}]$.

Théorème 4.8. $\forall x, L \cdot enum(x) \mapsto L \in M(x) \wedge ordre(L \mapsto Lv(x)) = 1 \Rightarrow (\exists y \cdot y \in ND \setminus \{x\} \wedge enum(y) = enum(x))$

Démenstration du Théorème 4.8. Si nous disposons d'un couple $enum(x) \mapsto L \in M(x)$, alors d'après l'Invariant 4.5, nous pouvons déduire qu'il existe un sommet y, ayant le même numéro que x. Ainsi, il reste à montrer que x est différent de y. Admettons que $x = y$, d'après l'Invariant 4.5, si $L \neq Lv(x)$ alors $ordre(Lv(x) \mapsto L) = 1$, ou encore $ordre(L \mapsto Lv(x)) = -1$ (Théorème 4.5). Toutefois, nous faisons l'hypothèse que $L \neq Lv(x)$, i.e., $ordre(L \mapsto Lv(x)) \neq 0$, mais $ordre(L \mapsto Lv(x)) = 1$. Par conséquent, $y \in ND \setminus \{x\}$.

Terminaison. Nous montrons par l'Invariant 4.6 que pour tout sommet x, si un numéro n figure dans $M(x)$ alors tous les numéros de l'intervalle $1..n$, figurent aussi dans $M(x)$. Par conséquent, il est facile de voir que l'algorithme termine. De plus, vue que le graphe est minimal pour les revêtements, cette terminaison globale pourra être détectée par un sommet x, dès que $card(ND)$ figure dans $M(x)$. La terminaison est vérifiée quand les numéros des sommets ne changent plus et aucun message n'est diffusé pour mettre à jour les mémoires. En effet, d'après les Invariants 4.6 et 4.5, nous montrons que les numéros des mémoires forment un intervalle $1..n$ où $n \leq card(ND)$ (Théorème 4.9). Par conséquent, il est bien clair qu'il existe une étape de calcul i_0, telle que pour tout sommet x et toute étape i, $i \geq i_0$, le numéro de x à l'étape $i+1$ est le même que celui de l'étape i ($enum_{i+1}(x) = enum_i(x)$). De plus, $Lv(x)$ et $M(x)$ ne peuvent prendre qu'un nombre fini de valeurs. En outre, il est facile de constater que l'évènement $ENUM''$ fait décroitre le nombre de sommets qui doivent faire des mises à jours sur leurs mémoires. Nous vérifions ainsi le variant suivant : $card(\{x.\exists y.y \in ND \setminus \{x\} \wedge M(y) \neq M(x)$.

Invariant 4.6. $\forall x, n, n1, L{\cdot}x \in ND \wedge n \mapsto L \in M(x) \wedge n1 \in 1 \,..\, n \Rightarrow (\exists L'{\cdot}n1 \mapsto L' \in M(x))$

Théorème 4.9. $\forall x, n{\cdot}x \in ND \wedge M(x) \neq \varnothing \wedge n = max(dom(M(x))) \Rightarrow dom(M(x)) = 1 \,..\, n$

De surcroit, étant donné que nous avons fait une hypothèse sur la minimalité du graph (Section 4.4), i.e., g est minimal pour les relations de revêtements, nous vérifions le Théorème 4.10 : pour tout sommet v, tel que $card(ND)$ figure dans $M(v)$, alors la mémoire de v doit satisfaire les propriétés suivantes : (1) l'ensemble des numéros de $dom(M(v))$, noté N, définit bien l'intervalle $1..card(ND)$, et (2) g est isomorphe à tout graphe dirigé symétrique D qui est construit à partir de N, via un homomorphisme h :

(i) La symétrie de D est définie par la fonction suivante :
$Sym_D \in D \to D \wedge (\forall n, m \cdot n \mapsto m \in D \Rightarrow Sym_D(n \mapsto m) = m \mapsto n)$

(ii) h est une fonction de ND vers N qui conserve la structure des arêtes :
$\forall x, y \cdot x \mapsto y \in g \Rightarrow h(x) \mapsto h(y) \in D$

(iii) D est défini avec un étiquetage de ports sur les arêtes :
$LE_D \in (D \to N \times N) \wedge (\forall x, y \cdot x \mapsto y \in g \Rightarrow LE(x \mapsto y) = LE_D(h(x) \mapsto h(y)))$

Théorème 4.10. $\forall v, N \cdot v \in ND \wedge card(ND) \in dom(M(v)) \wedge N = dom(M(v))$
$\Rightarrow N = 1 \,..\, card(ND) \wedge$
$(\forall D, h, LE_D, Sym_D \cdot$
$$\left(\begin{array}{l} D \subseteq N \times N \wedge Sym_D \in D \to D \wedge \\ (\forall n, m \cdot n \mapsto m \in D \Rightarrow Sym_D(n \mapsto m) = m \mapsto n) \wedge \\ h \in ND \to N \wedge \\ (\forall x, y \cdot x \mapsto y \in g \Rightarrow h(x) \mapsto h(y) \in D) \wedge \\ LE_D \in (D \to N \times N) \wedge \\ (\forall x, y \cdot x \mapsto y \in g \Rightarrow LE(x \mapsto y) = LE_D(h(x) \mapsto h(y))) \end{array} \right)$$
$\Rightarrow h \in ND \twoheadrightarrow N)$

Démonstration. Nous montrons d'abord que g est un revêtement de D. Ainsi, nous vérifions que (1) $ran(h) = dom(M(v))$, (2) pour tout sommet x, $\phi[g[\{x\}]] = D[\phi[\{x\}]]$, et (3) pour tout couple distincts (y, z) appartenant à la boule de centre x, $\phi(z) \neq \phi(y)$. Ensuite, en se basant sur l'hypothèse de minimalité de g, nous pouvons déduire que h est bijective.

4.5 Conclusion

Dans ce chapitre, nous avons traité l'algorithme de *Mazurkiewicz* pour résoudre le problème d'énumération sur des graphes qui sont minimaux pour

les revêtements [15]. Nous avons proposé une démarche progressive basée sur quatre niveaux d'abstractions. Pour chaque niveau, nous avons présenté les preuves nécessaires pour vérifier les propriétés de l'algorithme. Le dernier niveau constitue la preuve finale pour démontrer une énumération correcte de point de vue local. Le tableau 4.2 présente la difficulté de spécification en donnant les statistiques des obligations de preuves générées lors de la modélisation. Nous remarquons que le nombre total d'obligations ainsi que le nombre des obligations interactives commence à devenir plus élevé au fur et à mesure des raffinements. Cette augmentation est due aux détails et propriétés ajoutées pour relever le degré d'abstraction.

Machines	Total	O.P Automatiques		O.P Interactives	
GLOBAL_ENUM	7	5	71.42%	2	28.57%
GLOBAL_PROGRESS_ENUM	28	9	32.14%	19	67.85%
LOCAL_PROGRESS_ENUM	31	10	32.25%	21	67.74%
LOCAL_ENUM	78	23	29.48%	55	70.51%

TABLE 4.2 – Bilan des Obligations de Preuves (O.P)

Chapitre

5 Snapshots Locaux et Preuves du Calcul d'état Global

Sommaire

5.1 Introduction

Dans ce chapitre, nous étudions le problème du snapshot et du calcul d'état global dans les systèmes distribués anonymes. L'état global d'un système S est défini à un moment donné du déroulement du système. Il est caractérisé par l'ensemble des états locaux des processus ainsi que des canaux de communications. En effet, l'état local d'un processus est défini par l'état de sa mémoire locale et de l'historique de ses activités. L'état d'un canal de communication est caractérisé par l'ensemble de messages circulant dans ce canal. Plus précisément, si un processus p communique avec un processus q via un canal c entrant à q, l'état de c est défini par les messages qui sont envoyés par p et non encore reçus. Les changements d'états des processus et des canaux résultent généralement de trois types d'événements : un événement interne aux processus, un événement d'envoi et un événement de réception de messages.

Pour chaque processus p, les événements sont ordonnés d'une façon linéaire par leur ordre d'occurrence [59]. Un premier ordre causal local ($\prec_p$) transitif est défini sur les événements E_p de p. Considérons e_{p_i} et e_{p_j} deux événements de E_p, la relation $e_{p_i} \prec_p e_{p_j}$ désigne que e_{p_i} apparaît avant e_{p_j}. Un deuxième ordre causal global ($\prec$) transitif est défini entre les événements des processus. Il est caractérisé par la plus petite relation qui satisfait les propriétés suivantes :

- si e_{p_i} et e_{p_j} sont des événements locaux d'un processus p et $e_{p_i} \prec_p e_{p_j}$ alors $e_{p_i} \prec e_{p_j}$,
- si e_{p_i} est un événement d'envoi déclenché par un processus p pour un

processus q et e_{q_j} est l'événement de réception correspondant alors e_{p_i} se produit avant e_{q_j} ($e_{p_i} \prec e_{q_j}$).

Un état global est considéré comme une photographie instantanée du réseau. Il est défini par un calcul d'un snapshot consistant. En effet, le but d'un algorithme de snapshot est de collecter les états locaux des processus et des canaux de communications afin de construire cette photographie. La supervision d'un état global est motivée, dans certains cas, par le besoin de détecter les propriétés stables du réseau. Ces propriétés, dès qu'elles deviennent vraies, restent vraies tout au long de l'exécution du système. Dans d'autres cas, il pourrait avoir aussi un besoin de relancer l'exécution du système depuis le dernier état global mémorisé.

Le calcul d'état global est étudié dans plusieurs travaux de recherche. La plupart de ces travaux supposent que les éléments du système ont des identifiants uniques, ou qu'il existe un unique sommet initiateur du calcul. Cependant, en absence de ces hypothèses, il serait difficile de collecter localement les snapshots locaux et avoir un aperçu sur l'état global du système. Pour résoudre ce problème, nous dévoilons des propriétés de liaisons entre des algorithmes existants et nous prouvons que ces liaisons pourraient être mises en œuvre par des relations de raffinements.

Tout au long de ce chapitre, nous expliquons d'abord le principe du calcul des algorithmes de snapshots. Nous décrivons particulièrement l'algorithme de *Chandy-Lamport* [25]. Nous présentons ensuite notre méthodologie. Nous explorons principalement les travaux de Chalopin et al. [24] et Andriamiarina et al.[7] et nous proposons un schéma de preuve basé sur l'approche *correct-par construction*. Enfin, nous détaillons les spécification formelles correspondantes.

5.2 Algorithmes de Snapshot

Chaque processus peut calculer un snapshot local. Il s'agit d'enregistrer son état et les états de ses canaux entrants. L'ensemble des snapshots locaux, désigné par un snapshot global, constitue un état global que le système peut atteindre à un certain moment. Un état global est consistant si le snapshot l'est aussi. Par définition [40], cette consistance doit assurer les conditions suivantes :

- lors du calcul d'un snapshot local d'un processus p_i, un message m_{ij} peut être enregistré comme envoyé par p_i pour un processus p_j. Dans ce cas, m_{ij} doit être capturé dans le canal C_{ij} reliant p_i à p_j, ou bien enregistré comme un message reçu par p_j.
- si m_{ij} n'est pas enregistré comme un message envoyé par p_i alors il ne doit figurer ni dans le canal C_{ij} ni parmi les messages reçus par p_j.

La consistante d'un snapshot est vérifiée en fonction de la consistance des coupes. Pour tout processus p, une coupe C, nommée aussi $Coupe$, est induite par un snapshot local de p. Elle permet de scinder les événements E_p en deux catégories : des évènements antérieurs et des évènements postérieurs à la coupe. La première catégorie fait partie de la coupe et désigne les évènements qui sont déclenchés avant le calcul du snapshot local de p. Ces évènements sont désignés par évènements "pre-Snap". La deuxième catégorie définit les évènements qui sont déclenchés par p après le calcul de son snapshot. Ces évènements ne font pas partie de la coupe. Ils sont désignés par évènements "post-Snap". Formellement, une coupe est un sous-ensemble de E_p dont les évènements doivent satisfaire la relation suivante : $\forall e, f.e \in E_p \wedge f \in C \wedge e \preceq_p f \Rightarrow e \in C$.

Une coupe consistante C est un sous-ensemble d'évènements E du système

tel que : $\forall e, f.e \in E \wedge f \in C \wedge e \preceq f \Rightarrow e \in C$.
Nous citons le Théorème suivant pour définir les relations coupes/snapshot.

Théorème 5.1. *[59] étant donné une coupe C induite par un snapshot S, les assertions suivantes sont équivalentes :*

- *S est réalisable,*
- *C est une coupe consistante,*
- *S est significatif.*

Les algorithmes de snapshot proposés dans la littérature varient selon les caractéristiques du réseau. La plus part de ces travaux se basent sur des algorithmes classiques à savoir l'algorithme de *Morgan* pour les systèmes synchrones, et les algorithmes de *Chandy-Lamport* [25] et *Lai and Yang* pour les systèmes asynchrones. étant donné que l'algorithme de *Chandy-Lamport* constitue le point de départ de notre travail, nous introduisons dans la section suivante le principe de fonctionnement de cet algorithme. Nous présentons également une preuve par raffinements proposée récemment par *Andriamiarina et al.* [7].

