UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Hugues EVRARD

Thèse dirigée par **Frédéric LANG et Gwen SALAÜN**

préparée au sein de **l'équipe CONVECS – INRIA Grenoble Rhône-Alpes et Laboratoire d'Informatique de Grenoble (LIG)**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)**

Génération automatique d'implémentation distribuée à partir de modèles formels de processus concurrents asynchrones

Thèse soutenue publiquement le **10 Juillet 2015**,
devant le jury composé de :

Stephan MERZ
Directeur de Recherche, Rapporteur
Uwe NESTMANN
Professeur, Rapporteur
Guy LEDUC
Professeur, Examinateur
Yves DENNEULIN
Professeur, Président
Frédéric LANG
Chargé de recherche, Directeur de thèse
Gwen SALAÜN
Maître de conférence, Directeur de thèse

Remerciements

Je souhaite tout d'abord remercier profondément les membres du jury, à commencer par Stephan Merz et Uwe Nestmann dont les rapports fournis et précis m'ont permis de mettre en perspective les travaux réalisés et d'étoffer la version finale de ce mémoire. J'ai été honoré par la présence de Guy Leduc et Yves Denneulin, qui a assuré la présidence du jury, et je les remercie sincèrement pour leurs questions pertinentes et leurs avis éclairés sur de nombreuses facettes du sujet.

Cette thèse n'aurait pas vu le jour sans l'encadrement attentionné de Frédéric Lang, que je remercie pour avoir su me laisser explorer et mûrir de multiples idées tout en offrant un point d'appui solide à travers un dialogue toujours constructif. Grâce à ses innombrables relectures et corrections, Frédéric m'a patiemment appris à aiguiser mon écriture, ce qui est peut-être le plus grand bénéfice que je tire du doctorat. Je remercie chaleureusement Gwen Salaün pour son ouverture d'esprit et ses précieux conseils, ainsi que pour son engagement et son soutien qui ont été sans faille sur les plans scientifiques, administratifs et humains.

J'ai eu la chance de travailler au sein de l'environnement rigoureux et stimulant de l'équipe CONVECS. Je souhaite exprimer ma gratitude envers Wendelin Serwe pour sa disponibilité, son discernement et son efficacité face aux bogues mêmes les plus coriaces. Je serai toujours redevable à Hubert Garavel pour son écoute et ses suggestions lucides et perspicaces, par lesquelles il m'a permis de bénéficier de sa grande expérience, tant théorique que pratique, et qui couvre bien plus que l'informatique. Je suis très reconnaissant envers Radu Mateescu pour sa gentillesse, son expertise et sa souplesse, et je tiens à saluer le zèle avec lequel il orchestre le travail de l'équipe en toutes circonstances.

Mon séjour dans l'équipe a été aussi égayé par d'agréables échanges avec Matthias Güdemann, Abderahman Kriouile, Alexandre Dumont, Lina Ye, Jingyan Jourdan-Lu, Éric Léo, Rim Abid, Fatma Jebali, Remy Delanaux et Lakhdar Akroun ; et je salue encore plus particulièrement mes co-bureau qui ont parfois dû endurer mes états d'âme : merci à Kaoutar Hafdi, Raquel Oliveira, Imad-Seddick Arrada, José Ignacio Requeno et Zhen Zhang. Je remercie Helen Pouchot et Myriam Etienne pour m'avoir offert une interface conciliante avec la machinerie administrative. Je remercie toute l'équipe de documentation du centre, en particulier Audrey Genoud, pour leur professionnalisme, et je regrette la déportation de la bibliothèque qui a été pour moi un havre d'inspiration essentiel.

À travers le monitorat à l'Ensimag, j'ai eu l'opportunité de côtoyer Yves Denneulin, Roland Groz, Noël de Palma, Vivien Quéma, Franck Rousseau, Grégory Mounié, Frédéric Wagner et François Broquedis, que je remercie pour leur confiance.

Par périodes de déboires comme par temps de succès, j'ai pu compter sur des amis fidèles et sensibles, envers lesquels j'exprime toute ma gratitude. J'ai eu la chance de grandir dans une famille qui m'a toujours incité à me donner les moyens de faire ce que je souhaitais, et je serais ravi que cette thèse leur rende hommage. Enfin, Chloé, pour ta sincerité fortifiante, ton équilibre apaisant, et pour tout le reste, je te remercie du fond du cœur.

Ces travaux ont été financés en partie par le Fond national pour la Société Numérique (FSN), le pôle Minalogic, Systematics et SCS dans le cadre du projet OpenCloudware.

Certaines expériences ont été effectuées sur la plateforme Grid5000, soutenue par un groupe d'intérêt scientifique regroupant Inria, le CNRS, RENATER et autres universités et organisations (voir `https://www.grid5000.fr`).

Abstract

LNT is a recent formal specification language, based on process algebras, where several concurrent asynchronous processes can interact by multiway rendezvous (i.e., involving two or more processes), with data exchange. The CADP (*Construction and Analysis of Distributed Processes*) toolbox offers several techniques related to state space exploration, like model checking, to formally verify an LNT specification. This thesis introduces a distributed implementation generation method, starting from an LNT formal model of a parallel composition of processes. Taking advantage of CADP, we developed the new DLC (*Distributed LNT Compiler*) tool, which is able to generate, from an LNT specification, a distributed implementation in C that can be deployed on several distinct machines linked by a network.

The generated implementations require a synchronization protocol to handle multiway rendezvous with data exchange between distant processes. We set up a verification method for this kind of protocol, which, using LNT and CADP, can check that the protocol leads to valid interactions with respect to a given specification, without introducing livelocks or deadlocks. This method allowed us to identify possible deadlocks in a protocol from the literature, which we incrementally corrected, extended and improved, relying on our method to formally verify each iteration. We also designed a mechanism that enables the final user, by embedding user-defined C procedures into the implementation, to set up interactions between the generated implementation and other systems in the environment. Finally, we measured the performances of implementations generated by DLC on several examples, including a complete case-study on the new consensus algorithm Raft.

Résumé

LNT est un langage formel de spécification récent, basé sur les algèbres de processus, où plusieurs processus concurrents et asynchrones peuvent interagir par rendez-vous multiple, c'est-à-dire à deux ou plus, avec échange de données. La boite à outils CADP (*Construction and Analysis of Distributed Processes*) offre plusieurs techniques relatives à l'exploration d'espace d'états, comme le *model checking*, pour vérifier formellement une spécification LNT. Cette thèse présente une méthode de génération d'implémentation distribuée à partir d'un modèle formel LNT décrivant une composition parallèle de processus. En s'appuyant sur CADP, nous avons mis au point le nouvel outil DLC (*Distributed LNT Compiler*), capable de générer, à partir d'une spécification LNT, une implémentation distribuée en C qui peut ensuite être déployée sur plusieurs machines distinctes reliées par un réseau.

Les implémentations générées nécessitent un protocole de synchronisation pour assurer les rendez-vous multiples avec échange de données entre processus distants. Nous avons mis au point une méthode de vérification de ce type de protocole qui, en utilisant LNT et CADP, permet de vérifier que le protocole réalise, sans introduire ni boucle infinie ni interblocage, des rendez-vous cohérents par rapport à une spécification donnée. Cette méthode nous a permis d'identifier de possibles interblocages dans un protocole de la littérature, que nous avons ensuite corrigé, étendu et amélioré de manière incrémentale, en mettant notre méthode à profit pour vérifier formellement chaque itération. Nous avons aussi développé un mécanisme qui permet, en embarquant au sein d'une implémentation des procédures C librement définies par l'utilisateur, de mettre en place des interactions entre une implémentation générée et d'autres systèmes de son environnement. Enfin, nous avons mesuré les performances des implémentations générées par DLC sur plusieurs exemples, dont un cas d'étude complet sur le nouvel algorithme de consensus Raft.

Table des matières

Introduction **1**

1 Contexte et travaux existants **5**
1.1 Programmation concurrente 5
1.2 Méthodes formelles et génération d'exécutable 8
1.2.1 *Model checking* pour les langages d'implémentation 9
1.2.2 Génération de code pour les langages de spécification 10
1.3 Spécification et vérification formelles avec CADP 12
1.3.1 Introduction au langage LNT 12
1.3.2 Composition parallèle et vecteurs de synchronisation 22
1.3.3 Aperçu des outils CADP utilisés 23
1.3.4 Relations d'équivalences 25
1.4 Protocoles de synchronisation pour le rendez-vous multiple 29
1.4.1 Le rendez-vous multiple entre processus non déterministes 29
1.4.2 Acquisition de ressources partagées sans interblocage 31
1.4.3 Protocoles distribués pour le rendez-vous multiple 31

2 Protocole de synchronisation **35**
2.1 Version de Parrow et Sjödin 35
2.2 Rendez-vous en avance de phase 41
2.3 Généralisation du protocole 43
2.3.1 Simplifications 43
2.3.2 Extension aux communications asynchrones 44
2.3.3 Portes avec plusieurs vecteurs de synchronisation 47
2.3.4 Gestion des actions internes 49
2.3.5 Gestion des offres 50
2.3.6 Confirmation de réussite par la porte 52
2.4 Optimisation : auto-verrouillage des tâches 53
2.5 Cohabitation de l'auto-verrouillage et des rendez-vous en avance de phase . 55
2.5.1 Demande de verrou à une tâche auto-verrouillée 56
2.5.2 Purge de messages d'auto-verrouillage en transit 58
2.5.3 Purge de messages de résultats en transit 60

2.6 Coût du protocole 65
2.7 Spécification formelle du protocole 67
2.7.1 Types de données et canaux de communications 68
2.7.2 Spécification du comportement d'une porte 69
2.7.3 Spécification du comportement d'un manager 75

3 Interaction avec l'environnement 83
3.1 Les fonctions crochets 83
3.1.1 Effets de bord dans un programme LNT 84
3.1.2 Les trois types de fonctions crochets 85
3.1.3 Échange de données 87
3.1.4 Accès aux autres informations de la négociation 89
3.2 Exemples d'utilisation des fonctions crochets 89
3.2.1 Réalisation d'un effet de bord dans l'environnement 91
3.2.2 Contrôle de la réalisation d'une action 91
3.2.3 Envoi de données vers l'environnement 92
3.2.4 Réception de données depuis l'environnement 93
3.2.5 Autoriser une action après un délai 94

4 Génération automatique d'implémentation distribuée 97
4.1 La bibliothèque CÆSAR_NETWORK 98
4.2 Implémentation d'une tâche 100
4.2.1 Le compilateur EXEC/CÆSAR 100
4.2.2 Interface du programme généré par EXEC/CÆSAR 100
4.2.3 Branchement avec la partie manager du protocole 101
4.3 Implémentation du protocole 102
4.3.1 Choix non déterministe sur la réception de message 103
4.3.2 Compatibilité et fusion des offres 104
4.3.3 Action interne et option de progrès maximal 105
4.4 La bibliothèque DLC_SPECIFICATION 106
4.5 Limitations actuelles de DLC 106

5 Vérification formelle de protocole de synchronisation 109
5.1 Génération de modèle d'une implémentation 110
5.1.1 Modèle d'une tâche associée à un manager 110
5.1.2 Modèle des vecteurs de synchronisation 111
5.1.3 Modèle des communications asynchrones 112
5.1.4 Modèle général de l'implémentation 113
5.2 Vérifications conduites sur le modèle d'implémentation 114
5.2.1 Espace d'états du modèle de l'implémentation 115
5.2.2 Équivalence entre la spécification et son implémentation 115
5.2.3 Absence de livelock 117
5.2.4 Absence de deadlock 118

5.3 Application de la méthode de vérification 120
5.3.1 Base de tests 120
5.3.2 Analyse des résultats pour le protocole de DLC 122
5.3.3 Possibles interblocages pour deux protocoles existants 127

6 Performances des implémentations générées 129
6.1 Barrière de synchronisation distribuée 129
6.2 Le dîner des philosophes 131
6.2.1 Ressources partagées et rendez-vous multiple 131
6.2.2 Mesures de performances 133
6.3 Cas d'étude : l'algorithme de consensus Raft 134
6.3.1 Structure de l'algorithme Raft 135
6.3.2 Implémentation de Raft en LNT 138
6.3.3 Performances de l'implémentation générée 143
6.3.4 Consensus et rendez-vous distribué 145

Conclusion 149
Bilan 149
Perspectives 151

A Vérification de protocole de synchronisation 155
A.1 Spécification LNT du protocole utilisé dans DLC 155
A.1.1 Types de données 155
A.1.2 Fonctions 157
A.1.3 Canaux de communications (channels) 161
A.1.4 Spécification des processus 161
A.2 Script de vérification SVL 169

B Spécification de l'algorithme de consensus Raft 173

Introduction

Les méthodes formelles offrent des techniques pour s'assurer du bon comportement d'un système informatique. Elles s'appliquent sur des descriptions précises et non ambiguës de systèmes. En particulier, les algèbres de processus telles que CCS [Mil80], CSP [Hoa85] ou LOTOS [ISO89] sont des formalismes conçus spécifiquement pour l'étude des systèmes concurrents asynchrones, c'est-à-dire des systèmes contenant plusieurs processus qui s'exécutent de manière indépendante, sans partager d'horloge globale, et qui interagissent entre eux. Les algèbres de processus sont caractérisées notamment par leurs opérateurs de *composition parallèle*, qui définissent quelles interactions sont autorisées entre les processus, ainsi que par des processus *non déterministes*, qui peuvent être prêts sur plusieurs interactions au même moment bien qu'ils ne réalisent effectivement qu'une seule de ces interactions. De plus, la sémantique formelle des algèbres de processus offre un support à l'élaboration de techniques de vérification formelle. Les algèbres de processus fournissent ainsi une base rigoureuse pour spécifier un système concurrent asynchrone, puis pour vérifier les propriétés de cette spécification.

Les systèmes distribués, qui sont une forme de systèmes concurrents, sont réputés complexes à concevoir. Les méthodes formelles peuvent aider à détecter des bogues tôt dans le cycle de développement de ces systèmes. Toutefois, l'utilisation de méthodes formelles dans le cadre du développement d'un système distribué consiste encore souvent à vérifier la spécification du système, puis à traduire à la main cette spécification dans un langage d'implémentation. Si la phase de vérification formelle permet de s'assurer de la correction d'un système, la phase d'implémentation manuelle est propice à l'apparition d'écarts de sémantique entre la spécification de départ et l'implémentation finale. La génération automatique d'implémentation distribuée à partir d'une spécification formelle permet de réduire le risque d'apparition d'écarts de sémantique, et de rendre le développement plus fluide.

Nous nous focalisons, dans nos travaux, sur le langage formel LNT [CCG+14], basé sur les algèbres de processus. LNT offre une syntaxe proche des langages algorithmiques classiques, couplée à une sémantique formellement définie sous forme de LTS (*Labelled Transition System*, système de transitions étiquetées). L'opérateur de composition parallèle de LNT permet de définir des interactions par *rendez-vous multiple*, avec échange de données, entre plusieurs (2 ou plus) processus non déterministes. La boîte à outils CADP (*Construction*

and Analysis of Distributed Processes) [GLMS13] offre plusieurs techniques relatives à l'exploration d'espace d'états pour vérifier formellement une spécification LNT. Cependant, CADP ne contient pas d'outil permettant de générer une implémentation distribuée.

Dans cette thèse, nous proposons une technique de compilation permettant d'obtenir une implémentation distribuée, c'est-à-dire formée de plusieurs programmes pouvant être déployés sur des machines distinctes, à partir d'une spécification LNT. Nous avons identifié quatre difficultés majeures dans ce type de compilation :

1. **Un protocole de synchronisation général.** Le rendez-vous multiple est une interaction riche et complexe, qui n'est pas disponible dans les langages d'implémentation. La primitive d'interaction communément accessible entre programmes distants dans une implémentation distribuée est l'envoi de messages. Il est donc nécessaire de mettre en place un protocole de synchronisation, qui repose sur l'envoi de messages, pour implémenter les interactions par rendez-vous définies par la spécification LNT.

   Le protocole ne se résume pas à une barrière de synchronisation pour chaque rendez-vous, car il faut parfois choisir entre des rendez-vous mutuellement exclusifs. En effet, la nature non déterministe des processus peut amener un groupe de processus à pouvoir réaliser plusieurs rendez-vous au même moment, mais chaque rendez-vous doit obligatoirement concerner tout le groupe. Ces rendez-vous sont alors en *conflit*, et seul l'un d'entre eux doit être réalisé. Le protocole doit ainsi assurer la synchronisation des processus tout en garantissant l'exclusion mutuelle des rendez-vous en conflit. En outre, le protocole doit aussi prendre en charge les échanges de données entre processus lors d'un rendez-vous.
2. **La vérification formelle du protocole de synchronisation.** Le protocole de synchronisation doit garantir que la sémantique de la spécification de départ est respectée. Il doit permettre de réaliser tout rendez-vous possible au regard la spécification, mais il ne doit pas pouvoir mener à des rendez-vous invalides. Ce protocole est la clé de voûte d'une implémentation distribuée, il est donc souhaitable d'utiliser des méthodes formelles pour s'assurer de sa correction.
3. **L'interaction de l'implémentation générée avec son environnement.** L'implémentation produite doit être en mesure d'interagir avec son environnement, bien que cet environnement soir généralement abstrait dans la spécification formelle. Considérons par exemple un système dont un processus consulte une base de données externe. Cette base de données peut être abstraite par des opérations de lecture et d'écriture au niveau de la spécification formelle. Le programme généré devra néanmoins véritablement interagir avec une base de données réelle, qui existe indépendamment du système généré. La technique de compilation doit ainsi offrir un moyen de définir des interactions entre l'implémentation et son environnement d'exécution.
4. **Des performances non rédhibitoires.** L'implémentation produite doit atteindre des performances acceptables pour pouvoir être utilisée en pratique, au moins comme

prototype. Sans obligatoirement atteindre les performances d'une implémentation optimisée à la main, le code généré doit être suffisamment rapide pour pouvoir être placé dans un environnement et réagir avec d'autres systèmes. Dans une implémentation distribuée, les performances ne dépendent pas seulement de la rapidité d'exécution de chaque processus, mais aussi de l'efficacité du protocole employé pour assurer les interactions entre processus.

Nous avons développé un nouveau compilateur, nommé DLC (*Distributed LNT Compiler*), qui prend en entrée une composition parallèle de processus LNT et qui produit en sortie une implémentation distribuée en langage C [KR78]. DLC s'appuie sur certains outils existants de CADP pour traiter une spécification LNT, et apporte en plus toute la mécanique nécessaire pour obtenir automatiquement une implémentation distribuée. La méthode de compilation mise en œuvre dans DLC répond aux quatre principales difficultés identifiées plus haut :

1. **Protocole.** Chaque processus de la spécification de départ est transformé en un programme indépendant, et les interactions par rendez-vous sont assurées par un protocole de synchronisation entre ces programmes. En nous basant sur des travaux existants, nous avons développé un protocole de synchronisation qui implémente le rendez-vous multiple en minimisant le nombre de messages à échanger, et qui prend en compte les échanges de données entre processus.
2. **Vérification.** Nous avons élaboré une méthode de vérification formelle pour le protocole de synchronisation. Cette méthode utilise les outils de vérification énumérative (comme le *model checking*) de CADP pour valider le bon comportement d'un protocole qui implémente le rendez-vous multiple. Notre méthode nous a menés à la découverte de possibles interblocages dans un protocole proposé par Parrow et Sjödin [PS96]. De plus, cette méthode nous a permis, lors de la mise au point du protocole utilisé par DLC, de rapidement diagnostiquer – à plusieurs reprises – des problèmes parfois subtils.
3. **Interaction.** Nous avons mis en place un mécanisme de *fonctions crochets* qui permettent une interaction entre l'implémentation générée et son environnement. Les fonctions crochets sont des procédures définies en C par l'utilisateur ; elles sont ensuite embarquées directement au sein de l'implémentation produite par DLC. Ces fonctions crochets sont appelées à des moments précis lors de l'exécution, et elles permettent d'échanger des informations avec d'autres systèmes de l'environnement ou de contrôler l'exécution de l'implémentation générée.
4. **Performances.** Les implémentations produites atteignent des performances que nous considérons suffisantes pour pouvoir qualifier ces implémentations de prototypes. Nous avons testé le compilateur DLC sur un exemple concret et non trivial : le nouvel algorithme de consensus Raft [OO14]. Cette étude de cas nous permet notamment d'illustrer l'emploi des fonctions crochets sur un exemple réel, et de mesurer les performances atteintes par l'implémentation générée. Les optimisations

du protocole et le fait de générer directement du code C permettent aux implémentations de réaliser jusqu'à plusieurs milliers de rendez-vous à la seconde.

Contributions

La contribution principale de cette thèse est le développement du compilateur DLC, qui permet de générer automatiquement une implémentation distribuée à partir d'une spécification formelle LNT. Plus précisément, les contributions sont les suivantes :

- Le développement d'un protocole de synchronisation qui implémente le rendez-vous multiple, avec échange de données, entre processus non déterministes.
- La définition d'un mécanisme de fonctions crochets permettant des interactions entre l'implémentation générée et son environnement.
- L'automatisation complète, à travers le compilateur DLC, de la génération d'implémentation distribuée.
- La mise au point d'une méthode de vérification formelle de protocole de synchronisation, employée pour vérifier le protocole utilisé par DLC.
- La découverte, grâce à notre méthode de vérification, de possibles interblocages dans un protocole de synchronisation existant.
- La présentation de l'utilisation de DLC sur un cas d'étude concret de service répliqué par l'algorithme de consensus Raft.

Une partie de ces travaux a donné lieu à deux publications dans des conférences avec comité de lecture international. La première [EL13] présente la méthode de vérification formelle de protocole de synchronisation, l'application de cette méthode sur trois protocoles de la littérature, et la découverte d'interblocages dans l'un de ces protocoles. La deuxième [EL15] décrit la structure générale de DLC, et couvre succinctement l'application de DLC sur l'algorithme de consensus Raft. Nous sommes invités à étendre cette deuxième publication en un article du *Journal of Logical and Algebraic Methods in Programming* ; cette extension est en cours de relecture à l'heure de la rédaction de ce mémoire.

Ce mémoire de thèse est structuré de la manière suivante. Le chapitre 1 introduit le contexte et les travaux existants. Le chapitre 2 présente le protocole de synchronisation que nous avons développé. Le chapitre 3 couvre le mécanisme d'interaction par fonctions crochets. Le chapitre 4 décrit comment une implémentation distribuée est générée automatiquement. Le chapitre 5 détaille la méthode de vérification formelle de protocole de synchronisation. Enfin, le chapitre 6 expose trois expériences conduites pour mesurer les performances des implémentations générées par DLC, dont le cas d'étude sur l'algorithme de consensus Raft.

Chapitre 1

Contexte et travaux existants

Cette thèse s'inscrit dans le développement de techniques pour la conception rigoureuse de systèmes concurrents. La programmation concurrente est brièvement introduite, avant d'aborder les travaux existants sur la vérification formelle et la génération d'exécutables pour les systèmes concurrents. On présente ensuite le langage LNT et les outils de CADP utilisés dans le cadre de la thèse. En fin de chapitre, plusieurs protocoles proposés pour implémenter le rendez-vous multiple dans un contexte distribué sont passés en revue.

1.1 Programmation concurrente

La programmation concurrente traite de la commande de systèmes informatiques constitués de plusieurs processus d'exécution. Elle permet d'améliorer les performances ou la sûreté d'un système en répartissant le travail sur plusieurs processus. Les processus peuvent soit s'exécuter tour à tour sur un même processeur, comme c'est le cas des processus ordonnancés par un système d'exploitation préempteur, soit s'exécuter sur des processeurs distincts au sein d'une seule machine multiprocesseur, soit encore s'exécuter sur des processeurs de machines distantes reliées par un réseau. On appelle système *concurrent* un système qui possède plusieurs processus d'exécution.

Il est parfois sous entendu que le mot "concurrence" implique que les processus s'exécutent sur une même machine et ont accès à une mémoire partagée. Nous ne limitons pas la concurrence à ces caractéristiques. De plus, il ne faut pas confondre la notion de concurrence avec celle de *parallélisme*, qui indique que plusieurs processus s'exécutent *en même temps*. Le parallélisme est donc une manière d'exécuter un système concurrent. On précise aussi la notion de *distribution* qui indique généralement que les processus sont sur des machines distantes reliées par un réseau. Un système *distribué* est donc un système concurrent avec certaines hypothèses, telles qu'une grande latence dans les communications entre processus, ou encore la possibilité de pertes de messages sur le réseau.

Concrètement, la programmation concurrente nécessite des mécanismes d'interaction entre processus afin que ceux ci puissent se synchroniser ou échanger des informations. Il existe plusieurs manières de réaliser ces interactions, parmi lesquelles nous décrivons celles que nous estimons être les trois principales :

La mémoire partagée. Une partie de la mémoire est commune à plusieurs processus, qui peuvent lire ou écrire dans cette mémoire partagée pour se synchroniser ou échanger des informations. Afin d'organiser l'accès à la mémoire partagée, il existe plusieurs mécanismes d'exclusion mutuelle, tels que les verrous, les sémaphores, les moniteurs, ou encore la STM (*Software Transactional Memory*, mémoire transactionnelle logicielle).

La mémoire partagée est le mécanisme le plus répandu et le plus enseigné, notamment car il est utilisé depuis longtemps sur les machines à un seul processeur. Un des premiers programmes concurrents à utiliser des sémaphores est le système THE de Dijkstra [Dij68], dès 1965. Les techniques de verrouillage sont toutefois réputées difficiles à maîtriser, entre autres car une mauvaise gestion des verrous peut facilement mener à des interblocages ou encore à des modifications de la mémoire qui ne devraient pas être autorisés.

Ce type d'interaction se retrouve par exemple dans la bibliothèque des *threads* POSIX, qui permet d'écrire des programmes C où plusieurs processus partagent le même tas, dont l'accès est protégé typiquement par des verrous. En ce qui concerne la STM, elle est par exemple disponible par défaut dans Clojure [1], et proposée dans une bibliothèque standard pour Haskell [2].

Le passage de messages. Les processus peuvent interagir en s'envoyant des messages. Ce mécanisme d'interaction ne nécessite pas de mémoire partagée entre les processus, il est donc plus facile à mettre en place que la mémoire partagée pour des processus qui s'exécutent sur des machines distantes. Joe Armstrong, le créateur du langage Erlang [3], illustre dans sa thèse [Arm03] que la mémoire partagée entre les processus est la source de nombreux problèmes qui disparaissent quand seul le passage de messages est autorisé. Si ce mécanisme a gagné en popularité avec l'avènement des systèmes distribués, on peut noter qu'un des premiers systèmes à proposer le passage de messages entre processus est le noyau *nucleus* [Han70] destiné à la machine RC 4000, mis au point par Brinch Hansen dès 1969.

Le processus qui envoie un message peut soit directement continuer son exécution, on parle alors de passage de messages *asynchrones*, soit attendre la confirmation de la réception du message avant de continuer, ce qu'on désigne par passage de messages *synchrones*. Dans le cas asynchrone, il est souvent sous-entendu que l'ordre des messages est préservé entre les processus deux à deux, et que les messages ne peuvent pas être perdus, mais ces propriétés ne sont pas toujours vérifiées.

1. `http://clojure.org/concurrent_programming`
2. `http://wiki.haskell.org/STM`
3. `http://www.erlang.org`

Par exemple dans le langage Erlang, la seule interaction possible entre processus se fait par passage de messages asynchrones, mais le traitement d'un message à l'arrivée n'est pas garantie : un processus avec des messages en attente peut terminer sur une erreur puis être relancé (une opération courante en Erlang), les messages en attente sont alors perdus. Un exemple de passage de messages asynchrones non ordonnés et sans garantie de réception est l'utilisation du protocole UDP [4] pour la transmission des messages sur le réseau.

En ce qui concerne le passage de messages synchrones, on retrouve ce mécanisme par exemple dans les *channels* (canaux de communication) de CSP [Hoa85], repris plus tard dans Occam [Hoa88]. Les actions synchronisées de CCS [Mil80] et du π-calcul [Mil99] peuvent être considérée comme du passage de messages synchrones entre deux processus. Plus récemment, dans le langage Go [5], deux processus peuvent communiquer par un *channel* soit synchrone s'il est déclaré avec une taille égale à zéro, soit asynchrone si sa taille est supérieure à zéro.

Le rendez-vous. Deux processus ou plus peuvent se synchroniser et échanger des informations lors d'un rendez-vous. Un rendez-vous entre deux processus, appelé rendez-vous *binaire*, est proche d'un passage de message synchrone ; le rendez-vous permet en plus d'échanger des informations simultanément dans les deux sens alors que le passage de message synchrone transmet des informations uniquement d'un processus (l'envoyeur) vers un autre (le receveur). Le langage Ada [Taf00] par exemple offre une interaction par rendez-vous binaire, où un processus peut demander un rendez-vous sur une entrée d'un autre processus. Le processus qui accepte un rendez-vous sur une de ses entrées peut exécuter des instructions avant de conclure le rendez-vous, ce qui permet de calculer des informations à échanger.

Dans le cas général, un rendez-vous peut concerner un nombre arbitraire, supérieur ou égal à deux, de processus : on parle de rendez-vous *multiple*. On retrouve le rendez-vous multiple dans certains langages issus des algèbres de processus, tels que LOTOS [ISO89] et LNT [CCG+14]. À la différence d'Ada, dans ces deux langages un point de rendez-vous, aussi appelé *porte*, n'est pas rattaché à un processus en particulier. Ainsi, lorsqu'un processus réalise un rendez-vous sur une porte, il ne connaît pas l'identité des autres processus participants, ni leur nombre. Il est tout de même possible d'accéder à ces informations en les échangeant explicitement lors du rendez-vous.

Le rendez-vous multiple ne semble pas être populaire dans les langages de programmation courants. C'est toutefois la véritable primitive d'interaction des processus CSP, sur laquelle les *channels* sont ensuite construits. La majorité des langages inspirés de CSP ne semblent avoir retenus que les *channels* comme moyen d'interaction.

Il existe ainsi plusieurs mécanismes d'interaction, chacun présentant des avantages et des inconvénients pour programmer un système concurrent, et de nombreux langages de pro-

4. `http://tools.ietf.org/html/rfc768`
5. `http://golang.org`

grammation offrent un ou plusieurs de ces mécanismes. Cependant, quels que soient les moyens d'interaction utilisés, la complexité intrinsèque de la programmation concurrente vient du fait que l'ordonnancement des processus n'est pas maîtrisé, et qu'il varie entre les exécutions. Un système concurrent possède ainsi autant de chemins d'exécution possibles qu'il y a de façons d'ordonner l'exécution des processus.

La programmation concurrente est considérée comme difficile car il faut envisager *tous* les chemins d'exécution possibles pour s'assurer de l'absence de bogues. Lancer une exécution pour effectuer un test ne permet de valider ce test que sur un seul chemin d'exécution. Si l'ordonnancement des processus ne peut pas être contrôlé lors des phases de tests, il est alors laborieux de parcourir tous les chemins d'exécution. Afin de pouvoir s'assurer de l'absence de bogues dans les systèmes concurrents, des méthodes formelles d'analyses ont été développées.

1.2 Méthodes formelles et génération d'exécutable

Les méthodes formelles désignent de manière générale les techniques basées sur les mathématiques pour l'étude des systèmes informatiques. Il existe plusieurs types de méthodes formelles, comme par exemple l'analyse statique, la preuve de programme, ou le *model checking*.

Les techniques d'analyse statique, comme l'interprétation abstraite, permettent de vérifier certaines propriétés d'un programme sans avoir à l'exécuter. Parmi les outils reposant sur l'analyse statique, on peut citer Astrée [CCF+05], CPAchecker [BK11] ou encore, plus récemment, Verasco [JLB+15].

La preuve de programme consiste à raisonner sur la structure d'un programme pour démontrer, en utilisant des techniques mathématiques, que le programme respecte certaines propriétés. La preuve de programme peut être conduite à la main, mais il devient difficile de traiter de grands systèmes. Plusieurs techniques qui permettent d'aider le déroulement de preuves ont été mises au point, telles que les assistants de preuve Coq [CH88] ou Isabelle [NP92], ou encore le système de preuves TLAPS [CDLM10] pour les spécifications TLA+.

Le *model checking* consiste à vérifier une propriété sur un système en explorant les états atteignables lors de l'exécution de ce système. En particulier, dans le cadre de la programmation concurrente, tous les entrelacements possibles des différents fils d'exécutions sont parcourus. De plus, quand un programme ne respecte pas une certaine propriété, la plupart des outils de *model checking* peuvent retracer un chemin d'exécution qui mène à un comportement en désaccord avec la propriété. Cette capacité à générer un contre-exemple facilite le débogage des systèmes concurrents. On s'intéresse principalement au *model checking* par la suite.

Toutes les méthodes formelles sont basées sur des raisonnements mathématiques, elles nécessitent une description précise et non ambiguë du système à étudier, sous forme d'objet mathématique. Les langages de programmation dans lesquels les systèmes sont décrits doivent donc avoir une sémantique formellement définie afin de pouvoir construire un objet mathématique à partir d'un programme. La recherche en informatique a donné naissance à des langages de *spécification* conçus principalement pour décrire formellement un système, mais ces langages ne sont pas toujours équipés de compilateurs. Une grande majorité des langages de programmation utilisés dans l'industrie sont des langages *d'implémentation*, conçus principalement pour générer du code exécutable à partir d'un programme. Malheureusement, la plupart des langages d'implémentation ont une sémantique définie par un texte en langage naturel, non formel, ou par une certaine implémentation de référence.

Il n'existe pas de frontière nette entre les langages de spécification et les langages d'implémentation, certains langages étant équipés à la fois d'une sémantique formelle et de compilateurs. Dans la suite, nous présentons un aperçu des outils apparentés au *model checking* pour les langages d'implémentation, avant de s'intéresser aux langages de spécification équipés à la fois d'outils de vérification formelle ainsi que de générateurs de code exécutable.

1.2.1 *Model checking* pour les langages d'implémentation

Pour faire face à la complexité de la programmation concurrente, des outils de vérification formelle ont été développés pour les langages d'implémentation. Il existe de nombreux outils, nous évoquons ici uniquement quelques outils de *model checking* concernant les langages de programmation populaires en milieu industriel.

Le *model checking* procède en explorant l'espace d'états d'un système, ce qui nécessite non seulement le contrôle de l'ordonnancement des processus, mais aussi la capacité de sauvegarder l'état d'un système pour pouvoir explorer un chemin d'exécution et plus tard revenir à cet état pour explorer un autre chemin d'exécution. Pour les langages à sémantique formelle, ces opérations sont typiquement réalisées sur un objet mathématique obtenu à partir d'un programme. Dans le cas des langages d'implémentation sans sémantique formelle, il est souvent difficile d'extraire de manière sûre un objet mathématique à partir du code source. La technique généralement utilisée est alors d'observer et de manipuler directement le comportement du code exécutable produit par les compilateurs des langages d'implémentation : à défaut d'une sémantique formelle au niveau du programme source, on utilise la sémantique, souvent mieux définie, des instructions de plus bas niveau.

Par exemple, un programme Java est compilé vers du code octet (*byte code*) qui est ensuite interprété par la machine virtuelle Java. Cette machine virtuelle contrôle l'exécution du programme, dont l'ordonnancement des processus et les modifications de la mémoire. Le projet Java Pathfinder [VHBP00] de la NASA, disponible depuis 2000, exploite cette caractéristique en proposant une machine virtuelle Java modifiée pour permettre l'explo-

ration de l'espace d'états d'un programme exprimé en code octet. Une approche similaire est adoptée par les outils McErlang pour le langage Erlang, et CHESS [MQ06] ou MoonWalker [dBNR09] pour la plateforme Microsoft .NET.

Les langages sans machine virtuelle, tels que le C, génèrent des programmes qui utilisent des appels système notamment pour réaliser des interactions entre processus. L'approche vue précédemment est donc applicable en contrôlant l'exécution d'un programme et en observant les appels systèmes réalisés. Cette technique a été mise en place dès 1997, plusieurs années avant l'outil Pathfinder, par l'équipe de Patrice Godefroid aux laboratoires Bell, au travers de l'outil Verisoft [God05] capable d'analyser du code exécutable obtenu à partir de sources C, mais aussi C++ ou encore Tcl. Plus récemment, cette technique a été reprise pour développer un *model checker* couplé à la plate-forme SimGrid [MQR11].

Si ces outils sont aptes à analyser des systèmes concurrents directement au niveau du code compilé, la plupart profitent néanmoins de la possibilité pour l'utilisateur de placer des annotations au niveau du programme source pour, par exemple, préciser quelles interactions doivent être prises en compte ou ignorées lors de la construction de l'espace d'états. De plus, le traitement de langages d'implémentation entraîne parfois des complications qui limitent l'étendue des vérifications possibles.

1.2.2 Génération de code pour les langages de spécification

Nous abordons ici des travaux qui s'approchent de ceux réalisés dans le cadre de la thèse, à savoir générer une implémentation distribuée à partir de LNT, un langage de spécification de systèmes concurrents avec le rendez-vous multiple comme mécanisme d'interaction. On se focalise donc sur les langages de spécification conçus pour modéliser les systèmes concurrents, qui proposent à la fois des outils de vérification formelle, notamment de *model checking*, ainsi que de générateurs de code exécutable, et si possible distribué.

Du côté des langages synchrones, qui font l'hypothèse que tous les processus partagent une même horloge globale, Esterel [BG92] est un exemple d'environnement proposant à la fois des outils de vérification formelle et des compilateurs. L'hypothèse d'une horloge globale n'étant pas très pertinente pour traiter les systèmes distribués [Lam78], nous n'explorons pas plus en détail le domaine des langages synchrones.

Dans le monde des langages asynchrones, LOTOS [ISO89] est un langage formel basé sur les algèbres de processus, et normalisé par un standard ISO. Le projet Topo [MdMSA93] propose plusieurs outils de vérification formelle pour LOTOS, ainsi que de la génération de code en C ou en Ada. Cependant, les implémentations ne sont pas distribuées, et Topo n'est plus maintenu. La boîte à outils CADP offre un riche environnement de traitement et de vérification pour des spécifications LOTOS. En particulier, l'outil EXEC/CÆSAR [GVZ01] de CADP permet de générer un programme séquentiel en C.

Le langage Estelle [ISO88] est aussi un langage formel normalisé par un standard ISO, qui

permet de décrire des systèmes concurrents. Il est possible de vérifier des propriétés sur une spécification Estelle avec Sedos [DDG89]. Le projet Echidna [JJ92] permet de produire une implémentation distribuée à partir d'une spécification Estelle. Les outils concernant Estelle ne semble cependant plus être maintenus à ce jour.

L'environnement UPPAAL [BLM+01] est dédié à l'étude formelle des réseaux d'automates temporisés. L'outil associé Times [AFM+04] permet de générer du code C séquentiel, mais uniquement pour le système d'exploitation BrickOS destiné à la brique Mindstorms RCX de Lego. Moby [Tap98] est un autre projet connecté à UPPAAL qui peut générer des programmes à exécuter, mais encore une fois uniquement à destination d'un automate programmable industriel.

Le *model checker* SPIN [Hol04] permet de vérifier des propriétés sur des programmes Promela, un langage avec une syntaxe proche du C où l'interaction entre processus est réalisée par des canaux de communication, à la CSP. Il existe un projet de compilateur de Promela vers du C distribué [Löf96], dont l'implémentation a une structure client-serveur et pour laquelle l'utilisateur doit spécifier explicitement quels processus sont clients ou serveurs. Plus récemment, un raffinement de Promela vers C a aussi été proposé [Sha13], mais l'implémentation générée n'est pas distribuée.

Les langages de *coordination* se focalisent sur la définition d'interactions entre processus, distinctement du comportement des processus eux-mêmes. Parmi les langages de coordination, Reo [Arb04] est équipé d'outils de vérification formelle. Le projet Dreams [PCdVA12] a donné naissance à un générateur d'implémentation distribuée en Scala [Ode09], ce qui permet une exécution sur machine virtuelle Java. Il existe aussi un générateur de code concurrent qui vise l'environnement proto-runtime [HC11] plutôt que les classiques threads POSIX. Proto-runtime permet d'exécuter plusieurs fils d'exécution tout en court-circuitant le système d'exploitation afin d'obtenir de meilleures performances. Plus récemment, le langage Chor [CM13] permet de programmer et de vérifier des chorégraphies utilisant le passage de messages pour les interactions entre processus, et il est aussi équipé d'un générateur de code distribué.

L'environnement BIP [BBS06] permet de décrire formellement des systèmes sous la forme de composants qui interagissent par rendez-vous multiple avec une notion de priorité entre les rendez-vous. Il existe beaucoup d'articles qui présentent les techniques utilisées pour la génération de code distribué [BBJ+10a, BBJ+10b, BBQ11, BBQ13]. La thèse de Jean Quilbeuf [Qui13] est une bonne référence pour avoir une vue d'ensemble du travail réalisé. Le protocole α-core [PCT04] est utilisé pour l'implémentation des rendez-vous, et une surcouche sur ce protocole assure les priorités. En ce qui concerne la vérification formelle sur les modèles BIP, seule la détection de deadlock, via D-Finder [BGL+11], semble être disponible à ce jour.

Partitionnement. Lors de la génération de code distribué à partir d'une spécification de programme concurrent, il faut choisir un partitionnement du programme pour savoir

comment délimiter les parties à déployer sur différentes machines. Ce partitionnement n'est pas toujours évident à extraire du programme source. Dans le cas de Reo par exemple, de nombreux connecteurs sont susceptibles de devenir autant de programmes indépendants. Pour des raisons de performance, il n'est pas raisonnable de générer un programme pour chaque connecteur. Les auteurs de Reo développent donc des techniques de partitionnement plus élaborées [JSA14] qui permettent de définir des regroupements de connecteurs de manière à limiter les interactions entre processus distants dans l'implémentation générée. Pour le cas de BIP, le générateur de code distribué ne peut pas choisir par lui-même comment partitionner le système, un partitionnement devant nécessairement être fourni par l'utilisateur en supplément des sources BIP. On note que les langages basés sur les algèbres de processus fournissent naturellement une partition – sinon optimale, du moins pertinente – via les points de synchronisation nommés, tels que les portes en LOTOS ou en LNT.

1.3 Spécification et vérification formelles avec CADP

La boîte à outils CADP regroupe de nombreuses fonctionnalités pour la spécification et la vérification formelle de systèmes concurrents. Dans CADP, l'objet mathématique qui définit formellement le comportement d'un système est un LTS (*Labelled Transition System*, système de transition étiquetées). Un LTS permet de représenter l'espace d'états d'un système concurrent, en incluant les différents chemins d'exécutions possibles.

Il existe plusieurs langages formels dont la sémantique est définie sous forme de LTS. CADP offre des flots de compilation capables de générer le LTS correspondant à une spécification exprimée dans certains de ces langages, notamment LOTOS et LNT. Ensuite, la plupart des outils de CADP reposent sur l'exploration ou la transformation de LTS, pour par exemple vérifier une propriété ou réduire un espace d'états modulo une relation d'équivalence.

Dans cette section, nous commençons par présenter le langage LNT. Nous définissons ensuite la notion de vecteurs de synchronisation pour une composition parallèle de processus, puis nous introduisons les outils CADP utilisés dans le cadre de la thèse. Enfin, nous rappelons la définition des relations d'équivalence disponibles dans CADP.

1.3.1 Introduction au langage LNT

Le langage LNT [CCG+14] est développé par l'équipe INRIA/CONVECS pour offrir un moyen de décrire formellement un système dans une syntaxe proche des langages classiques d'implémentation. Le langage historique de CADP est LOTOS [ISO89], dont la prise en main est réputée difficile. Une révision en profondeur du langage LOTOS a conduit à la standardisation du langage E-LOTOS [ISO01]. LNT s'inspire des améliorations proposées dans le cadre de E-LOTOS dans le but de proposer un langage pratique pour la spécification

de systèmes concurrents. Depuis 2010, LNT est le langage de spécification recommandé pour les utilisateurs de CADP.

La définition formelle du langage LNT et de sa sémantique est disponible dans le manuel de référence de l'outil LNT2LOTOS [CCG+14]. L'introduction au langage LNT que nous présentons ici est informelle. Elle vise principalement à (1) permettre au lecteur de comprendre les extraits de spécifications qui apparaissent dans le reste de ce mémoire, et (2) établir une partie du vocabulaire utilisé dans la suite du mémoire.

Dans ce document, le code LNT est généralement en anglais, et les commentaires sont eux rédigés en français. Il existe deux types de commentaires en LNT. Le double tiret "`--`" marque le début d'un commentaire qui termine en fin de ligne. Un commentaire multi-lignes commence par "`(*`" et termine par "`*)`". Les commentaires sont représentés en italique (exemple : *(∗ commentaire ∗)*). Enfin, les mots-clés de LNT sont marqués en gras (exemple : **process**).

Type de données

Il est possible de déclarer des types de données abstraits. Certains types sont déjà prédéfinis, tels que les booléens (bool), les entiers naturels (nat), les entier relatifs (int), les réels (real), les caractères (char) et les chaînes de caractères (string).

Par exemple, LNT permet de définir le type "drink", dont les valeurs sont différentes boissons :

```
type drink is
   tea, coffee, milk
end type
```

Il est possible de définir des types qui sont les listes de valeurs d'un autre type, ce que nous appelons un "type liste" :

```
type drink_list is
   list of drink
end type
```

D'une manière similaire, on peut déclarer des types listes ordonnées (**sorted list of**), ensemble (**set of**), ou ensemble ordonnées (**sorted set of**). Les types tableaux sont aussi disponibles (**array** [0 .. n] **of**).

On peut définir des types enregistrement, dont les constructeurs contiennent des valeurs d'autres types. Par exemple, le constructeur du type "point2d" suivant définit un couple de coordonnées entières :

```
type point2d is
   point2d (x, y: int)
end type
```

Fonctions

Les opérations sur les types sont définies par des fonctions. La passage de paramètres à une fonction peut se faire par copie (paramètre **in**), ou par référence (paramètre **out** ou bien **inout**). Par défaut, un paramètre est passé par copie. Une fonction peut retourner une valeur d'un certain type. Les appels récursifs sont autorisés. Les structures de contrôle, telles que le branchement conditionnel ou les boucles, sont disponibles.

Par exemple, on peut définir la factorielle d'un entier naturel de manière récursive :

```
function fact_rec (x: nat) : nat is   -- un paramètre passé par copie, retourne un entier naturel
   if x == 0 then                     -- branchement conditionnel
      return 1
   else
      return x * (fact_rec (x-1))     -- appel récursif
   end if
end function
```

Il est possible de déclarer des variables locales au sein d'une fonction, et d'affecter une valeur à une variable avec l'opérateur ":=". De plus, l'opérateur ";" désigne la composition séquentielle d'instructions (et non la fin d'une instruction comme en C par exemple). Voici une version itérative de la factorielle :

```
function fact_iter (x: nat) : nat is
   var res: nat in          -- variable locale
      res := 1;             -- affectation
      while x > 0 loop      -- boucle
         res := res * x;
         x   := x - 1
      end loop;
      return res
   end var
end function
```

L'opérateur "**case**" permet de réaliser du *pattern matching* (filtrage par motif) sur des valeurs de n'importe quel type. Dans le cadre du *pattern matching*, le mot clé **any** (*wildcard*) correspond toujours à la valeur évaluée. La fonction suivante retourne vrai si la boisson passée en argument fait partie des boissons chaudes, c'est-à-dire thé ou café :

```
function hot_drink (d: drink) : bool is
   case d in
      tea | coffee -> return True
    | any -> return False
   end case
end function
```

Fonctions prédéfinies pour les types

Lors de la définition d'un type, certaines fonctions peuvent être automatiquement définies. Les fonctions prédéfinissables dépendent du type déclaré. Voici un aperçu des fonctions disponibles :

- Pour tous les types exceptés les types ensembles et ensembles ordonnés, des fonctions de comparaison (==, !=, <=, etc) peuvent être prédéfinies.
- Pour les types listes et ensembles, ordonnés ou non, les constructeurs "nil" (liste ou ensemble vide) et "cons" (ajout d'une valeur à une liste ou un ensemble) sont toujours prédéfinis. Il existe des raccourcis syntaxiques pour ces constructeurs. Le constructeur "nil" peut être exprimé par le raccourci syntaxique "{}". Une liste de valeurs entourée d'accolades est un raccourci syntaxique pour une série d'appels au constructeur "cons" (voir exemple ci-dessous).
- Pour les types listes ordonnées, ensembles, et ensembles ordonnés, le constructeur "insert" est toujours prédéfini. Ces types sont gérés comme des listes, et le constructeur "insert" a l'avantage par rapport au constructeur "cons" de vérifier qu'il ne crée pas de doublons dans les ensembles, et qu'il maintient une liste ordonnée à l'ajout d'une valeur.
- Pour les types listes et ensembles, ordonnés ou non, les fonctions suivantes peuvent être prédéfinies :

Nom	Description
empty	teste la présence d'au moins un élément
length	retourne le nombre d'éléments
member	teste la présence d'un certain élément
access	retourne l'élément d'un certain index
delete	retourne la liste (l'ensemble) privé de la première occurrence d'un certain élément
remove	retourne la liste (l'ensemble) privé de toutes les occurrences d'un certain élément
head	retourne l'élément en tête
tail	retourne la liste (l'ensemble) privé de l'élément en tête
union	retourne l'union de deux ensembles ou la concaténation de deux listes

- Pour les types ensembles, ensembles ordonnés, et listes ordonnées, on peut automatiquement définir les fonctions suivantes :

Nom	Description
inter	retourne l'intersection de deux ensembles ou deux listes ordonnées
diff	retourne l'ensemble (la liste ordonnée) passé en premier argument en ayant retiré les éléments de l'ensemble (la liste ordonnée) passé en deuxième argument

— Pour les types enregistrements, on peut automatiquement obtenir les fonctions “get” pour accéder à un champ, et “set” pour mettre à jour un champ. Le nom des fonctions effectivement prédéfinies comporte le nom du champ correspondant, et il existe du sucre syntaxique pour ces fonctions (voir exemple ci-dessous).

Voici par exemple des valeurs du type de liste de boisson :

```
{}          -- sucre syntaxique pour "nil"
{tea, milk} -- sucre syntaxique pour "cons (tea, cons (milk, nil ))"
```

La liste des fonctions à prédéfinir est introduite par le mot clé **with**. La définition du type ensemble de boissons “drink_set” suivant requiert que les fonctions “member” et “inter” soient prédéfinies :

```
type drink_set is
   set of drink
   with "member", "inter"
end type
```

En ce qui concerne les types enregistrements, voici un exemple de sucre syntaxique pour les fonctions “get” et “set” :

```
type point3d is
   point3d (x, y, z: int)
   with "get", "set"
end type

p = Point3D (1, 2, 3);
get_x (p)     -- retourne : 1
p.z           -- sucre syntaxique pour "get_z (p)", retourne : 3
set_y (p, 4) -- retourne : point3d (1, 4, 3)
p.{x => 5}  -- sucre syntaxique pour "set_x (p, 5)", retourne : point3d (5, 2, 3)
```

Actions et offres

Un processus réalise des *actions*, et une action peut soit être *interne*, soit être réalisée sur une *porte*. L’action interne est désignée par “i”. Une action sur porte est désignée par l’identifiant de la porte, suivi éventuellement d’*offres* entre parenthèses.

Les offres permettent d’échanger des données lors d’une action sur porte. Une offre en *mode envoi* permet d’envoyer une valeur, elle est notée par un “ !”, optionnel[6], suivi d’une expression de valeur. Une offre en *mode réception* permet de recevoir une valeur dans une variable, elle est notée par un “ ?” suivi d’une variable. Une action interne ne peut pas avoir d’offre.

6. De manière générale, nous omettons donc ce signe pour les offres en mode envoi.

Voici par exemple un comportement qui effectue une action interne, suivie par une action sur la porte A où la première offre désigne l'envoi d'un entier naturel, et la deuxième une réception d'un booléen :

```
var b : bool in
   i ;
   A (123, ?b)
end var
```

Garde de communication

Il est possible de conditionner la réalisation d'une action par une fonction booléenne sur les variables couramment définies, y compris les offres de l'action. On appelle cette condition une *garde de communication*, ou plus simplement une garde. Une garde est introduite à la suite d'une action par le mot clé **where**.

Voici une action sur la porte A, qui n'est autorisée que lorsque la valeur de x est positive et la valeur reçue dans y est inférieure à z :

```
var x, y, z : int in
   ... ;          -- affecte les valeurs de x et z
   A (x, ?y) where (x > 0) and (y < z);
   ...
end var
```

Channels

Un *channel* permet de limiter les *profils d'offres* acceptés sur une porte. Un profil d'offres correspond à une liste de types, qui indique le nombre et le type des offres autorisées. Un channel peut spécifier plusieurs profils.

Voici différents exemples de définitions de channels :

```
channel none is () end channel      -- aucune offre autorisée
channel int is (int) end channel    -- une offre de type entier autorisée
channel misc is   -- plusieurs profils autorisés
   (nat, int),
   (bool, char, bool)
end channel
```

Le channel **any** est prédéfini et autorise n'importe quel profil d'offres. De plus, on considère dans la suite que pour chaque type défini, un channel du même nom qui autorise une offre de ce type (comme par exemple le channel “int” ci-dessus) est aussi défini.

Processus

Le processus est l'entité d'exécution d'une spécification LNT. Un processus reçoit en argument une liste de portes sur lesquelles il peut réaliser des actions. Chaque porte est caractérisée par un channel. Un processus peut aussi recevoir une liste de valeurs en argument. La liste de portes est passée entre crochets, la liste de valeurs est passée entre parenthèses.

Voici la définition du processus P0, qui reçoit en argument une porte G et un entier x, et dont le comportement consiste à effectuer une action sur la porte G, avec en offre la valeur de x en mode envoi :

```
process P0 [G : int] (x : int) is
   G (x)
end process
```

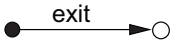
Les actions sont les événements observables de l'exécution d'un processus. La sémantique formelle de LNT définit comment, à partir de la spécification d'un processus, on peut construire le LTS qui représente les comportements possibles de ce processus. Dans la suite, les LTS correspondants aux exemples de processus sont illustrés. L'état initial du LTS est marqué d'un disque plein. Une étiquette accompagne chaque transition, elle indique l'action réalisée au passage de cette transition.

Le processus P1 ne réalise aucune action, il bloque son exécution avec l'opérateur **stop** :

```
process P1 is
   stop
end process
```

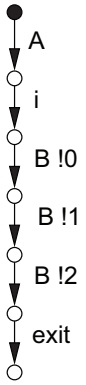

Un processus qui termine sans se bloquer effectue implicitement l'action spéciale " exit ". Le processus P2 n'effectue aucune action spécifique (opérateur **null**) et termine sans se bloquer.

```
process P2 is
   null
end process
```



Tout comme pour les fonctions, l'opérateur ";" désigne la composition séquentielle. De plus, il est possible d'appeler des fonctions depuis un processus, et les constructions des fonctions (branchement conditionnel, boucle, *pattern matching*, etc) sont aussi utilisables directement au sein d'un processus. Le processus P3 réalise une action sur la porte A, puis une action interne, puis une boucle de 3 actions sur la porte B :

```
process P3 [A, B: none] is
   var n : nat in
      A;
      i ;
      for n := 0 while n < 3 by n := n+1 loop
         B (n)
      end loop
   end var
end process
```

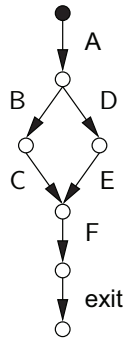


Choix non déterministe

Un processus peut faire un choix non déterministe entre plusieurs actions à l'aide de l'opérateur **select**. Cet opérateur permet d'introduire plusieurs comportements possibles, qui sont délimités par "**[]**".

Le processus P4 effectue un choix non déterministe entre l'action sur la porte B et celle sur la porte D. Une fois qu'il a réalisé une action d'une branche de ce choix non déterministe, il est contraint d'exécuter le reste de cette branche.

```
process P4 [A, B, C, D, E, F: none] is
   A;
   select
      B;
      C
   []
      D;
      E
   end select;
   F
end process
```

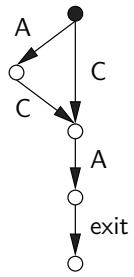


On constate que le choix non déterministe résulte, dans le LTS, en un état à partir duquel il existe plusieurs transitions sortantes.

Le choix non déterministe permet de réaliser le comportement de l'une des branches qu'il introduit. Il faut alors se méfier des choix non déterministes contenant des constructions **if** sans composante **else**, car la composante **else** par défaut est **null**, c'est-à-dire un comportement toujours exécutable. La construction **only if** est du sucre syntaxique pour un branchement **if** dont la composante **else** est un blocage **stop**.

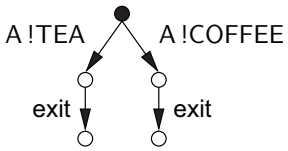
Le processus P5 illustre l'utilité de l'opérateur **only if** : dans le premier **select**, la deuxième branche peut être traversée via le **null** implicite, et donc une action sur la porte C peut directement avoir lieu. Dans le deuxième select, le **stop** implicite évite la possibilité de traverser cette branche sans action.

```
process P5 [A, B, C: none] is
   select
      A
   []
      if false then
         B
      -- implicite : else null
      end if
   end select;
   C; -- peut avoir lieu sans A avant
   select
      A
   []
      only if false then
         B
      -- implicite : else stop
      end if
   end select
end process
```



Dans une expression, le mot clé **any** autorise n'importe quelle valeur d'un type, avec éventuellement une restriction introduite par **where**. Cette construction peut aussi faire apparaître du non-déterminisme. Par exemple, le processus P6 peut réaliser une action sur la porte A en passant une valeur de boisson chaude :

```
process P6 [A: drink] is
   var d : drink in
      d := any drink where hot_drink (d);
      -- ici, d vaut une valeur de drink
      -- pour laquelle hot_drink est vraie
      A (d)
   end var
end process
```

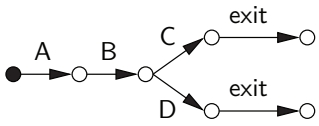


Composition parallèle

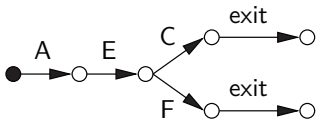
L'opérateur de composition parallèle **par** permet de déclarer plusieurs processus qui s'exécutent indépendamment, en parallèle, tout en spécifiant sur quelles portes les processus doivent synchroniser leurs actions. Les actions sur les portes pour lesquelles aucune synchronisation n'est imposée sont réalisées indépendamment par chaque processus. L'action de terminaison "exit" est toujours synchronisée entre tous les processus d'une composition parallèle.

L'exemple suivant montre la composition parallèle des processus P7 et P8, qui doivent synchroniser leurs actions sur les portes A, C, D et F :

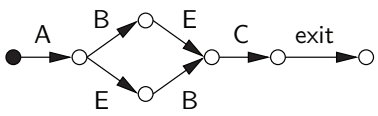
```
process P7 [A, B, C, D: none] is
   A;
   B;
   select
      C
   [] D
   end select
end process
```




```
process P8 [A, E, C, F: none] is
   A;
   E;
   select
      C
   [] F
   end select
end process
```




```
-- Composition parallèle
par A, C, D, F in
   P7 [A,B,C,D]
|| P8 [A,E,C,F]
```



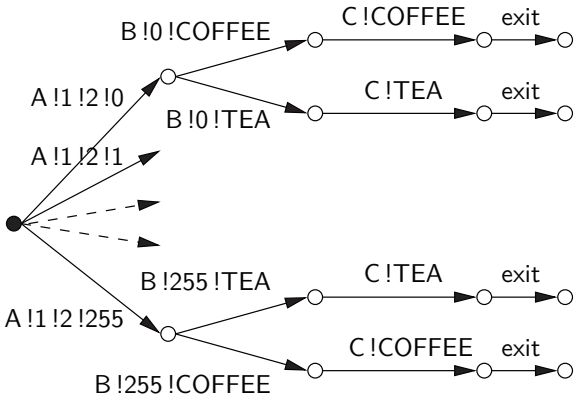
La première action sur la porte A est commune aux deux processus. Ensuite, les actions sur les portes B et E n'ont pas à être synchronisées, elles peuvent donc être réalisées dans n'importe quel ordre. Les actions sur les portes C, D, et F doivent être synchronisées. Lorsque le processus P7 est prêt pour une action sur la porte C ou la porte D, et le processus P8 est prêt pour une action sur la porte C ou la porte F, la seule action qui peut être réalisée par les deux processus est celle sur la porte C. Enfin, les deux processus synchronisent leur action "exit".