Algorithme de Chandy-Lamport L'algorithme de *Chandy-Lamport* [25] permet de calculer un snapshot consistant pour les systèmes asynchrones qui communiquent via des canaux FIFO, i.e., les messages sont traités par ordre d'émission. La particularité de cet algorithme consiste à utiliser des messages de contrôle spécifiques, nommés "marqueur", afin de distinguer les messages "pre-Snap" des messages "post-Snap". En effet, au moins un processus p peut initialiser le calcul du snapshot : il mémorise son état, envoie un "marqueur" pour chacun de ses voisins et enregistre tous les messages entrants. Les canaux

portant ces messages sont marqués vides. L'envoi d'un message "marqueur" permet au processus p de signaler à ses voisins le calcul de son snapshot local. étant donné que les canaux sont FIFO, les messages qui sont envoyés par p avant l'envoi du "marqueur" sont identifiés comme messages "pre-Snap". Les autres sont considérés comme messages "post-Snap". Quand un processus q reçoit un "marqueur", il se rend compte qu'un snapshot est en train de s'effectuer. Ainsi, il commence à calculer le sien. Il répète les mêmes étapes de calculs effectuées par p. Au bout d'un temps fini, quand tous les processus effectuent leurs snapshots locaux, nous obtenons un snapshot consistant contenant tous les états des processus et des canaux de communications.

Andriamiarina et al. [7] ont proposé un patron formel pour la modélisation et la preuve des algorithmes de snapshot. Ce patron est basé sur une succession de raffinements qui permet de démontrer et analyser les liens sémantiques qui relient les trois algorithmes : *Morgan* , *Chandy-Lamport* [25] et *Lai and Yang* . étant donné que notre méthodologie de raffinement se base sur la spécification de *Chandy-Lamport*, nous pensons que le patron proposé par *Andriamiarina et al.* [7] servirait de point de départ pour mettre en œvre notre approche.

5.3 Calcul d'état Global : Plan du développement

Dans cette section, nous expliquons le processus de composition que nous avons suivi pour calculer anonymement un état global du système. Cette composition est basée sur les algorithmes de *Chandy-Lamport* [25] pour le calcul du snapshot, l'algorithme *SSP* pour une détection de terminaison globale [58], et l'algorithme de *Mazurkiewicz* [48] pour la construction d'une

carte du réseau. En effet, à partir d'une spécification détaillée de *Chandy-Lamport*, nous ajoutons les étapes de calcul de SSP afin de permettre aux processus de détecter la terminaison globale du snapshot (voir Chapitre 3). Une fois cette propriété est détectée par un processus p, ce dernier peut lancer les étapes de calcul de *Mazurkiewicz* (voir Chapitre 4). Nous rappelons que cet algorithme converge vers une énumération si le réseau est minimal pour les revêtements. Dans tous les cas, il permet de construire des boites aux lettres contenants toutes les informations reçues lors de l'exécution. À partir de ces boites aux lettres, les processus peuvent calculer un graphe D qui a pour revêtement le graph initial. Dans un travail récent, Chalopin et al. [24] ont montré que si la taille du réseau est connue, et si un processus p détecte l'instant où sa boite aux lettre est stable, alors p peut calculer D. À partir de ce graphe, p peut déduire certaines propriétés stables.

La composition de ces trois algorithmes *Chandy-Lamport*, *SSP*, et *Mazurkiewicz* est basée sur des relations de raffinements présentées dans la FIGURE 5.1 : chaque processus peut lancer le calcul de *Chandy-Lamport*. Ce dernier est spécifié sur trois machines Event-B de différents niveaux d'abstraction : $SYSTEM - OBSERVATION$, $ASYNC - PROCESS$ et $FIFO - PROCESS$. La première machine présente les activités que peut subir un processus, à savoir un changement local d'état, traitement local de messages stockés, et envoi/réception de messages. Elle présente également une abstraction du calcul d'un snapshot global à un instant donné, i.e., prise d'une une photographie des états du systèmes. La deuxième machine décrit une construction asynchrone du snapshot : les processus ne calculent pas simultanément leurs snapshots locaux. La troisième machine vérifie la consistance des snapshots locaux en introduisant la notion de messages "marqueur" qui circulent dans des canaux FIFO.

Chaque processus qui termine localement son snapshot procède à l'application de l'algorithme *SSP*. Après avoir détecté la terminaison globale du snapshot, chaque processus peut lancer le calcul de *Mazurkiewicz* pour construire les boites aux lettres et collecter ainsi les snapshots locaux.

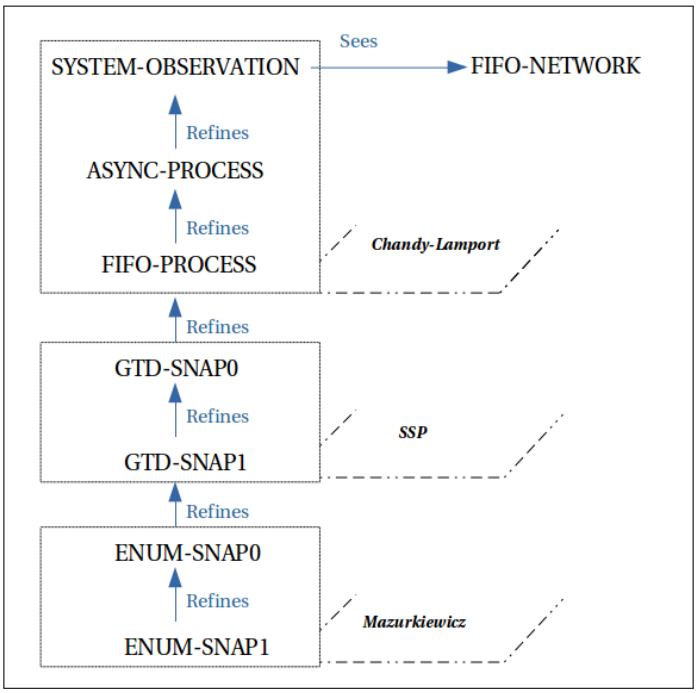

FIGURE 5.1 – Composition d'Algorithmes pour un Calcul D'état Global

5.3.1 Preuves par Raffinement de Chandy-Lamport

Nous présentons dans cette section une application du patron proposé par *Andriamiarina et al.* [7]. Considérons un ensemble de processus P ($axm1$) inter-connectés via des canaux de communications notés C. Le réseau est supposé simple ($axm2$, $axm3$, $axm4$) et connecté ($axm5$). Tout couple $(p \mapsto q)$ appartenant à C désigne qu'un processus p peut envoyer des messages au processus q via le canal $(p \mapsto q)$. Le processus p peut également recevoir des messages du processus q via le canal $(q \mapsto p)$. Si $(p \mapsto q)$ est un canal dans

C alors $(q \mapsto p)$ l'est aussi $(axm6)$. L'ensemble des messages circulant dans C est noté M $(axm7)$. L'ensemble des états des processus est désigné par $EtatP$ $(axm8)$.

$axm1 : finite(P) \wedge P \neq \varnothing$

$axm2 : finite(C) \wedge C \neq \varnothing$

$axm3 : C \subseteq P \times P \wedge dom(C) = P$

$axm4 : P \lhd id \cap C = \varnothing$

$acm5 : \forall s \cdot s \subseteq P \wedge s \neq \varnothing \wedge C[s] \subseteq s \Rightarrow P \subseteq s$

$axm6 : \forall p, q . p \mapsto q \in C \Rightarrow q \mapsto p \in C$

$axm7 : finite(M) \wedge M \neq \varnothing$

$axm8 : finite(EtatP) \wedge EtatP \neq \varnothing$

5.3.2 Niveau 0 : Observation des Activités du Système

Ce premier niveau est spécifié par la machine $SYSTEM-OBSERVATION$. Il introduit les différents types d'activités effectuées par les processus. Il décrit aussi une abstraction d'un snapshot global consistant. Des variables sont introduites comme suit :

- l spécifie l'état de chaque processus : $l \in P \rightarrow EtatP$,
- es définit l'estampille de sa dernière activité : $es \in P \rightarrow \mathbb{N}$. Pour chaque processus p $es(p)$ est initialisé à 0,
- h mémorise l'historique des processus : $h \in P \rightarrow (\mathbb{N} \nrightarrow (P \times P))$. Chaque évènement effectué par p est mémorisé instantanément dans $h(p)$ avec l'estampille courant de p. En d'autres termes, $h(p)$ est composé d'évènements numérotés de 0 à $es(p)$ (Invariant 5.1).
- Env décrit les messages qui sont envoyés : $Env \subseteq M$.
- Can caractérise les messages qui circulent dans les canaux de communications : $Can \in C \rightarrow \mathbb{P}(M)$. Ces messages doivent figurer dans

l'ensemble des messages envoyés (Invariant 5.2).

- *Stock* définit les messages qui sont enregistrés : $Stock \in P \rightarrow \mathbb{P}(M)$. Chaque message reçu par un processus q est enregistré. Ce message doit aussi figurer dans l'ensemble des messages envoyés (Invariant 5.3),
- e caractérise l'envoi d'un message circulant dans un canal. $e \in C \times \mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow M \wedge ran(e) = Env$. Un élément de e est un quadruplet $p \mapsto q \mapsto i \mapsto m$. Il désigne qu'un processus p envoie un message m au processus q à l'estampille i. L'envoi de m doit être mémorisé dans $h(p)$ (Invariant 5.4).
- r caractérise la réception d'un message : $r \in C \times \mathbb{N} \rightarrowtail\!\!\!\!\!\!\rightarrow M \wedge ran(r) \subseteq Env$. Un élément de r est un quadruplet $p \mapsto q \mapsto j \mapsto m$. Il désigne qu'un processus q reçoit un message m d'un processus p à l'estampille j. Chaque message reçu doit être déjà envoyé ($\forall p, q, m, j \cdot p \mapsto q \mapsto j \mapsto m \in r \Rightarrow (\exists i \cdot p \mapsto q \mapsto i \mapsto m \in e)$). La réception de m doit être mémorisée dans $h(q)$ (Invariant 5.5).
- *Coupe* désigne le calcul d'un snapshot global : $Coupe \in P \rightarrow \mathbb{N}$. Cette fonction est initialisée à l'ensemble vide et prendra pour valeur les estampilles du calcul des snapshots locaux.

Invariant 5.1. $\forall p \cdot p \in P \Rightarrow dom(h(p)) = 0 \mathinner{..} es(p)$.

Invariant 5.2. $\forall m, c \cdot c \in C \wedge m \in Can(c) \Rightarrow m \in Env$.

Invariant 5.3. $\forall m, p \cdot m \in store(p) \Rightarrow m \in send$.

Invariant 5.4. $\forall p, q, i, m . p \mapsto q \mapsto i \mapsto m \in s \Rightarrow i \in dom(h(p))$.

Invariant 5.5. $\forall p, q, j, m . p \mapsto q \mapsto j \mapsto m \in r \Rightarrow j \in dom(h(q))$.

Les activités qui sont internes aux processus sont spécifiées par les deux évènements suivants $TraitEtat$ et $TraitMsg$. Le premier évènement est observé

quand un nouvel état ne est associé à un processus p. Le deuxième évènement spécifie le traitement local d'un message stocké $m \in Stock(p)$. Ce message est ensuite retiré de l'ensemble $Stock(p)$. Dans les deux cas d'évènements, l'estampille courant du processus est incrémenté et l'activité correspondante est mémorisée dans $h(p)$.

EVENT $TraitEtat$
any p, ne
where
$grd1 : p \in P$
$grd2 : ne \in EtatP$
then
$act1 : l(p) := ne$
$act2 : es(p) := es(p) + 1$
$act3 : h(p) := h(p) \cup$
$\{(es(p) + 1) \mapsto (p \mapsto p)\}$

EVENT $TraitMsg$
any p, m
where
$grd1 : p \in P$
$grd2 : m \in Stock(p)$
then
$act1 : Stock(p) := Stock(p) \setminus \{m\}$
$act2 : es(p) := es(p) + 1$
$act3 : h(p) := h(p) \cup$
$\{(es(p) + 1) \mapsto (p \mapsto p)\}$

Les activités d'envoi et de réception de messages sont spécifiées respectivement par les événements $Envoi$ et $Reception$. Chaque processus p peut envoyer un message m à son voisin via un canal c sortant de p. Formellement, ce canal doit vérifier l'égalité suivante : $prj1(c) = p$. Les fonctions $prj1$ et $prj2$ sont des fonctions pré-définies dans Event-B. Elles associent respectivement le premier et le deuxième élément à chaque couple. Une activité d'envoi d'un message m doit assurer que m n'est pas encore envoyé ($m \notin Env$). Par conséquent, m ne circule pas dans le canal c ($m \notin Can(c)$). Une fois le message est envoyé, m est ajouté aux ensembles Env et $Can(c)$. De plus, l'estampille ainsi que l'historique de p sont mis à jour : $es(p)$ est incrémenté et le couple $es(p) + 1 \mapsto c$ est mémorisé dans $h(p)$. Une autre mise à jour doit être effectuée sur e pour mémoriser l'estampille d'envoi du message m par le processus p au processus q : le triplet $c \mapsto es(p) + 1 \mapsto m$ est ajouté à e.

EVENT *Envoi*

any p, m, c

where

$grd1 : p \in P$

$grd2 : c \in C \land prj1(c) = p$

$grd3 : m \notin Env$

$th1 : m \notin Can(c)$

then

$act1 : Env := Env \cup \{m\}$

$act2 : Can(c) := Can(c) \cup \{m\}$

$act3 : es(p) := es(p) + 1$

$act4 : h(p) := h(p) \cup$
$\{es(p) + 1 \mapsto c\}$

$act5 : e := e \cup$
$\{c \mapsto es(p) + 1 \mapsto m\}$

EVENT *Reception*

any q, m, c, i

where

$grd1 : q \in P$

$grd2 : c \in C \land prj2(c) = q$

$grd3 : m \in Can(c)$

$grd4 : i \in \mathbb{N} \land (c \mapsto i \mapsto m) \in e$

then

$act1 : Stock(q) := Stock(q) \cup \{m\}$

$act2 : Can(c) := Can(c) \setminus \{m\}$

$act3 : es(q) := es(q) + 1$

$act4 : h(q) := h(q) \cup$
$\{es(q) + 1 \mapsto c\}$

$act5 : r := r \cup$
$\{c \mapsto es(q) + 1 \mapsto m\}$

L'événement *Recption* est observé quand un processus q reçoit un message via un canal c entrant à q ($prj2(c) = q$). Dans ce cas, il faut vérifier que m circule dans le canal c ($m \in Can(c)$). Suite au déclenchement de l'événement, m est retiré du canal et stocké dans $Stock(q)$ et l'estampille ainsi que l'historique de q sont mis à jour. Pour mémoriser les détails de réception de m par q au moment $es(q) + 1$, le triplet $c \mapsto es(q) + 1 \mapsto m$ est ajouté à r.

Le calcul d'un snapshot global est spécifié par un seul évènement abstrait nommé *Snapshot* : toute fonction *aCoupe* ($grd1$), qui représente une coupe consistante, est considérée comme un snapshot global ($act1$). Pour chaque processus p, La dernière activité effectuée par p au moment de la coupe devrait déjà être mémorisée dans $h(p)$ ($grd3$). La consistance de *aCoupe* est vérifiée par $grd2$ de l'évènement. En effet, un processus p peut envoyer un message m au processus q au moment i ($p \mapsto q \mapsto i \mapsto m \in e$). La réception

de m pourra être effectuée à un instant j ($p \mapsto q \mapsto j \mapsto m \in r$). Dans ce cas, si j fait partie de la coupe induite par le snapshot local de q ($j \leq aCoupe(q)$) alors i doit également faire partie de la coupe ($i \leq aCoupe(p)$).

EVENT *Snapshot*

any *aCoupe*

where

$grd1 : aCoupe \in P \rightarrow \mathbb{N}$

$$grd2 : \forall p, q, i, j, m \cdot \left(\begin{array}{l} \{p, q\} \subseteq P \wedge m \in M \wedge \\ i \in dom(h(p)) \wedge j \in dom(h(q)) \wedge \\ (p \mapsto q) \mapsto i \mapsto m \in e \wedge \\ (p \mapsto q) \mapsto j \mapsto m \in r \wedge \\ j \leq aCoupe(q) \end{array} \right) \Rightarrow i \leq aCoupe(p)$$

$grd3 : \forall p \cdot p \in P \Rightarrow aCoupe(p) \in dom(h(p))$

then

$act1 : Coupe := aCoupe$

5.3.3 Niveau 1 : Calcul Asynchrone du Snapshot

Une nouvelle machine $ASYNC - PROCESS$ raffine la machine du niveau précédent. Elle introduit l'aspect asynchrone du calcul d'un snapshot. De nouvelles variables sont introduites pour caractériser les snapshots locaux et spécifier les étapes du calcul de l'algorithme de *Chandy-Lamport* :

- $SnapEtatP$ définit les états des processus qui sont mémorisés lors du snapshot : $SnapEtatP \in P \nrightarrow EtatP$.
- $EtatC$ représente les messages enregistrés par les processus avant le calcul du snapshot : $EtatC \in C \nrightarrow \mathbb{P}(M)$. Un triplet $p \mapsto q \mapsto m$ est un élément de $EtatC$ signifie que m est un message entrant à un processus q, via le canal $p \mapsto q$, et enregistré par q. Ce message est considéré comme un message "pre-Snap".

– $e\_marqueur$ mémorise les canaux de communication via lesquels un message "marqueur" a été envoyé : $e\_marqueur \subseteq C$. Le couple $p \mapsto q$ est un élément de $e\_marqueur$ signifie qu'un processus p a envoyé un "marqueur" au processus q. Si ce type d'envoi a été effectué alors le processus p devrait avoir enregistré les messages "pre-Snap" qui lui sont entrants via le canal $q \mapsto p$ (Invariant 5.6).

– $PostSnapMsg$ définit les messages qui sont envoyés après le calcul d'un snapshot : $PostSnapMsg \subseteq Env$.

– $pCoupe$ caractérise une construction progressive d'un snapshot global. Il s'agit d'une coupe qui contient les processus ayant calculé leurs snapshots locaux : $pCoupe \in P \nrightarrow \mathbb{N}$. Pour chaque processus p, le calcul d'un snapshot local signifie que p a enregistré son état (Invariant 5.7 (l1)), envoyé un marqueur à tous ses voisins (l2), et a enregistré tous les messages entrants (l3).

Invariant 5.6. $\forall p, q \cdot p \mapsto q \in e\_marqueur \Rightarrow q \mapsto p \in dom(EtatC)$.

Invariant 5.7. $\forall p \cdot p \in dom(pCoupe) \Rightarrow$

$$\begin{array}{ll} (l1) & \left(\begin{array}{ll} p \in dom(SnapEtatP) & \wedge \\ \forall i \cdot p \mapsto i \in C \Rightarrow p \mapsto i \in e\_marqueur & \wedge \\ \forall i \cdot i \mapsto p \in C \Rightarrow i \mapsto p \in dom(EtatC) & \end{array} \right). \\ (l2) & \\ (l3) & \end{array}$$

La consistance de $pCoupe$ est spécifiée par les Invariants 5.8 et 5.9 : un processus p ayant calculé son snapshot peut envoyer un message m à un processus q au moment i (Invairant 5.8 (l2)). Si q a également calculé son snapshot, mais il a reçu le message à un instant j (l3), tel que j se situe avant $pCoupe(q)$ (l4), alors l'estampille d'envoi i doit également se situer avant $pCoupe(p)$. Si q n'a pas calculé son snapshot (Invariant 5.9 (l3)) et le message envoyé par p n'est pas marqué comme un message "post-Snap" (l4)

alors l'estampille d'envoi doit aussi se situer avant le calcul du snapshot de p.

Invariant 5.8. $\forall p, q, m, i, j \cdot$

$$\left(\begin{array}{ll} (l1) & \{p, q\} \subseteq P \wedge p \neq q \wedge m \in M \wedge \\ (l2) & p \in dom(pCoupe) \wedge p \mapsto q \mapsto i \mapsto m \in e \wedge \\ (l3) & q \in dom(pCoupe) \wedge p \mapsto q \mapsto j \mapsto m \in r \wedge \\ (l4) & j \leq pCoupe(q) \end{array}\right) \Rightarrow i \leq pCoupe(p).$$

Invariant 5.9. $\forall p, q, i, m \cdot$

$$\left(\begin{array}{ll} (l1) & \{p, q\} \subseteq P \wedge m \in M \wedge p \neq q \wedge \\ (l2) & p \in dom(pCoupe) \wedge p \mapsto q \mapsto i \mapsto m \in e \wedge \\ (l3) & q \notin dom(pCoupe) \wedge \\ (l4) & m \notin PostSnapMsg \end{array}\right) \Rightarrow i \leq pCoupe(p).$$

Un nouvel évènement $InitSnap$ est introduit pour procéder au calcul d'un snapshot : l'évènement est observé quand aucun processus n'a initialisé ce calcul ($grd2$). Cette initialisation pourrait être déclenchée par un processus quelconque p ($grd1$). Dans ce cas, p est considéré comme un initiateur du snapshot : il enregistre sont état ($act1$), mémorise tous les messages entrants ($grd3$, $grd4$ et $act2$), envoie un marqueur à tous ses voisins ($grd5$, $grd6$ et $act3$), et enregistre son estampille courant pour mémoriser le moment du calcul de son snapshot ($act4$).

La progression du calcul du snapshot s'effectue quand un processus q reçoit un marqueur pour la première fois. Cette étape de calcul est spécifiée par l'événement $ProgSnap$: la réception d'un marqueur par un processus q est vérifiée par les gardes $grd1$ et $grd2$ de l'évènement. De plus, il faut vérifier que q n'a pas encore calculé son snapshot $grd3$. Les messages qui circulent dans le canal, via lequel q a reçu un marqueur, sont envoyés par un voisin ayant calculé son snapshot. Ces messages sont ainsi considérés comme des

messages “post-Snap” ($grd4$). Afin de progresser vers un snapshot global, q enregistre sont état ($act1$), mémorise tous les messages entrants ($grd5$, $grd6$ et $act2$), envoie un marqueur à tous ses voisins ($grd7$, $grd8$ et $act3$), et enregistre son estampille courant pour mémoriser le moment du calcul de son snapshot ($act4$).

Les événements *Envoi* et *Reception* sont raffinés dans ce niveau pour distinguer les activités d'envoi/réception avant et après le snapshpt. En effet, événement *Envoi* est remplacé par deux événements : $EnvoiAvantSnap$ et $EnvoiApreSnap$. Le premier spécifie un envoi d'un message m par un processus p qui n'a pas encore calculé son snapshot ($\oplus grd4$). Dans ce cas, nous pouvons déduire que m ne fait pas partie des messages $PostSnapMsg$ ($\oplus th2$). Toutefois, le deuxième événement spécifie un envoi d'un message m après un calcul d'un snapshot local ($\oplus grd5$). Dans ce cas, m est ajouté à l'ensemble $PostSnapMsg$ ($\oplus act6$).

EVENT $InitSnap$

any p, ne, em

where

$grd1 : p \in P$

$grd2 : pCupe = \varnothing$

$grd3 : ne \in C \nrightarrow \mathbb{P}(M)$

$grd4 : ne = \{d.d \in C \wedge prj2(d) = p \mid d \mapsto \varnothing\}$

$grd5 : em \subseteq C$

$grd6 : em = \{d.d \in C \wedge prj1(d) = p \mid d\}$

then

$act1 : EtatP(p) := l(p)$

$act2 : EtatC := EtatC \cup ne$

$act3 : e\_marqueur := e\_marqueur \cup em$

$act4 : pCoupe(p) := es(p)$

EVENT $ProgSnap$

any c, q, ne, em

where

$grd1 : c \in C \wedge q = prj2(c)$

$grd2 : c \in e\_marqueur$

$grd3 : q \notin dom(pCoupe)$

$grd4 : can(c) \subseteq PostSnapMsg$

$grd5 : ne \in C \nrightarrow \mathbb{P}(M)$

$grd6 : ne = \{d \in C \wedge prj2(d) = q \mid d \mapsto \varnothing\}$

$grd7 : em \subseteq C$

$grd8 : em = \{d.d \in C \wedge prj1(d) = q \mid d\}$

then

$act1 : EtatP(q) := l(q)$

$act2 : EtatC := EtatC \cup ne$

$act3 : e\_marqueur := e\_marqueur \cup em$

$act4 : pCoupe(q) := es(q)$

EVENT $EnvoiAvantSnap$

refines $Envoi$

where

$\ldots$

$\oplus grd4 : p \notin dom(pCoupe)$

$\oplus th2 : m \notin PostSnapMsg$

$\ldots$

EVENT $EnvoiApreSnap$

refines $Envoi$

where

$\ldots$

$\oplus grd5 : p \in dom(pCoupe)$

then

$\ldots$

$\oplus act6 : PostSnapMsg := PostSnapMsg \cup \{m\}$

L'événement *Reception* est remplacé par les trois événements suivants : $ReceptionPreSnapMsg$, $ReceptionPostSnapMsg$ et $ReceptionAvantSnap$. Les deux premiers événements spécifient la réception d'un message m par un processus q qui a déjà calculé son snapshot local ($\oplus grd5$). Toutefois, m pourrait être un message "pre-Snap" ou bien un message "post-Snap". L'événement $ReceptionPreSnapMsg$ caractérise la réception des messages "Pre-Snap". Une nouvelle garde $\oplus grd6$ est ainsi ajoutée pour vérifier que m ne fait pas partie de l'ensemble $PostSnapMsg$. Dès réception, m devrait être enregistré parmi l'ensemble $EtatC(c)$ ($\oplus act6$). L'événement $ReceptionPostSnapMsg$ caractérise la réception des messages "post-Snap". Une nouvelle garde ($\oplus grd6$) est ajoutée pour vérifier que m fait partie de l'ensemble $PostSnapMsg$. Dès réception, m n'est plus considéré comme un message "post-Snap". Il est ainsi retiré de la liste correspondante ($\oplus act6$).

EVENT $ReceptionPreSnapMsg$
refines $Reception$
where
$\cdots$
$\oplus grd5 : q \in dom(pCoupe)$
$\oplus grd6 : m \notin PostSnapMsg$
then
$\cdots$
$\oplus act6 : EtatC(c) :=$
$EtatC(c) \cup \{m\}$

EVENT
$ReceptionPostSnapMsg$
refines $Reception$
where
$\cdots$
$\oplus grd5 : q \in dom(pCoupe)$
$\oplus grd6 : m \in PostSnapMsg$
then
$\oplus act6 : PostSnapMsg :=$
$PostSnapMsg \setminus \{m\}$

L'événement $ReceptionAvantSnap$ spécifie la réception d'un message m par un processus q qui n'a pas encore calculé son snapshot ($\oplus grd5$). Dans ce cas, m ne doit pas faire partie des messages $PostSnapMsg$ ($\oplus grd6$) car, à cet instant de réception, q n'a pas encore reçu un marqueur. L'événement *Snapshot* est raffiné en remplaçant le paramètre abstrait $aCoupe$ par la nouvelle va-

riable *pCoupe* (*with*). Les gardes de l'événement sont ainsi remplacées par de nouvelles gardes spécifiées en fonction de la variable *pCoupe*. En effet, un snapshot global est effectué sous la condition que tous les processus ont localement calculé leurs snapshots ($\oplus grd1$). De plus, tous les processus doivent avoir signalé ce calcul : l'ensemble des canaux de communication, via lesquels un message "marqueur" a été envoyé, doit constituer tous les canaux du réseau ($\oplus grd2$). En outre, il faut s'assurer que tous les messages qui circulent via ces canaux sont des messages "post-Snap" ($\oplus grd3$).