Lors d'une action synchronisée, les éventuelles offres sont confrontées. Une action ne peut avoir lieu que lorsque les offres de tous les processus sont *compatibles*. Deux offres sont compatibles lorsqu'elles ont le même type, et lorsqu'elles ont la même valeur si elles sont toutes les deux en mode envoi. L'action synchronisée résultante est réalisée avec la *fusion* des offres. La fusion de deux offres compatibles impose une valeur si au moins l'une des deux offres est en mode envoi. Si les deux offres sont en mode réception, alors l'offre résultant de la fusion reste en mode réception. La présence de garde de communication peut limiter les valeurs qui sont acceptées pour une offre en réception.

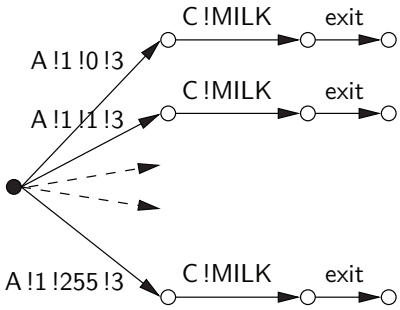
L'exemple suivant illustre des synchronisations d'actions avec des offres. Les LTS des processus P9 et P10 ne sont représentés que partiellement, car l'énumération des valeurs entières possibles créent beaucoup de transitions[7].

7. Par défaut, les entiers naturels sont bornés à 255, il est cependant possible de régler cette borne à des valeurs plus élevées ou plus basses.

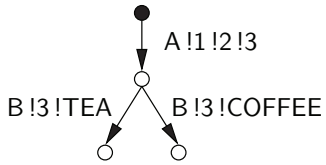
```
process P9 [A, B, C: any] is
   var x: nat, d: drink in
      A (1, 2, ?x);
      B (x, ?d) where hot_drink (d);
      C (d)
   end var
end process
```




```
process P10 [A, C: any] is
   var y: nat in
      A (1, ?y, 3);
      C (milk)
end process
```




```
par A, C in
   P9 [A,B,C]
|| P10 [A,C]
end par
```



Pour l'action sur la porte A, les offres des deux processus P9 et P10 sont compatibles, et leur fusion fixe une valeur à toutes les offres. Ensuite, le processus P9 réalise une action non synchronisée sur la porte B. La première offre de cette action est en mode envoi, et impose la valeur entière de x, c'est-à-dire 3. La deuxième offre est en réception, sa valeur peut être n'importe laquelle parmi celles de son type, "drink". La garde contraint cette valeur à faire partie des boissons chaudes. Comme aucun autre processus ne vient imposer de valeur avec une offre en mode envoi, deux actions sont possibles, pour les deux valeurs possibles de boisson chaude. Enfin, la composition parallèle impose la synchronisation des actions sur la porte C. Toutefois, l'offre proposée par le processus P10 n'est compatible avec aucune des offres possibles pour les actions du processus P9. Cette action ne peut donc pas avoir lieu, et les deux processus sont bloqués.

1.3.2 Composition parallèle et vecteurs de synchronisation

Une composition parallèle permet d'exprimer quelles synchronisations sont possibles pour chaque porte. Cette information peut être représentée pour chaque porte sous la forme d'une liste d'ensembles d'identifiants de processus. Chacun de ces ensembles représente un groupe de processus qui peuvent réaliser une action synchronisée sur cette porte. Nous

appelons un ensemble d'identifiants de processus un *vecteur de synchronisation* (*synchronization vector*) [Lan05].

L'opérateur de composition parallèle **par** permet d'exprimer précisément les rendez-vous qui sont possibles sur chaque porte. Nous présentons ici la syntaxe complète de cet opérateur :

- Lorsqu'une porte est indiquée entre **par** et **in**, les actions sur cette porte sont synchronisées entre tous les processus de la composition (comme dans les exemples de la section 1.3.1).
- Lorsqu'une porte est placée en préfixe d'un processus (avant "–>"), alors les actions sur cette porte sont synchronisées entre tous les processus préfixés par cette porte.
- Lorsqu'une porte est indiquée entre **par** et **in**, et est suivie d'un "**#**" lui-même suivi d'un nombre n, alors les action sur cette porte sont synchronisées entre n'importe quel ensemble de n processus parmi les m processus de la composition. On parle dans ce cas de *rendez-vous à n-parmi-m* [8].

Nous illustrons cette syntaxe sur la composition parallèle suivante, où on ne préoccupe pas du comportement des processus :

```
par A, B #2 in
   C     -> P1 [A,B,C]
|| D     -> P2 [A,B,D]
|| C, D -> P3 [A,B,C,D]
end par
```

Cette composition parallèle exprime les listes de vecteurs de synchronisation suivantes :

- pour la porte A : {P1, P2, P3}
- pour la porte B : {P1, P2}, {P1, P3}, {P2, P3}
- pour la porte C : {P1, P3}
- pour la porte D : {P2, P3}

Au sein de DLC, nous utilisons l'outil EXP.OPEN [Lan05] pour extraire automatiquement les vecteurs de synchronisation d'une composition parallèle.

1.3.3 Aperçu des outils CADP utilisés

La boîte à outils CADP offre plus d'une cinquantaine d'outils. Nous présentons ici un aperçu de l'architecture de CADP et nous décrivons les principaux outils utilisés dans le cadre de cette thèse. Pour plus d'informations, les pages manuels des outils sont disponibles en ligne sur le site de CADP [9].

8. Le rendez-vous à n-parmi-m n'est pas encore disponible directement en LNT, il est cependant utilisable par le biais de l'outil EXP.OPEN, présenté par la suite.

9. `http://cadp.inria.fr`

La plupart des outils de CADP opèrent sur des espaces d'états de processus, sous forme de LTS. Ces LTS peuvent être représentés *explicitement* ou *implicitement*. La représentation explicite d'un LTS consiste à stocker tous les états et toutes les transitions de l'espace d'états. Au sein de CADP, le format BCG (*Binary Coded Graph*) a été développé afin d'encoder la représentation explicite d'un LTS de manière compacte. BCG est un format binaire, et CADP offre une bibliothèque pour manipuler des fichiers au format BCG. Un LTS au format implicite consiste en un état initial et une fonction *post*, qui permet de parcourir des fragments de l'espace d´états, selon le but recherché. CADP est capable de produire un programme qui représentant le LTS implicite correspondant à une spécification formelle. L'accès à ce LTS est réalisé via l'interface OPEN/CÆSAR [Gar98]. Les outils de CADP peuvent ainsi être invoqués soit sur des LTS implicites, soit sur des LTS explicites.

Voici un aperçu des outils CADP utilisés dans le cadre de la thèse :

EXEC/CÆSAR [GVZ01] Le compilateur EXEC/CÆSAR produit un programme C séquentiel à partir d'une spécification de processus LNT. Le programme généré par EXEC/CÆSAR liste l'ensemble des actions immédiatement atteignables après une séquence d'instructions internes. En particulier, lorsqu'un processus fait face à un choix non déterministe, le programme permet de lister les différents actions possibles. Ce programme généré ne se suffit pas à lui même, et il a besoin de code complémentaire pour implémenter le choix d'action à réaliser. L'interface offerte par le code produit par EXEC/CÆSAR est présentée en détails au chapitre 4.

Le compilateur EXEC/CÆSAR permet d'obtenir aisément un programme pour chaque processus d'une composition parallèle. Nous tirons bénéfice de l'existence de ce compilateur pour nous concentrer, au sein de DLC, sur l'implémentation des interactions entre processus par rendez-vous multiple.

EXP.OPEN [Lan05] L'outil EXP.OPEN permet de définir des compositions parallèles de LTS stockés au format BCG. Cet outil permet d'utiliser des opérateurs de composition parallèle de différentes algèbres de processus (CCS, CSP, LOTOS, LNT, ...) pour exprimer la composition parallèle.

Nous utilisons une option de l'outil EXP.OPEN pour obtenir la liste des vecteurs de synchronisation à partir d'une composition parallèle LNT.

EVALUATOR4 et MCL [MT08, MS13] EVALUATOR4 est un des *model checker* de CADP. Il prend en entrée un LTS et une propriété de logique temporelle, et indique en sortie si la propriété est vérifiée par le LTS. EVALUATOR4 est capable de produire des graphes de diagnostique, qui permettent d'obtenir des contre-exemples lorsqu'une propriété n'est pas vérifiée par un LTS.

Les propriétés sont exprimées dans le langage MCL (*Model Checking Langage*) [MT08]. Ce langage permet d'exprimer des propriétés de logique temporelle, avec la particularité de pouvoir capturer les données d'une action et de faire référence à la donnée capturée plus tard dans la propriété.

Nous utilisons le *model checker* EVALUATOR4 au chapitre 5 pour vérifier différentes propriétés.

BISIMULATOR [MO08] L'outil BISIMULATOR permet de conclure si deux LTS sont équivalents modulo une certaine relation d'équivalence. Lorsqu'ils ne le sont pas, un diagnostique est disponible pour illustrer quel genre de trace est réalisable dans un des LTS mais pas dans l'autre. Nous utilisons cet outil au cours du chapitre 5.

SVL [Lan02] Enfin, l'outil SVL réalise des scénarios de vérification exprimés dans le langage de script SVL (*Script Verification Langage*). Les outils de CADP sont traditionnellement appelés en ligne de commande, et chaque outil possède plusieurs options. Le langage SVL permet de décrire aisément des scénarios de vérifications qui font intervenir plusieurs outils CADP. De plus, il est possible d'intercaler n'importe quelle commande de shell UNIX au sein d'un scénario SVL, ce qui facilite l'automatisation de vérifications formelles.

Nous utilisons un script SVL pour décrire notre méthode de vérification de protocole de synchronisation au chapitre 5. Ce script nous évite d'avoir à nommer explicitement certains outils de CADP, SVL se chargeant d'invoquer les outils adéquats pour conduire les vérifications requises.

1.3.4 Relations d'équivalences

L'espace d'états d'un système est représenté sous forme de LTS. Pour comparer le comportement de deux systèmes, on définit plusieurs relations d'équivalence entre LTS. Ces relations diffèrent selon leur manière de déterminer si deux états sont équivalents au regard des actions réalisables à partir de ces états. Dans cette section, les actions internes sont identifiées par l'étiquette τ.

Formalisation d'un LTS

Un LTS est un quadruplet $M = (Q, A, T, q_0)$, où :
- Q est l'ensemble des états
- A est l'ensemble des actions (étiquettes de transitions, représentant la porte et les offres d'une action)
- $T \in Q \times A \times Q$ est la relation de transition
- q_0 est l'état initial

L'ensemble A contient l'action interne, dénotée par τ. Une transition $(p, a, q) \in T$, aussi notée $p \xrightarrow{a} q$ signifie que le système peut passer de l'état p à l'état q en réalisation l'action a. Si L est un langage inclus dans A^*, alors $p \xrightarrow{L} q$ dénote une séquence de transitions $p \xrightarrow{a1} q1 \xrightarrow{a2} ... \xrightarrow{an} q$ telle que le mot $a1a2...an$ appartient à L. On considère sans perte de généralité que tous les états $q \in Q$ sont atteignables depuis l'état initial q_0 par une séquence de transition dans T (c'est-à-dire, $q_0 \xrightarrow{A^*} q$). Dans la suite, a dénote une action sur porte, et b dénote une action soit sur porte, soit interne. La fermeture transitive et réflexive de T est dénotée par T^*.

Deux LTS $M1 = (Q1, A1, T1, q_01)$ et $M2 = (Q2, A2, T2, q_02)$ sont équivalents selon une relation d'équivalence $\sim$ (noté $M1 \sim M2$) si et seulement si leurs états initiaux sont équivalents modulo $\sim$ (noté $q_01 \sim q_02$).

Équivalence forte (*strong equivalence*) [Par81]

Deux états p et q sont équivalents modulo l'équivalence forte (noté $p \sim_{str} q$) si et seulement si il existe une relation R_{str} telle que :

1. pour chaque transition $p \xrightarrow{b} p' \in T1$, il existe une transition $q \xrightarrow{b} q' \in T2$ telle que $p' \, R_{str} \, q'$
2. pour chaque transition $q \xrightarrow{b} q' \in T2$, il existe une transition $p \xrightarrow{b} p' \in T1$ telle que $p' \, R_{str} \, q'$

Équivalence de branchement (*branching equivalence*) [vGW89, vGW96]

Deux états p et q sont équivalents modulo l'équivalence de branchement (noté $p \sim_{bra} q$) si et seulement si il existe une relation R_{bra} telle que :

1. pour chaque transition $p \xrightarrow{b} p' \in T1$,
   (a) soit $b = \tau$ et $p' \, R_{bra} \, q$
   (b) soit il existe une séquence $q \xrightarrow{\tau^*} q' \xrightarrow{b} q'' \in T2^*$ telle que $p \, R_{bra} \, q'$ et $p' \, R_{bra} \, q''$
2. pour chaque transition $q \xrightarrow{b} q' \in T2$,
   (a) soit $b = \tau$ et $p \, R_{bra} \, q'$
   (b) soit il existe une séquence $p \xrightarrow{\tau^*} p' \xrightarrow{b} p'' \in T1^*$ telle que $p' \, R_{bra} \, q$ et $p'' \, R_{bra} \, q'$

L'équivalence de branchement ne distingue pas une inaction et un cycle d'actions internes. L'équivalence de branchement sensible à la divergence est introduite pour prendre en compte les cycles d'actions internes.

Équivalence de branchement sensible à la divergence (*divergence sensitive branching equivalence*) [vGW89, vGW96]

Deux états p et q sont équivalents modulo l'équivalence de branchement sensible à la divergence, (noté $p \sim_{divbra} q$) si et seulement si il existe une relation R_{divbra} telle que R_{divbra} est une relation d'équivalence de branchement, et :

1. s'il existe une séquence infinie $p \xrightarrow{\tau} p^{(1)} \xrightarrow{\tau} p^{(2)} \xrightarrow{\tau} ... \in T1^*$ avec $(p^{(i)}, q) \in R_{divbra}$ pour tout $i \geq 0$, alors il existe une séquence infinie $q \xrightarrow{\tau} q^{(1)} \xrightarrow{\tau} q^{(2)} \xrightarrow{\tau} ... \in T2^*$ telle que $(p^{(i)}, q^{(j)}) \in R_{divbra}$ pour tout $i \geq 0$ et pour tout $j \geq 0$.
2. s'il existe une séquence infinie $q \xrightarrow{\tau} q^{(1)} \xrightarrow{\tau} q^{(2)} \xrightarrow{\tau} ... \in T2^*$ avec $(q^{(i)}, p) \in R_{divbra}$ pour tout $i \geq 0$, alors il existe une séquence infinie $p \xrightarrow{\tau} p^{(1)} \xrightarrow{\tau} p^{(2)} \xrightarrow{\tau} ... \in T1^*$ telle que $(q^{(i)}, p^{(j)}) \in R_{divbra}$ pour tout $i \geq 0$ et pour tout $j \geq 0$.

Par la suite, la relation d'équivalence de branchement sensible à la divergence est aussi appelée "*divbranching*".

Équivalence observationnelle (*observational equivalence*) [Mil89]

Deux états p et q sont équivalents modulo l'équivalence observationnelle (noté $p \sim_{obs} q$) si et seulement si il existe une relation R_{obs} telle que :

1. (a) pour chaque transition $p \xrightarrow{\tau} p' \in T1$, il existe une séquence $q \xrightarrow{\tau^*} q' \in T2^*$ telle que $p' \, R_{obs} \, q'$
   (b) pour chaque transition $p \xrightarrow{a} p' \in T1$, il existe une séquence $q \xrightarrow{\tau^*.a.\tau^*} q' \in T2^*$ telle que $p' \, R_{obs} \, q'$
2. (a) pour chaque transition $q \xrightarrow{\tau} q' \in T2$, il existe une séquence $p \xrightarrow{\tau^*} p' \in T1^*$ telle que $p' \, R_{obs} \, q'$
   (b) pour chaque transition $q \xrightarrow{a} q' \in T2$, il existe une séquence $q \xrightarrow{\tau^*.a.\tau^*} q' \in T1^*$ telle que $p' \, R_{obs} \, q'$

Équivalence $\tau^*.a$ ($\tau^*.a$ *equivalence*) [FM91]

Deux états p et q sont équivalents modulo l'équivalence $\tau^*.a$ (noté $p \sim_{tau} q$) si et seulement si il existe une relation R_{tau} telle que :

1. pour chaque séquence $p \xrightarrow{\tau^*.a} p' \in T1^*$, il existe une séquence $q \xrightarrow{\tau^*.a} q' \in T2^*$ telle que $p' \, R_{tau} \, q'$
2. pour chaque séquence $q \xrightarrow{\tau^*.a} q' \in T2^*$, il existe une séquence $p \xrightarrow{\tau^*.a} p' \in T1^*$ telle que $p' \, R_{tau} \, q'$

On dit aussi que p est inclus dans q modulo le préordre $\tau^*.a$ (noté $p \sqsubseteq_{tau} q$) si et seulement si il existe R_{tau} telle que la condition 1 est satisfaite.

Équivalence de sûreté (*safety equivalence*) [BFG+91]

Deux états p et q sont équivalents modulo l'équivalence de sûreté (noté $p \sim_{saf} q$) si et seulement si :

1. $p \sqsubseteq_{tau} q$
2. $q \sqsubseteq_{tau} p$

L'équivalence de sûreté est définie en fonction du pré-ordre $\sqsubseteq_{tau}$ de l'équivalence $\tau^*.a$. C'est une *simulation d'équivalence* plutôt qu'une *bisimulation*[10], car elle requiert seulement que les états p et q soient inclus l'un dans l'autre modulo $\sqsubseteq_{tau}$, et elle ne nécessite pas que chaque $\tau^*.a$-successeur de p (respectivement de q) soit équivalent à un $\tau^*.a$-successeur de q (respectivement de p).

Équivalence de trace (*trace equivalence*)

L'équivalence de trace est aussi connue sous le nom d'équivalence de langage (*language equivalence*).

Deux états p et q sont équivalents modulo l'équivalence de trace (noté $p \sim_{tra} q$) si et seulement si :

1. pour tout $n \geq 0$ et toute séquence $p \xrightarrow{b1...bn} p' \in T1^*$, il existe une séquence $q \xrightarrow{b1...bn} q' \in T2^*$
2. pour tout $n \geq 0$ et toute séquence $q \xrightarrow{b1...bn} q' \in T2^*$, il existe une séquence $p \xrightarrow{b1...bn} p' \in T1^*$

Équivalence faible (*weak trace equivalence*) [BHR84]

Deux états p et q sont équivalents modulo l'équivalence faible (noté $p \sim_{wtr} q$) si et seulement si :

1. pour tout $n \geq 0$ et toute séquence $p \xrightarrow{\tau^*.a1...\tau^*.an} p' \in T1^*$, il existe une séquence $q \xrightarrow{\tau^*.a1...\tau^*.an} q' \in T2^*$
2. pour tout $n \geq 0$ et toute séquence $q \xrightarrow{\tau^*.a1...\tau^*.an} q' \in T2^*$, il existe une séquence $p \xrightarrow{\tau^*.a1...\tau^*.an} p' \in T1^*$

10. L'équivalence $\tau^*.a$ est, par exemple, une bisimulation.

Classement des relations d'équivalence selon leur force

Une relation d'équivalence $R1$ est considérée plus *forte* qu'une relation d'équivalence $R2$ si et seulement si $p\,R1\,q$ implique $p\,R2\,q$ pour n'importe quels états p et q. La figure 1.1 illustre le classement, selon leur force, des relations d'équivalence que nous venons de décrire.

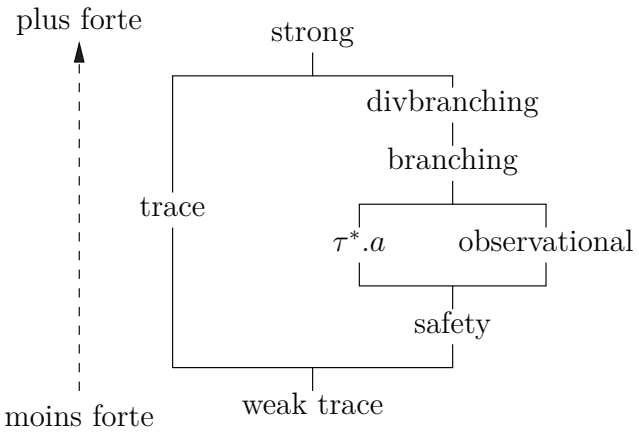


FIGURE 1.1 – Classement des relations d'équivalence selon leur force

1.4 Protocoles de synchronisation pour le rendez-vous multiple

En LNT, la primitive d'interaction entre processus est le rendez-vous multiple, ce qui n'est pas le cas des langages mentionnés dans la section précédente, à l'exception de BIP. Une implémentation distribuée d'un programme LNT nécessite un protocole pour assurer les rendez-vous entre processus. Chaque processus pouvant s'exécuter sur une machine distincte, on cherche un protocole qui repose sur du passage de messages entre processus. Plus précisément, on estime que le mécanisme d'interaction disponible pour l'implémentation est le passage de messages asynchrones, sans perte, avec ordre préservé entre deux processus. On fait également l'hypothèse que les machines ne tombent pas en panne.

Dans cette section, on explore plus en détails les difficultés de l'implémentation distribuée du rendez-vous multiple, avant de présenter différentes solutions existantes.

1.4.1 Le rendez-vous multiple entre processus non déterministes

Le rendez-vous multiple est une action synchronisée entre plusieurs processus. En LNT, les rendez-vous ont lieu sur des portes. On qualifie les processus LNT de non déterministes car ils peuvent être prêts sur plusieurs actions à la fois. Une action sur une porte concerne un

vecteur de synchronisation, c'est-à-dire un ensemble non vide d'identifiants de processus. Lorsque l'intersection des vecteurs de synchronisation de deux rendez-vous n'est pas vide, ces rendez-vous sont alors en conflit. Ce genre de synchronisation est plus complexe que des barrières de synchronisation classiques, car il faut assurer l'exclusion mutuelle des rendez-vous en conflit.

En 1988, Chandy et Misra [CM88] ont résumé ce type de synchronisation par le problème de coordination des comités, en anglais *committee coordination problem*, défini de la manière suivante. On considère un ensemble de professeurs qui se réunissent dans des comités. Chaque professeur fait partie d'un ou plusieurs comités, et chaque comité comprend au moins un professeur. Une réunion de comité peut avoir lieu uniquement si tous les professeurs du comité participent, et un professeur ne peut participer qu'à un seul comité à la fois.

Ce problème est similaire à la gestion des rendez-vous, les professeurs correspondant aux processus LNT et les comités aux rendez-vous. On relève tout de même une légère différence : l'ensemble de rendez-vous auquel un processus LNT peut prendre part varie au fur et à mesure que ce processus s'exécute, alors qu'un professeur peut toujours participer à n'importe lequel des comités dont il est membre. En d'autres termes, pour un système formé d'un composition parallèle de processus LNT, chaque état du système nécessite de résoudre une instance du problème de coordination des comités.

Un protocole qui organise les rendez-vous doit garantir les propriétés suivantes :

exclusion mutuelle des rendez-vous en conflit : un processus ne peut participer qu'à un seul rendez-vous à la fois, deux rendez-vous en conflit ne peuvent donc pas avoir lieu en même temps.

absence d'interblocage : le système ne doit pas bloquer indéfiniment si l´état des processus permet au moins un rendez-vous dans le système. Cela concerne les interblocages où tous les processus sont bloqués (*deadlock*) ainsi que ceux où les processus négocient entre eux indéfiniment sans parvenir à réaliser un rendez-vous (*livelock*).

De manière générale, les protocoles comportent 2 phases principales :

1. détection de rendez-vous possibles
2. négociation afin de confirmer la réalisation d'un rendez-vous tout en garantissant l'exclusion mutuelle des autres rendez-vous en conflit

Un premier protocole naïf consiste en un unique processus auxiliaire qui joue le rôle de synchroniseur central. Quand un processus LNT est prêt sur un ou plusieurs rendez-vous, il avertit le synchroniseur central. Celui ci peut décider quel rendez-vous a lieu, et l'indiquer aux processus concernés. Cette solution n'est pas satisfaisante car elle force une exécution séquentielle de tous les rendez-vous du système, alors que les rendez-vous qui ne sont pas en conflit peuvent avoir lieu de manière concurrente. De plus, la structure centralisée est un goulot d'étranglement évident pour les systèmes avec de nombreux processus LNT. De nombreux travaux ont donc été menés pour résoudre le problème de la coordination des comités de professeurs de manière distribuée.

1.4.2 Acquisition de ressources partagées sans interblocage

Avant d'explorer les différents protocoles, nous rappellons brièvement une technique d'acquisition de ressources partagées qui évite les interblocages.

Dans les systèmes concurrents, il arrive souvent que plusieurs processus doivent acquérir des ressources partagées, ce qui peut facilement mener à des situations d'interblocage. Par exemple, on considère deux processus P1 et P2 qui cherchent tous les deux à acquérir deux ressources R1 et R2. D'abord P1 verrouille R1 et P2 verrouille R2, ensuite P1 cherche à verrouiller R2 et P2 cherche à verrouiller R1 : c'est un interblocage, car chaque processus veut acquérir une ressource déjà verrouillée par l'autre.

Il existe une technique simple pour éviter ces interblocages : si tous les processus verrouillent les ressources *dans le même ordre*, alors aucun interblocage ne peut avoir lieu. Cela nécessite de pouvoir ordonner les ressources, et que tous les processus respectent l'ordre de verrouillage. Dans notre exemple, on impose de verrouiller d'abord R1, puis R2. Le premier de P1 ou P2 qui arrive à verrouiller R1 pourra ensuite verrouiller R2, tandis que l'autre attendra que le verrou sur R1 soit relâché avant de s'intéresser à R1, puis à R2.

Cette technique est utilisée dans plusieurs protocoles pour le rendez-vous multiple. Elle trouverait son origine dans la solution proposée par Dijkstra pour le problème des philosophes, et elle est notamment formalisée en 1968 par Havender [Hav68].

1.4.3 Protocoles distribués pour le rendez-vous multiple

L'implémentation distribuée du rendez-vous multiple a été résolue dans différents domaines qui ont chacun leur terminologie, ce qui ne facilite pas la recherche bibliographique. Nous présentons les travaux existants dans un ordre chronologique, en essayant d'indiquer leur relation le cas échéant.

Dès 1983, des études sur l'implémentation distribuée de réseaux de Petri donnent naissance à des propositions [Win83, Tau87]. Les protocoles définis ont pour but d'assurer la progression de jetons entre les places du réseau. Chaque transition du réseau est équivalente à un rendez-vous : elle ne peut avoir lieu que si il y a des jetons dans les places qui la précède, et elle peut être en concurrence avec d'autres transitions qui nécessitent les mêmes jetons. Les protocoles envisagés reposent sur le verrouillage des jetons des places par les transitions pour assurer l'exclusion mutuelle des transitions en conflit. Pour éviter les interblocages lors du verrouillage, plusieurs stratégies sont envisagées, notamment l'élection d'une transition gagnante parmi celles qui verrouillent les mêmes jetons [Win83], ou le verrouillage dans un ordre fixe des places [Tau87], ce qui revient à la technique expliquée à la section précédente. Certains des protocoles passés en revue par Taubner [Tau87] sont pertinents pour l'implémentation du rendez-vous multiple, mais ils ne semblent pas être connus de la communauté des algèbres de processus, car nous n'avons pas vu de référence à ces travaux dans les autres articles présentés dans la suite.

En 1988, dans leur ouvrage "*Parallel Programming Design : a Foundation*"[CM88], Chandy et Misra traitent le rendez-vous multiple dans la continuité de celui du dîner des philosophes. Ils proposent une solution distribuée qui utilise des échanges de jetons, et qui se ramène au problème des philosophes pour assurer l'exclusion mutuelle. Chaque rendez-vous est représenté par un processus auxiliaire, et une fourchette est introduite pour chaque paire de rendez-vous en conflit. Ces rendez-vous sont alors comme des philosophes voisins de table. Un processus envoie un jeton à tous les rendez-vous sur lesquels il est prêt. Un rendez-vous qui a reçu suffisamment de jetons demande les fourchettes de tous ses voisins, afin d'assurer l'exclusion mutuelle des rendez-vous en conflit. Une fois qu'un rendez-vous a obtenu les fourchettes de ses voisins, il doit encore vérifier qu'aucun des processus concernés n'a déjà participé à un autre rendez-vous entre temps : ceci est réalisé par un échange d'autres jetons avec les rendez-vous voisins. Les solutions connues au problème des philosophes permettent d'assurer qu'un rendez-vous qui demande les fourchettes de ses voisins finira toujours par les recevoir.

L'année suivante, en 1989, Bagrodia [Bag89] décrit deux autres protocoles. Le premier, nommé "*Event Manager*", utilise des processus auxiliaires managers pour un ou plusieurs rendez-vous. Un unique jeton circule sur un anneau virtuel entre les managers, et ce jeton contient pour chaque processus un compteur du nombre de fois où ce processus a participé à un rendez-vous. Quand un processus est prêt sur un ou plusieurs rendez-vous, il prévient les managers correspondants, qui maintiennent chacun un compteur du nombre de fois où ils ont été prévenus. Quand un manager a détecté suffisamment de processus prêts, il attend le jeton puis compare les compteurs pour déterminer si les processus concernés ont déjà participé à un autre rendez-vous. Si ce n'est pas le cas, alors le manager réalise le rendez-vous, prévient les processus concernés, incrémente les compteurs correspondants du jeton, avant de transférer le jeton au prochain manager. Le jeton unique garantit l'exclusion mutuelle des rendez-vous, mais il empêche l'exécution en parallèle des rendez-vous qui ne sont pas en conflit. Pour remédier à ce problème, le deuxième protocole baptisé "*Modified Event Manager*"(MEM), abandonne l'idée du jeton unique pour garantir l'exclusion mutuelle et utilise à la place l'approche des philosophes proposée par Chandy et Misra. Le système de compteur est toujours utilisé, mais les compteurs de nombre d'actions réalisées par les processus sont cette fois stockés sur les fourchettes.