EVENT *ReceptionAvantSnap*
refines *Reception*
where
...
$\oplus grd5 : q \notin dom(pCoupe)$
$\oplus grd6 : m \notin PostSnapMsg$
...

EVENT *Snapshot′*
refines *Snapshot*
where
$\ominus grd1, \ominus grd2, \ominus grd3$
$\oplus grd1 : dom(pCoupe) = P$
$\oplus grd2 : e\_marqueur = C$
$\oplus grd3 : \forall c \cdot c \in C$
$\Rightarrow Can(c) \subseteq PostSnapMsg$
with
$aCoupe : aCoupe = pCoupe$
then
$\ominus act1 : Coupe := aCoupe$
$\oplus act1 : Coupe := pCoupe$

5.3.4 Niveau 2 : Communication FIFO

La machine du niveau précédent est raffinée par une nouvelle machine nommée $FIFO - PROCESS$. Cette nouvelle machine dévoile la propriété FIFO des canaux de communication. De nouvelles variables sont introduites pour ordonner les canaux en des files FIFO et caractériser les indices d'envoi et de réception des messages, notamment le message "marqueur" :

– $File$ remplace la variable Can du niveau précédent. Elle associe à chaque canal une numérotation des messages qui circulent dans ce canal : $File \in C \rightarrow (\mathbb{N} \nrightarrow M) \wedge \forall d \cdot d \in C \Rightarrow ran(File(d)) = Can(d)$. Les canaux de communication sont traités comme des files ordonnés. Chaque message de la file, i.e., du canal est associé à un numéro unique (Invariant 5.10).

– iR associe à chaque canal l'indice du dernier message reçu via ce canal : $iR \in C \rightarrow \mathbb{N}$. Un triplet $p \mapsto q \mapsto i$ est un élément de iR signifie que i est l'indice courant du dernier message reçu par un processus q via un canal $p \mapsto q$, entrant à q. L'indice i est toujours en tête de la file $p \mapsto q$ (Invariant 5.12).

– iE associe, à chaque canal l'indice courant d'envoi d'un message via ce canal : $iE \in C \rightarrow \mathbb{N}$. Un triplet $p \mapsto q \mapsto j$ est un élément de iE signifie que j est l'indice courant d'envoi d'un processus p. L'indice $j - 1$ est l'indice du dernier message envoyé par p via un canal $p \mapsto q$, sortant de p. Il est toujours est situé en queue de file (Invariant 5.12). Notons que l'indice tête doit être inférieur ou égale à l'indice queue (Invariant 5.11). En cas d'égalité, la file est considérée vide (Invariant 5.13).

– iM associe à chaque canal l'indice du message marqueur diffusé dans ce canal : $iM \in C \nrightarrow \mathbb{N} \wedge dom(iM) = e\_marqueur$. Un triplet $p \mapsto q \mapsto i$ est un élément de iM signifie que p a envoyé un message "marqueur" à un processus voisin q et ce message ce situe à l'indice i de la file $p \mapsto q$. Dans ce cas, p doit avoir calculé son snapshot (Invariant 5.14). Notons que l'indice d'un message "marqueur" ne doit pas dépasser la queue de file (Invariant 5.15).Si l'indice du dernier message reçu par un processus q dépasse l'indice d'un message "marqueur", qui a été diffusé dans un canal $p \mapsto q$, alors tous les messages courants qui circulent

dans ce canal sont des messages "post-shot" (Invariant 5.16). De plus, si un message m circule dans un canal avec un indice inférieur à celui du "marqueur" alors m est envoyé avant le calcul du snapshot. Il ne fait pas partie des messages "post-shot" (Invariant 5.17). Dans le cas contraire, i.e., m est associé à un indice supérieur à celui du marqueur, alors m fait partie des messages "post-Snap" (Invariant 5.18).

Invariant 5.10. $\forall d, i1, i2, m \cdot d \in C \land i1 \mapsto m \in File(d) \land i2 \mapsto m \in File(d) \Rightarrow i1 = i2.$

Invariant 5.11. $\forall d \cdot d \in C \Rightarrow iR(d) \leq iE(d).$

Invariant 5.12. $\forall d \cdot d \in C \land File(d) \neq \varnothing \Rightarrow dom(File(d)) = iR(d)..iE(d)-1.$

Invariant 5.13. $\forall d \cdot d \in C \Rightarrow (iR(d) = iE(d) \Leftrightarrow File(d) = \varnothing).$

Invariant 5.14. $dom(pCoupe) = dom(dom(iM)).$

Invariant 5.15. $\forall d \cdot d \in dom(iM) \Rightarrow iM(d) \leq iE(d).$

Invariant 5.16. $\forall d \cdot d \in e\_marqueur \land iR(d) \geq iM(d)$
$\Rightarrow (\forall m \cdot m \in ran(File(d)) \Rightarrow m \in PostSnapMsg).$

Invariant 5.17. $\forall m, d, i \cdot d \in e\_marqueur \land i < iM(d) \land i \mapsto m \in File(d)$
$\Rightarrow m \notin PostSnapMsg.$

Invariant 5.18. $\forall m, d, i \cdot d \in e\_marqueur \land i > iM(d) \land i \mapsto m \in File(d)$
$\Rightarrow m \in PostSnapMsg.$

Les événements $InitSnap$ et $ProgSnap$ sont raffinés dans la machine $FIFO - PROCESS$ pour spécifier les indices des messages "marqueur" couplés avec les canaux via lesquels ils ont été envoyés. En effet, lors du calcul d'un snapshot par un processus p, un "marqueur" est envoyé par p via tous

les canaux sortants de p. Ces canaux sont spécifiés par l'ensemble suivant : $(\{d|d \in C \wedge prj1(d) = p\})$. Admettons que iEm contient les indices des messages "marqueur" envoyés par p. Le contenu de iEm correspond aux indices courants d'envoi (iE) projetés sur les canaux sortants de p. iEm est spécifié, dans l'événement raffiné, $InitSnap'$ (respectivement $ProgSnap'$), par les nouvelles gardes $\oplus grd7$ et $\oplus grd8$ (respectivement $\oplus grd9$ et $\oplus grd10$).

Dans les deux cas d'événements, le contenu de iM doit être mis à jour. S'il s'agit d'une initialisation du snapshot, iM devrait être vide. Ainsi, suite au déclenchement de $InitSnap'$, iM prendra pour valeur le contenu de iEm ($\oplus act5$ de $InitSnap'$). Dans le deuxième cas d'événement ($ProgSnap'$), une mise à jour est effectuée sur une partie de iM qui correspond aux indices des messages marqueurs circulant via les canaux sortant de p ($\oplus act5$ de $ProgSnap'$).

EVENT $InitSnap'$
refines $InitSnap$
any $\cdots$
$\oplus iEm$
where
$\cdots$
$\oplus grd7 : iEm \in C \nrightarrow \mathbb{N}$
$\oplus grd8 : iEm = (\{d\|d \in C \wedge prj1(d) = p\}) \lhd iE$
then
$\cdots$
$\oplus act5 : iM := iEm$

EVENT $ProgSnap'$
refines $ProgSnap$
any $\cdots$
$\oplus iEm$
where
$\cdots$
$\oplus grd9 : iEm \in C \nrightarrow \mathbb{N}$
$\oplus grd10 : iEm = (\{d\|d \in C \wedge prj1(d) = i\} \lhd iE)$
then
$\cdots$
$act5 : iM := iM \lhd\!\!+ iEm$

Les événements $EnvoiAvantSnap$ et $EnvoiApreSnap$ sont respectivement raffinés par $EnvoiAvantSnap'$ et $EnvoiApreSnap'$: l'indice courant d'envoi doit être mis à jour après déclenchement de chacun des deux événe-

ments ($\oplus act6$). De plus, étant donné que la variable Can est remplacée par une file ordonnée $File$, les éléments $th1$ et $act2$ des deux événements abstraits doivent être également remplacés : un message m ne circule pas dans $Can(c)$ ($\ominus th1$) signifie que le couple $iE(c) \mapsto m$ ne circule pas dans $File(c)$ ($\oplus th1$). Suite au déclenchement de l'événement, ce couple est ajouté à la file ($\oplus act2$).

EVENT $EnvoiAvantSnap'$
refines $EnvoiAvantSnap$
...
where
...
$\ominus th1 : m \notin Can(c)$
$\oplus th1 : iE(c) \mapsto m \notin File(c)$
then
...
$\ominus act2 : Can(c) := Can(c) \cup \{m\}$
$\oplus act2 : File(c) := File(c) \cup \{iE(c) \mapsto m\}$
$\oplus act6 : iE(c) := iE(c) + 1$

EVENT $EnvoiApreSnap'$
refines $EnvoiApreSnap$
...
where
...
$\ominus th1 : m \notin Can(c)$
$\oplus th1 : iE(c) \mapsto m \notin File(c)$
then
...
$\ominus act2 : Can(c) := Can(c) \cup \{m\}$
$\oplus act2 : File(c) := File(c) \cup \{iE(c) \mapsto m\}$
$\oplus act6 : iE(c) := iE(c) + 1$

Les événements, $ReceptionPreSnapMsg$, $ReceptionPostSnapMsg$ ainsi que $ReceptionAvantSnap$ sont respectivement raffinés dans ce niveau par les trois événements $ReceptionPreSnapMsg'$, $ReceptionPostSnapMsg'$ et $ReceptionAvantSnap'$: l'indice courant de réception doit être mis à jour après déclenchement de chacun des trois événements ($\oplus act6$). De plus, un message m circule pas dans $Can(c)$ ($\ominus grd3$) signifie que le couple $iR(c) \mapsto m$ circule pas dans $File(c)$ ($\oplus grd3$). Suite au déclenchement de l'événement, ce couple est retiré de la file ($\ominus \oplus act2$). D'autre part, nous rappelons que les événements $ReceptionPreSnapMsg$ et $ReceptionAvantSnap$ traitent le cas

où le message courant de réception ne fait pas partie des messages "post-shot" ($m \notin PostSnapMsg$). Pour vérifier si un message a été envoyé avant/après le snapshot, nous utilisons l'indice du message "marqueur" iM plutôt que l'ensemble $PostSnapMsg$ ($\ominus grd6$ des trois événements de réception et $\ominus act6$ de l'événement $ReceptionPostSnapMsg'$). Il suffit de comparer l'indice du message par rapport à celui du message "marqueur" ($\oplus grd7$ des événements $ReceptionPreSnapMsg'$ et $ReceptionAvantSnap'$).

EVENT
$ReceptionPreSnapMsg'$
refines $ReceptionPreSnapMsg$
...
where
...
$\ominus grd3 : m \in Can(c)$
$\oplus grd3 : iR(c) \mapsto m \in File(c)$
$\ominus grd6 : m \notin PostSnapMsg$
$\oplus grd6 : c \notin e\_marqueur \vee$
$iR(c) < iM(c)$
then
...
$\ominus act2 : Can(c) := Can(c) \setminus \{m\}$
$\oplus act2 : File(c) := File(c) \setminus$
$\{iR(c) \mapsto m\}$
$\oplus act6 : iR(c) := iR(c) + 1$

EVENT
$ReceptionPostSnapMsg'$
refines $ReceptionPostSnapMsg$
...
where
...
$\ominus grd3 : m \in Can(c)$
$\oplus grd3 : iR(c) \mapsto m \in File(c)$
$\ominus grd6 : m \in PostSnapMsg$
$\oplus grd6 : c \in e\_marqueur \wedge$
$iR(c) \geq iM(c)$
then
...
$\ominus act2 : Can(c) := Can(c) \setminus \{m\}$
$\oplus act2 : File(c) := File(c) \setminus$
$\{iR(c) \mapsto m\}$
$\ominus act6 : PostSnapMsg :=$
$PostSnapMsg \setminus \{m\}$
$\oplus act6 : iR(c) := iR(c) + 1$

L'événement $ReceptionPostSnapMsg$ est déclenché quand m fait partie des

messages “post-shot”. Dans ce cas, un “marqueur” devrait avoir circulé dans c et la réception de m devrait être effectuée après la réception du “marqueur” ($\oplus grd6$ de l'événement $ReceptionPostSnapMsg$). Un snapshot global est calculé si tous les messages qui circulent dans l'ensemble Can font parties des messages “post-shot”. Cette propriété devrait être vérifiée pour tous les canaux lors du calcul d'un snapshot global. Ainsi, un événement $Snapshot''$ raffine l'événement $Snapshot'$ du niveau précédent pour remplacer la garde $grd3$ de l'événement abstrait : $Can(c) \subseteq PostSnapMsg$ signifie que $iR(c) \geq iM(c)$.

EVENT $ReceptionAvantSnap'$

refines $ReceptionAvantSnap$

...

where

...

$\ominus grd3 : m \in Can(c)$

$\oplus grd3 : iR(c) \mapsto m \in File(c)$

$\oplus grd7 : c \notin e\_marqueur \vee$
$iR(c) < iM(c)$

then

...

$\ominus act2 : Can(c) := Can(c) \setminus \{m\}$

$\oplus act2 : File(c) := File(c) \setminus$
$\{iR(c) \mapsto m\}$

$\oplus act6 : iR(c) := iR(c) + 1$

EVENT $Snapshot''$

refines $Snapshot'$

...

where

...

$\ominus grd3 : \forall c \cdot c \in e\_marqueur$
$\Rightarrow Can(c) \subseteq PostSnapMsg$

$\oplus grd3 : \forall c \cdot c \in C$
$\Rightarrow iR(c) \geq iM(c)$

...

5.4 Détection de Terminaison Globale de Chandy-Lamport

Dans cette section, nous spécifions une composition de Chandy-Lamport avec l'algorithme SSP [58] afin de détecter la terminaison globale du snapshot. Cette composition est spécifiée par une suite de raffinements appliquée au dernier niveau de la spécification de *Chandy-Lamport*. Ainsi, nous introduisons deux autres niveaux de raffinements détaillés dans les sections suivantes (Section 5.4.1, Section 5.4.2).

5.4.1 Niveau 3 : Vers une Détection de Terminaison du Snapshot Global

Dans ce niveau, la machine $FIFO - PROCESS$ est raffinée par une autre machine $GTD - SNAP_0$. Lors de ce raffinement, nous introduisons les interactions locales effectués par les processus afin de procéder à une détection de terminaison globale. De nouvelles variables est événements sont introduits pour spécifier les informations nécessaires autour de la terminaison locale des processus et leur diffusion progressive dans le réseau :

- $Local\_Snapshot$ caractérise la convergence des processus par rapport au calcul de leurs snapshots locaux : $Local\_Snapshot \in P \rightarrow BOOL$. Pour tout processus p, $Local\_Snapshot(p) = FALSE$ signifie que p n'a pas calculé son snapshot (Invariant 5.19, Invariant 5.20). Dans ce cas, nous pouvons déduire que p n'a pas mémorisé son état ou bien il n'a pas encore enregistré l'un de ses canaux entrants (Théorème 5.2).
- a est un rayon de confiance qui permet de calculer le nombre de voisins qui ont terminé localement : $a \in P \rightarrow \mathbb{N} \cup \{-1\}$. Pour tout processus p, $a(p)$ est un entier qui est initialisé à -1 et qui ne dépasse pas $card(P)$

$(\forall e \cdot e \in ran(a) \Rightarrow e \leq diameter)$.

- e_a caractérise l'envoi d'un rayon de confiance circulant dans un canal de communication : $e\_a \in (C \times \mathbb{N}) \nrightarrow \mathbb{N}$. Un élément de a est un quadruplet $p \mapsto q \mapsto i \mapsto e$. Il désigne qu'un processus p envoie un rayon de confiance e à un processus voisin q, au moment i. Le rayon de confiance envoyé doit être toujours inférieur ou égale au rayon de confiance courant (Invariant 5.21). De plus, l'estampille d'envoi de p ne doit pas dépasser son estampille courant (Invariant 5.22).
- r_a caractérise la réception d'un rayon de confiance.$r\_a \in C \times \mathbb{N} \nrightarrow \mathbb{N}$. Un élément de r_a est un quadruplet $p \mapsto q \mapsto j \mapsto e$. Il désigne qu'un processus q reçoit un entier e au moment j. Nous nous assurons que e est un rayon de confiance qui a été envoyé par un processus voisin p (Invariant 5.23). Ainsi, e ne doit pas dépasser $a(p)$ (Invariant 5.24). De plus, il faut vérifier que l'estampille de réception de q est toujours inférieur ou égale à l'estampille courant (Invariant 5.25).

Invariant 5.19. $\forall p \cdot p \in P \wedge p \notin dom(pCoupe) \Rightarrow Local\_Snapshot(p) = FALSE$.

Invariant 5.20. $\forall p \cdot Local\_Snapshot(p) = FALSE \Rightarrow p \notin dom(pCoupe)$.

Théorème 5.2. $\forall p \cdot Local\_Snapshot(p) = FALSE \Rightarrow p \notin dom(SnapEtatP) \vee (\exists c \cdot c \in C \wedge prj2(c) = p \wedge c \notin dom(EtatC))$.

Invariant 5.21. $\forall c, p, i, n \cdot c \in C \wedge p = prj1(c) \wedge n \geq 0 \wedge$
$c \mapsto i \mapsto n \in e\_a \Rightarrow n \leq a(p)$.

Invariant 5.22. $\forall p, q, n \cdot p \mapsto q \mapsto n \in dom(e\_a) \Rightarrow n \leq es(p)$.

Invariant 5.23. $\forall p, q, n, j \cdot p \mapsto q \mapsto j \mapsto n \in r\_a \Rightarrow (\exists i \cdot p \mapsto q \mapsto i \mapsto n \in e\_a)$.

Invariant 5.24. $\forall p, q, i, n \cdot p \mapsto q \mapsto i \mapsto n \in r\_a \Rightarrow n \leq a(p)$.

Invariant 5.25. $\forall p, q, n \cdot p \mapsto q \mapsto n \in dom(r\_a) \Rightarrow n \leq es(q)$.

Pour chaque processus p, nous vérifions sa détection de terminaison locale par rapport aux changements de valeur de son rayon de confiance. En effet, p détecte sa terminaison locale si, est seulement si, son rayon de confiance prendra pour valeur un entier positif (Invariant 5.26, Invariant 5.27).

Invariant 5.26. $\forall p \cdot a(p) \geq 0 \Leftrightarrow Local\_Snapshot(p) = TRUE.$

Invariant 5.27. $\forall p \cdot a(p) = -1 \Leftrightarrow Local\_Snapshot(p) = FALSE.$

Nous vérifions, par Invariant 5.27 et Invariant 5.28, que la différence des rayons de confiance qui sont calculés par deux processus voisins est toujours inférieur ou égale à 1.

Invariant 5.28. $\forall p, q \cdot p \mapsto q \in C \wedge a(p) \leq a(q) \Rightarrow a(p) - a(q) \leq 1.$

Invariant 5.29. $\forall p, q \cdot p \mapsto q \in C \wedge a(p) > a(q) \Rightarrow a(q) - a(p) \leq 1.$

Les deux événements $InitSnap'$ est $ProgSnap'$ sont respectivement raffinés par $InitSnap''$ et $ProgSnap''$. En effet, nous rappelons que le calcul d'un snapshot local résulte du déclenchement de ces deux événements. Chaque processus p qui calcul son snapshot, détecte sa terminaison locale. Ainsi, p doit procéder aux étapes de calcul de l'algorithme de détection de terminaison global. Donc, il met à jour son étiquette $Local\_Snapshot$ et son rayon de confiance $a(p)$ ($\oplus act6, \oplus act7$).

EVENT $InitSnap''$
refines $InitSnap'$
...
then
...
$\oplus act6 : Local\_Snapshot(p) := TRUE$
$\oplus act7 : a(p) := 0$

EVENT $ProgSnap''$
refines $InitSnap'$
...
then
...
$\oplus act6 : Local\_Snapshot(q) := TRUE$
$\oplus act7 : a(q) := 0$

Nous ajoutons un nouvel événement $DetectLocalSnap$ pour vérifier la correction d'une détection de terminaison locale. Cet événement se déclenche dés qu'un processus p calcule son snapshot local ($grd1$). Dans ce cas, p devrait avoir enregistré son état ainsi que tous les messages entrants ($\oplus th1$). Cet événement consiste en un traitement interne d'états des processus. Par conséquent, il est spécifié par un raffinement de l'événement $TraitEtat$ (voir Section 5.3.2).

EVENT $DetectLocalSnap$ **refines** $TraitEtat$

any p

where

$grd1 : p \in P \wedge p \in dom(pCoupe)$

$th1 : p \in dom(SnapEtatP) \wedge (\forall c \cdot c \in C \wedge prj2(c) = p \Rightarrow c \in dom(EtatC))$

$grd2 : Local\_Snapshot(p) = FALSE \wedge EtatCa(p) = -1$

with $ne : ne = l(p)$

then

$act1 : es(p) := es(p) + 1$

$act2 : h(p) := h(p) \cup \{(es(p) + 1) \mapsto (p \mapsto p)\}$

$act3 : Local\_Snapshot(p) := TRUE$

$act4 : a(p) := 0$

Contrairement à la description présentée dans le Chapitre 3, nous spécifions les étapes de calcul de SSP sans faire une abstraction du mode de communication. Dans ce cas, les processus doivent effectuer certaines activités (d'envoi, de réception ou de traitement locale) afin de collecter des informations sur leurs voisins.

Un nouvel événement $Envoi\_a$ est ajouté à la machine $GTD-SNAP_0$ afin de spécifier une activité d'envoi d'un rayon de confiance d'un processus p. Cette activité désigne que p voudrait informer ses voisins de sa terminaison locale. Ainsi, p devrait avoir terminé le calcul de son snapshot. Par conséquent, $Envoi\_a$ doit être déclenché quand le prédicat $Local\_Snapshot(p)$ est satisfait ($\oplus grd6$). Dans ce cas, le message qui correspond à cet d'envoi constitue un message "post-shot". D'où, l'événement $Envoi\_a$ est spécifié par un raffinement de l'événement $EnvoiApreSnap'$.

Afin de ne pas avoir des envois redondants, il faut vérifier que $a(p)$ n'a pas été envoyé ($\oplus grd8$). Une fois déclenché, l'événement mémorise l'action d'envoi ($\oplus act7$). Notons que, pour cet événement, nous traitons le cas où $a(p)$ n'a pas encore atteint le diamètre du réseau ($\oplus grd7$).

EVENT $Envoi\_a$ **refines** $EnvoiApreSnap'$

...

where

...

$\oplus grd6 : Local\_Snapshot(p) = TRUE$

$\oplus grd7 : a(p) \geq 0 \wedge a(p) < diameter$

$\oplus grd8 : \forall j, a' \cdot c \mapsto j \mapsto a' \in e\_a \Rightarrow a(p) > a'$

then

...

$\oplus act7 : e\_a := e\_a \cup \{c \mapsto es(p) + 1 \mapsto a(p)\}$

Nous rappelons qu'un message envoyé par un processus p pourrait être reçu par un processus q selon différents événements de réception : une réception avant le calcul du snapshot de q ($ReceptionAvantSnap'$), ou une réception après le snapshot de q. Dans ce deuxième cas, le message envoyé pourrait être un message "pre-shot"($ReceptionPreSnapMsg'$), ou un message "post-shot" (selon l'événement $ReceptionPostSnapMsg'$). étant donné que l'envoi d'un rayon de confiance est considéré comme un envoi d'un message "post-shot", sa réception doit être effectuée selon l'événement $ReceptionPostSnapMsg'$, dans le cas où q a calculé son snapshot. Dans le cas contraire, le message est reçu selon $ReceptionAvantSnap'$.

Cette activité de réception est spécifiée par un nouvel événement $Reception\_a$ qui raffine $ReceptionPostSnapMsg'$ et $ReceptionAvantSnap'$. En effet, ces deux événements disposent des mêmes gardes que l'événement abstrait $Reception$, i.e., $grd1 \cdots grd4$ (voir Section 5.3.2). Nous rappelons que l'événement $Reception$ est raffiné dans un premier niveau par les événements $ReceptionPostSnapMsg$ et $ReceptionAvantSnap$. Un autre raffinement est introduit ensuite pour déclencher

$ReceptionPostSnapMsg'$ et $ReceptionAvantSnap'$. Ce raffinement a fait l'objet de nouvelle gardes spécifiées indifféremment, selon le cas d'événement ($\oplus grd5$ et $\oplus grd6$ dans $ReceptionPostSnapMsg$ et $ReceptionAvantSnap$/ $\oplus grd6$ dans $ReceptionPostSnapMsg'$ et $ReceptionAvantSnap'$).

EVENT $Reception\_a$

refines $ReceptionAvantSnap', ReceptionPostSnapMsg'$

any $q, m, c, i, ancien\_re, e$

where

$grd1 : q \in P$

$grd2 : c \in C \land prj2(c) = q$

$grd3 : iR(c) \mapsto m \in File(c)$

$grd4 : i \in \mathbb{N} \land (c \mapsto i \mapsto m) \in e$

$$grd5 : \left(\begin{array}{l} q \in dom(pCoupe) \land \\ c \in e\_marqueur \land \\ j \geq iM(c) \end{array} \right) \lor \left(\begin{array}{l} q \notin dom(pCoupe) \land \\ (c \notin e\_marqueur \lor j < iM(c)) \end{array} \right)$$

$grd6 : \forall k \cdot c \mapsto k \mapsto e \notin r\_a$

$grd7 : ancien\_re = \{k, n \cdot \{k, n\} \subseteq \mathbb{N} \land c \mapsto k \mapsto n \in r\_a | c \mapsto k \mapsto n\}$

then

$act1 : Stock(q) := Stock(q) \cup \{m\}$

$act2 : File(c) := File(c) \setminus \{iR(c) \mapsto m\}$

$act3 : es(q) := es(q) + 1$

$act4 : h(q) := h(q) \cup \{es(q) + 1 \mapsto c\}$

$act5 : r := r \cup \{c \mapsto es(q) + 1 \mapsto m\}$

$act6 : iR(c) := iR(c) + 1$

$act7 : r\_a := (r\_a \setminus ancien\_re) \cup \{c \mapsto es(q) + 1 \mapsto e\}$

La fusion du raffinement de $ReceptionAvantSnap'$ et $ReceptionPostSnapMsg'$ en un seul événement concret $Reception\_a$ doit vérifier les conditions de déclenchement des deux événements et produire les mêmes actions. Ainsi, l'événement

Reception_a doit être déclenché si la disjonction des gardes des deux événements abstraits est vérifiée ($grd5$ de l'événement *Reception_a*). De plus, il faut vérifier que le rayon de confiance, en cours de réception, n'a pas été reçu auparavant ($grd6$). Une fois déclenché, l'événement produit les mêmes actions que dans les deux événements abstraits ($act1 \cdots act6$). Il mémorise en plus l'action de réception en mettant à jour r_a ($\oplus grd7, \oplus act7$).

Suite au déclenchement d'un nouvel événement *ProgGtdSnap*, chaque processus commence à détecter la terminaison des autres processus distants. En effet, *ProgGtdSnap* raffine l'événement $TraitEtat$. Il est observé quand un processus p, qui a calculé son snapshot ($grd2$), commence à recevoir des informations sur les rayons de confiance de ses voisins ($grd3$). Si la valeur minimale des ces rayons, noté n, est plus grande que $a(p)$ ($grd4$ et $grd5$), alors le rayon de confiance de p augmente et prend pour valeur $n+1$. ($act3$). Par conséquent, p détecte la terminaison de $n+1$ processus distants.