En 1990, Kumar [Kum90] montre qu'un protocole proposé par Bagrodia en 1985 [Bag85] n'est pas à l'abri de livelock, et présente une version corrigée. Dans ce protocole, chaque rendez-vous est modélisé par un jeton et un anneau virtuel entre les processus. Le protocole fonctionne en deux temps : d'abord on cherche à établir la faisabilité d'un rendez-vous, ensuite on confirme ce rendez-vous aux processus participants. Un processus qui est prêt sur un rendez-vous dont il a le jeton envoie ce jeton marqué de son identifiant de processus sur l'anneau virtuel. Un processus de l'anneau virtuel du rendez-vous transfère le jeton s'il est aussi prêt à participer au rendez-vous ; sinon, il annule le rendez-vous, prévient les processus précédents de l'échec du rendez-vous et garde le jeton. Si le jeton parvient à faire un tour de l'anneau virtuel du rendez-vous, alors une confirmation d'interaction est envoyée à tous les processus de l'anneau. Le point délicat de ce protocole est de savoir

comment un processus traite l'arrivée d'un jeton pour un rendez-vous valide alors qu'il est en attente d'une réponse pour un autre rendez-vous dont il a déjà transféré le jeton. Il ne peut pas transférer plus d'un jeton à la fois, car cela pourrait mener à la réalisation de rendez-vous en conflit sur ce processus. Il ne peut pas simplement retarder le traitement en attendant la réponse du jeton déjà transmis car cela peut facilement créer des situations d'interblocages. Kumar choisit de retarder les réponses, tout en imposant un ordre sur les processus pour éviter les interblocages. Les processus qui sont prêts sur un rendez-vous avertissent les processus les plus petits selon l'ordre, afin qu'une chaîne de verrouillage commence.

Des travaux sur l'implémentation distribuée de LOTOS ont donné naissance à plusieurs autres propositions. En 1989, Von Bochmann [BGW89] propose un protocole dont la structure est proche de celui de Kumar. A chaque rendez-vous correspond un anneau virtuel entre les processus. Un processus prêt sur un rendez-vous envoie sur l'anneau correspondant un message où il indique son identifiant, qui sert d'index de priorité. Un processus relaie le message si son identifiant est plus petit que celui du message, et s'il est effectivement prêt sur le rendez-vous (sinon, il annule la négociation). Ainsi, seuls les messages les plus prioritaires peuvent effectuer un tour complet d'anneau virtuel. Quand un processus relaie un message, il ajoute au message une information indiquant s'il a déjà relayé un autre message pour un autre rendez-vous. Quand un message parvient à effectuer un tour d'anneau, le processus peut ainsi observer quels sont les rendez-vous potentiellement en conflit. Il entame alors une autre négociation avec les rendez-vous en conflit pour assurer leur exclusion mutuelle, et selon l'issue de cette négociation, avertit les processus concernés du succès ou de l'échec du rendez-vous.

En 1990, Sjödin [Sjö91] termine sa thèse intitulée "*From LOTOS Specifications to Distributed Implementations*", dans laquelle il propose un protocole où chaque rendez-vous est représenté par un processus auxiliaire, et où chaque processus LOTOS est identifié par un entier, ce qui permet de les ordonner. Un processus LOTOS prévient tous les rendez-vous sur lesquels il est prêt. Quand un rendez-vous a détecté suffisamment de processus prêts, il va chercher à les verrouiller dans l'ordre. Un processus LOTOS accepte un verrou uniquement s'il n'est pas déjà verrouillé, sinon il attend la réponse du verrou courant. Si un rendez-vous parvient à verrouiller tous les processus, alors il les avertit de la réalisation du rendez-vous. Les processus concernés qui avaient des verrous d'autres rendez-vous en attente préviennent ces autres rendez-vous de l'échec de leur tentative. Ceux-ci déverrouillent alors tous les processus qu'ils avaient réussi à verrouiller.

Six ans plus tard, Sjödin accompagné de Parrow revoit sa copie en proposant une légère modification de son protocole qui permet de diminuer le nombre de messages nécessaires à la réalisation d'un rendez-vous. Dans la version de 1990, lors de la phase de verrouillage des processus, le rendez-vous envoie un message à chaque processus et attend sa réponse. Dans la nouvelle version, le rendez-vous amorce la négociation en demandant le verrou au premier processus, et ensuite cette demande de verrou est transmise directement entre les processus à verrouiller, dans l'ordre de verrouillage.

En 1994, Joung et Smolka [JS94] proposent un protocole pour des interactions multiples dites du premier ordre, qui sont comparables aux rendez-vous multiples de LNT. Dans ce protocole, un coordinateur est associé à chaque processus et est en charge de décider si une interaction peut avoir lieu, en verrouillant les processus nécessaires à l'interaction. Pour départager des interactions en conflit, ce protocole associe une horloge logique [Lam78] à chaque couple processus-coordinateur, ce qui permet de définir un âge à chacun de ces couples. Les coordinateurs les plus vieux sont prioritaires dans la résolution de conflit : un processus peut révoquer son verrou courant en faveur d'une demande de verrou émanent d'un coordinateur plus ancien. Ces possibles phases d'annulation de verrou peuvent engendrer de nombreux échanges de messages dans les pire cas, où le protocole présente une complexité quadratique.

En 2004, Perez, Corchuelo et Toro proposent le protocole α-core [PCT04]. L'approche générale du protocole est similaire à la proposition de 1990 de Sjödin. La différence remarquable est qu'un rendez-vous ne prend pas la peine de verrouiller un processus qui est prêt uniquement sur ce rendez-vous. On peut considérer que le processus se verrouille de lui-même sur le rendez-vous, nous avons choisi de nommer cette technique l'*auto-verrouillage*. En effet, il n'existe pas de risque de conflit de rendez-vous pour les processus qui sont prêts uniquement sur un seul rendez-vous, il n'y a donc pas de nécessité de s'assurer d'une exclusion mutuelle par un verrou.

Il existe deux autres protocoles qui ont été étudiés dans le cadre de l'implémentation distribuée de LOTOS, mais qui ne reposent pas sur du passage de messages asynchrones. En 1991, Sisto, Ciminiera et Valenzano [SCV91] proposent un protocole où les rendez-vous sont structurés selon l'arborescence de la composition parallèle LOTOS. Ce protocole utilise des canaux de communication unidirectionnels, qui assurent que si un message est envoyé de A vers B, alors il n'y a pas de message en transit de B vers A tant que le premier message n'est pas arrivé à B. En 2001, Yasumoto, Higashino et Taniguchi [YHT01] décrivent une technique basée sur une primitive de diffusion de message ordonné (*ordered broadcast*) disponible entre tous les programmes de l'implémentation distribuée. Les processus étant assurés de tous recevoir les messages dans le même ordre, ils peuvent tous décider du rendez-vous à effectuer en sachant que tous les processus concernés feront le même choix.

Chapitre 2

Protocole de synchronisation

> Simplify, simplify.
>
> Henry D. Thoreau
> *Walden*

Le protocole de synchronisation est une partie clé de l'implémentation distribuée : il doit assurer la gestion des rendez-vous entre les processus de manière correcte et efficace. Parmi les protocoles de la littérature que nous avons étudiés, celui proposé par Parrow et Sjödin [PS96] a retenu notre attention car il utilise peu de messages. Nous nous sommes donc basés sur ce protocole pour développer celui utilisé au sein de DLC.

Dans ce chapitre, nous commençons par introduire le protocole de Parrow et Sjödin, avant de détailler le phénomène de synchronisation "en avance de phase" que ce protocole autorise. Nous présentons ensuite différentes améliorations que nous avons apportées à ce protocole pour le généraliser. Nous exposons l'optimisation d'auto-verrouillage, inspirée du protocole α-core [PCT04], ainsi que le système de purge que nous avons mis au point pour permettre la cohabitation de l'auto-verrouillage avec les synchronisations en avance de phase. Nous comparons le coût de notre protocole avec celui de Parrow et Sjödin et α-core. Enfin, nous présentons la spécification formelle de notre protocole.

2.1 Version de Parrow et Sjödin

Le protocole de Parrow et Sjödin nécessite des processus auxiliaires en plus des processus à synchroniser. Pour simplifier le vocabulaire, nous appelons désormais les processus de la spécification de départ des *tâches*. Le but du protocole est donc de synchroniser les actions entre les tâches en respectant les vecteurs de synchronisation de chaque porte, définis par la composition parallèle comme nous l'avons vu à la section 1.3.2.

Le protocole de Parrow et Sjödin est défini pour des systèmes où chaque porte a un unique vecteur de synchronisation, et où le traitement des échanges de données n'est pas pris en compte. Le protocole est constitué, en plus des processus tâches, d'un processus auxiliaire "porte" pour chaque porte du système et d'un processus auxiliaire "manager" pour chaque tâche. La figure 2.1 illustre la structure du protocole.

Pour donner une vue d'ensemble, on peut dire que chaque porte est en charge d'assurer les rendez-vous qui la concernent, tandis que chaque manager est responsable de conduire les négociations pour sa tâche. La technique retenue pour assurer l'exclusion mutuelle entre portes en conflit est le verrouillage ordonné des tâches, représentées ici par leur manager. Les vecteurs de synchronisation sont donc des ensembles *ordonnés* d'identifiants de tâches.

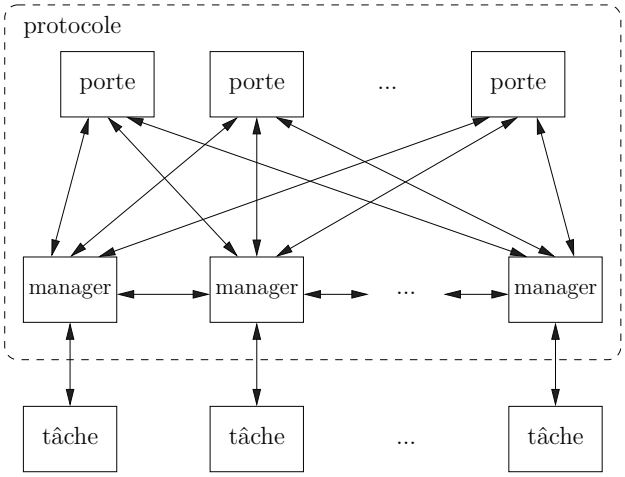


FIGURE 2.1 – Structure du protocole de Parrow et Sjödin qui emploie, en plus des tâches, des processus auxiliaires managers et portes.

Le déroulement du protocole peut être divisé en 3 phases principales, qui mettent en jeu 8 types de messages, résumés dans la table 2.1 :

Phase d'annonce. Une tâche annonce à son manager les actions qu'elle peut effectuer par un message "request". Le manager transmet alors cette information aux portes concernées par un message "ready".

Phase de verrouillage. Lorsqu'une porte a reçu un message "ready" de la part de toutes les tâches de son vecteur de synchronisation, elle sait qu'un rendez-vous est possible. La porte peut alors amorcer une *négociation* en envoyant une demande de verrou qui va suivre une *chaîne de verrouillage*, afin d'assurer que les tâches ne participent pas à un autre rendez-vous en conflit. Plus précisément, la porte envoie un message "query" au premier manager du vecteur de synchronisation. Les managers transmettent ensuite la demande de verrou le long de la chaîne de verrouillage par des messages "lock". Les managers sont globalement ordonnés par l'identifiant de leur tâche associée, et les messages "lock" sont transmis en suivant cet ordre entre

Phase	Nom	Direction	Description
Ann.	request	T → M	annonce des actions qu'une tâche peut effectuer
Ann.	ready	M → P	annonce : la tâche du manager M est prête sur la porte P
Verr.	query	P → M	amorce d'une chaîne de verrouillage
Verr.	lock	M → M	négociation : transmission de la demande de verrou
Res.	commit	M → M	transmission d'une chaîne de commit
Res.	abort	M → M	transmission d'une chaîne d'annulation
Res.	yes	M → P	annonce du succès d'une négociation
Res.	no	M → P	annonce de l'échec d'une négociation
Res.	confirm	M → T	confirme à la tâche quelle action elle doit effectuer

TABLE 2.1 – Les différents messages utilisés dans le protocole de Parrow et Sjödin. Dans la colonne direction, T indique une tâche, M un manager et P une porte.

tous les managers des tâches du vecteur de synchronisation de la porte. Un manager accepte une seule demande de verrou à la fois, il ne s'engage pas dans plusieurs négociations simultanément.

Phase de résultat. Si le dernier manager d'une chaîne de verrouillage accepte le verrou, alors il annonce le succès de la négociation à la porte qui a lancé la chaîne de verrouillage par un message "yes". De plus, il lance un message "commit" qui va être relayé par les managers en remontant la chaîne de verrouillage, ce qui forme une *chaîne de commit*. Tous les managers d'une négociation qui a aboutit préviennent leur tâche de l'action décidée avec un message "confirm".

Par contre, si un des managers brise une chaîne de verrouillage en refusant le verrou, alors il annonce l'échec de la négociation à la porte concernée par un message "no". Dans ce cas, il envoie aussi un message "abort" qui va être relayé par les managers verrouillés jusqu'à présent sur la chaîne de verrouillage, ce qui forme une *chaîne d'annulation*. Un manager qui reçoit un message "abort" se déverrouille de la négociation dans laquelle il s'était engagé, et peut accepter une autre demande de verrou par la suite.

Un des points clés du protocole est la gestion des verrous par les managers. Lorsqu'un manager reçoit un verrou, par un message "query" ou "lock", il regarde d'abord si ce verrou concerne une porte sur laquelle sa tâche est prête. Si sa tâche n'est pas prête sur cette porte, alors le manager refuse le verrou. Si sa tâche est prête sur cette porte, son comportement dépend de son état de verrouillage :

— S'il n'est pas déjà verrouillé pour une autre négociation, alors il accepte le verrou et continue la chaîne de verrouillage, ou bien, s'il est le dernier manager de la chaîne, il envoie "yes" à la porte et lance la chaîne de commit.

— S'il est déjà verrouillé par une autre négociation n, alors il place en attente les demandes de verrou jusqu'à recevoir le résultat de n. S'il reçoit un message "commit", alors n est un succès : il transmet la chaîne de commit avant de refuser tous les verrous en attente. S'il reçoit un message "abort", alors n est un échec : il transmet

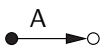
la chaîne d'annulation avant de traiter une demande de verrou en attente, s'il y en a.

Exemple de déroulement du protocole

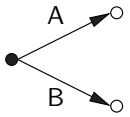
Nous utilisons des exemples pour illustrer différents comportement du protocole au cours de ce chapitre. Afin de garder les exemples court, les processus tâches terminent généralement sur un "stop" pour d'éviter d'avoir à traiter les synchronisations sur la porte "exit".

Le comportement du protocole de Parrow et Sjödin est illustré sur le système suivant :

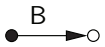
```
process T1 [A: none] is
   A;
   stop
end process
```



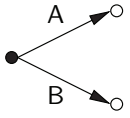

```
process T2 [A, B: none] is
   select
      A
   [] B
   end select;
   stop
end process
```




```
process T3 [B: none] is
   B;
   stop
end process
```




```
par
   A    -> T1 [A]
|| A, B -> T2 [A,B]
||    B -> T3 [B]
end par
```



Ce système comprend trois tâches T1, T2 et T3, ainsi que deux portes A et B. Les tâches T1 et T3 peuvent effectuer une action sur respectivement A et B, tandis que la tâche T2 peut effectuer une action sur n'importe laquelle des deux portes. Le vecteur de synchronisation de la porte A est {T1,T2} et celui de la porte B est {T2,T3}. Les portes A et B sont donc en conflit car la tâche T2 est présente dans le vecteur de synchronisation de ces deux portes. Globalement, le système effectue un choix non déterministe entre un rendez-vous des tâches T1 et T2 sur la porte A et un rendez-vous des tâches T2 et T3 sur la porte B.

La figure 2.2 illustre un échange de messages possible selon le protocole. Une fois passée la phase d'annonce, les portes A et B détectent toutes les deux un rendez-vous possible

et lancent donc une chaîne de verrouillage. Ces demandes de verrou sont transmises dans l'ordre entre les managers. La porte B est la première à verrouiller le manager M2, qui est aussi nécessaire à la porte A pour valider le rendez-vous. Quand le manager M2 reçoit "lock(A)" de la part du manager M1, le manager M2 place cette demande de verrou en attente. Lorsque le manager M3 accepte le verrou pour la négociation concernant l'action sur la porte B, c'est le succès de cette négociation. Le manager M2 reçoit ensuite la confirmation que la négociation sur la porte B est un succès, via le message "commit" du manager M3. Le manager M2 peut alors annoncer l'échec de la négociation sur la porte A, en envoyant "no" à la porte A et "abort" au manager M1. Au final, les tâches T2 et T3 reçoivent la confirmation d'une action sur la porte B, tandis que la tâche T1 ne pourra pas effectuer son action sur la porte A.

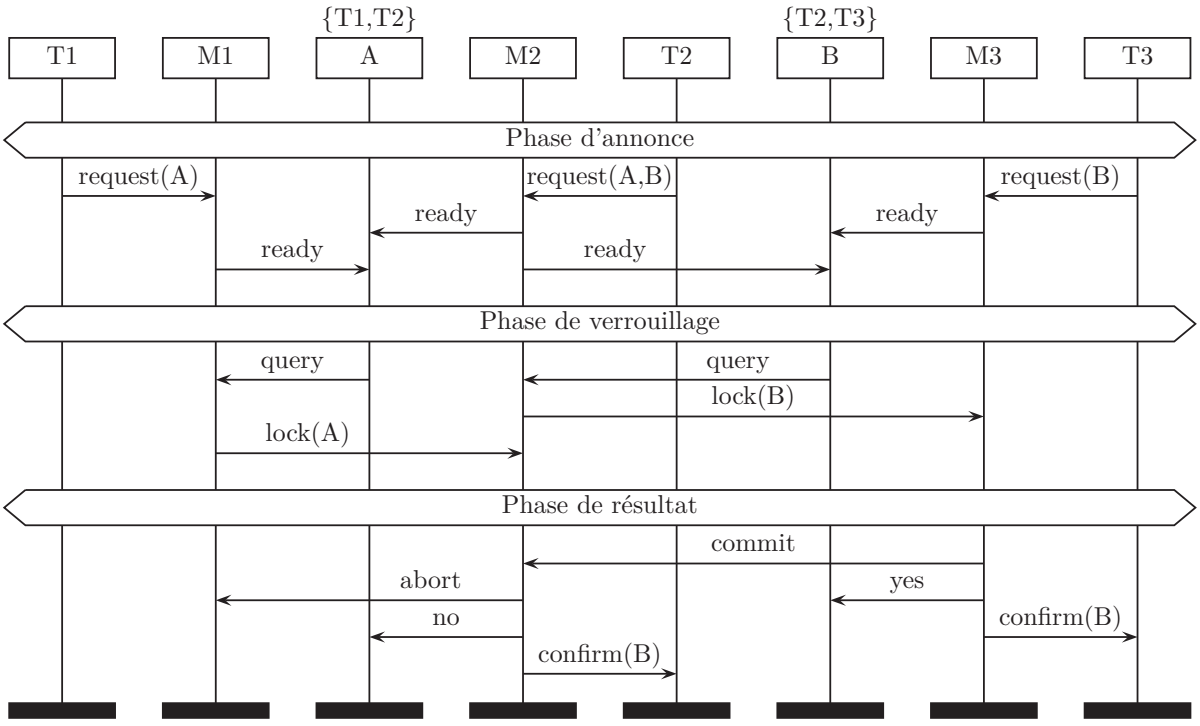


FIGURE 2.2 – Exemple de déroulement du protocole de Parrow et Sjödin. Au dessus d'une porte, les vecteurs de synchronisation de cette porte sont rappelés.

On note que le protocole permet aussi que l'action sur la porte A se réalise au détriment de celle sur la porte B, il suffit pour cela que le manager M2 reçoive le message "lock(A)" du manager M1 avant le message "query" de la porte B. En d'autres termes, il faut que la chaîne de verrouillage de la porte A atteigne le manager M2 avant celle de la porte B. En appliquant le verrouillage ordonné des managers, le protocole assure que seule l'une de ces deux chaînes de verrouillage aboutit ; il préserve ainsi l'exclusion mutuelle des rendez-vous

en conflit sans créer d'interblocage.

On remarque aussi que la réalisation d'une action est décidée lorsque la dernière tâche du vecteur de synchronisation accepte le verrou. Ainsi, ce n'est pas le processus porte qui est le premier au courant de la réalisation d'une action.

Nombres de messages requis

Pour un rendez-vous entre n tâches, une négociation nécessite $5n$ messages :
- la phase d'annonce consomme $2n$ messages (1 "request" et 1 "ready" par tâche),
- la chaîne de verrouillage en consomme n (1 "query" puis $n-1$ "lock"),
- la chaîne de commit consomme $n-1$ "commit" entre les tâches,
- et enfin la phase de résultat consomme 1 "yes" pour prévenir la porte et n "confirm" pour avertir les tâches.

Si on ignore les messages entre une tâche et son manager, qui ont tout intérêt à être localisés sur la même machine, le protocole ne requiert alors plus que $3n$ messages.

On remarque que le protocole économise des messages en propageant une chaîne de verrouillage directement entre les managers. En effet, la plupart des protocoles qui ont un processus auxiliaire pour chaque porte et qui adoptent une approche de verrouillage ordonné, tel que α-core [PCT04] par exemple, font transiter toutes les requêtes de verrouillage par le processus porte. La figure 2.3 permet de comparer ces deux approches de verrouillage. Avec l'approche "α-core" (à gauche), la porte doit attendre la réponse d'une tâche avant d'envoyer un verrou à la suivante, la phase de verrouillage consomme donc 2 messages par tâche. Dans le protocole de Parrow et Sjödin (à droite), la porte amorce une chaîne de verrouillage qui se propage au sein des managers, la phase de verrouillage consomme donc 1 message par tâche. On peut aussi considérer le message "yes" qui permet d'atteindre le même état que dans l'approche α-core, à savoir la porte est avertie du succès du verrouillage, pour atteindre un total de $n+1$ messages pour n tâches.

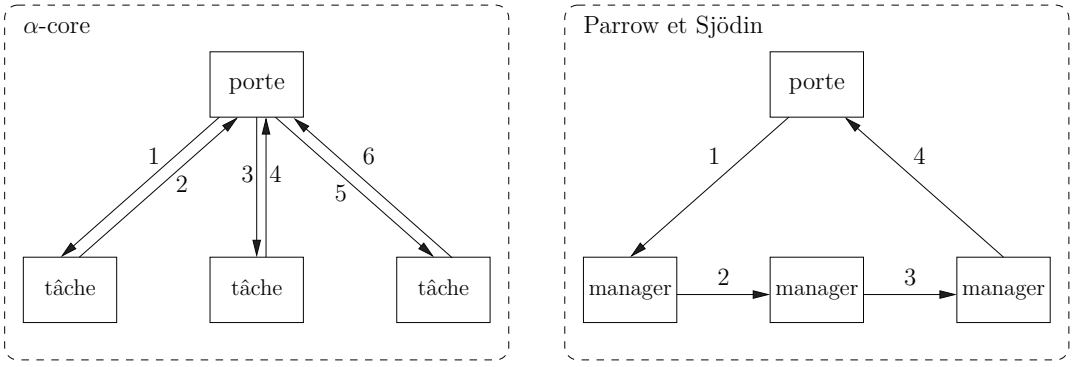


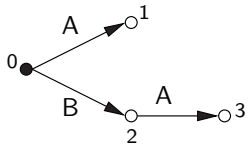
FIGURE 2.3 – Deux approches de verrouillage qui ne nécessitent pas le même nombre de messages.

2.2 Rendez-vous en avance de phase

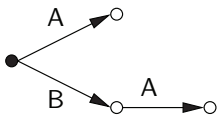
Une des caractéristiques du protocole de Parrow et Sjödin est de pouvoir réaliser des rendez-vous à partir d'informations qui ne sont plus à jour. Nous avons nommé ce phénomène un rendez-vous (ou une synchronisation) "en avance de phase". Nous illustrons ce phénomène sur le système suivant :

```
process T1 [A: none] is
   A;
   stop
end process
```

```
process T2 [A, B: none] is
    select
        A
    []  B ; A
    end select;
    stop
end process
```




```
par A in
    T1 [A]
||  T2 [A,B]
end par
```



Ce système comporte deux tâches T1 et T2 qui peuvent réaliser des actions sur les portes A et B. Pour l'espace d'états de la tâche T2, un index est associé à chaque état pour faciliter la discussion. La composition parallèle du système indique que les tâches T1 et T2 doivent synchroniser leurs actions sur la porte A, mais pas sur la porte B. On note donc que la tâche T2 peut effectuer une action sur la porte B sans avoir à se synchroniser avec une autre tâche.

La figure 2.4 illustre un déroulement possible du protocole pour ce système. Au début, une phase d'annonce avertit les portes que les tâches sont prêtes. La porte A lance alors une négociation n_0 qui commence par le manager M1 (la négociation n_0 est identifiée en gras). La porte B amorce une négociation sur le manager M2, qui accepte le verrou et conclut la négociation. La tâche T2 effectue donc une action sur la porte B et passe dans son état 2, où elle est prête sur la porte A. Une phase d'annonce a lieu pour avertir la porte A. La négociation n_0 atteint maintenant le manager M2 : comme la tâche T2 est prête sur la porte A, son manager accepte le verrou. Lorsque la tâche T2 est dans l'état 2, le manager M2 conclut donc la négociation n_0 qui a été débutée quand la tâche T2 était dans l'état 0 : c'est un rendez-vous en avance de phase pour la tâche T2.

Les rendez-vous en avance de phase sont possibles car les managers ne préviennent pas toutes les portes quand ils ont conclu une négociation. Dans le protocole α-core par

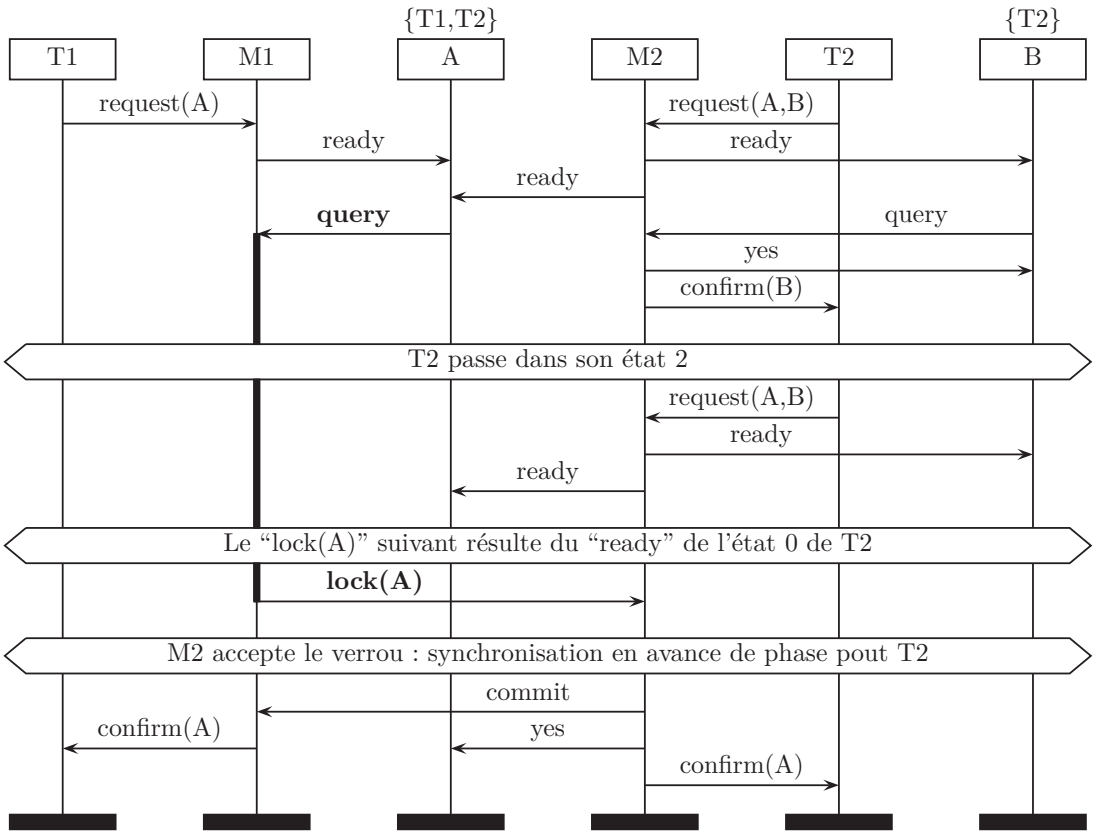


FIGURE 2.4 – Exemple de synchronisation en avance de phase, identifiée en gras.

exemple, lorsqu'une tâche effectue un rendez-vous sur une porte, elle prévient toutes les autres portes sur lesquelles elle était prête qu'elle ne l'est plus, et attend une confirmation de ces portes avant d'effectivement réaliser l'action et de passer dans l'état suivant. Selon le protocole de Parrow et Sjödin, lorsqu'un manager conclut une négociation, il prévient uniquement les portes dont les chaînes de verrouillage l'ont atteint, en indiquant qu'ils refusent le verrou par un message “no”. Cela permet au manager d'accepter des chaînes de verrouillage qui ont pu être lancées à partir d'un “ready” relatif à un état précédent de la tâche. On remarque que les synchronisations en avance de phase sont des rendez-vous valides, étant donné que toutes les tâches qui participent sont effectivement prêtes à réaliser une action sur la porte concernée.

Nous prenons ici le temps de détailler le phénomène de rendez-vous en avance de phase car les auteurs du protocole ne l'évoquent pas. Les synchronisations en avance de phase sont pourtant relativement courantes en pratique. En effet, une tâche a des chances de réaliser un rendez-vous en avance de phase lorsqu'elle est souvent prête sur les mêmes portes. Or, les systèmes distribués comportent souvent des serveurs qui réagissent à des requêtes, ce

qui se traduit typiquement par une boucle infinie sur un choix non déterministe d'actions sur différentes portes, dont chacune correspond à un type de requête. Ce comportement de boucle infinie réactive sur plusieurs types de messages est par exemple à la base du patron de serveur générique de la bibliothèque Erlang OTP pour systèmes distribués [1].

2.3 Généralisation du protocole

Nous avons apporté plusieurs modifications au protocole afin de généraliser sa portée. Dans cette section, nous commençons par décrire trois simplifications avant de présenter l'extension du protocole aux communications asynchrones. Nous exposons ensuite la gestion de plusieurs vecteurs de synchronisation par porte, et on explique enfin comment la présence d'actions internes et d'offres impacte le protocole.

2.3.1 Simplifications

Nous présentons ici trois simplifications que nous avons apportées au protocole.