EVENT *ProgGtdSnap*
refines $TraitEtat$
any p, n
where
$grd1 : p \in P$
$grd2 : Local\_Snapshot(p) = TRUE$
$grd3 : \exists c, i, k \cdot prj2(c) = p \wedge c \mapsto i \mapsto k \in r\_a$
$grd4 : n = min(\{c, i, k \cdot prj2(c) = p \wedge c \mapsto i \mapsto k \in r\_a | k\})$
$grd5 : n < diameter \wedge n \geq a(p)$
$grd6 : \forall q \cdot q \in P \wedge distance(p \mapsto q) \leq n+1 \Rightarrow a(q) \in ran(r\_a)$
with $ne : ne = l(p)$
then
$act1 : es(p) := es(p) + 1$
$act2 : h(p) := h(p) \cup \{(es(p)+1) \mapsto (p \mapsto p)\}$
$act3 : a(p) := n + 1$

5.4.2 Niveau 4 : Détection de Terminaison du snapshot Global

Dans ce niveau, nous introduisons une nouvelle machine $GTD - SNAP_1$ qui raffine la machine du niveau précédent. L'objectif est de converger vers une détection de terminaison du snapshot global. Une nouvelle variable $Global\_Snapshot$ est introduite afin de caractériser les processus qui ont détecté cette terminaison : $Global\_Snapshot \in P \rightarrow BOOL$. Le rayon de confiance d'un processus p atteint le diamètre du réseau, signifie que tous les processus, qui sont distants de p d'une valeur égale au diamètre, ont calculé leurs snapshots locaux. Par conséquent, p détecte la terminaison global du snapshot et le prédicat $Global\_Snapshot(p) = TRUE$ (Invariant 5.30).

Invariant 5.30. $\forall p \cdot (a(p) \geq diameter \Leftrightarrow Global\_Snapshot(p) = TRUE)$

Considérons deux processus p et q, tel que p a calculé son snapshot, nous vérifions par l'Invariant 5.31 que si la distance de $p \mapsto q$ est inférieur au rayon de confiance de p alors q a également calculé son snapshot.

Invariant 5.31. $\forall p, q \cdot \{p, q\} \subseteq P \wedge a(p) \geq 0 \wedge distance(p \mapsto q) \leq a(p) \Rightarrow a(q) \geq 0$

Nous vérifions par le Théorème 5.3 que si le prédicat $Global\_Snapshot$ est satisfait pour un processus p, alors tous les autres processus du réseau ont localement terminé le calcul de leurs snapshots. De plus, il est bien clair que, dans ce cas, p préserve aussi sa terminaison locale (Théorème 5.4).

Théorème 5.3. $\forall p, q \cdot a(p) \geq 0 \wedge q \in P \wedge distance(p \mapsto q) \leq a(p) \Rightarrow a(q) \geq 0$

Théorème 5.4. $\forall p \cdot Global\_Snapshot(p) = TRUE \Rightarrow Local\_Snapshot(p) = TRUE$

Un nouvel événement $GtdSnap$ est ajouté afin de déclencher la détection de terminaison global du snapshot par un processus p. En effet, cette propriété est vérifiée si le rayon de confiance de p atteint le diamètre du réseau.

EVENT $GtdSnap$

refines $TraitEtat$

any p

where

$grd1 : p \in P$

$grd2 : a(p) \geq diameter$

$grd3 : Global\_Snapshot(p) = FALSE$

with $ne : ne = l(p)$

then

$act1 : es(p) := es(p) + 1$

$act2 : h(p) := h(p) \cup \{(es(p) + 1) \mapsto (p \mapsto p)\}$

$act3 : Global\_Snapshot(p) = TRUE$

5.5 Échange d'Informations et Collection Des Snapshots Locaux

Nous spécifions une composition de l'algorithme d'énumération de *Mazurkiewicz* [48] avec le calcul du snapshot, après avoir détecté la terminaison globale du calcul. L'objectif de cette composition consiste à construire des mémoires (boites aux lettres) afin de collecter le maximum d'informations sur les processus du réseau (voir la description de l'algorithme de *Mazurkiewicz* dans Chapitre 4, Section 4.2). En effet, chaque élément du réseau est étiqueté par son état qui l'a enregistré lors du calcul du snapshot. Après avoir effectué les étapes de calculs de l'algorithme de *Mazurkiewicz*, chaque processus devrait construire une vue locale, puis une mémoire contenant les numéros des autres processus, ainsi que leurs snapshots locaux (Voir Annexe A).

5.6 Conclusion

Dans ce chapitre, nous avons abordé le problème du calcul d'état global dans un réseau anonyme. Un état global est caractérisé par l'ensemble des états locaux des processus ainsi que des canaux de communications. L'état local d'un processus est défini par l'état de sa mémoire et de l'historique de ses activités. L'état d'un canal de communication est caractérisé par l'ensemble de messages circulant dans ce canal.

Cependant, dans un réseau anonyme, il est difficile de collecter anonymement des informations sur l'état global du système. Nous avons montré qu'une solution à ce problème consiste à réutiliser des algorithmes et preuves existantes en appliquant une certaine liaison de composition entre le calcul du snapshot, la détection de terminaison globale et la construction d'une cartographie du réseau. Nous avons montré que cette liaison pourrait être spécifiée et prouvée par un développement *correct-par-construction*, basée sur une succession de raffinements.

Conclusion Générale

Dans ce mémoire, nous avons étudié certains problèmes de l'algorithmique distribué. Ce concept, qui a fait preuve d'une révolution informatique, soulève des problèmes fondamentaux de vérification, de modélisation, d'analyse, etc. Les défis associés à ces problèmes font l'objet de plusieurs travaux de recherche qui abordent le sujet de différentes manières. Certains donnent des solutions selon un contexte technologique bien déterminé et d'autres traitent les problèmes d'une manière générale, tout en proposant des solutions applicables dans différents contextes, et sous certaines hypothèses.

Nos travaux se situent au deuxième type de contributions. Nous avons abordé la vérification formelle des algorithmes de base qui permettent de résoudre quelques problèmes du distribué, notamment la détection de terminaison, l'énumération et le calcul d'état global. Nous avons considéré un réseau fiable et de processus anonymes qui communiquent par échange de message. Dans le chapitre 1, nous avons présenté les différents aspects de l'algorithmique distribuée et nous avons mis l'accent sur un modèle particulier : le modèle des calculs locaux. Ce modèle fait une abstraction du mode de communication en utilisant des règles qui permettent de modifier les états des éléments du réseaux. Nous avons utilisé ce modèle lors de l'étude des problèmes de détection de terminaison et de calcul d'énumération.

Nous avons suivi l'approche *correct-par-construction* pour partager la complexité du développement et traiter les spécifications selon différents nivaux d'abstraction. Le degré d'abstraction est réduit, progressivement, par une succession de

raffinements. Nous avons également eu recours à des approches par *composition* et *modularisation* afin de profiter des preuves existantes et fournir des composants (ou modules) réutilisables. Ces types d'approches sont supportés par la méthode formelle Event-B et son assistant de preuve Rodin. La *modularisation* consiste à encapsuler certaines propriétés et les prouver indépendamment puis les incorporer dans diverses spécifications. Un aperçu sur ces approches et méthode à été présenté dans le chapitre 2.

Les chapitres 3, 4 et 5 font l'objet de nos contributions. Dans un premier temps, nous avons étudié le problème de détection de terminaison. Nous avons proposé dans le chapitre 3, un patron formel qui permet de transformer un calcul "avec une détection de terminaison locale" vers un calcul "avec une détection de terminaison globale". La transformation est effectuée en se basant sur une modularisation des preuves de calcul d'un algorithme classique de détection de terminaison.

Dans un deuxième temps, nous avons traité le problème d'énumération qui consiste à attribuer des identifiants uniques aux processus. L'identification des entités dans un système distribué est parfois une condition indispensable pour un fonctionnement correct de divers algorithmes. Dans le chapitre 4, nous avons proposé un schéma de preuve du calcul d'énumération, basé sur une succession de raffinement. Le travail peut sembler quelque part à un cas d'étude. Cependant, il peut être considéré comme un composant de base pour certains algorithmes qui ne fonctionnent que sous l'hypothèse d'une énumération totale des processus. Dans un troisième temps, nous avons étudié le calcul d'état global dans un réseau anonyme. Nous avons exploité les résultats de Chalopin et al. [24] et Andriamiarina et al. [7]. Nous avons montré, dans le chapitre 5, qu'une liaison de composition entre les algorithmes de snapshots, de détection de terminaison et d'énumération pourrait être mise en œuvre dans le cadre d'une approche correct-par-construction, afin de résoudre le calcul d'état global.

Comme perspectives à nos travaux de thèse, nous pensons qu'elles peuvent être résumées en plusieurs idées : la première idée consiste à proposer un plugin in-

tégré à Rodin pour systématiser davantage la transformation d'algorithmes d'une détection locale vers une détection globale de terminaison. Ce plugin se base sur le patron proposé dans le chapitre 3. Il pourra prendre en entrée deux paramètres : un algorithme basé sur le modèle des calculs locaux et un prédicat qui déclare la condition de terminaison locale des processus. Le plugin génèrera, en toute transparence, le même algorithme avec détection de terminaison globale. La deuxième idée consiste à utiliser des outils existants [50] afin de générer du code source à partir des modèles proposés. La troisième idée consiste à étendre le modèle du chapitre 5 afin de détecter différentes propriétés stables du réseau. Cette extension pourra être effectuée par raffinement du dernier niveau. Il suffit d'appliquer une composition avec l'algorithme de détection de terminaison globale (*SSP* [58]) et raffiner ensuite pour tester certaines propriétés. Dans ce cas, l'application de *SSP* permettra aux processus de vérifier la stabilisation des informations échangées. Plus précisément, il faut s'assurer que l'état des messages, qui sont reçus par les processus, sera stable à partir d'une certaine étape de calcul. Toutefois, une telle vérification ne pourra pas être établie en utilisant uniquement la méthode Event-B. Cette dernière se limite à la vérification des propriétés de sûreté. Par conséquent, il serait intéressant de l'utiliser avec une autre méthode, qui permet la vérification des propriétés de vivacité, à savoir TLA+ [44].

En effet, l'utilisation de méthodes variées nous paraît très utiles pour aboutir à de nouveaux résultats de vérifications formelles. Ainsi, la quatrième idée consiste à étudier les limites de certains calculs distribués, par le moyen de preuves d'impossibilités. Ce type de preuves pourra être étudié en se basant sur une logique de preuve du second ordre, à savoir *Coq* [10]. Finalement, la cinquième idée consiste à étudier d'autres problèmes du distribué, à savoir ceux qui sont soulevés suite à des choix probabilistes des processus.

A Annexe

A.1 Collection Des Snapshots Locaux

A.1.1 Niveau 5 : Construction des Vues Locales

La machine $GTD-SNAP_1$ du chapitre 5 est raffinée dans un cinquième niveau par une nouvelle machine $ENUM_0 - SNAP$ pour introduire les premiers étapes de calcul de *Mazurkiewicz*. Dans ce niveau, chaque processus essaie de choisir un numéro entre 1 et la taille du réseau, et procède à la construction de sa vue locale. Cette dernière comportera les numéros de ses voisins ainsi que leurs états. De nouvelles variables sont ainsi introduites :

- $enum$ permet de définir, pour chaque processus, son numéro courant.
  $enum \in P \rightarrow 0\,..\,card(P)$.
- e_enum caractérise l'envoi d'un numéro courant d'un processus.
  $e\_enum \in (C \times \mathbb{N}) \nrightarrow \mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)$. Un élément de e_enum est un quadruplet $p \mapsto q \mapsto i \mapsto (n \mapsto k)$. Il désigne qu'un processus p envoie son numéro courant n au processus q, à l'estampille i, via le port k. Pour caractériser les numéros de ports, nous utilisons la constante LE que nous avons défini dans le chapitre 4, section 4.4.1. Ainsi, j devrait prendre pour valeur $LE(p \mapsto q)$ (Invariant A.1). Il faut s'assurer que chaque numéro, qui est envoyé par un processus p, ne doit pas dépasser $enum(p)$ (Invariant A.2). De plus, nous vérifions que l'estampille d'envoi est toujours inférieur ou égale à l'estampille courant (Invariant A.3).

Invariant A.1. $\forall p, q, i, j, n \cdot p \mapsto q \mapsto i \mapsto (n \mapsto j) \in e\_enum \Rightarrow j = (LE(p \mapsto q))$

Invariant A.2. $\forall p, q, i, j, n \cdot p \mapsto q \mapsto i \mapsto (n \mapsto j) \in e\_enum \Rightarrow n \leq enum(p)$

Invariant A.3. $\forall p, q, i, j, n \cdot p \mapsto q \mapsto i \mapsto (n \mapsto j) \in e\_enum \Rightarrow i \leq o(p)$

- r_enum caractérise la réception d'un rayon de confiance :
  $r\_enum \in (C \times \mathbb{N}) \nrightarrow \mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)$. Un élément de r_enum est un quadruplet $p \mapsto q \mapsto j \mapsto (n \mapsto z)$. Il désigne qu'un processus q reçoit un numéro n d'un processus p, à l'estampille j, via le port z. Chaque numéro reçu doit être déjà envoyé par un processus p (Invariant A.4), et il ne doit pas dépasser le numéro courant de p (Invariant A.5). De plus, l'estampille de réception doit être toujours inférieur ou égale à l'estampille courant (Invariant A.6).

Invariant A.4. $\forall p, q, i1, n \cdot p \mapsto q \in C \wedge p \mapsto q \mapsto i1 \mapsto (n \mapsto LE(q \mapsto p)) \in r\_enum \Rightarrow (\exists i2 \cdot p \mapsto q \mapsto i2 \mapsto (n \mapsto LE(p \mapsto q)) \in e\_enum)$

Invariant A.5. $\forall p, q, i, j, n \cdot p \mapsto q \mapsto i \mapsto (n \mapsto j) \in r\_enum \Rightarrow n \leq enum(p)$

Invariant A.6. $\forall p, q, i, j, n \cdot p \mapsto q \mapsto i \mapsto (n \mapsto j) \in r\_enum \Rightarrow i \leq o(q)$

- Lv constitue la vision locale des processus sur les numéros des voisins :
  $Lv \in P \rightarrow \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1))$. Plus précisément, pour tout processus q, $Lv(q)$ contient les numéros des voisins de q qui sont reçus par ce dernier, couplés avec les numéros des ports correspondants. (Invariant A.7).
- e_Lv caractérise l'envoi d'une vue locale d'un processus :
  $e\_Lv \in (C \times \mathbb{N}) \nrightarrow \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1))$. Un élément de e_Lv est un quadruplet $p \mapsto q \mapsto i \mapsto N$. Il désigne qu'un processus p a envoyé sa vue locale N, au processus q, et à l'estampille i. Nous vérifions par l'Invariant A.10, que la vue locale envoyée par p est plus faible que la vue locale courante. Nous utilisons la même constante *ordre* définie dans le chapitre 4, section 4.4.1.
- r_Lv caractérise la réception d'une vue locale par un processus :
  $r\_Lv \in (C \times \mathbb{N}) \nrightarrow \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1))$. Un élément de r_Lv est un quadruplet $p \mapsto q \mapsto j \mapsto N$. Il désigne qu'un processus q a reçu une vue locale N

d'un processus p, au moment j. Chaque vue locale reçue doit être déjà envoyée (Invariant A.11). L'estampille d'envoi ou de réception d'une vue locale doit être toujours inférieur ou égale à l'estampille courant (Invariant A.12 et Invariant A.13).

Invariant A.7. $\forall p, q, i, n, j.$
$p \mapsto q \in C \wedge p \mapsto q \mapsto i \mapsto (n \mapsto j) \in r\_enum \Rightarrow n \mapsto j \in Lv(q).$

Chaque processus, qui procède à l'envoi de son numéro courant, devrait avoir calculé son snapshot local (Invariant A.8).

Invariant A.8. $\forall p, q, i \cdot p \mapsto q \mapsto i \in dom(e\_enum) \Rightarrow p \in dom(pCoupe) \wedge q \mapsto p \in dom(EtatC).$

Invariant A.9. $\forall x, n, i, j \cdot x \in P \wedge n \mapsto (i \mapsto j) \in Lv(x) \Rightarrow (\exists y \cdot y \mapsto x \in C \wedge enum(y) \geq n \wedge LE(y \mapsto x) = j \mapsto i).$

Invariant A.10. $\forall p, q, i, N \cdot p \mapsto q \mapsto i \mapsto N \in e\_Lv \Rightarrow ordre(N \mapsto Lv(p)) \in \{-1, 0\}.$

Invariant A.11. $\forall c, i, N \cdot c \mapsto i \mapsto N \in r\_Lv \Rightarrow (\exists i2 \cdot c \mapsto i2 \mapsto N \in e\_Lv).$

Invariant A.12. $\forall p, q, i, N \cdot p \mapsto q \mapsto i \mapsto N \in e\_Lv \Rightarrow i \leq o(p).$

Invariant A.13. $\forall p, q, i, N \cdot p \mapsto q \mapsto i \mapsto N \in r\_Lv \Rightarrow i \leq o(q).$

- e_Etat caractérise l'envoi des états des processus et des canaux de communication qui sont enregistrés lors du snapshot :
  $e\_Etat \in (C \times \mathbb{N}) \nrightarrow (EtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1)$. Un élément de e_Etat est de la forme de $(p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e)$. Il désigne qu'un processus p envoi son état ep ainsi que l'état de ses canaux entrants ec, à un processus q, au moment i, via le port e (Invariant A.14, A.15, A.16).
- r_Etat caractérise la réception des états des processus et des canaux de communication qui sont enregistrés lors du snapshot :

$r\_Etat \in (C \times \mathbb{N}) \nrightarrow (EtatP \times \mathbb{P}(M) \times (\mathbb{N}_1 \times \mathbb{N}_1))$. Un élément de r_Etat est de la forme de $(p \mapsto q \mapsto j) \mapsto (ep \mapsto ec \mapsto e)$. Il désigne qu'un processus q a reçu l'état d'un processus p ainsi que l'état des canaux entrants à p. La réception est effectuée au moment j, via le port e (Invariant A.17, A.18, A.19). Ce message devrait être déjà envoyé par le processus p (A.20). L'estampille d'envoi ou de réception des états doit être toujours inférieur ou égale à l'estampille courant (Invariant A.21, A.22).

Invariant A.14. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in e\_Etat$
$\Rightarrow p \in dom(SnapEtatP) \wedge ep = SnapEtatP(p)$.

Invariant A.15. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in e\_Etat$
$\Rightarrow ec = union(\{c \cdot c \in dom(cstate) \wedge prj2(c) = p | cstate(c)\})$.

Invariant A.16. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in e\_Etat$
$\Rightarrow e = LE(p \mapsto q)$.

Invariant A.17. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in r\_Etat$
$\Rightarrow p \in dom(SnapEtatP) \wedge ep = SnapEtatP(p)$.

Invariant A.18. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in r\_Etat$
$\Rightarrow ec = union(\{c \cdot c \in dom(cstate) \wedge prj2(c) = p | cstate(c)\})$.

Invariant A.19. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in r\_Etat$
$\Rightarrow e = LE(q \mapsto p)$.

- $LvEtat$ constitue la vision locale des processus sur les snapshots locaux des voisins : $LvEtat \in P \rightarrow \mathbb{P}((SnapEtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1))$. Pour chaque processus q, $LvEtat(q)$ est construite à partir des informations reçues par q et mémorisées dans r_Etat (Invariant A.23).
- e_LvEtat caractérise l'envoi d'une vue locale sur les états des processus : $e\_LvEtat \in (C \times \mathbb{N}) \nrightarrow \mathbb{P}((SnapEtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1))$.

– r_LvEtat caractérise la réception d'une vue locale sur les états des processus : $r\_LvEtat \in (C \times \mathbb{N}) \nrightarrow \mathbb{P}((SnapEtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1))$.

Invariant A.20. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in r\_Etat$
$\Rightarrow (\exists j, e2 \cdot (p \mapsto q \mapsto j) \mapsto (ep \mapsto ec \mapsto e2) \in e\_Etat)$.

Invariant A.21. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in e\_Etat$
$\Rightarrow i \leq o(p)$.

Invariant A.22. $\forall p, q, i, ep, ec, e \cdot (p \mapsto q \mapsto i) \mapsto (ep \mapsto ec \mapsto e) \in r\_Etat$
$\Rightarrow i \leq o(q)$

Invariant A.23. $\forall q \cdot q \in P \Rightarrow LvEtat(q) = \{p, i, j, L \cdot p \mapsto q \in C \wedge j =$
$LE(p \mapsto q) \wedge p \mapsto q \mapsto i \mapsto (L \mapsto j) \in r\_Etat | L \mapsto j\}$

Un nouvel événement *InitEnum* est introduit pour initialiser les étapes de calculs de *Mazurkiewicz*. Cette initialisation doit être effectuée par un processus qui a détecté la terminaison globale du snapshot.

EVENT $InitEnum$
refines $TraitEtat$
any p
where
$grd1 : Global\_Snapshot(p) = TRUE$
$grd2 : enum(p) = 0$
with $ns : ns = l(p)$
then
$act1 : enum(p) := 1$
$act2 : es(p) := es(p) + 1$
$act3 : h(p) := h(p) \cup \{(es(p) + 1) \mapsto (p \mapsto p)\}$

EVENT $Envoi\_Enum$ **refines** $EnvoiApreSnap'$

any $\cdots$

e, nc

where

$\cdots$

$grd6 : Global\_Snapshot(p) = TRUE$

$grd7 : \forall n, j, w \cdot c \mapsto j \mapsto (n \mapsto w) \in e\_enum \Rightarrow enum(p) > n$

$grd8 : e = LE(c)$

$grd9 : nc = union(\{d \cdot d \in dom(cstate) \wedge prj2(d) = p | cstate(d)\})$

then

$\cdots$

$act7 : e\_enum := e\_enum \cup \{c \mapsto o(p) + 1 \mapsto (enum(p) \mapsto e)\}$

$act8 : e\_Etat := e\_Etat \cup \{c \mapsto o(p) + 1 \mapsto (SnapEtatP(p) \mapsto nc \mapsto e)\}$

EVENT $Reception\_Enum$ **refines** $ReceptionPostSnapMsg'$

any $\cdots$

$e, e1, n, nc, Ne$

where

$\cdots$

$grd6 : c \mapsto i \mapsto (n \mapsto e) \in e\_enum$

$grd7 : \forall w \cdot c \mapsto w \mapsto (n \mapsto e1) \notin r\_enum$

$grd8 : Ne = \{i', n' \cdot \{n', i'\} \subseteq \mathbb{N} \wedge c \mapsto i' \mapsto (n' \mapsto e) \in r\_enum |$
$c \mapsto i' \mapsto (n' \mapsto e)\}$

$grd9 : n \mapsto e \notin Lv(q)$

$Th1 : ordre(Lv(q) \mapsto (Lv(q) \cup \{n \mapsto e\})) = -1$

$grd10 : nc = union(\{d \cdot d \in dom(cstate) \wedge prj2(d) = prj1(c) | cstate(d)\})$

$grd11 : e = LE(c) \wedge e1 = LE(q \mapsto prj1(c))$

then

$\cdots$

$act7 : r\_enum := (r\_enum \setminus Ne) \cup \{c \mapsto o(q) + 1 \mapsto (n \mapsto e1)\}$

$act8 : Lv(q) := Lv(q) \cup \{n \mapsto e1\}$

$act9 : r\_Etat := r\_Etat \cup \{c \mapsto o(q) + 1 \mapsto (SnapEtatP(prj1(c)) \mapsto nc \mapsto e1)\}$

$act10 : LvEtat(q) := LvEtat(q) \cup \{SnapEtatP(prj1(c)) \mapsto nc \mapsto e1\}$

EVENT $Envoi\_Lv$ **refines** $EnvoiApreSnap'$

any $\cdots$

where

$\cdots$

$grd6 : Lv(p) \neq \varnothing$

$grd7 : \forall k \cdot c \mapsto k \mapsto Lv(p) \notin e\_Lv$

$grd8 : \forall k \cdot c \mapsto k \mapsto LvEtat(p) \notin e\_LvEtat$

then

$\cdots$

$act7 : e\_Lv := e\_Lv \cup \{c \mapsto o(p) + 1 \mapsto Lv(p)\}$

$act8 : e\_LvEtat := e\_LvEtat \cup \{c \mapsto o(p) + 1 \mapsto LvEtat(p)\}$

EVENT $Reception\_Lv$ **refines** $ReceptionPostSnapMsg'$

any $\cdots$

$Lv1, nq, np, N, e, Le$

where

$\cdots$

$grd6 : \exists z \cdot c \mapsto z \mapsto Lv(prj1(c)) \in e\_Lv$

$grd7 : Lv1 = \{k, L \cdot k \in \mathbb{N} \land L \in \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)) \land c \mapsto k \mapsto L \in r\_Lv$
$\quad |c \mapsto k \mapsto L\}$

$grd8 : N = \{d, z, x, t \cdot d \in C \wedge prj2(d) = q \wedge d \mapsto z \mapsto (x \mapsto t) \in r\_enum | x\}$

$grd9 : e = LE(c)$

$grd10 : np \in \{z, n \cdot c \mapsto z \mapsto (n \mapsto e) \in r\_enum | n\}$

$grd11 : nq \in 0 \,..\, card(P)$

$grd12 : nq \in \{1 + max(N \cup \{enum(q)\}), enum(q)\}$

$grd13 : ordre(Lv(q) \mapsto Lv(prj1(c))) = -1$

$\quad \Rightarrow nq = 1 + max(N \cup \{enum(q)\})$

$grd14 : ordre(Lv(q) \mapsto Lv(prj1(c))) = 1 \Rightarrow nq = enum(q)$

$grd15 : \exists z \cdot c \mapsto z \mapsto LvEtat(prj1(c)) \in e\_LvEtat$

$grd16 : Le = \{k, L \cdot k \in \mathbb{N} \wedge L \in \mathbb{P}((EtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1))$

$\quad \wedge c \mapsto k \mapsto L \in r\_LvEtat | c \mapsto k \mapsto L\}$

then

$\ldots$

$act7 : r\_Lv := (r\_Lv \setminus Lv1) \cup \{c \mapsto o(q) + 1 \mapsto Lv(prj1(c))\}$

$act8 : enum(q) := nq$

$act9 : r\_LvEtat := (r\_LvEtat \setminus Le) \cup \{c \mapsto o(q) + 1 \mapsto LvEtat(prj1(c))\}$

A.1.2 Niveau 6 :Construction des Mémoires

La machine $ENUM_0 - SNAP$ est raffinée dans un sixième niveau par une nouvelle machine $ENUM_1 - SNAP$. Dans cette machine, chaque processus procède à la construction d'une mémoire. Cette dernière comportera les numéros et les vues locales reçus lors du déroulement de l'algorithme. De nouvelles variables et événements sont introduits afin de permettre aux processus d'échanger leurs mémoires et garder la trace de toute mémoire reçue.