Fusion des managers avec leur tâche

La première simplification consiste à réunir chaque manager avec sa tâche respective. La séparation d'une tâche et de son manager permet d'isoler clairement le comportement du protocole de celui de la tâche. Cependant, le comportement du manager peut être inclus au sein de la tâche, ce qui évite des communications inter-processus. Une partie de la logique du protocole se retrouve donc attachée à la tâche, qui participe activement aux négociations. Nous utilisons désormais les mots "tâche" ou "manager" pour désigner l'union de ces deux entités.

Réduction du nombre de types de messages

Nous avons réduit le nombre de types de messages, pour le ramener à quatre. La fusion des managers avec leur tâche élimine déjà les messages de type "request" et "confirm". Ensuite, le type de message "query" est abandonné pour utiliser à la place des messages de type "lock". Un message "query" diffère d'un message "lock" par le fait que le message "query" n'a pas besoin de transmettre la porte qui demande le verrou : en effet, le manager peut déduire cette information de la porte qui lui envoie le message. En pratique, transmettre un message "lock" avec un identifiant de porte requiert seulement quelques octets de plus qu'un message "query", ce qui n'impacte pas le temps de transmission du message.

1. `http://www.erlang.org/doc/man/gen_server.html`

Nous avons aussi éliminé les types de messages “yes” et “no”, qui sont respectivement remplacés par des messages de type “commit” et “abort”. La seule différence entre “commit” et “yes”, respectivement “abort” et “no”, est le destinataire : c'est soit un manager pour les messages “commit” ou “abort”, soit une porte pour les messages “yes” ou “no”. Le rôle principal de ces messages est d'informer du *résultat* (succès ou échec) d'une négociation, il semble superflu de différencier les messages selon leur destinataire.

Au final, notre protocole utilise donc quatre types de messages :
- “ready” pour annoncer qu'une tâche est prête
- “lock” pour demander de verrouiller une tâche
- “commit” et “abort” pour indiquer respectivement le succès ou l'échec d'une négociation

Diffusion en parallèle des messages de résultat

Enfin, nous avons choisi de diffuser les messages de résultat “commit” ou “abort” directement à tous les destinataires concernés. Cette diffusion en parallèle améliore les performances par rapport aux chaînes de confirmation ou d'annulation. En effet, ces chaînes imposent un ordre dans le passage des messages “commit” ou “abort”, qui remontent séquentiellement une chaîne de verrouillage, tâche après tâche. Cependant, la tâche qui détecte le succès ou l'échec d'une négociation peut directement prévenir, en plus de la porte, toutes les tâches déjà verrouillées le long de la chaîne de verrouillage.

En diffusant les messages “commit” et “abort” en parallèle, on ne peut que réduire la durée de la phase de résultat. Le seul type de message qui doit être transmis selon un certain ordre est le type “lock”, afin d'éviter les interblocages. Il est inutile et pénalisant d'imposer une séquence de transmission entre les tâches pour les autres types de messages.

La figure 2.5 représente un déroulement similaire à celui de la figure 2.2, mais avec notre version simplifiée du protocole : les managers ont disparu, seuls quatre types de messages sont présents, et les messages “commit” et “abort” sont diffusés en parallèle. Sur cet exemple, nos simplifications permettent de ramener le déroulement à 12 messages, au lieu de 17 pour la version originale du protocole.

2.3.2 Extension aux communications asynchrones

Le comportement des portes tel qu'il est spécifié dans la publication [PS96] de Parrow et Sjödin est correct sous l'hypothèse de communications synchrones, mais il peut mener à des interblocages dans le cas de communications asynchrones. Ces interblocages sont dûs à la manière dont une porte enregistre l'état des tâches.

Pour pouvoir détecter si un rendez-vous est possible, une porte enregistre les tâches qui se sont manifestées par un message “ready” dans un ensemble “rdyset” (nommé “R” dans

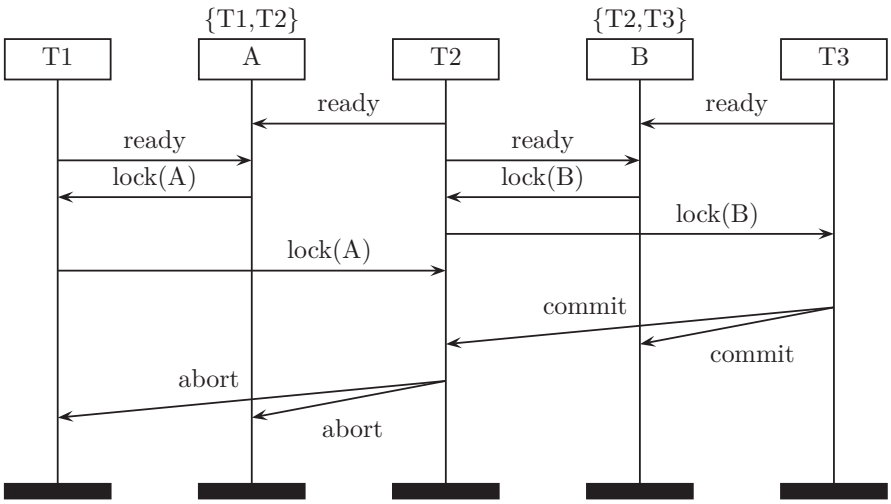


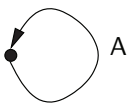
FIGURE 2.5 – Le déroulement de la figure 2.2 après les trois simplifications.

la publication de Parrow et Sjödin [PS96]). La spécification de Parrow et Sjödin précise qu'une porte doit vider cet ensemble quand elle reçoit un message de confirmation de négociation. L'idée derrière cette opération est la suivante : toutes les tâches ont participé au rendez-vous, elles ne sont donc plus prêtes pour une action sur la porte.

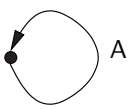
Cependant, dans le cadre de communications asynchrones, il se peut que le message de confirmation "commit" arrive à la porte après un certain délai pendant lequel des messages "ready" sont reçus par la porte. Dans ce cas, vider l'ensemble rdyset à la réception du "commit" peut faire oublier à une porte qu'une ou plusieurs tâches sont prêtes, et mener à un interblocage.

Nous illustrons cette possibilité d'interblocage avec le système suivant :

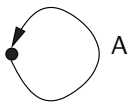
```
process T1 [A: none] is
   loop
      A
   end loop
end process
```




```
process T2 [A: none] is
   loop
      A
   end loop
end process
```



```
par A in
   T1 [A]
|| T2 [A]
end par
```



Le système comporte deux tâches T1 et T2 qui peuvent indéfiniment effectuer des actions sur une porte A. La composition parallèle indique que ces deux tâches doivent synchroniser leurs actions sur la porte A. Globalement, ce système peut donc réaliser une infinité d'actions sur la porte A.

La figure 2.6 illustre un déroulement possible du protocole, en faisant apparaître l'ensemble rdyset de la porte A. Les trois simplifications présentées précédemment sont utilisées, mais le même déroulement peut facilement être transcrit dans la version de Parrow et Sjödin du protocole.

À la réception des messages "ready" des deux tâches, la porte A les enregistre dans l'ensemble rdyset. La porte A détecte un rendez-vous possible et lance une négociation qui atteint les deux tâches. La tâche T2 accepte le verrou et envoie un message "commit" à la porte A et à la tâche T1. On considère que le message "commit" de la tâche T2 vers la porte A reste en transmission sur le réseau pendant un certain délai.

Pendant ce temps, la tâche T1 a le temps d'effectuer une action sur la porte A puis de se manifester à nouveau sur cette porte. La porte A enregistre donc la tâche T1 dans son ensemble rdyset, qui vaut ainsi {T1,T2}. Le message "commit" de la tâche T2 arrive alors à la porte A : c'est une confirmation de négociation, l'ensemble rdyset est donc réinitialisé. La porte A perd au passage l'état de la tâche T1, et elle ne peut pas détecter de synchronisation possible par la suite, même une fois qu'elle a reçu le prochain message "ready" de la tâche T2. La porte A est bloquée en attente de message "ready", et les deux tâches sont bloquées en attente d'une demande de verrouillage : le système qui devrait pouvoir effectuer une action sur la porte A ne progresse plus, il y a un interblocage dû au protocole.

Nous avons modifié le comportement des portes afin de rendre le protocole correct pour des communications asynchrones. En plus de l'ensemble rdyset, chaque porte a aussi un ensemble "deal_rdyset" où sont enregistrées les tâches qui se manifestent durant une négociation. Une porte réinitialise son ensemble deal_rdyset à chaque fois qu'elle amorce une chaîne de verrouillage. Entre le début de la négociation et la réception d'un message de résultat, une porte enregistre les tâches qui se signalent comme prêtes dans son ensemble deal_rdyset plutôt que dans son ensemble rdyset. Lorsqu'une porte reçoit un message "commit" de la part d'une tâche T, elle retire d'abord de l'ensemble rdyset toutes les tâches concernées par le rendez-vous, puis elle ajoute les éléments de l'ensemble deal_rdyset dans l'ensemble rdyset. Enfin, elle retire la tâche T de l'ensemble rdyset car la réception du message "commit" indique que cette tâche n'est plus prête sur la porte. À la réception d'un message "abort" de la part d'une tâche T, l'ensemble rdyset reçoit les tâches présentes dans deal_rdyset, et la tâche T est retirée de l'ensemble rdyset.

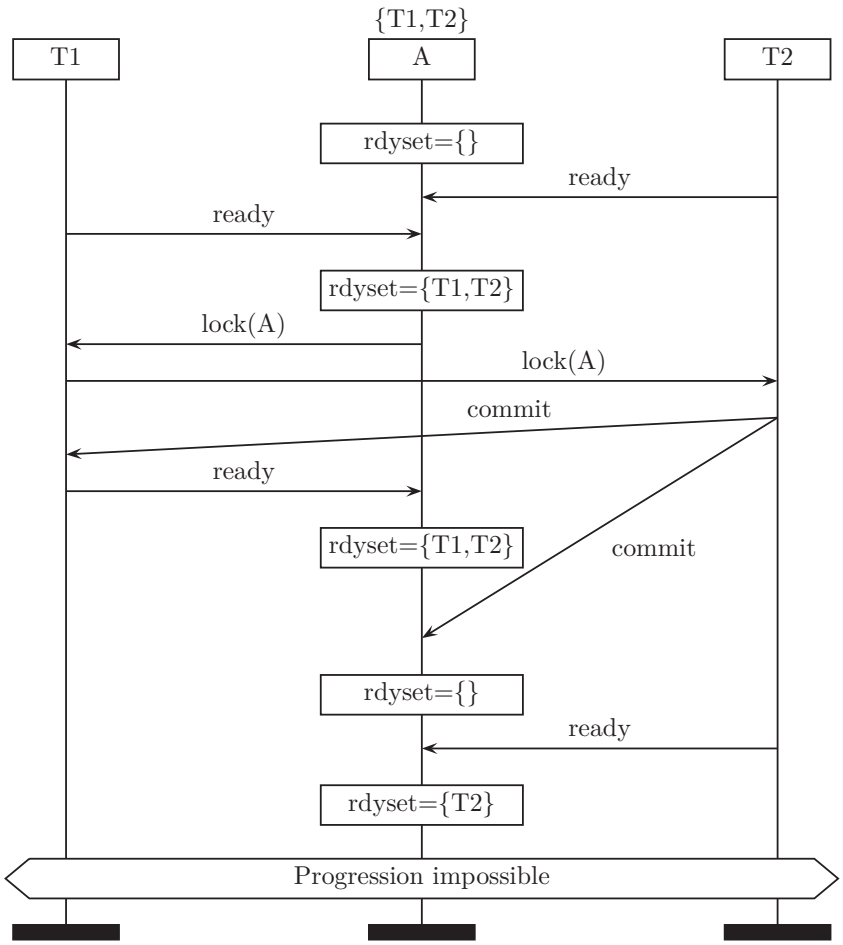


FIGURE 2.6 – Un interblocage est possible en cas de communications asynchrones.

2.3.3 Portes avec plusieurs vecteurs de synchronisation

Nous étendons le comportement des portes afin qu'une porte puisse gérer plusieurs vecteurs de synchronisation, par rapport à la version de Parrow et Sjödin du protocole qui considère que chaque porte n'a qu'un seul ensemble de tâches à synchroniser. Chaque porte est équipée d'un tableau qui peut stocker ses vecteurs de synchronisation. Lorsqu'une porte recherche si un rendez-vous est possible, elle teste tous les vecteurs de synchronisation de son tableau. Afin de ne pas favoriser un rendez-vous en particulier, la porte tire au hasard un index du tableau avant de commencer sa recherche à cet index.

Lorsqu'une tâche accepte un verrou, elle doit pouvoir déterminer si elle est la dernière tâche du vecteur ou bien si elle doit encore transmettre la demande de verrou à la tâche suivante. Pour cela, les vecteurs de synchronisation de chaque porte, qui sont connus à la compilation,

sont répliqués sur chaque tâche. De plus, lorsqu'une porte P amorce une négociation pour son vecteur de synchronisation à l'index "i", elle envoie un message "lock(P,i)" afin que les tâches puissent récupérer l'index du vecteur de synchronisation. Nous rappelons qu'un vecteur de synchronisation stocke les identifiants de tâche de manière ordonnée, ce qui permet aux tâches de connaître leur place dans la chaîne de verrouillage.

La gestion de plusieurs vecteurs de synchronisation par porte permet directement de traiter les rendez-vous à n-parmi-m. En effet, ceux-ci se traduisent simplement en une énumération de vecteurs de synchronisation. Considérons par exemple la composition parallèle suivante :

```
par A #2 in
   T1 [A]
|| T2 [A]
|| T3 [A]
end par
```

Cette composition se traduit par les vecteurs de synchronisation {T1,T2}, {T2,T3} et {T1,T3} pour la porte A.

Partition du système

La gestion de plusieurs vecteurs de synchronisation par porte soulève la discussion de la partition du protocole, c'est-à-dire du nombre de processus auxiliaires dédiés à la réalisation des rendez-vous. Nous dressons une liste non exhaustive de stratégies de partition, de la plus centralisée à la plus distribuée :

Un unique synchroniseur centralisé : un seul processus auxiliaire regroupe toutes les portes et traite tous les rendez-vous. Cette solution limite le parallèlisme de l'implémentation, car toutes les actions sont décidées par le synchroniseur centralisé.

Un processus auxiliaire par porte : chacun de ces processus gère l'ensemble des vecteurs de synchronisation de la porte qu'il représente. Cela permet de regrouper des vecteurs de synchronisation sous le contrôle d'une porte, qui autorise une négociation seulement pour un de ses vecteurs à la fois.

Un processus auxiliaire par vecteur de synchronisation : chaque vecteur de synchronisation est représenté par un processus indépendant, qui gère uniquement les rendez-vous correspondant à ce vecteur. Cette approche permet de potentiellement exécuter de nombreuses actions en parallèle. Cependant, elle augmente les chances d'avoir plusieurs négociations en cours au même moment pour des rendez-vous en conflit.

Pour notre protocole, nous avons retenus la partition selon les portes du système. Nous estimons que, de manière générale, les vecteurs de synchronisation rattachés à une porte ont des chances d'être en conflit, c'est-à-dire d'avoir au moins une tâche en commun. Par exemple, en cas de rendez-vous à n-parmi-m sur une porte, pour toute partition de

l'ensemble des vecteurs en deux sous-ensembles E_1 et E_2, pour tout vecteur $v_1 \in E_1$, il existe un vecteur $v_2 \in E_2$ en conflit avec v_1, et vice-versa. Dans ce cas, il est inutile de démarrer plus d'une négociation pour différents vecteurs de synchronisation en conflit, car seule l'une de ces négociations peut aboutir. La porte permet donc, à travers un regroupement pertinant des vecteurs de synchronisation, et de modérer le nombre de négociations en cours dans le système.

Pour des travaux futurs, il serait toutefois intéressant d'étudier différentes stratégies de partition et leur impact sur les performances du protocole. Nous estimons que la configuration des vecteurs de synchronisation d'un système est susceptible d'influencer l'efficacité d'une technique de partition. Nous pensons qu'une bonne stratégie regroupe autant que possible les vecteurs de synchronisation en conflit, et limite le nombre de processus auxiliaires dans l'implémentation, tout en permettant aux actions indépendantes d'être réalisées en parallèle.

2.3.4 Gestion des actions internes

Nous traitons les actions internes "i" directement au niveau d'une tâche, sans négociation. Ce type d'action n'est autorisée que lorsque la tâche n'est pas verrouillée. En effet, quand la tâche est verrouillée, elle s'est déjà engagée à prendre part à une autre action qui aura lieu si la chaîne de verrouillage aboutit.

Lorsque la seule action listée par la tâche est une action interne, la tâche prend le temps de refuser les éventuelles demandes de verrou en attente avant de réaliser cette action interne. Lorsqu'au contraire, d'autres actions sont aussi possibles, la tâche prévient les portes correspondantes, puis attend des demandes de verrou. Lors de cette attente, lorsque la tâche n'est pas verrouillée, elle peut à tout moment décider de réaliser l'action interne.

Nous précisons la manière de traiter les actions internes car peu des protocoles étudiés dans la littérature évoquent ce point. De plus, certaines approches peuvent s'avérer inefficaces, voire incorrectes. Par exemple, on peut considérer l'action interne comme une porte dont la liste des vecteurs de synchronisation est constituée des singletons de chaque tâche. Cette approche est inefficace car elle entraîne des échanges de messages entre la tâche et la porte qui représente les actions internes, alors que le choix de l'action interne peut toujours se résoudre localement au niveau de la tâche.

Une autre manière de procéder est, dans le cas où une action interne est possible parmi d'autres actions, de choisir au hasard, une fois pour toute, de réaliser l'action interne ou non. Cette approche peut mener à des interblocages : si, étant donné l'état du système, l'action interne est la seule action possible pour la tâche et que celle-ci a fait le choix de ne pas la réaliser, alors elle va bloquer indéfiniment en attendant qu'une négociation la concernant réussisse. C'est pourquoi dans le cas général il est important de pouvoir réaliser une action interne si aucune négociation n'aboutit pour une autre action.

Remarque 2-1

Les actions sur portes qui ne nécessitent pas de synchronisation entre plusieurs tâches pourraient être traitées de manière similaire aux actions internes, c'est-à-dire directement au niveau des tâches. Cela permet notamment d'éviter une négociation pour ce type d'actions. Nous n'avons cependant pas retenu cette approche, car elle n'est pas compatible avec le mécanisme d'interaction, présenté au chapitre 3, qui requiert, pour chaque porte, un point unique de décision de réalisation d'actions. Néanmoins, on pourrait envisager une optimisation dans le cas d'action non synchronisée qui n'interagit pas avec l'environnement. ■

2.3.5 Gestion des offres

Nous avons modifié le protocole pour prendre en compte les offres d'actions afin d'assurer les échanges de données lors de rendez-vous. Les offres de chaque tâche sont remontées aux portes qui se chargent de tester la compatibilité des offres pour savoir si un rendez-vous est possible. La porte est aussi responsable de fusionner les offres avant de les diffuser aux tâches concernées par une synchronisation.

Lorsqu'une tâche est prête pour une action comportant des offres, elle transmet ces offres à travers un message "ready(*offres*)". En particulier, lorsqu'une offre est en mode réception, la tâche n'énumère pas toutes les valeurs possibles pour le type de cet offre : elle signale simplement que l'offre est en mode réception, et que sa valeur n'est donc pas fixée.

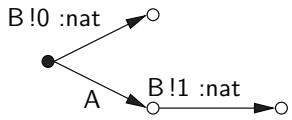
Quand une porte teste la possibilité d'un rendez-vous sur un vecteur de synchronisation, elle doit vérifier non seulement que toutes les tâches sont prêtes, mais aussi que leurs offres sont compatibles. Dans ce cas, une porte fusionne les offres des tâches pour obtenir des *offres de négociation*, qui sont transmises le long de la chaîne de verrouillage. Une tâche accepte un verrou seulement si elle est prête sur la porte et que les offres de négociation sont compatibles avec les siennes. Si la négociation aboutit, toutes les tâches concernées réalisent alors une action avec les offres de négociation.

Nous considérons pour le moment que les offres de négociations sont toujours en mode envoi, et que leur valeur est donc fixée et connue. Nous traitons en détail le cas où la fusion des offres par la porte produit des offres en mode réception dans la section 3.1.3.

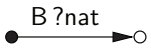
La présence d'offres nécessite de modifier le comportement des portes lors de la réception d'un message "abort", sous peine de possibilité d'interblocage. En effet, jusqu'à maintenant une porte P qui reçoit un message "abort" d'une tâche T retire T de l'ensemble des tâches prêtes, étant donné que cette tâche a refusé le verrou. Cependant, dans le cas de présence d'offre, il se peut que la tâche T refuse le verrou non pas car elle n'est pas prête sur la porte P, mais uniquement car ses offres courantes ne sont pas compatibles avec les offres transmises avec la demande de verrou. La tâche T refuse donc le verrou, et si la porte P retire T de l'ensemble des tâches prêtes, elle ne peut plus négocier aucun rendez-vous qui

concerne T, ce qui peut mener à un interblocage. Le système suivant permet d'illustrer cette possibilité :

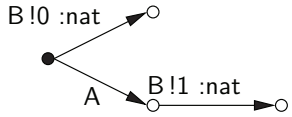
```
process T1 [A: none, B: nat] is
   select
      B (0 of nat)
   [] A ; B (1 of nat)
   end select;
   stop
end process
```




```
process T2 [B: nat] is
   var n : nat in
      B (?n);
      stop
   end var
end process
```




```
par B in
   T1 [A,B]
|| T2 [B]
end par
```



La figure 2.7 illustre un déroulement possible du protocole sur ce système. La porte B reçoit les offres des deux tâches T1 et T2. Elle constate que ces offres sont compatibles et démarre une négociation avec l'offre obtenue en fusionnant les offres des tâches. Le temps que la demande de verrou atteigne la tâche T1, celle-ci effectue une action sur la porte A. La tâche T1 est maintenant prête sur la porte B, avec une offre différente et incompatible avec l'offre de négociation véhiculée dans la demande de verrou. La tâche T1 refuse donc le verrou et envoie un message "abort" à la porte B. À la réception de ce message "abort", si la porte B retire la tâche T1 de la liste des tâches prêtes, alors elle ne démarrera pas de nouvelle négociation. Le système est ainsi bloqué, alors qu'un rendez-vous des tâches T1 et T2 sur la porte B devrait être possible.

Nous résolvons ce problème en modifiant le comportement des portes. Comme présenté dans la section 2.3.2, chaque porte possède un ensemble rdyset qui stocke les tâches considérées comme prêtes par la porte, ainsi qu'un ensemble deal_rdyset qui stocke les tâches qui ont annoncé être prêtes durant une négociation. À la fin de la section 2.3.2, nous avons précisé qu'à la réception d'un message "abort" de la part d'une tâche T, cette tâche T est retirée de l'ensemble rdyset. Nous modifions ce comportement de la manière suivante : lorsqu'une porte reçoit un message "abort" de la part d'une tâche T, alors cette tâche T est retirée de l'ensemble rdyset *uniquement si elle n'est pas dans l'ensemble deal_rdyset*. Cette modification permet de gérer les offres sans risque d'interblocage.

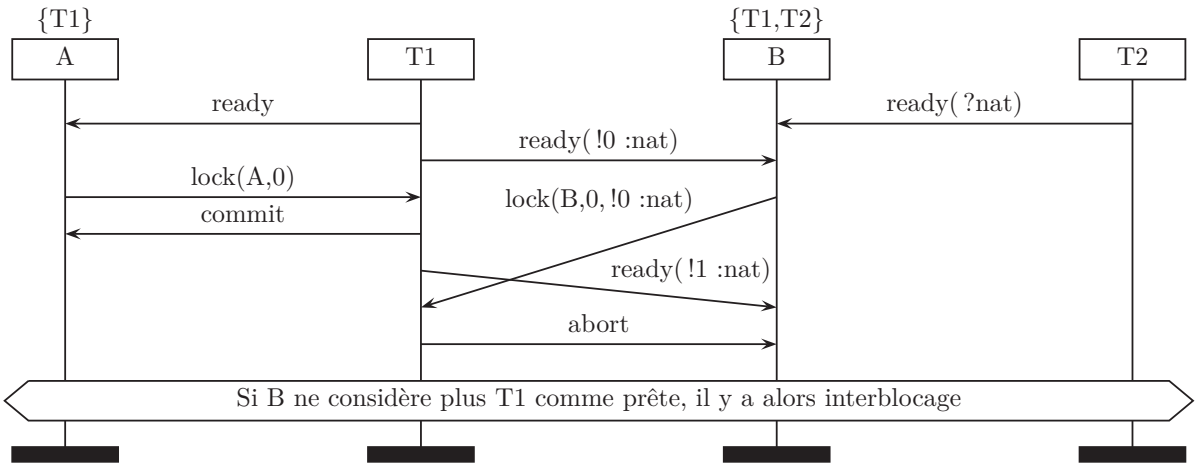


FIGURE 2.7 – Une tâche peut être prête deux fois de suite sur une porte, mais avec des offres différentes. Dans ce cas, la porte ne doit pas toujours traiter un message “abort” comme signifiant que la tâche n’est plus prête.

2.3.6 Confirmation de réussite par la porte

Le processus qui décide de la réalisation d’une action est la dernière tâche à accepter de se verrouiller, en fin de négociation. Cependant, le mécanisme d’interaction qui sera présenté au chapitre 3 nécessite que la porte puisse elle-même décider si une négociation qui a abouti entraîne la réalisation d’une action ou non. Nous exposons cette extension dès maintenant afin de décrire en totalité le protocole véritablement utilisé dans DLC.

Nous souhaitons que, pour certaines négociations, le choix de réaliser ou non une action se fasse au niveau de la porte. Jusqu’ici, c’est la dernière tâche de la chaîne de verrouillage qui, si elle accepte le verrou, déclenche la réussite de la négociation et avertit ensuite la porte et les autres tâches concernées. Désormais, lorsqu’une porte P démarre une négociation, elle peut indiquer qu’elle souhaite confirmer la réussite de la négociation via un message “lock(P, confirm, ...)”.

Lorsque la tâche en fin de chaîne de verrouillage accepte un verrou qui requiert une confirmation de la porte, cette tâche transmet le message “lock” à la porte et attend son verdict. À la réception de ce message “lock”, la porte peut soit décider de réaliser cette action, auquel cas elle diffuse un message “commit” à toutes les tâches concernées, soit se raviser et annuler la négociation, auquel cas elle diffuse un message “abort” à toutes les tâches concernées.

2.4 Optimisation : auto-verrouillage des tâches

Dans le protocole, chaque porte doit verrouiller les tâches qu'elle souhaite synchroniser, afin d'assurer l'exclusion mutuelle des rendez-vous en conflit. Toutefois, il est inutile de verrouiller les tâches qui sont prêtes sur une seule porte, car de toute manière la tâche n'acceptera aucun autre verrou que celui de la porte sur laquelle elle est prête. En suivant cette idée, une tâche peut indiquer à une porte qu'elle est prête uniquement sur cette porte, la porte n'a alors plus besoin de verrouiller la tâche. Nous avons nommé ce genre de verrouillage de la tâche par elle-même l'"*auto-verrouillage*" d'une tâche. Une idée similaire est présente dans le protocole α-core [PCT04] [2].

En pratique, quand une tâche est prête uniquement sur une seule porte, sans possibilité d'action interne, elle envoie à cette porte un message "ready(locked)" qui indique son auto-verrouillage. Ensuite, une négociation peut avoir lieu pour un certain vecteur de synchronisation, où les tâches auto-verrouillées n'ont pas besoin d'être explicitement verrouillées. Pour cela, lorsqu'une porte amorce une négociation, elle indique le *chemin de verrouillage*, c'est-à-dire le vecteur de synchronisation duquel les tâches auto-verrouillées sont retirées. Un chemin de verrouillage est donc un ensemble ordonné d'identifiants de tâches, qui indique quelles tâches doivent être verrouillées lors de la négociation. Une porte démarre une négociation avec un message "lock(P,i,*chemin*)" qui indique le chemin de verrouillage.

Lors d'une négociation, les demandes de verrous sont transmises uniquement le long des tâches qui font partie du chemin de verrouillage, toujours en respectant l'ordre global des identifiants de tâches. Si la dernière tâche du chemin de verrouillage accepte le verrou, alors la négociation réussit, et cette tâche envoie un message "commit" à la porte ainsi qu'un message "commit(i)" à toutes les tâches du vecteur de synchronisation. Le message de résultat "commit(i)" contient l'index "i" afin que les tâches auto-verrouillées puissent savoir selon quel vecteur de synchronisation l'action a été réalisée. Quand une ou plusieurs tâches sont auto-verrouillées, le chemin de verrouillage est plus court que le vecteur de synchronisation complet, et la négociation nécessite ainsi moins de messages.

Remarque 2-2

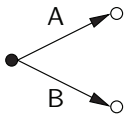
Avec cette optimisation et l'éventuelle confirmation de réussite par la porte, l'événement qui marque le succés d'une négociation peut avoir lieu sur différents processus. Lorsque la porte ne requiert pas de confirmation de réussite, nous sommes sûr qu'une négociation est un succès à partir du moment où la demande de verrou est acceptée par *la dernière tâche du chemin de verrouillage*. Lorsque la porte requiert une confirmation de réussite, la réalisation de l'action est décidée *au niveau de la porte*, une fois que toutes les tâches à verrouiller ont accepté le verrou. ■

Nous illustrons l'auto-verrouillage par la figure 2.8, qui expose un comportement possible

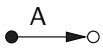
2. Dans le protocole α-core, les messages de type "PARTICIPATE" indiquent ce que nous appelons un auto-verrouillage.

du protocole sur l'exemple suivant :

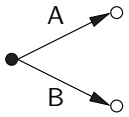
```
process T1 [A, B: none] is
   select
      A
   [] B
   end select;
   stop
end process
```




```
process T2 [A: none] is
   A;
   stop
end process
```

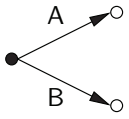



```
process T3 [A, B: none] is
   select
      A
   [] B
   end select;
   stop
end process
```




```
process T4 [A: none] is
   A;
   stop
end process
```

```
par A in
   B -> T1 [A,B]
||       T2 [A]
|| B -> T3 [A,B]
||       T4 [A]
end par
```



Les tâches T1 et T3 se déclarent prêtes sur les portes A et B. Les tâches T2 et T4 se déclarent prêtes et auto-verrouillées sur la porte "A", via un message "ready(locked)". La porte A lance ensuite une négociation dont le chemin de verrouillage ne contient que T1 et T3, soit un sous-ensemble du vecteur de synchronisation complet qui contient les quatre tâches, avec le message "lock(A, 0, {T1,T3})". Lorsque T3 accepte la demande de verrou, elle détecte qu'elle est la dernière tâche du chemin de verrouillage, elle transmet donc un message de confirmation à la porte ainsi qu'à toutes les tâches du vecteur de synchronisation.