- Mb constitue la mémoire des processus qui contient les numéros des voisins et leurs vue locales sur la numérotation. $Mb \in P \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)))$.
- e_Mb caractérise l'envoie d'une mémoire $Mb(p)$ par un processus p. $e\_Mb \in (C \times \mathbb{N}) \nrightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)))$.
- Mb_Etat constitue la mémoire des processus qui contient les numéros des

voisins accompagnés de leurs vue locales sur le calcul des snapshots locaux. $Mb\_Etat \in P \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}((EtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1)))$

– e_MbEtat caractérise l'envoie d'une mémoire $Mb\_Etat(p)$ par un processus. $e\_MbEtat \in (C \times \mathbb{N}) \rightarrow \mathbb{P}(\mathbb{N}_1 \times \mathbb{P}((EtatP \times \mathbb{P}(M)) \times (\mathbb{N}_1 \times \mathbb{N}_1)))$

Nous vérifions les invariants suivants :

Invariant A.24. $\forall p, q, i, j, n, N \cdot p \mapsto q \in C \wedge p \mapsto q \mapsto i \mapsto (n \mapsto LE(q \mapsto p)) \in r\_enum \wedge c \mapsto j \mapsto N \in r\_Lv \Rightarrow n \mapsto N \in Mb(q)$.

Invariant A.25. $\forall p, n, L \cdot n \mapsto L \in Mb(p) \Rightarrow (\exists y \cdot y \in P \wedge enum(y) \geq n)$.

Invariant A.26. $\forall p, q, i, M0 \cdot p \mapsto q \mapsto i \mapsto M0 \in e\_Mb \Rightarrow i \leq o(p)$.

Certains événements sont raffinés et d'autres sont introduits comme suit : l'événement $Reception\_Lv$ est raffiné par $Reception\_Lv'$ pour construire, pour chaque processus q, les mémoires $Mb(q)$ et $Mb\_Etat(q)$. Ces mémoires sont définies, respectivement, à partir des vues locales sur la numérotation et sur les états enregistrés lors du snapshot. L'événement $Envoi\_Mb$ est observé quand un processus p envoie ses mémoires $Mb(p)$ et $Mb\_Etat(p)$, à un processus voisin.

EVENT $Reception\_Lv'$
refines $Reception\_Lv$
...
then
...
$\oplus act10 : Mb(q) := Mb(q) \cup \{np \mapsto Lv(prj1(c))\}$
$\oplus act11 : Mb\_Etat(q) := Mb\_Etat(q) \cup$
$\{np \mapsto LvEtat(prj1(c))\}$

EVENT $Envoi\_Mb$

refines

$EnvoiApreSnap'$

$\cdots$

where

$\oplus grd6 : Mb(p) \neq \varnothing$

$\oplus grd7 : \forall j \cdot c \mapsto j \mapsto Mb(p) \notin e\_Mb$

$\oplus grd8 : \forall j \cdot c \mapsto j \mapsto Mb\_Etat(p) \notin e\_Mb\_Etat$

then

$\cdots$

$\oplus act7 : e\_Mb := e\_Mb \cup \{c \mapsto o(p) + 1 \mapsto Mb(p)\}$

$\oplus act8 : e\_MbEtat := e\_MbEtat \cup \{c \mapsto o(p) + 1 \mapsto Mb\_Etat(p)\}$

EVENT $Reception\_Mb$ **refines** $Reception\_Lv$

any $\cdots$

L

where

$\cdots$

$\oplus grd17 : \exists z \cdot c \mapsto z \mapsto Mb(prj1(c)) \in e\_Mb$

$\oplus grd18 : Mb(prj1(c)) \nsubseteq Mb(q)$

$\oplus grd19 : L \in \mathbb{P}(\mathbb{N}_1 \times (\mathbb{N}_1 \times \mathbb{N}_1)) \wedge L \neq Lv(prj2(c))$

$\oplus grd20 : nq \in \{1 + max(dom(Mb(q)) \cup dom(Mb(prj1(c)))), enum(q)\}$

$\oplus grd21 : enum(q) \mapsto L \in Mb(q) \wedge ordre(Lv(q) \mapsto L) = -1$

$\Rightarrow nq = 1 + max(dom(Mb(q)) \cup dom(Mb(prj1(c))) \cup \{0\})$

$\oplus grd22 : (enum(q) \mapsto L \in Mb(q) \wedge ordre(Lv(q) \mapsto L) = 1)$

$\vee (enum(q) \mapsto L \notin Mb(q)) \Rightarrow nq = enum(q)$

then $\cdots$

$\oplus act10 : Mb(q) := Mb(q) \cup Mb(prj1(c))$

$\oplus act11 : Mb\_Etat(q) := Mb\_Etat(q) \cup Mb\_Etat(prj1(c))$

Bibliographie

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[3] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin : an open toolset for modelling and reasoning in event-b. *STTT*, 12(6) :447–466, 2010.

[4] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In *ZB 2003 : Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, pages 457–476, 2003.

[5] Karine Altisen and Pierre Corbineau. Composition certifiée dŠalgorithmes autostabilisants silencieux. In *19 èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel 2017), May 2017, Quiberon, France.*, 2017.

[6] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Integrating proved state-based models for constructing correct distributed algorithms. In *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, pages 268–284, 2013.

[7] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Revisiting snapshot algorithms by refinement-based techniques. *Comput. Sci. Inf. Syst.*, 11(1) :251–270, 2014.

[8] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*, pages 82–93, 1980.

[9] Michel Bauderon, Stefan Gruner, Yves Métivier, Mohamed Mosbah, and Afif Sellami. Visualization of distributed algorithms based on graph relabelling systems. *Electr. Notes Theor. Comput. Sci.*, 50(3) :227–237, 2001.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[11] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 145–164, 2010.

[12] Lélia Blin and Franck Butelle. The first approximated distributed algorithm for the minimum degree spanning tree problem on general graphs. *Int. J. Found. Comput. Sci.*, 15(3) :507–516, 2004.

[13] Maha Boussabbeh, Mohamed Tounsi, Ahmed Hadj Kacem, and Mohamed Mosbah. Enhancing proofs of local computations through formal event-b modularization. In *2014 IEEE 23rd International WETICE Conference, WETICE 2014, Parma, Italy, 23-25 June, 2014*, pages 50–55, 2014.

[14] Maha Boussabbeh, Mohamed Tounsi, Ahmed Hadj Kacem, and Mohamed Mosbah. Towards a general framework for ensuring and reusing proofs of termination detection in distributed computing. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 504–511, 2016.

[15] Maha Boussabbeh, Mohamed Tounsi, Ahmed Hadj Kacem, and Mohamed Mosbah. Formal development of distributed enumeration algorithms by refinement-based techniques. In *SCSS 2017, The 8th International Symposium on Symbolic Computation in Software Science 2017, April 6-9, 2017, Gammarth, Tunisia*, pages 96–106, 2017.

[16] Maha Boussabbeh, Mohamed Tounsi, Mohamed Mosbah, and Ahmed Hadj Kacem. Formal proofs of termination detection for local computations by refinement-based compositions. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, pages 198–212, 2016.

[17] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. The unified policy framework (UPF). *Archive of Formal Proofs*, 2014, 2014.

[18] Dominique Cansell and Dominique Méry. Tutorial on the event-based b method : Concepts and case studies. In *Logics of Formal Software Specification Languages - LFSL2004*. none, 2004.

[19] Pierre Castéran and Vincent Filou. Tasks, types and tactics for local computation systems. *Stud. Inform. Univ.*, 9(1) :39–86, 2011.

[20] Jérémie Chalopin, Emmanuel Godard, and Yves Métivier. Local terminations and distributed computability in anonymous networks. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 47–62, 2008.

[21] Jérémie Chalopin, Emmanuel Godard, Yves Métivier, and Gerard Tel. About the termination detection in the asynchronous message passing model. In *SOFSEM 2007 : Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings*, pages 200–211, 2007.

[22] Jérémie Chalopin and Yves Métivier. An efficient message passing election algorithm based on mazurkiewicz's algorithm. *Fundam. Inform.*, 80(1-3) :221–246, 2007.

[23] Jérémie Chalopin and Yves Métivier. On the power of synchronization between two adjacent processes. *Distributed Computing*, 23(3) :177–196, 2010.

[24] Jérémie Chalopin, Yves Métivier, and Thomas Morsellino. On snapshots and stable properties detection in anonymous fully distributed systems (extended abstract). In *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2, 2012, Revised Selected Papers*, pages 207–218, 2012.

[25] K. Mani Chandy and Leslie Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[26] Rance Cleaveland and Steve Sims. The NCSU concurrency workbench. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 394–397, 1996.

[27] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Inf. Process. Lett.*, 11(1) :1–4, 1980.

[28] Jean-François Dufourd. Discrete jordan curve theorem : A proof formalized in coq with hypermaps. In *STACS 2008, 25th Annual Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, February 21-23, 2008, Proceedings*, pages 253–264, 2008.

[29] ClearSy System Engineering. Atelier b, manuel utilisateur. Technical report, 2001.

[30] Vincent Filou. *Une étude formelle de la théorie des calculs locaux á l'aide de l'assistant de preuve Coq.* PhD thesis, Ecole Doctorale de mathématiques et d'informatique -Université de BORDEAUX, 2012.

[31] Vincent Filou, Mohamed Mosbah, and Mohamed Tounsi. Towards proved distributed algorithms through refinement, composition and local computations. In *2013 Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, June 17-20, 2013*, pages 353–358, 2013.

[32] Allyx Fontaine and Akka Zemmari. Certified impossibility results and analyses in coq of some randomised distributed algorithms. In *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, pages 69–81, 2016.

[33] Thomas Geffroy. Certifications de transformations d'algorithmes distribués : l'exemple de la transformation ssp, 2014.

[34] Emmanuel Godard and Yves Métivier. A characterization of families of graphs in which election is possible. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, pages 159–172, 2002.

[35] Emmanuel Godard, Yves Métivier, Mohamed Mosbah, and Afif Sellami. Termination detection of distributed algorithms by graph relabelling systems. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, pages 106–119, 2002.

[36] Emmanuel Godard, Yves Métivier, and Gerard Tel. Termination detection of local computations. *CoRR*, abs/1001.2785, 2010.

[37] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Supporting reuse in event B development : Modularisation approach. In *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 174–188, 2010.

[38] Tobias Isenberg, Dominik Steenken, and Heike Wehrheim. Bounded model checking of graph transformation systems via SMT solving. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, pages 178–192, 2013.

[39] Savas Konur, Clare Dixon, and Michael Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2) :199–213, 2012.

[40] Ajay D. Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4) :224, 1995.

[41] Gary T. Leavens, Jean-Raymond Abrial, Don S. Batory, Michael J. Butler, Alessandro Coglio, Kathi Fisler, Eric C. R. Hehner, Cliff B. Jones, Dale Miller, Simon L. Peyton Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, pages 221–236, 2006.

[42] Dominique Lecomte, D. Thierry, and Dominique Méry. Patrons de conception prouvés. *Génie Logiciel - Magazine de língéniérie du logiciel et des systèmes*, pages 14–18, 2007.

[43] Lamport Leslie. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 1994.

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

[45] Lamport Leslie. The pluscal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing.* Springer-Verlag, 2009.

[46] Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova, and Elena Troubitsyna. Patterns for representing FMEA in formal specification of control systems. In *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, pages 146–151, 2011.

[47] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151, 1987.

[48] Antoni W. Mazurkiewicz. Distributed enumeration. *Inf. Process. Lett.*, 61(5) :233–239, 1997.

[49] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Ransomware steals your phone. formal methods rescue it. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 212–221, 2016.

[50] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011*, pages 179–188, 2011.

[51] Yves Métivier, Nasser Saheb, and Akka Zemmari. Randomized local elections. *Inf. Process. Lett.*, 82(6) :313–320, 2002.

[52] Yves Métivier and Éric Sopena. Graph relabelling systems : A general overview. *Computers and Artificial Intelligence*, 16(2), 1997.

[53] Jayadev Misra. Detecting termination of distributed computations using markers. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 290–294, 1983.

[54] Mohamed Mosbah and Rodrigue Ossamy. A programming language for local computations in graphs : Computational completeness. In *5th Mexican International Conference on Computer Science (ENC 2004), 20-24 September 2004, Colima, Mexico*, pages 12–19, 2004.

[55] V. Natarajan and Gerard J. Holzmann. Outline for an operational semantics of promela. In *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*, pages 133–152, 1996.

[56] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[57] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS : A prototype verification system. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 748–752, 1992.

[58] Boleslaw K. Szymanski, Yuan Shi, and Noah S. Prywes. Terminating iterative solution of simultaneous equations in distributed message passing systems. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*, pages 287–292, 1985.

[59] Gérard Tel. *Introduction of Distributed Algorithms*. Cambridge University Press.

[60] Mohamed TOUNSI. *Preuves d'algorithmes distribués par raffinement*. PhD thesis, Ecole Doctorale de mathématiques et d'informatique Université de BORDEAUX, 2012.

[61] Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. Proving distributed algorithms by combining refinement and local computations. *ECEASST*, 35, 2010.

[62] Laurent Voisin and Jean-Raymond Abrial. The rodin platform has turned ten. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 1–8, 2014.

[63] Roel Wieringa and Wiebren de Jonge. Object identifiers, keys, and surrogates : Object identifiers revisited. *TAPOS*, 1(2) :101–114, 1995.

[64] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks : Part i-characterizing the solvable cases. *IEEE Trans. Parallel Distrib. Syst.*, 7(1) :69–89, 1996.

[65] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla$^+$ specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 54–66, 1999.