Dans le cas où une tâche t refuse une demande de verrou, elle doit prévenir les tâches qui se sont déjà verrouillées : la tâche t renvoie alors un message "abort" aux tâches qui la précèdent au sein du chemin de verrouillage. Il est inutile d'envoyer un message "abort" aux

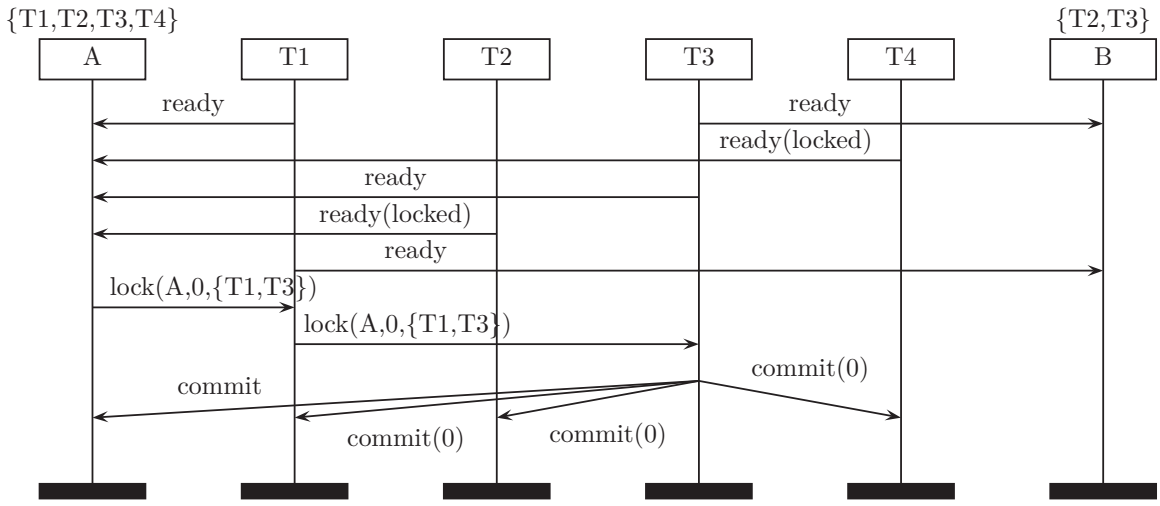


FIGURE 2.8 – Auto-verrouillage des tâches prêtes sur une seule porte : T2 et T4 sont auto-verrouillées sur A, qui n'a donc pas besoin de les verrouiller pour conclure une négociation.

tâches qui la précèdent dans le vecteur de synchronisation mais qui ne sont pas présentes dans le chemin de verrouillage, car ces tâches auto-verrouillées n'ont pas eu à traiter une demande de verrou.

Cas limite : barrière de synchronisation. Dans le cas limite où toutes les tâches d'un vecteur de synchronisation d'index "i" sont auto-verrouillées, il suffit à la porte d'indiquer que l'action a eu lieu en diffusant un message "commit(i)" à toutes les tâches concernées. On remarque que ce cas limite est alors équivalent à l'implémentation classique d'une barrière de synchronisation distribuée entre les processus tâches.

2.5 Cohabitation de l'auto-verrouillage et des rendez-vous en avance de phase

> The solutions all are simple—after you have arrived at them.
> But they're simple only when you know already what they are.
>
> R. M. Pirsig
> *Zen and the Art of Motorcycle Maintenance*

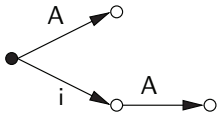
Le principe de l'auto-verrouillage est relativement simple, mais sa mise en place au sein de notre protocole engendre toutefois des situations complexes qui nécessitent quelques

modifications pour que le protocole reste correct. Ces situations sont notamment dues à la présence de rendez-vous en avance de phase. Ces derniers ne sont pas possibles dans le protocole α-core, qui n'a donc pas à gérer les situations que nous exposons ici.

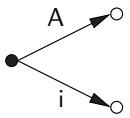
2.5.1 Demande de verrou à une tâche auto-verrouillée

Une tâche auto-verrouillée peut recevoir des demandes de verrou valides, et elle doit traiter ces demandes de verrou. La figure 2.9 illustre cette éventualité par un déroulement possible du protocole sur le système suivant (nous ne représentons par l'espace d'états de la composition parallèle car il n'est pas pertinent pour notre exemple) :

```
process T1 [A: none] is
   select
      A
   [] i ; A
   end select;
   stop
end process
```

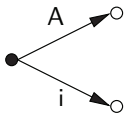



```
process T2 [A: none] is
   select
      A
   [] i
   end select;
   stop
end process
```




```
process T3 [A: none] is
   select
      A
   [] i
   end select;
   stop
end process

par A in
   T1 [A]
||
   par
      T2 [A]
   || T3 [A]
   end par
end par
```



La tâche T1 commence par annoncer à la porte A qu'elle est prête, puis elle effectue une action interne. La tâche T1 envoie ensuite un message d'auto-verrouillage “ready(locked)”

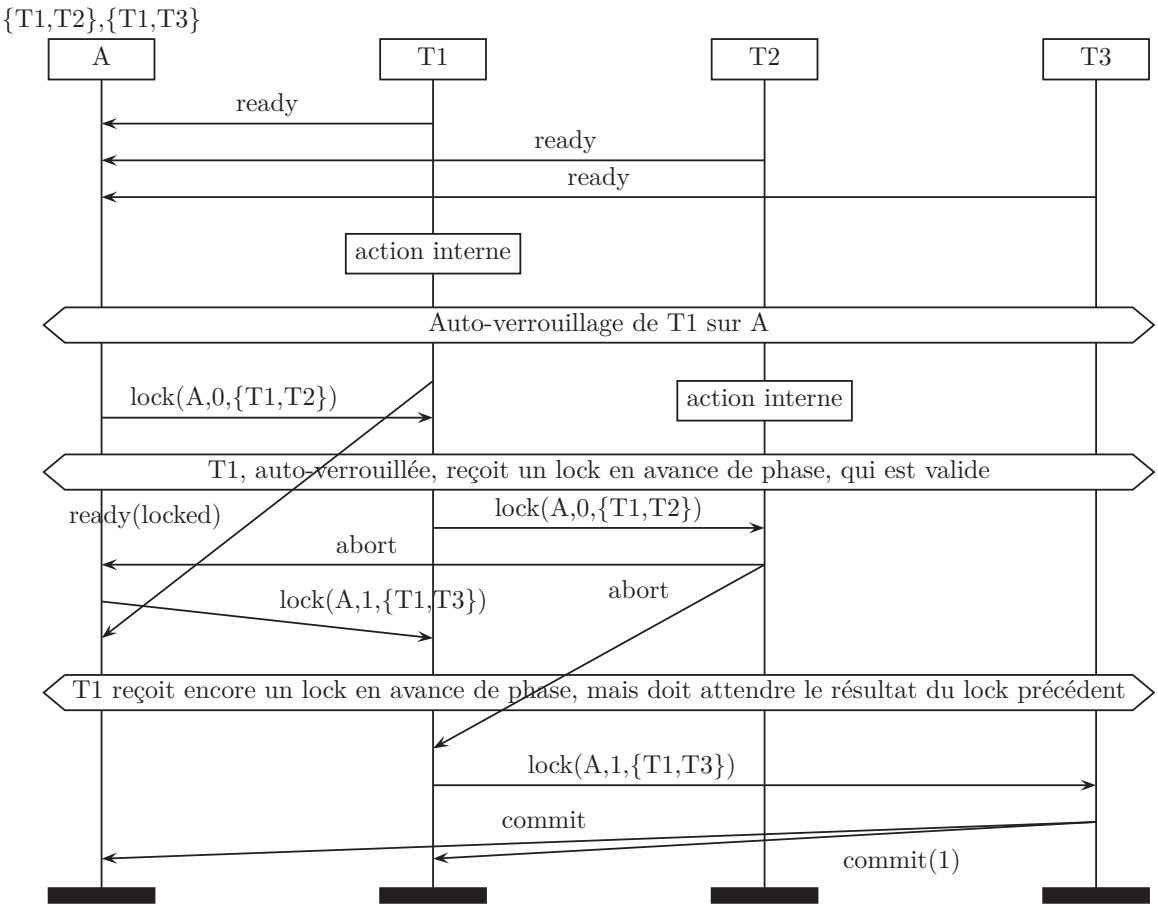


FIGURE 2.9 – Bien qu'une tâche soit auto-verrouillée, elle peut toujours recevoir des demandes de verrous en avance de phase qui soient valides, et qu'elle doit traiter.

qui va mettre beaucoup de temps à atteindre la porte A. Pendant ce temps, les tâches T2 et T3 se sont manifestées sur la porte A. La porte A lance une négociation pour le vecteur de synchronisation d'index 0, égal à {T1,T2}. La porte A n'a pas encore reçu le message d'auto-verrouillage de la tâche T1, la porte considère donc que T1 est prête et doit être verrouillée, en conséquence la tâche T1 fait partie du chemin de verrouillage. Cette négociation est en avance de phase, car elle est démarrée selon un état passé de la tâche T1. Nous sommes alors dans un cas où la tâche T1 est auto-verrouillée, et elle reçoit une demande de verrou valide qu'elle doit traiter.

La suite de l'exemple illustre qu'une tâche auto-verrouillée peut non seulement recevoir une demande de verrou valide, mais elle peut même en recevoir plusieurs à la suite. En effet, si la tâche T2 réalise une action interne, elle n'est alors plus prête sur la porte A. La tâche T2 refuse donc la demande de verrou et envoie un message "abort" à la porte, ainsi qu'à

la tâche T1. Il se peut que ce le message “abort” à destination de T1 mette beaucoup de temps à arriver. Pendant ce temps, la porte A reçoit la réponse de T2, et lance une nouvelle négociation cette fois pour le vecteur de synchronisation {T1, T3}. La porte A n'a toujours pas reçu le message d'auto-verrouillage de T1, cette tâche est donc toujours présente dans le chemin de verrouillage. On atteint ainsi un cas où la tâche T1 est auto-verrouillée, a accepté une demande de verrou valide, et reçoit une nouvelle demande de verrou valide. Avant de traiter cette nouvelle demande de verrou, la tâche T1 doit attendre la réponse de la demande précédente, qu'elle a acceptée. Faute de quoi, elle s'engagerait à participer à deux actions distinctes.

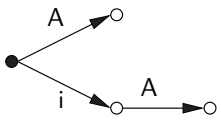
Il apparaît à travers cet exemple que l'auto-verrouillage d'une tâche ne lui évite pas d'avoir à traiter les demandes de verrou. Le comportement d'une tâche auto-verrouillée est caractérisé par le fait qu'elle peut recevoir un message “commit” sans avoir accepté de demande de verrou au préalable. Par contre, une tâche auto-verrouillée ne peut pas recevoir de message “abort” sans avoir auparavant accepté une demande de verrou, car les messages “abort” sont diffusés uniquement aux tâches déjà verrouillées et présentes dans le chemin de verrouillage d'une négociation.

2.5.2 Purge de messages d'auto-verrouillage en transit

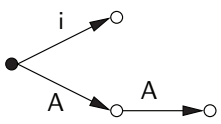
En présence de synchronisation en avance de phase, une porte peut recevoir de la part d'une tâche un message d'auto-verrouillage périmé, il est alors crucial d'ignorer ce message pour éviter une action invalide.

La figure 2.10 illustre ce genre de situation par une exécution possible du protocole sur le système suivant (encore une fois, nous omettons le LTS de la composition) :

```
process T1 [A: none] is
   select
      A
   [] i ; A
   end select;
   stop
end process
```




```
process T2 [A: none] is
   select
      A ; A
   [] i
   end select;
   stop
end select
```




```
par A in
   T1 [A]
```

```
|| T2 [A]
end par
```

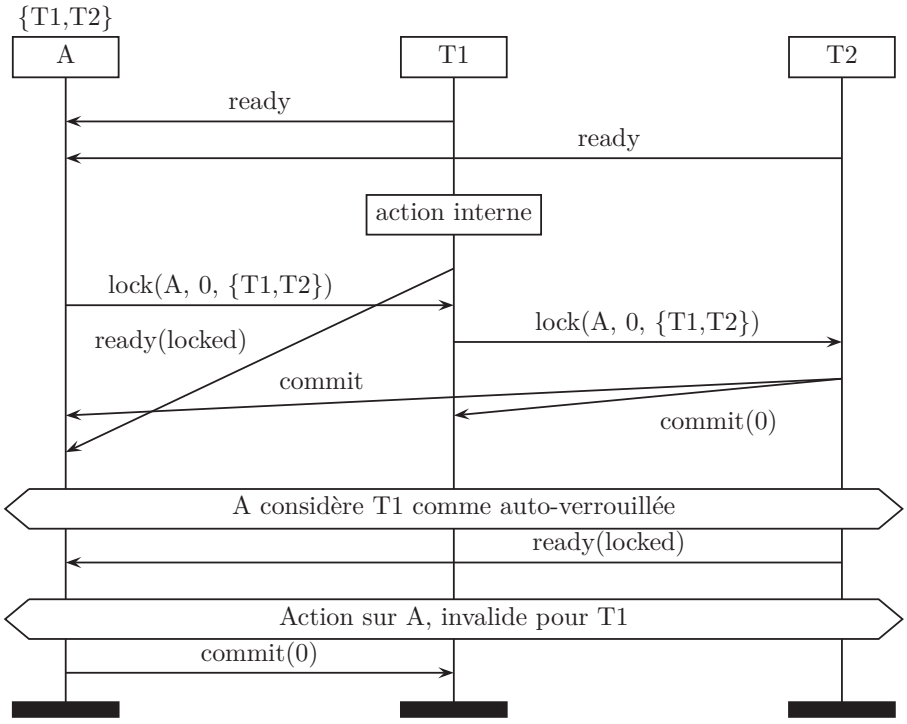


FIGURE 2.10 – Une porte peut considérer une tâche comme auto-verrouillée alors qu'elle ne l'est pas, et ainsi décider d'une action pour cette tâche alors que cette action n'est pas possible.

Les tâches T1 et T2 sont prêtes sur la porte A, qui lance une négociation pour synchroniser l'action de ces deux tâches. Avant que la demande de verrou n'atteigne la tâche T1, celle-ci effectue une action interne puis envoie un message "ready(locked)" à la porte A pour lui indiquer son auto-verrouillage ; ce message met du temps à arriver à la porte A. Pendant ce temps, la demande de verrou est acceptée par la tâche T1 puis la tâche T2, et cette dernière annonce la réussite de la négociation. La porte A reçoit le message "commit" de la part de la tâche T2, puis elle reçoit enfin l'annonce d'auto-verrouillage de la tâche T1. À ce moment, la porte A considère donc que la tâche T1 est auto-verrouillée, alors que cette tâche n'est en fait pas prête à effectuer une action sur A. La tâche T2 est, quant à elle, véritablement auto-verrouillée sur la porte A, et la porte A peut alors conclure qu'une action synchronisée entre les tâches T1 et T2 est possible, alors que la tâche T1 ne peut pas réaliser cette action.

L'origine de ce genre de situation apparaît lorsqu'une tâche t auto-verrouillée sur une porte

P accepte une demande de verrou d'une négociation amorcée par la porte P. La tâche t fait partie du chemin de verrouillage, car elle reçoit la demande de verrou. Cela indique que, au moment de démarrer la négociation, la porte P considérait la tâche t comme prête, mais pas comme auto-verrouillée. Cela signifie que la porte P n'avait pas encore reçu, et n'a peut-être toujours pas reçu, le message "ready(locked)" de la tâche t. Or, si la négociation est un succès, la tâche t va "consommer" son action sur la porte P. Dans ce cas, la porte ne doit pas considérer le message d'auto-verrouillage qu'elle a reçu, ou bien qu'elle va recevoir, de la part de t comme valide. Il est donc nécessaire que la tâche t signale à la porte P que le prochain message "ready(locked)" ne doit pas être pris en compte. La tâche t ne peut pas signaler cette situation à la porte P par un message direct car, les messages étant ordonnés entre deux processus, ce message arrivera systématiquement après le message "ready(locked)", que la porte P aura peut-être déjà pris en compte pour une autre négociation.

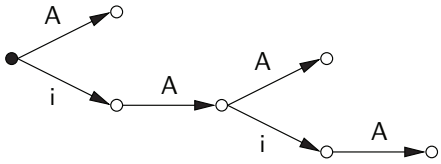
Pour indiquer à la porte P d'ignorer le prochain message "ready(locked)" de la tâche t, nous avons ajouté un nouveau champ *purge* dans les messages de demande de verrou et de résultat de négociation. Lorsque la tâche t auto-verrouillée sur la porte P accepte une demande de verrou de la porte P, cette tâche ajoute son identifiant au champ purge des messages suivants de la négociation. La porte P finit donc par recevoir un message de résultat de négociation, dont le champ purge indique toutes les tâches qui ont reçu une demande de verrou alors qu'elles étaient auto-verrouillées. La porte est ainsi prévenue qu'il faut ignorer les messages "ready" de ces tâches jusqu'à leur prochain message "ready(locked)" ; en d'autres termes, il faut *purger* les messages "ready" qui sont potentiellement encore en train de transiter sur le réseau, et transformer le message "ready(locked)" en simple "ready" afin de forcer une demande de verrou à la tâche. Le signal de purge est acheminé par les messages de la négociation courante pour s'assurer qu'ils sont reçus par la porte avant que celle-ci ne décide d'une autre négociation.

2.5.3 Purge de messages de résultats en transit

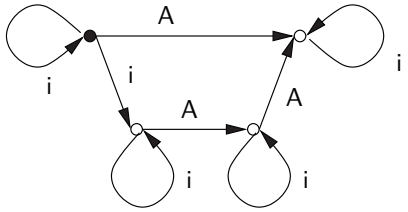
Lorsque la négociation est un échec, les tâches marquées à purger ne vont pas effectuer d'action suite à cette négociation. Dans ce cas, on peut envisager qu'une porte ignore le message de purge. Cependant, ce comportement de la part d'une porte pourrait mener à des actions invalides, comme le montre la figure 2.11, qui illustre un déroulement du protocole sur le système suivant :

```
process T1 [A: none] is
   loop
      select
         A
      [] i
      end select
   end loop
end process
```

```
process T2 [A: none] is
   select
      A
   [] i ;
      A;
      select
         A
      [] i ;
         A
      end select
   end select;
   stop
end process
```




```
par A in
   T1 [A]
|| T2 [A]
end par
```



Les tâches T1 et T2 se signalent prêtes sur la porte A, qui lance une première négociation pour laquelle elle demande une confirmation. La tâche T1 effectue une action interne puis s'auto-verrouille sur la porte A, par un message "ready(locked)" qui met du temps à être délivré. Entre temps, les tâches T1 et T2 acceptent la demande de verrou de la porte A, et T2 demande confirmation à la porte via le message "lock(A,confirm,0,{T1,T2},{T2})" qui indique la purge de la tâche T2. La porte A refuse alors la négociation pour des raisons externes, ignore le champ de purge et envoie un message "abort" aux deux tâches T1 et T2. On suppose que le message à destination de T2 est reçu après un certain délai. La porte A reçoit entre temps la notification d'auto-verrouillage de la tâche T2. Une nouvelle négociation contenant seulement la tâche T1 dans son chemin de verrouillage est alors lancée par la porte A. La tâche T1 accepte cette demande de verrou, et une première action sur A est ainsi réalisée. La tâche T1 envoie un message "commit" à la tâche T2, qui n'a toujours pas reçu le message "abort" de la porte A. La tâche T2 considère donc ce message "commit" comme le résultat de la toute première négociation.

Ensuite, les deux tâches T1 et T2 sont de nouveau prêtes sur la porte A, qui lance une nouvelle négociation, en demandant à nouveau une confirmation. Les tâches T1 et T2

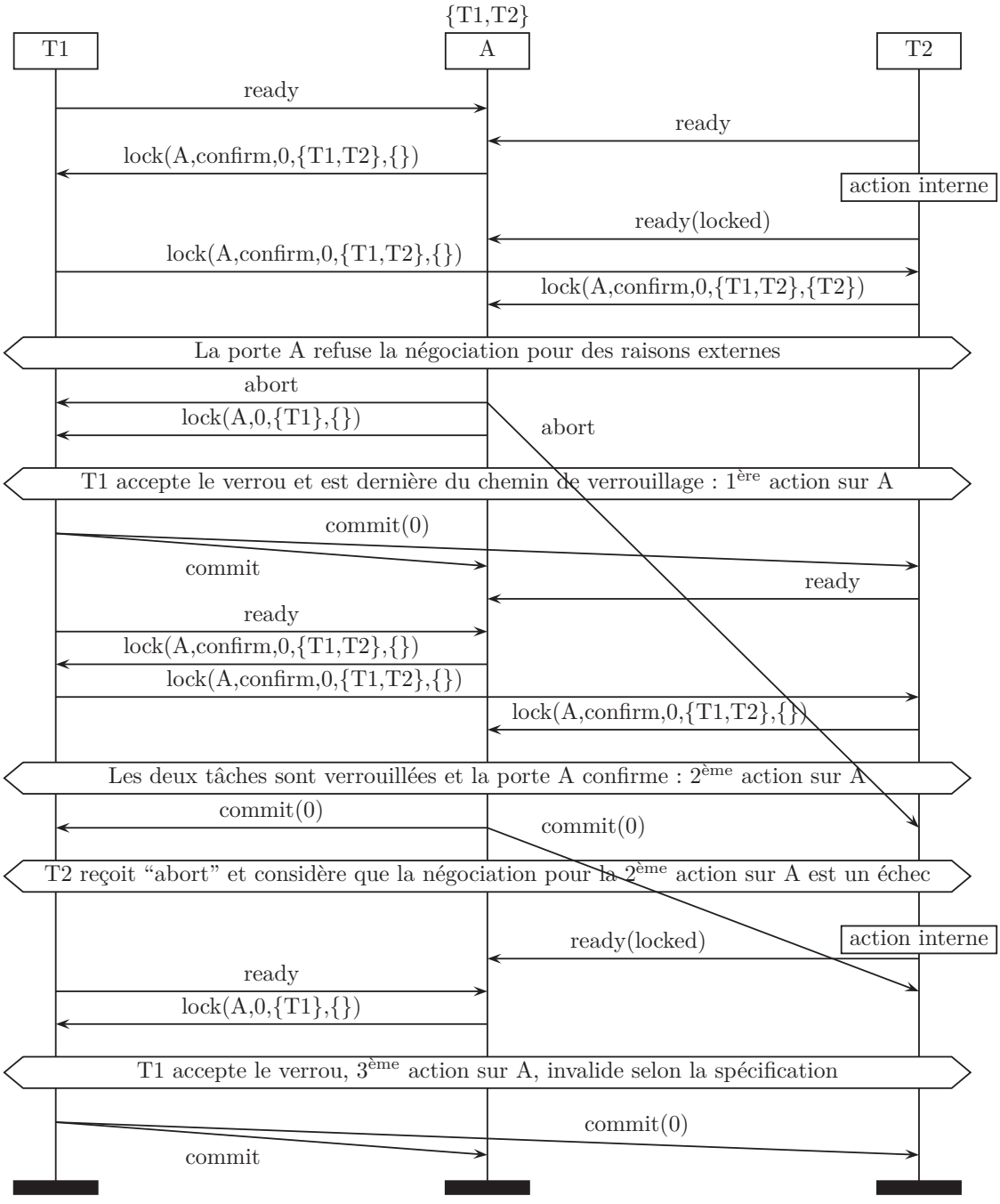


FIGURE 2.11 – Une action invalide peut avoir lieu si la porte ignore le signal de purge d'une négociation qui échoue.

acceptent les demandes de verrou, et la tâche T2 transmet le message de demande de verrou à la porte A pour confirmation. La porte A confirme la négociation : une deuxième action sur A est ainsi réalisée et la porte envoie un message "commit(0)" aux deux tâches. Cependant, la tâche T2 va obligatoirement recevoir le message "abort" en attente *avant* le message "commit" correspondant à la deuxième action sur la porte A, car les messages de la porte A vers la tâche T2 restent ordonnés. La tâche T2 considère donc que la porte n'a pas confirmé la négociation, et elle décide d'effectuer une action interne. On remarque ici qu'aux yeux de la tâche T2, la deuxième action sur A n'a pas eu lieu.

La tâche T2 se signale ensuite auto-verrouillée sur la porte A. Entre temps, la tâche T1 a reçu la confirmation pour la deuxième action sur A, et s'est signalée de nouveau prête. La porte A considère alors la tâche T1 comme prête et la tâche T2 comme auto-verrouillée, elle lance donc une négociation qui ne contient que la tâche T1 dans son chemin de verrouillage. La tâche T1 accepte le verrou et annonce une troisième action sur A. Or la spécification de la tâche T2 est telle que le système peut réaliser au plus deux actions sur la porte A. Cette troisième action est donc invalide, ce qui montre qu'une porte ne doit pas ignorer le signal de purge en cas d'échec de négociation.

Dans le contre-exemple précédent, le problème vient d'un décalage entre l'avancement de la tâche T2 et le reste du système. Ce décalage est déclenché par le croisement du message "abort" concernant la première négociation, et le message "commit" concernant la deuxième. Ce message "commit" résulte de la deuxième négociation qui n'a pas effectué de demande de verrou à la tâche T2, car la porte A considérait à ce moment cette tâche comme auto-verrouillée. Cependant, la porte A a reçu un signal de purge pour la tâche T2 avant de lancer la deuxième négociation.

Ce problème est résolu par la prise en compte de la purge même en cas d'échec de négociation. La figure 2.12 illustre le protocole sur le même exemple que précédemment, mais cette fois ci la porte A prend en compte la purge de la première négociation. Ainsi, la deuxième négociation contient la tâche T2 dans son chemin de verrouillage. Quand la tâche T2 reçoit la demande de verrou pour la deuxième négociation, elle place cette demande en attente car elle s'est déjà engagée pour la première négociation. La tâche T2 ne va pas traiter la demande de verrou pour la deuxième négociation tant qu'elle n'a pas reçu la réponse pour la première, ce qui empêche le croisement de message du contre-exemple de la figure 2.11. En d'autres termes, le fait de retirer l'auto-verrouillage de la tâche T2 lorsqu'elle s'est signalée à purger impose de lui envoyer une demande de verrou pour toute négociation ultérieure, et cette demande de verrou ne sera pas traitée par la tâche T2 avant qu'elle ne reçoive la réponse de la négociation sur laquelle elle s'est signalée à purger. Nous forçons ainsi la purge du message de résultat négatif, ce qui permet d'éviter les croisements de messages de résultats.

Remarque 2-3

Le protocole α-core n'a pas besoin de purger des messages en attente, car lorsqu'une action est confirmée à une tâche, cette tâche avertit toutes les autres portes sur lesquelles

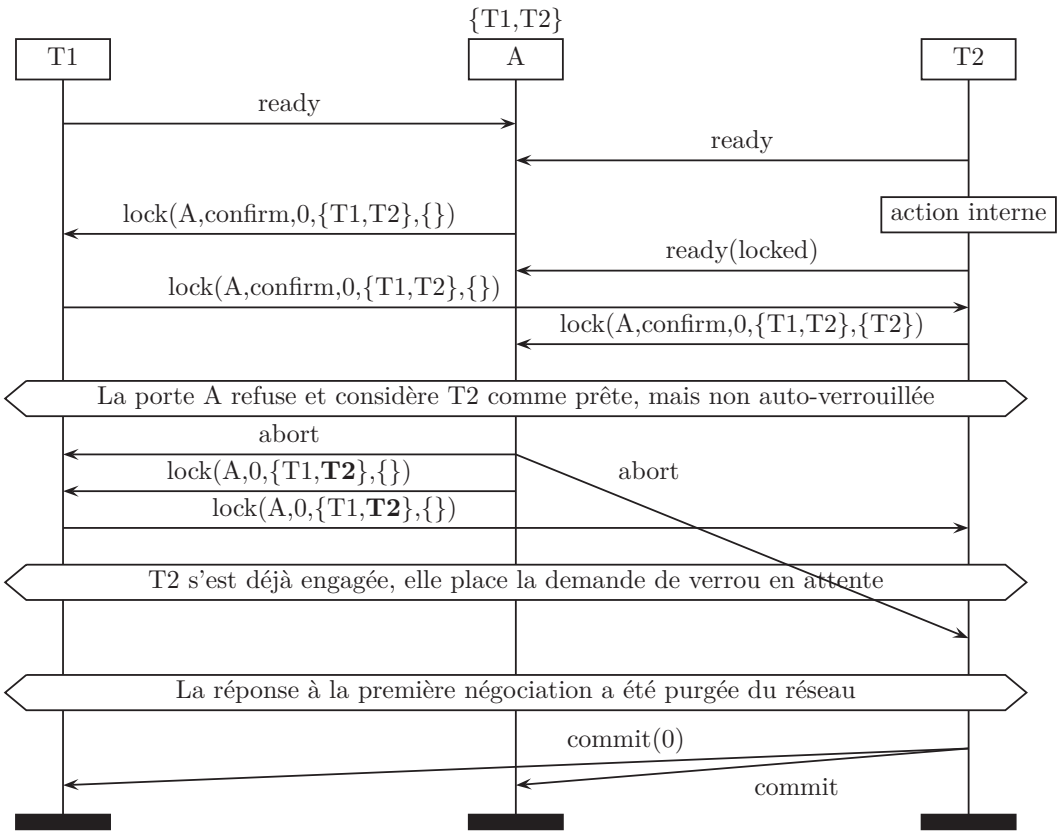


FIGURE 2.12 – À la réception d'un signal de purge suite à une négociation qui échoue, la porte A continue de considérer la tâche T2 comme prête, mais pas comme auto-verrouillée. Ce comportement force à effectuer une nouvelle demande de verrou à la tâche T2, et permet de *purger* la réponse à la première négociation.

elle était prête et attend une confirmation de ces portes avant d'effectivement réaliser l'action. Le protocole α-core impose ainsi une sorte de re-synchronisation entre tâches et portes à chaque action. Cette re-synchronisation empêche le phénomène de synchronisation en avance de phase, et facilite la mise en place de l'auto-verrouillage. La phase de re-synchronisation du protocole α-core requiert toutefois des messages supplémentaires, qui ralentissent l'exécution du protocole. Notre mécanisme de signal de purge permet de combiner l'auto-verrouillage et les synchronisations en avance de phase sans nécessiter de messages supplémentaires. ■

2.6 Coût du protocole

Nous comparons le coût du protocole de Parrow et Sjödin [PS96], de α-core [PCT04], et enfin du protocole que nous avons mis au point pour DLC. La table 2.2 résume le nombre de messages requis par chaque protocole pour synchroniser n tâches dont k peuvent s'auto-verrouiller. Comme nous l'expliquons dans la suite, les quantités entre crochets représentent des messages additionnels qui peuvent éventuellement être pris en compte.

Protocole	Total messages	Plus longue séquence
Parrow et Sjödin [PS96]	$3n \quad [+2n]$	$2n \quad [+2]$
α-core [PCT04]	$4n - 2k + 2\sum_{i=1}^{n}(p_i - 1)$	si $n = k$: $\quad 2$ si $n > k$: $\quad 4 + 2(n-k)$
DLC	$3n - k \quad [+2]$	$2 + n - k \quad [+1]$

TABLE 2.2 – Comparaison de coût des protocoles : nombre total de messages requis et taille de la plus longue série de messages en séquence pour synchroniser n tâches, dont k sont en situation d'auto-verrouillage. La quantité p_i dénote le nombre de portes sur lesquelles la tâche i est prête.

Remarque 2-4

Nous n'avons pas pris en compte les perturbations qui peuvent être causées par des échecs de négociation. Ces perturbations sont difficiles à estimer, notamment dans le protocole de Parrow et Sjödin et celui de DLC, car les synchronisations en avance de phase permettent parfois à une négociation démarrée dans un état passé du système de tout de même aboutir à un succès dans l'état courant du système. Pour des travaux futurs, il serait néanmoins intéressant d'étudier, pour chaque protocole, la rapidité de détection et d'annulation d'une négociation qui échoue. ■

Dans le cadre de nos mesures, deux quantités sont relevées : d'une part le nombre de messages total requis pour assurer une synchronisation, et d'autre part la taille de la plus longue série de messages qui doivent nécessairement être transmis en séquence. En effet, sur le réseau la diffusion de messages en parallèle n'est en général pas beaucoup plus coûteuse en temps que la transmission d'un seul message. C'est pourquoi il nous semble pertinent d'évaluer un protocole par le plus grand nombre de messages qui sont transmis les uns après les autres, tels que les demandes de verrouillage par exemple.

Nous présentons le calcul qui conduit aux résultats de chaque protocole :

Parrow et Sjödin. Le calcul du nombre total de messages requis a déjà été discuté dans la section 2.1. Pour la négociation entre les n managers et une porte, $3n$

messages sont requis. Ensuite, les communication entre chaque manager et sa tâche associée totalisent $2n$ messages. Nous comptons ces messages comme optionnels, car en général les communications entre un manager et sa tâche seront locales et n'impacteront pas la performance du protocole.

Concernant la plus longue séquence, elle débute par les messages des managers qui avertissent une porte qu'une tâche est prête. Ces messages peuvent être diffusés en parallèle, nous comptons donc une seule transmission de message pour cette étape. Ensuite, la négociation requiert n demandes de verrous en séquence (le protocole de Parrow et Sjödin ne traite pas l'auto-verrouillage). Lorsque le dernier manager accepte le verrou, il envoie une confirmation qui remonte la chaîne de verrouillage, ce qui consomme $n-1$ messages. La plus longue séquence contient donc $1+n+(n-1) = 2n$ messages, auxquels les 2 messages (de requête et de confirmation) entre un manager et sa tâche peuvent éventuellement être ajoutés.

α-core. Dans la section 2.1, nous avons rapidement discuté le nombre de messages requis pour la phase de verrouillage. Cependant le protocole possède aussi une phase de "re-synchronisation" : lorsqu'une tâche prête sur plusieurs portes reçoit la confirmation d'une porte, la tâche envoie un message "refuse" à toutes les autres portes sur lesquelles elle était prête. De plus, la tâche attend un message "ack" en réponse à chacun de ses messages "refuse".

Le protocole α-core nécessite donc : n messages qui annoncent que les tâches sont prêtes, $2(n-k)$ messages de verrouillage pour les tâches qui ne sont pas auto-verrouillées, et n messages de confirmation à destination des tâches, et les messages de re-synchronisation des portes. On note p_i le nombre de portes sur lesquelles la tâche i est prête. Après confirmation, chaque tâche doit prévenir toutes les portes sur lesquelles elle était prête, sauf la porte qui lui a envoyé une confirmation, la phase de re-synchronisation nécessite donc $2\sum_{i=1}^{n}(p_i-1)$ messages. Au total, le protocole α-core requiert ainsi $4n-2k+2\sum_{i=1}^{n}(p_i-1)$ messages.

En ce qui concerne la plus longue séquence de messages, dans le cas où toutes les tâches sont auto-verrouillées ($n = k$), les seuls messages échangés sont ceux qui préviennent une porte qu'une tâche est prête et ceux qui confirment l'action à une tâche. Ces deux types de messages sont diffusés en parallèle, la plus longue séquence est donc de 2 messages. Dans le cas où au moins une tâche nécessite d'être verrouillée ($n > k$), il faut compter pour chaque tâche à verrouiller une demande de verrou et sa réponse, ce qui ajoute $2(n-k)$ messages. De plus, la présence d'une tâche non auto-verrouillée entraîne une phase de re-synchronisation, qui consomme 2 messages ("refuse" et "ack") en séquence. La séquence la plus longue atteint ainsi $4+2(n-k)$ messages dans ce cas.

DLC. Pour synchroniser n tâches, notre protocole nécessite n messages "ready", suivis de $n-k$ demandes de verrou, et enfin n messages de confirmation "commit". De plus, en cas de demande de confirmation par la porte, il faut ajouter le transfert du message "lock" à la porte pour confirmation, et un message "commit" supplémentaire à destination de la dernière tâche du chemin de verrouillage. Le total des

messages requis est donc de $3n-k$, et la confirmation par la porte ajoute un surcoût éventuel de 2 messages.

La plus longue séquence de messages comptabilise un des messages "ready" qui peuvent être diffusés en parallèle, suivi par $(n-k)$ demandes de verrous pour les tâches non auto-verrouillées, puis par un des messages "commit" diffusés par la dernière tâche à accepter le verrou, ou bien par la porte si aucune tâche n'a besoin d'être verrouillée. La taille de la plus longue séquence atteint donc $2+n-k$ messages, auxquels il faut ajouter un message supplémentaire en cas de demande de confirmation par la porte.

On remarque que notre protocole combine la propagation de verrou proposée par Parrow et Sjödin et l'auto-verrouillage employé dans le protocole α-core. D'ailleurs, lorsque toutes les tâches sont auto-verrouillées (c'est-à-dire $n = k$, ce qui implique $\forall i, p_i = 1$) et que la porte ne demande pas de confirmation, le coût du protocole α-core et celui de notre protocole sont équivalents. Parmi les simplifications que nous avons apportées, la diffusion des messages de résultat permet de réduire la taille de la plus longue séquence par rapport au protocole de Parrow et Sjödin. Comparé au protocole α-core, nous n'avons pas de phase de re-synchronisation : cela permet non seulement de réduire le nombre de messages nécessaires, mais cela autorise aussi les synchronisations en avance de phase.

Bien qu'elles ne soient pas prises en compte dans nos mesures ici, car il est difficile d'estimer leur impact, les synchronisations en avance de phase peuvent raccourcir le nombre de messages nécessaires pour atteindre une synchronisation. En effet, une synchronisation en avance de phase représente une demande de verrou émise dans un état passé du système, qui permet tout de même de réaliser une synchronisation selon l'état courant des tâches ; elle peut donc faire économiser le démarrage d'une nouvelle négociation. À l'opposé, le protocole α-core impose une re-synchronisation qui force les portes à rester en phase avec les tâches.

2.7 Spécification formelle du protocole

Cette section résume le protocole obtenu après les améliorations et l'optimisation d'auto-verrouillage présentées dans les sections précédentes. Le comportement d'une porte et celui d'un manager sont exposés en intercalant des extraits de spécification LNT au sein d'une description en langage naturel, dans un style qui peut évoquer la programmation lettrée (*literate programming*) [Knu84]. Cette description est relativement longue, il n'est pas strictement nécessaire de la lire en détail avant d'aborder la suite du mémoire de thèse. Nous invitons le lecteur à revenir, par la suite, aux détails de la spécification du protocole en cas de nécessité.

Les processus LNT présentés ici sont extraits de la spécification véritablement utilisée au chapitre 5 pour vérifier le protocole. La spécification complète est disponible dans l'An-

nexe A. Nous avons fait le choix d'utiliser directement cette spécification dès maintenant afin d'assurer la cohérence de la description du protocole au sein de ce mémoire.

2.7.1 Types de données et canaux de communications

En plus des types prédéfinis de LNT, la spécification du protocole utilise quelques autres types de données. Ces nouveaux types sont introduits ici pour faciliter la compréhension du reste de la spécification.

Nous commençons par une description succincte des types suivants :

- DLC_ID : identifiants de portes ou de tâches
- id_set : ensemble *ordonné* ("sorted set") d'identifiants DLC_ID, utilisé notamment pour représenter un vecteur de synchronisation ou un chemin de verrouillage
- id_list : liste d'identifiants DLC_ID, utilisé pour stocker les signaux de purge
- sync_vect_list : liste de vecteurs de synchronisation (liste de id_set)
- dlc_action : une action, qui se résume à un identifiant de porte

Il existe ensuite plusieurs types de données plus élaborés, dont on donne la définition complète en LNT :

```
-- demande de verrou
type lock is
   lock (action : dlc_action, index : nat, path : id_set, confirm : bool, purge : id_set)
   with "get", "set"
end type

type message is
   READY (autolocked : bool),
   LOCK (lock : lock),
   COMMIT,
   COMMIT (purge : id_set),
   ABORT,
   ABORT (purge : id_set)
end type

-- état de la porte
type gate_state is
   idle, dealing
   with "=="
end type

-- état du manager
type manager_state is
   free, locked, autolock_free, autolock_locked
   with "==", "!="
end type

-- Stockage de l'espace d'états d'une tâche
```

```
type transition is
    nil_transition ,
    transition (action : dlc_action, next_states : nat_set)
   with "get", "=="
end type

type transition_list is
    list of transition
end type

type state is
    nil_state ,
    state ( id : nat, transitions : transition_list )
   with "get"
end type

-- Stockage des vecteurs de synchronisation
type sync_vect_list is
    list of id_set with "head", "access", "length"
end type

type sync_map_entry is
   sync_map_entry (gate : DLC_ID, vect_list : sync_vect_list ) with "get"
end type

type sync_map is
    list of sync_map_entry with "access", "length"
end type
```

Certains type de données ont une déclinaison sous forme de liste ou d'ensembles, nous ajoutons alors " _list " et "_set" à leur nom, par convention.

Enfin, les deux canaux de communications utilisés dans la spécification sont les suivants :

```
channel com is
  (DLC_ID, message)
end channel

channel annonce is
  (DLC_ID, id_set)
end channel
```

2.7.2 Spécification du comportement d'une porte

Le comportement générique d'une porte est décrit par un processus LNT qui est paramétré par un identifiant de porte et la liste des vecteurs de synchronisation de cette porte, ainsi que par des portes qui lui permettent de transmettre des messages et d'effectuer des annonces. Ce processus contient plusieurs variables locales, dont notamment des ensembles

d'identifiants qui stockent quelles tâches sont prêtes, auto-verrouillées, ou à purger. Les variables qui ont un rapport avec une négociation sont préfixées par "deal".

```
process GATE [SEND, RECV : com, ACTION, HOOK_REFUSE : annonce]
            (gate : DLC_ID, vectors : sync_vect_list)
is
   var
      state          : gate_state, -- etat de la porte
      readyset       : id_set,     -- taches pretes
      autolock       : id_set,     -- taches autoverrouillees
      dealreadyset   : id_set,     -- taches pretes pdt une negociation
      dealautolock   : id_set,     -- taches autoverrouillees pdt une negociation
      dealvect       : id_set,     -- vecteur de synchro de la negociation
      dealindex      : nat,        -- index du vecteur de negociation
      dealpath       : id_set,     -- chemin de verrouillage
      purgelist      : id_list ,   -- le prochain rdy(lck) devient rdy
      -- variables de stockage temporaire
      n              : nat,
      task           : DLC_ID,
      lock           : lock,
      confirm        : bool,
      purge          : id_list ,
      autolocked     : bool,
      vectindexes    : nat_set
   in
```

Le processus commence par une phase d'initialisation : la porte commence en état de repos (elle n'est pas en train de conduire une négociation), et les ensembles relatifs à l'état des tâches sont vides. La porte rentre ensuite dans sa boucle principale, qui ne termine jamais. Cette boucle est constituée d'un choix non déterministe entre la réception d'un message et le démarrage d'une négociation. Les sous-sections suivantes exposent chacune des branches de ce choix non déterministe.

```
      -- initialisation
      state          := idle ;
      readyset       := {};
      autolock       := {};
      dealreadyset   := {};
      dealautolock   := {};
      dealvect       := {};
      purgelist      := {};
      dealpath       := {};

      -- boucle principale
      loop
         select
```

Réception d'un message "ready"

À la réception d'un message "ready", la porte enregistre la tâche correspondante dans l'ensemble des tâches prêtes. Il existe un ensemble des tâches prêtes distinct pour enregistrer les tâches qui se signalent durant une négociation. De la même manière, la tâche est aussi ajoutée dans l'ensemble des tâches auto-verrouillées si le message "ready" a son drapeau "autolocked" à vrai. De plus, le message ready peut être ignoré ou l'auto-verrouillage annulé selon le signal de purge associé à la tâche.

```
-- Recoit un message READY
RECV (?task, ?READY (autolocked));

if member (task, purgelist) and (autolocked) then
   purgelist := delete (task, purgelist);
   -- ignore l'auto- verrouillage
   autolocked := false
end if;

if state == dealing then
   dealreadyset := insert (task, dealreadyset);
   if autolocked then
      dealautolock := insert (task, dealautolock)
   end if
else
   readyset := insert (task, readyset);
   if autolocked then
      autolock := insert (task, autolock)
   end if
end if
```

Démarrage d'une négociation

Lorsque la porte n'est pas déjà en train de négocier et qu'au moins un rendez-vous est possible, elle peut amorcer une négociation. Il se peut que l'ensemble des tâches prêtes autorise plusieurs rendez-vous parmi les vecteurs de synchronisation de la porte ; dans ce cas on choisit au hasard l'un de ces rendez-vous.

```
[]
-- Commence une négociation
only if (state == idle) and (possible_rdv (readyset, vectors)) then
   vectindexes := list_rdv_index (readyset, vectors);
   -- Choisit au hasard parmi les synchros possibles
   dealindex := any nat where member (dealindex, vectindexes);
   dealvect  := access (vectors, dealindex);
```

Le chemin de verrouillage est obtenu en retirant les tâches auto-verrouillées du vecteur de synchronisation. Si toutes les tâches du vecteur sont auto-verrouillées, alors ce chemin est vide et la porte peut directement décider si une action a lieu. Soit elle refuse cette action

(pour des raisons externes dues au mécanisme d'interaction, qui n'est pas détaillé ici), soit elle indique qu'une action a lieu, avant de diffuser des messages "commit" auprès des tâches concernées.

```
dealpath := diff (dealvect, autolock);
if empty (dealpath) then
   -- Toutes les tâches sont autoverrouillées
   select
      -- Refuse l'action pour des raisons externes
      HOOK_REFUSE (gate, dealvect)
   []
      -- Accepte l'action et diffuse le résultat
      ACTION (gate, dealvect);
      for n := 1 while n <= length (dealvect) by n := n+1 loop
         SEND (access (dealvect, n), COMMIT)
      end loop;
      readyset := diff (readyset, dealvect);
      autolock := diff (autolock, dealvect)
   end select
```

Sinon, la porte amorce une négociation en envoyant une demande de verrou à la première tâche du chemin de verrouillage. La porte peut indiquer qu'elle souhaite confirmer cette négociation, et l'ensemble de purge (le dernier champ du type "lock") est vide au démarrage d'une négociation. Elle passe ensuite dans l'état "dealing" (en train de négocier), en ayant pris soin de réinitialiser les variables utilisées dans cet état.

```
   else
      task := head (dealpath);
      -- La nécessité d'une confirmation est choisie au hasard
      confirm := any bool;
      SEND (task, LOCK (lock (action(gate), dealindex, dealpath, confirm, {})));
      -- Passe dans l'état négociation
      dealreadyset := {};
      dealautolock := {};
      state := dealing
   end if
end if
```

Réception d'un message "commit"

La porte peut recevoir un message "commit" seulement si elle a amorcé une négociation dont elle attend maintenant la réponse. À la réception d'un message "commit", la porte met à jour les ensembles qui stockent les tâches prêtes : toutes les tâches concernées par l'action ne sont plus considérées comme prêtes, hormis celles qui se sont manifestées durant la négociation. De plus, nous sommes sûr que la tâche qui a envoyé le message "commit" n'est plus prête. Ensuite, la porte met à jour l'état des tâches selon le signal de purge. Enfin, la porte repasse dans son état de repos, ce qui l'autorise à pouvoir amorcer une nouvelle négociation dans le futur.

La mise à jour de la liste de purge est effectuée à plusieurs endroits du processus. Nous avons donc abstrait cette mise à jour dans la procédure “update_purge”, qui est définie de la manière suivante :

```
function update_purge (in out purgelist : id_list , purge : id_list , in out autolock : id_set) is
   var id : DLC_ID, newpurge : id_list in
       purgelist := union ( purgelist , purge);
      newpurge := {};
      while not (empty ( purgelist )) loop
         id := head ( purgelist );
         if member (id, autolock) then
            autolock := remove (id, autolock)
         else
            newpurge := cons (id, newpurge)
         end if;
         purgelist := tail ( purgelist )
      end loop;
      purgelist := newpurge
   end var
end function
```

On retrouve cette procédure dans le traitement d’un message “commit” :

```
[]
   -- Recoit un message COMMIT
   only if state == dealing then
      RECV (?task, ?COMMIT (purge) of message);

      readyset := diff (readyset, dealvect );
      readyset := union (readyset, dealreadyset );
      readyset := remove (task, readyset );
      -- idem pour autolock
      autolock := diff (autolock, dealvect );
      autolock := union (autolock, dealautolock );
      autolock := remove (task, autolock );

      eval update_purge (!?purgelist , purge, !?autolock );
      state := idle
   end if
```

Réception d’un message “abort”

Un message “abort”, tout comme un message “commit”, peut être reçu uniquement si la porte est en attente de résultat de négociation. À la réception d’un message “abort”, la porte retire cette fois ci uniquement la tâche qui a refusé le verrouillage, et si elle ne s’est pas signalée comme prête durant la négociation, conformément à l’influence des offres discutée dans la section 2.3.5. Ensuite, la porte prend en compte le signal de purge, avant de passer dans son état de repos.

```
[]
  -- Recoit un message ABORT
  only if state == dealing then
    RECV (?task, ?ABORT (purge) of message);

    readyset := remove (task, readyset);
    readyset := union (readyset, dealreadyset);
    -- idem pour autolock
    autolock := remove (task, autolock);
    autolock := union (autolock, dealautolock);

    eval update_purge (!?purgelist, purge, !?autolock);
    state := idle
  end if
```

Réception d'un message "lock"

La dernière branche du choix non déterministe concerne la réception d'un message "lock", qui n'est possible que si la porte est en train de négocier. À la réception de ce message, la porte peut refuser ou accepter la réalisation de l'action. En cas de refus, la porte envoie un message "abort" à chaque tâche du chemin de verrouillage. Si elle accepte l'action, elle envoie un message "commit" à chaque tâche concernée par l'action, et elle retire ces tâches de l'ensemble des tâches prêtes ; de plus, nous sommes certain dans ce cas que la tâche qui a envoyé le message "lock" n'est plus prête. Dans les deux cas, la porte met ses ensembles de tâches prêtes à jour en prenant en compte le signal de purge. Cette branche étant la dernière, la déclaration du processus porte est ensuite terminée.

```
[]
  -- Recoit un message LOCK
  only if state == dealing then
    RECV (?task, ? LOCK (lock) of message);
    select
      HOOK_REFUSE (gate, dealvect);
      for n := 1 while n <= length (dealpath) by n := n+1 loop
        SEND (access (dealpath, n), ABORT)
      end loop;
      -- update idem qu'a la reception d'un ABORT
      readyset := union (readyset, dealreadyset);
      autolock := union (autolock, dealautolock)

    []
      ACTION (gate, dealvect);
      for n := 1 while n <= length (dealvect) by n := n+1 loop
        SEND (access (dealvect, n), COMMIT)
      end loop;

      readyset := diff (readyset, dealvect);
      readyset := union (readyset, dealreadyset);
```

```
                readyset := remove (task, readyset);
                -- idem pour autolock
                autolock := diff (autolock, dealvect);
                autolock := union (autolock, dealautolock);
                autolock := remove (task, autolock)

            end select;

            eval update_purge (!?purgelist, lock.purge, !?autolock);
            state := idle
        end if
    end select
  end loop
 end var
end process
```

2.7.3 Spécification du comportement d'un manager

Le comportement générique d'un manager est paramétré par l'identifiant de la tâche qui lui est associée, l'espace d'états de cette tâche ainsi que la carte des vecteurs de synchronisations. De plus, ce processus contient plusieurs variables locales, qui stockent notamment des informations à propos des demandes de verrouillage.

```
process MANAGER [SEND, RECV : com, ACTION : annonce]
                (task : DLC_ID, statespace : state_list, map : sync_map)
is
  var
    manager    : manager_state, -- état du manager
    actions    : action_set,    -- actions possibles pour la tâche
    arriv_list : arrival_list,  -- liste de paires (action, destination)
    taskstate  : nat,           -- état courant de la tâche
    waitlock   : lock_list,     -- verrous en attente
    lock       : lock,          -- verrou actif
    action     : dlc_action,    -- prochaine action à réaliser
    internal   : bool,          -- une action interne est possible
    sigpurge   : bool,          -- on doit se signaler a la purge
    -- variables de stockage temporaire
    n          : nat,
    l          : lock,
    to, gate   : DLC_ID,
    vect       : id_set
  in
```

Le manager a accès aux actions possibles de sa tâche en connaissant l'espace d'états ainsi que l'état courant de sa tâche. Au début, l'état courant de tâche est mis à jour à l'état initial de la tâche, qui est toujours zéro. De plus, le manager commence avec aucun verrou en attente.

```
-- initialisation
taskstate := 0;
waitlock  := {};
```

Nous entrons ensuite dans la boucle principale du manager, qui ne termine jamais. À chaque début de cette boucle, le manager est réinitialisé : il est dans son état libre (c'est-à-dire non verrouillé), et nous ne savons pas si la tâche autorise une action interne ou non. Certaines variables sont initialisées à une valeur par défaut dans le seul but d'éviter une alerte à la compilation.

```
-- boucle principale
loop
   -- Réinitialisation du manager
   manager := free;
   internal := false;
   -- Evite les warnings du compilateur
   action   := action (DLC_NULL_ID);
   sigpurge := false;
```

Le manager récupère ensuite les actions possibles de la tâche selon l'état courant de celle-ci, en consultant l'espace d'états de la tâche. Pour une raison d'équivalence de comportement qui sera détaillée au chapitre 5, pour chaque action qui permet de rejoindre plusieurs états différents de manière non déterministe, il faut tirer au hasard dès maintenant lequel de ces états sera atteint si cette action est réalisée. Ensuite, si la tâche est prête sur une seule action, et que cette action n'est pas interne, alors le manager s'auto-verrouille et permet de se signaler comme à purger. Le manager envoie ensuite un message "ready" à la porte de chaque action. Dans le cas d'une implémentation réelle, quand plusieurs actions sont pour la même porte avec des offres différentes, nous factorisons ces actions pour n'envoyer qu'un seul message "ready" à la porte concernée.

```
-- Récupère les actions possibles pour la tâche
actions := possible_actions (statespace, taskstate);

-- le choix interne est résolu dès maintenant : lorsqu'une action peut mener
-- à plusieurs états différents, on choisit quel état sera atteint si cette
-- action a lieu.
arriv_list := {};
for n := 1 while n <= length (actions) by n := n+1 loop
   var dest_set : nat_set, dest : nat, act : dlc_action in
      act      := access (actions, n);
      dest_set := get_next (statespace, taskstate, act);
      -- on choisi au hasard un etat de destination
      dest     := any nat where member (dest, dest_set);
      arriv_list := cons ( arrival (act, dest), arriv_list )
   end var
end loop;

if (length (actions) == 1) and ((get_gate (access (actions, 1))) != DLC_GATE_I)
then
```

```
        -- Auto-verrouillage
        action := access (actions, 1);
        SEND (action.gate, READY (true));
        manager := autolock_free;
        sigpurge := true
    else
        for n := 1 while n <= length (actions) by n := n+1 loop
            gate := get_gate (access (actions, n));
            if (gate == DLC_GATE_I) then
                internal := true
            else
                SEND (gate, READY (false))
            end if
        end loop
    end if;

    -- Boucle de négociation
    loop NEGOCIATION in
        select
```

Le manager rentre ensuite dans une boucle de négociation, qui est constituée d'un choix non déterministe entre plusieurs comportements. Les sous-sections suivantes décrivent chacun de ces comportements.

Réception d'un message "lock"

À la réception d'une demande de verrou, le manager stocke simplement cette demande dans la file des demandes en attente.

```
-- Reçoit un message LOCK
RECV (? any DLC_ID, ?LOCK (l) of message);
waitlock := append (l, waitlock)
```

Traitement d'une demande de verrou

Le manager traite la plus ancienne demande de verrou de sa file d'attente. Ce traitement est possible si le manager est libre, mais aussi s'il est auto-verrouillé, pour les raisons discutées dans la section 2.4. Ensuite, le manager vérifie que la demande de verrouillage est valide, c'est-à-dire que l'action du verrou est couramment possible par la tâche. Si c'est le cas et que la tâche est auto-verrouillée, alors celle-ci est ajoutée au champ "purge" de la demande de verrou. La variable "sigpurge" permet au manager de ne se signaler, dans son état courant, qu'une seule fois dans la purge : si le manager se signalait plusieurs fois à purger, la porte pourrait considérer trop de messages "ready(locked)" comme à purger. Le manager marque aussi l'action de la demande de verrou comme l'action candidate à effectuer.

```
[]
  -- Traitement du plus vieux verrou en attente
  only if not (empty (waitlock))
     and ((manager == free) or (manager == autolock_free))
  then
     lock      := head (waitlock);
     waitlock  := tail (waitlock);

     if member (lock.action, actions) then

        -- on se signale au plus une fois à purger
        if (manager == autolock_free) and (sigpurge) then
           lock := lock.{purge => cons (task, lock.purge)};
           sigpurge := false
        end if;

        action := lock.action;
```

Le manager teste ensuite si sa tâche correspond à la dernière tâche du chemin de verrouillage. Dans ce cas, si le verrou requiert une confirmation de la porte, le manager transmet le verrou à la porte et se place dans l'état verrouillé. Sinon, le manager peut conclure la négociation en envoyant les messages de résultat puis en quittant la boucle de négociation.

```
        if task == access (lock.path, length (lock.path)) then
           -- Fin du chemin de verrouillage
           if lock.confirm then
              -- Demande confirmation à la porte
              SEND (lock.action.gate, LOCK (lock));
              eval lock_state (!?manager)
           else
              -- Conclut la négociation
              vect := get_sync_vect (lock, map);
              ACTION (lock.action.gate, vect);
              SEND (lock.action.gate, COMMIT (lock.purge));
              for n := 1 while n <= length(vect) by n := n+1 loop
                 to := access(vect, n);
                 if to != task then
                    SEND (to, COMMIT)
                 end if
              end loop;
              break NEGOCIATION
           end if
```

Lorsque la tâche associée au manager n'est pas en fin de chemin de verrouillage, la demande de verrou est transmise à la prochaine tâche sur le chemin, et le manager se place dans l'état verrouillé.

```
        else
           -- Transmission du verrou
           to := next_task (task, lock.path);
           SEND (to, LOCK (lock));
```

```
            eval lock_state (!?manager)
        end if
```

Enfin pour cette branche, lorsque la demande de verrou n'est pas valide, elle est refusée. Le manager diffuse alors un message "abort" à la porte concernée ainsi qu'aux tâches qui ont été verrouillées par cette demande.

```
    else
        -- Refus du verrou
        SEND (lock.action.gate, ABORT (lock.purge));
        for n := 1 while n <= length (lock.path) by n := n+1 loop
            to := access (lock.path, n);
            if to < task then
                SEND (to, ABORT)
            end if
        end loop
    end if
end if
```

Réception d'un message "commit"

Un manager peut recevoir un message "commit" dès lors qu'il n'est pas dans son état libre. Dans une implémentation réelle, si des offres sont en jeu, il est possible de vérifier par prudence que les offres du message "commit" sont bien compatibles avec celles de l'action sur laquelle le manager est verrouillé. À la réception de ce message, la boucle de négociation est terminée.

```
[]
    -- Reçoit un message COMMIT
    only if manager != free then
        RECV (? any DLC_ID, COMMIT);
        break NEGOCIATION
    end if
```

Réception d'un message "abort"

À la réception d'un message "abort", ce qui est possible uniquement si le manager a accepté une demande de verrou, le manager est averti que la négociation a échoué. Il repasse dans un état à partir duquel il peut accepter une autre demande de verrou.

```
[]
    -- Reçoit un message ABORT
    only if (manager == locked) or (manager == autolock_locked) then
        RECV (? any DLC_ID, ABORT);
        if manager == locked then
            manager := free
        elsif manager == autolock_locked then
```

```
                manager := autolock_free
            end if
        end if
```

Réalisation d'une action interne

Enfin la dernière branche possible de la boucle de négociation est la décision d'effectuer une action interne. Cela est possible à condition que le manager soit libre et qu'une action interne soit possible. Si le manager prend cette décision, il annonce une action interne de la part de sa tâche et sort de la boucle de négociation.

```
        []
            -- Réalise une action interne
            only if (manager == free) and (internal) then
                ACTION (DLC_GATE_I, {task} of id_set);
                action := action (DLC_GATE_I);
                break NEGOCIATION
            end if
        end select
    end loop; -- Fin de boucle NEGOCIATION
```

Fin de boucle principale

Le manager quitte la boucle de négociation seulement quand il a décidé d'une action à effectuer pour sa tâche. À la sortie de cette boucle, le manager commence par refuser les éventuelles demandes de verrou en attente. Puis, le manager récupère l'état d'arrivée correspondant à l'action réalisée. Enfin, le manager recommence sa boucle principale avec sa tâche dans un nouvel état.

```
        while not (empty (waitlock)) loop
            l := head (waitlock);
            waitlock := tail (waitlock);
            SEND (l.action.gate, ABORT (l.purge));
            for n := 1 while n < length (l.path) by n := n+1 loop
                to := access (l.path, n);
                if to < task then
                    SEND (to, ABORT)
                end if
            end loop
        end loop;

        -- Récupère l'état d'arrivée choisi pour l'action réalisée
        taskstate := arrival_state ( arriv_list , action)

    end loop -- Fin de boucle principale
  end var
end process
```

Nous remarquons qu'une fois un état puits atteint par la tâche, le manager rentre dans sa boucle de négociation sans envoyer de nouveau message "ready". Il reste alors indéfiniment dans sa boucle de négociation, ce qui lui permet de refuser les éventuelles demandes de verrou qui peuvent encore lui parvenir.

Chapitre 3

Interaction avec l'environnement

Ce chapitre présente le mécanisme que nous avons mis en place pour permettre l'interaction entre une implémentation générée par DLC et son environnement. Pour pouvoir être utilisée au sein d'un système réel, l'implémentation doit être en mesure d'interagir avec d'autres systèmes présents dans son environnement. Par exemple, un processus d'une implémentation doit pouvoir accéder au système de fichier de la machine sur laquelle il s'exécute, ou aller consulter une base de données distante, ou encore actionner un bras robotisé.

Nous avons équipé DLC d'un mécanisme de *fonctions crochets* (*hook functions*) qui permet d'embarquer au sein de l'implémentation générée des procédures définies en C par l'utilisateur. Les fonctions crochets sont *optionnelles*, elles ne sont pas requises par DLC pour pouvoir générer une implémentation à partir d'une spécification LNT. Ainsi, l'utilisateur définit uniquement les fonctions crochets dont il a besoin pour mettre en place une interaction, lorsque cela est nécessaire.

La première section de ce chapitre introduit les différents types de fonctions crochets, et leur place au sein de l'implémentation générée. La deuxième section illustre, sur un exemple, différents cas d'utilisation classiques des fonctions crochets.

3.1 Les fonctions crochets

Les fonctions crochets, parfois plus simplement nommées "crochets" dans la suite, sont des procédures écrites en C qui sont automatiquement embarquées dans l'implémentation. Les crochets sont déclenchés à des moments précis de l'exécution, en relation avec la réalisation d'actions sur porte. L'utilisateur de DLC est libre d'implémenter le corps de ces fonctions crochets comme bon lui semble, ce qui lui permet de mettre en place une interaction entre l'implémentation et son environnement. Nous avons défini trois types de fonctions crochets :

les crochets *pré-négociation* et *post-négociation* peuvent être définis pour chaque porte, et le crochet *local* peut être défini pour chaque tâche.

Dans cette section, nous discutons tout d'abord le choix d'attacher les fonctions crochets aux actions du système. Nous présentons ensuite les trois différents types de crochets. Enfin, nous décrivons comment les crochets rendent possible l'échange de données entre le système et son environnement, et nous précisons quelles informations sont mises à disposition des fonctions crochets.

3.1.1 Effets de bord dans un programme LNT

Le langage LNT permet de définir le corps d'une fonction en C, ce qui permet d'optimiser certaines fonctions ou bien de réutiliser des fonctions C déjà existantes. Cependant, cette implémentation en C doit respecter l'hypothèse qu'une fonction LNT est pure du point de vue de l'environnement, dans le sens où elle ne réalise pas d'effet de bord en dehors du processus.

Nous illustrons la nécessité de cette hypothèse sur l'exemple suivant :

```
-- Déclaration d'une fonction LNT "f" implémentée en C par la fonction "ma_fonction_f"
function f (y: nat): nat is !external !implementedby "ma_fonction_f"
  null    -- le corps de la fonction est défini en C dans un autre fichier
end function

-- idem pour la fonction g
function g (z: nat): nat is !external !implementedby "ma_fonction_g"
  null
end function

process T [A, B: nat] is
   ...
   select
      x := f(y);
      A (x)
   []
      x := g(z);
      B (x)
   end select
   ...
end process
```

La déclaration des fonctions "f" et "g" précise que leur comportement est implémenté par les procédures C "`ma_fonction_f`" et "`ma_fonction_g`", respectivement. L'implémentation du corps de ces fonctions est réalisée par l'utilisateur, et celui-ci peut envisager d'utiliser ces fonctions pour interagir avec l'environnement et réaliser des effets de bord. Dans chaque branche du choix non déterministe, une de ces fonctions est nécessairement appelée pour obtenir la valeur de "x", qui est utilisée en offre des actions sur les portes A et B.

Lorsque la tâche liste ses actions possibles, si les fonctions “f” et “g” déclenchent chacune des effets de bord lorsqu'elles sont appelées, alors les effets de bord de chacune de ces deux fonctions sont réalisés. Or, cela ne respecte pas la sémantique du choix non déterministe, qui indique que la seule branche qui est exécutée est celle qui mène à l'action finalement réalisée par la tâche. Ce genre de situation illustre pourquoi les fonctions LNT ne sont pas adaptées pour réaliser des interactions avec l'environnement. De plus, le manuel Lnt2Lotos [CCG+14], dans sa section 2.5 “Including external C code”, met explicitement en garde contre l'utilisation de fonction LNT pour réaliser des effets de bords.

Les fonctions crochets évitent ce problème car elles sont rattachées aux *actions* sur porte, plutôt qu'à une fonction LNT. Les crochets sont déclenchés à des moments clés du protocole de synchronisation, autour de la réalisation d'une action. Nous avons jugé pertinent d'associer les fonctions crochets aux actions sur porte car les actions sont les événements observables par l'environnement. De plus, du point de vue d'un processus, une porte est un point de synchronisation et de communication avec l'extérieur, que ce soit avec un ou plusieurs autres processus, ou bien avec l'environnement en général.

3.1.2 Les trois types de fonctions crochets

Nous avons défini trois types de fonctions crochets, qui diffèrent notamment par le processus qui les appelle, le moment où elles sont appelées par rapport au déroulement du protocole, et enfin par une éventuelle valeur attendue en retour.

Crochet pré-négociation. Pour chaque porte du système, il est possible de définir un crochet pré-négociation associé à cette porte. Dans le déroulement du protocole, une porte appelle son crochet pré-négociation lorsqu'elle a reçu suffisamment de messages “ready” pour pouvoir démarrer une négociation. La porte passe en argument au crochet les caractéristiques de l'action pour laquelle elle envisage de lancer une négociation. La fonction crochet doit retourner un booléen qui indique si oui ou non une négociation doit être démarrée.

Crochet post-négociation. Il existe un crochet post-négociation associé à chaque processus porte pour laquelle un crochet pré-négociation est défini. Lorsque l'utilisateur déclare des fonctions crochets pour une porte, toutes les négociations pour les actions sur cette porte requièrent la confirmation de la porte en fin de négociation, comme nous l'avons décrit dans la section 2.3.6. Une porte appelle son crochet post-négociation lorsqu'elle a reçu une demande de confirmation. La fonction crochet reçoit en argument les caractéristiques de l'action pour laquelle la négociation a été conduite, et elle doit retourner un booléen qui indique si oui ou non cette action est réalisée par le système. La porte diffuse des messages de confirmation ou d'annulation de la négociation en fonction de la valeur retournée par le crochet post-négociation.

Crochet local. Il est possible de définir un crochet local pour chaque processus tâche. Une tâche appelle sa fonction crochet local à chaque fois qu'elle réalise une action, ou plus précisément à chaque fois qu'elle reçoit la confirmation de réussite d'une négociation.

On résume à quel moment chaque type de fonction crochet est appelé au cours d'une négociation en suivant le déroulement canonique du protocole :

1. Chaque tâche liste ses actions possibles, puis envoie un message "ready" aux portes sur lesquelles elle est prête à réaliser une action.
2. Lorsqu'une porte a détecté suffisamment de tâches prêtes à se synchroniser suivant un de ses vecteurs de synchronisation, elle appelle alors sa fonction crochet de pré-négociation pour savoir si une négociation peut être démarrée. La fonction crochet de pré-négociation permet principalement d'*éviter les négociations, rendues inutiles ou indésirables selon des facteurs externes à la spécification du système.*
3. À la première action pour laquelle le crochet de pré-négociation répond par vrai, la porte amorce une négociation en requérant une confirmation finale [1].
4. Si la négociation est un succès, alors la demande de verrou est renvoyée à la porte pour une confirmation finale. C'est à ce moment que la porte appelle son crochet de post-négociation, qui décide si l'action est réalisée ou non. La fonction crochet de post-négociation offre ainsi à l'utilisateur une primitive *qui a le dernier mot sur la réalisation d'une action du système.*
5. (a) Si l'action est validée par le crochet de post-négociation, la porte diffuse un message "commit" à toutes les tâches concernées par l'action. À la réception de ce message, chaque tâche déclenche son crochet local avant de passer dans son état suivant. La fonction de crochet local fournit un moyen d'*effectuer des interactions avec l'environnement au niveau d'une tâche.*

   (b) Si l'action est refusée par le crochet de post-négociation, la porte diffuse un message "abort" à toutes les tâches verrouillées durant la négociation. Ce cas correspond à l'annonce "HOOK_REFUSE" que nous n'avions pas vraiment détaillée dans la spécification LNT du comportement d'une porte, présentée en section 2.7.2.

Remarque 3-1

Il est envisageable d'avoir une fonction crochet post-négociation qui est appelée non pas par la porte, mais plutôt par la dernière tâche du chemin de verrouillage. Cette variante évite notamment d'avoir à modifier le protocole de synchronisation pour permettre une confirmation par la porte. Nous avons cependant décidé de placer le crochet de post-négociation au niveau d'une porte car, en pratique, il est souvent utile que les crochets

1. Lorsqu'aucun crochet n'a été déclaré pour la porte, celle-ci ne demande pas de confirmation

de pré- et de post-négociation soient appelés depuis le même processus. Cela permet non seulement d'avoir le même "point de vue" sur le système, mais aussi de pouvoir facilement échanger des informations, par variables globales, entre les crochets de pré- et de post-négociation. ■

L'implémentation du corps des fonctions crochets est laissée libre à l'utilisateur. Celui-ci peut ainsi utiliser les crochets pour réaliser des interactions avec l'environnement à l'occasion d'une action du système. Les fonctions crochets offrent non seulement un moyen de suivre les actions réalisées par le système, mais elles permettent aussi de contrôler, dans une certaine mesure, quelles actions sont effectuées. En effet, les crochets de pré-négociation et de post-négociation peuvent interdire l'exécution d'une action en retournant "faux". Ces crochets sont comme des gardes externes au système : ils ne créent pas d'autres comportements que ceux possibles selon la spécification, ils peuvent uniquement interdire la réalisation de certains de ces comportements.

3.1.3 Échange de données

Les fonctions crochets peuvent échanger des données entre le système et son environnement grâce aux offres des actions. Plus précisément, un crochet reçoit en argument une structure de données qui décrit une action, et qui permet d'accéder aux offres de cette action. Une fonction crochet peut ainsi, soit récupérer une donnée du système en accédant à la valeur d'une offre, soit passer une donnée au système en affectant la valeur d'une offre.

En pratique, les structures de données qui représentent une offre et une action du système sont définies en C de la manière suivante :

```
typedef struct DLC_OFFER DLC_OFFER;
typedef struct DLC_ACTION DLC_ACTION;

struct DLC_OFFER {
    DLC_MODE   mode;    // valeurs possibles : DLC_MODE_SEND, DLC_MODE_RECV
    DLC_NAT    type;    // identifiant de type
    DLC_VALUE  value;   // valeur de l'offre, valide si mode == DLC_MODE_SEND
};

struct DLC_ACTION {
    DLC_NAT    gate;    // identifiant de porte
    DLC_NAT    nboffer; // nombre d'offres
    DLC_OFFER *offer;   // tableau d'offres
};
```

Chaque fonction crochet reçoit en arguments un pointeur vers une structure de données "`DLC_ACTION`", qui décrit l'action en cours de traitement. Les offres associées à cette action résultent de la fusion des offres des tâches concernées par la négociation. L'utilisateur peut aisément accéder aux champs d'offres pour récupérer une valeur, et ensuite utiliser cette valeur lors d'une interaction avec l'environnement.

Pour passer une valeur depuis l'environnement vers le système, nous utilisons des offres en mode réception. Nous traitons ici le cas où la fusion des offres des tâches produit des offres dont certaines sont encore en mode réception ; ce cas avait été laissé en suspens à la section 2.3.5. Les fonctions crochets de pré- et de post-négociation peuvent accéder à ces offres et en modifier le mode et la valeur pour les transformer en offres en mode envoi.

Considérons par exemple une tâche voulant réceptionner une valeur entière dans une variable "n" via une action sur une porte A :

```
process ... is
   ...
   A (?n)
   ...
end process
```

Si aucune autre tâche ne fixe de valeur pour cette offre, alors l'offre est toujours en mode réception lorsque les crochets de pré- ou de post-négociation y ont accès. L'utilisateur peut alors employer les fonctions crochets pour fixer une valeur à cette offre, par exemple avec le code C suivant :

```
// la variable "action" est un pointeur vers une structure DLC_ACTION
action->offre[0].value = ... ; // affecte une valeur, lue dans l'environnement
action->offre[0].mode = DLC_MODE_SEND; // passe l'offre en mode "envoi"
```

Une fonction de crochet local n'est pas autorisée à modifier des offres car elle est appelée au niveau d'une seule tâche, or une action peut concerner plusieurs tâches, et il faut garantir que toutes ces tâches perçoivent les mêmes offres pour cette action. En conséquence, seules les fonctions crochets de pré- et de post-négociation, qui sont appelées alors que l'action est encore en cours de négociation, peuvent modifier des offres.

Cas des offres en mode réception non mises à jour par les fonctions crochets. Les fonctions crochets, qui sont optionnelles de manière générale, sont requises pour fixer une valeur aux offres qui restent en mode réception après la fusion des offres listées par les tâches. L'implémentation générée par DLC effectue des vérifications après avoir appelé les crochets de pré- et de post-négociation. Plus particulièrement, chaque porte vérifie que les offres de l'action avant et après l'appel à un de ces deux crochets sont compatibles entre elles. Cette compatibilité garantit que la valeur des offres en mode envoi n'a pas été modifiée par le crochet. Enfin, l'implémentation lève une erreur à l'exécution s'il reste une ou plusieurs offres en mode réception après avoir appelé le crochet de post-négociation. Les fonctions crochets, qui sont optionnelles de manière générale, sont ainsi nécessaires pour fixer une valeur aux offres qui restent en mode réception après la fusion des offres listées par les tâches.

3.1.4 Accès aux autres informations de la négociation

En plus de la description de l'action en cours de négociation, les crochets de pré- et de post-négociation reçoivent aussi en argument un pointeur vers une structure définie de la manière suivante :

```
typedef struct DLC_HOOK_INFO DLC_HOOK_INFO;

struct DLC_HOOK_INFO {
    DLC_NAT *vect;   // tableau d'identifiants des tâches du vecteur de synchro.
    DLC_NAT vectsize; // taille du vecteur de synchronisation
    Bool   *newrdy;  // newrdy[i] vrai : la porte a reçu "ready" de la tâche i
};
```

Chaque tâche du système est identifiée de manière unique par un entier naturel, qu'on appelle son identifiant de tâche. Les deux premiers champs de la structure permettent d'avoir accès au vecteur de synchronisation de la négociation : le tableau "`vect`" contient l'identifiant des tâches qui font parties de ce vecteur, et la variable "`vectsize`" stocke la taille de ce tableau.

Le tableau "`newrdy`" indique si la porte vient de recevoir un message "ready" de la part d'une des tâches du vecteur de synchronisation. Plus précisément, la valeur de "`newrdy[i]`" est vraie si :

— la tâche "`i`" fait partie du vecteur de synchronisation
— la porte a reçu un message "ready" de la part de la tâche "`i`" depuis la dernière fois qu'un crochet pré- ou post-négociation a été appelé pour une négociation dont le vecteur de synchronisation contient la tâche "`i`".

Le rôle du tableau "`newrdy`" est de pouvoir détecter lorsqu'un message "ready" est reçu par la porte durant une négociation. L'intérêt pratique de ce tableau est présenté dans l'exemple de la section 3.2.5 qui traite des actions retardées.

3.2 Exemples d'utilisation des fonctions crochets

Dans cette section, nous illustrons différents cas d'utilisation classiques des fonctions crochets. Nous nous appuyons sur un exemple de système de protection par de multiples cadenas numériques.

On veut protéger l'accès à un coffre par cinq cadenas de manière à ce que le coffre soit accessible uniquement lorsque trois cadenas sur cinq sont simultanément ouverts. Chaque cadenas est caractérisé par une clé correcte, que l'on représente par un simple entier naturel dans notre exemple. Lorsqu'on insère une clé correcte dans le cadenas auquel elle est associée, celui-ci peut accepter d'ouvrir le coffre dans un délai de trois secondes, après quoi le cadenas requiert d'insérer à nouveau la clé. Pour ouvrir le coffre, il faut donc insérer

dans trois cadenas la clé correcte correspondante, le tout dans un délai de trois secondes. De plus, chaque cadenas mémorise le nombre de tentatives d'accès qui ont été réalisées.

Ce système de cadenas mutliples peut être spécifié de la manière suivante en LNT :

```
channel two_nat is (nat, nat) end channel

process CADENAS
   [INSERER, NB_ACCES: two_nat, OUVRIR, FERMER, DELAI: none]
   (id, cle_correcte : nat)
is
   var
      acces : nat, -- compteur de tentatives d'accès
      cle   : nat  -- stocke temporairement une clé
   in
      acces := 0;
      loop -- boucle principale
         select
            INSERER (id, ?cle);
            acces := acces+1;
            if cle == cle_correcte then
               select
                  OUVRIR; -- ouverture du coffre
                  FERMER -- fermeture du coffre
               []
                  DELAI  -- action retardée, interdit l'ouverture après un délai
               end select
            end if
         []
            NB_ACCES (id, acces)
         end select
      end loop
   end var
end process

-- Composition
par OUVRIR#3 , FERMER#3 in
   CADENAS [INSERER, NB_ACCES, OUVRIR, FERMER, DELAI] (1, 12)
|| CADENAS [INSERER, NB_ACCES, OUVRIR, FERMER, DELAI] (2, 34)
|| CADENAS [INSERER, NB_ACCES, OUVRIR, FERMER, DELAI] (3, 56)
|| CADENAS [INSERER, NB_ACCES, OUVRIR, FERMER, DELAI] (4, 78)
|| CADENAS [INSERER, NB_ACCES, OUVRIR, FERMER, DELAI] (5, 90)
end par
```

La composition parallèle impose que les actions sur les portes OUVRIR et FERMER synchronisent trois cadenas parmi les cinq. Chaque cadenas, identifié par un entier naturel, peut recevoir une clé via une action sur la porte INSERER. Si cette clé est correcte, alors le cadenas peut effectuer une action sur la porte OUVRIR, mais elle peut aussi effectuer une action sur la porte DELAI, suite à quoi elle n'est plus prête à ouvrir le coffre. Enfin, un cadenas maintient un compteur du nombre de tentatives d'accès, dont la valeur peut

être consultée par une action sur la porte NB_ACCES. La suite de cette section illustre l'utilisation de fonctions crochets sur ce système.

3.2.1 Réalisation d'un effet de bord dans l'environnement

Les fonctions crochets permettent de modifier l'état de l'environnement. Pour cela, l'utilisateur peut insérer au sein du corps des fonctions crochets des appels à des procédures qui ont un effet sur l'environnement.

Dans notre exemple, on souhaite qu'une action sur la porte OUVRIR ouvre effectivement un coffre, qui est un élément de l'environnement et qui n'est pas explicitement modélisé dans la spécification. Pour réaliser cet effet de bord, le crochet de post-négociation de la porte OUVRIR est le plus approprié car il s'exécute quand on est certain qu'une action sur la porte OUVRIR a lieu.

```
bool DLC_OUVRIR_POST_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    // Effet de bord dans l'environnement: ouverture du coffre
    ouvrir_coffre ();

    return TRUE; // Réalise l'action
}
```

De manière similaire, le coffre est refermé lors d'une action sur la porte FERMER.

Cet exemple expose aussi l'utilité de rattacher certains crochets aux portes pour avoir un point de vue global sur une action qui concerne plusieurs processus. Une action sur la porte OUVRIR met en jeu trois processus cadenas, chaque action sur cette porte déclenche donc autant de crochets locaux. Or, on souhaite parfois avoir un seul point d'interaction associé à une action, indépendamment du nombre de processus qui prennent part à cette action. C'est le cas dans notre exemple qui ne contient qu'un seul coffre, et le crochet de post-négociation offre un point unique de définition d'effet de bord pour une action qui synchronise plusieurs processus cadenas.

3.2.2 Contrôle de la réalisation d'une action

Les fonctions crochets de pré- et de post-négociation permettent de contrôler quelles actions sont autorisées à être réalisées. Nous illustrons cette possibilité sur la porte NB_ACCES : d'après la spécification formelle du processus cadenas, une action sur cette porte est possible lorsque la boucle principale débute. Si une action sur la porte NB_ACCES est possible sans contrainte, un processus cadenas peut boucler sur des réalisations successives de cette action. Cependant, nous souhaitons contrôler la réalisation de cette action et ne l'autoriser que lorsqu'une requête de consultation de nombre d'accès est en attente. La présence d'une

requête en attente est un exemple d'état de l'environnement du système qui influe sur la réalisation d'actions.

Le crochet de pré-négociation de la porte NB_ACCES est utilisé pour interdire les négociations si aucune requête de consultation n'a été détectée, ou bien si la requête concerne un cadenas différent de celui qui s'identifie par la première offre de l'action.

```
bool DLC_NB_ACCES_PRE_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    // Consulte l'environnement pour une éventuelle requête en attente
    int cadenas = requete_consultation_en_attente ();

    if (cadenas == -1) {
         // pas de requête, refuse de démarrer une négociation
        return FALSE;
    } else {
        // autorise une négociation si l'identifiant de cadenas de la requête
        // correspond à celui de la première offre de l'action
        return cadenas == (int) action->offer[0].value;
    }
}
```

De manière similaire, il suffit à la fonction crochet de post-négociation de retourner faux pour empêcher la réalisation d'une action s'il n'est pas souhaitable de la réaliser étant donné l'état actuel de l'environnement. Le fait d'interdire une action dès le crochet de pré-négociation permet d'éviter des négociations, qui peuvent entraver le déroulement d'autres négociations et donc dégrader les performances du système, lorsque l'action visée ne sera de toute manière pas autorisée par l'environnement.

3.2.3 Envoi de données vers l'environnement

Les trois types de fonctions crochets peuvent envoyer des données du système vers l'environnement en récupérant la valeur des offres d'une action. Nous continuons d'utiliser la porte NB_ACCES pour illustrer ce mécanisme. La fonction crochet de post-négociation de cette porte n'a pas à vérifier qu'il existe une requête en attente, car le crochet de pré-négociation s'en est chargé. Le crochet de post-négociation va accéder à la deuxième offre de l'action pour récupérer le nombre de tentatives d'accès effectuées sur le cadenas, et transmettre cette valeur à un processus tiers dans l'environnement. De plus, cette fonction crochet marque la requête en attente comme traitée.

```
bool DLC_NB_ACCES_POST_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    // Le nombre de tentatives d'accès est passé dans la deuxième offre
    DLC_VALUE nb_tentatives = action->offer[1].value;

    envoi_a_un_process_de_l_environnement (nb_tentatives);
    requete_consultation_en_attente_traitee ();
```

```
    return TRUE;
}
```

Le nombre de tentatives d'accès au cadenas est aussi disponible au crochet local du cadenas, qui peut l'utiliser pour effectuer un effet de bord local au processus cadenas, comme par exemple une écriture dans un fichier de log propre à la tâche.

3.2.4 Réception de données depuis l'environnement

Les fonctions crochets permettent de passer des données depuis l'environnement vers le système, en utilisant les offres en mode réception. Nous employons cette fois les actions sur la porte INSERER pour illustrer cette fonctionnalité.

Une action sur la porte INSERER permet d'essayer d'ouvrir un cadenas à l'aide d'une clé. Cette opération nécessite de passer une clé, représentée ici par un entier naturel, depuis l'environnement vers un processus cadenas. Pour réaliser ce transfert de donnée, nous utilisons les fonctions crochets de la porte INSERER : le crochet de pré-négociation va autoriser une négociation uniquement quand une requête d'insertion de clé correspond à l'action visée sur INSERER.

```
struct requete_cle {
    DLC_NAT  cadenas;
    DLC_VALUE cle;
};

bool DLC_INSERER_PRE_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    struct requete_cle *rc = requete_cle_en_attente ();

    if (rc == NULL) {
        // pas de requête en attente
        return FALSE;
    } else {
        // autorise uniquement si l'identifiant de cadenas est cohérent
        return rc->cadenas == action->offer[0].value;
    }
}
```

Ensuite, le crochet de post-négociation va fixer la valeur de la clé à passer au cadenas, et marquer la requête comme traitée.

```
bool DLC_INSERER_POST_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    struct requete_cle *rc = requete_cle_en_attente ();

    // Fixe la valeur de l'offre correspondant à la clé
    action->offer[1].value = rc->cle;
```

```
    // Change le mode en "envoi"
    action->offer[1].mode = DLC_MODE_SEND;

    // Marque la requête comme traitée
    requete_cle_en_attente_traitee ();

    return TRUE;
}
```

Lors d'une action sur la porte INSERER, un cadenas va ainsi récupérer la clé issue de l'environnement dans sa variable "cle" utilisée en réception en deuxième offre.

3.2.5 Autoriser une action après un délai

La capacité à limiter les actions possibles peut être employée pour retarder la réalisation d'une action, et ainsi offrir un mécanisme de délai (*timeout*) dans l'implémentation générée. Nous présentons comment mettre en place une action retardée sur la porte DELAI de notre exemple.

Le crochet de pré-négociation est utilisé pour interdire une action sur la porte DELAI pendant une certaine durée, ici trois secondes. Pour cela, le crochet stocke la date de début d'attente de chaque cadenas dans un tableau en variable globale, et autorise le lancement d'une négociation uniquement lorsque le cadenas concerné attend depuis au moins trois secondes. Cependant, un cadenas peut être prêt sur la porte DELAI, puis effectuer une autre action et être prêt à nouveau sur cette porte, le tout en moins de trois secondes. C'est ici que l'information sur les réceptions de messages "ready" par la porte est utile : le crochet peut savoir si un cadenas s'est à nouveau manifesté comme prêt, auquel cas la date de début d'attente est mise à jour.

Étant donné qu'une action sur la porte DELAI ne concerne qu'un seul cadenas à la fois, le vecteur de synchronisation ne contient qu'un seul identifiant de tâche, qui correspond au cadenas visé par la négociation. Le crochet de pré-négociation récupère ainsi cet identifiant de tâche via le vecteur de synchronisation, met à jour le début d'attente si ce cadenas est nouvellement prêt, et autorise le démarrage d'une négociation si le cadenas attend depuis plus de trois secondes.

```
#define DELAI_ATTENTE 3

// Dates de début d'attente, 0 == jamais commencé d'attendre
time_t debut[DLC_SPEC_MAX_TASK];

bool DLC_DELAI_PRE_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    DLC_NAT cadenas; // identifiant de tâche du cadenas concerné par l'action
    time_t duree_attente;

    // L'unique identifiant de tâche du vecteur de synchronisation
```

```
    // correspond au cadenas visé pour une action sur la porte DELAI
    assert (hookinfo->vectsize == 1);
    cadenas = hookinfo->vect[0];

    // Mise à jour date de début d'attente
    if (hookinfo->newrdy[cadenas]) {
        debut[cadenas] = time();
    } else {
        // un cadenas qui ne vient pas d'envoyer un message "ready"
        // est forcément déjà en train d'attendre
        assert (debut[cadenas] != 0);
    }

    // Autorise une négociation si le cadenas attend depuis suffisamment longtemps
    duree_attente = time() - debut[cadenas];
    return duree_attente >= DELAI_ATTENTE;
}
```

Entre le moment où le crochet de pré-négociation autorise le démarrage d'une négociation pour un cadenas et le moment où la demande de confirmation de cette négociation atteint la porte, il se peut qu'un nouveau message "ready" de la part de ce cadenas soit reçu par la porte. En conséquence, le crochet de post-négociation doit lui aussi vérifier que le cadenas concerné par l'action ne s'est pas manifesté comme de nouveau prêt durant la négociation. En d'autres termes, le comportement du crochet de post-négociation est en fait identique à celui de pré-négociation.

```
bool DLC_DELAI_POST_NEGOTIATION (DLC_ACTION *action, DLC_HOOK_INFO *hookinfo)
{
    return DLC_DELAI_PRE_NEGOTIATION (action, hookinfo);
}
```

Les fonctions crochets peuvent ainsi être utilisées pour implémenter un mécanisme de délai de réalisation d'action. Cependant, une fois le délai passé, il n'est pas garanti que l'action sur la porte DELAI ait lieu, car le cadenas a encore une chance de participer à une action sur la porte OUVRIR. De plus, le mécanisme de délai de réalisation d'action ne permet pas d'être précis, et son utilisation est inappropriée pour des systèmes avec de fortes contraintes temporelles. En effet, la mesure de durée est effectuée au niveau de la porte, et cette mesure démarre à partir du moment où la porte reçoit un message "ready" de la part du cadenas. Les délais résultants des transmissions de messages sur le réseau impactent donc la durée d'attente réelle.

Remarque 3-2

Lorsqu'un crochet de pré-négociation retourne faux, la porte relève d'éventuels nouveaux messages en réception, puis consulte les tâches prêtes pour détecter une action possible et appelle à nouveau la fonction crochet de pré-négociation pour la nouvelle action visée. Le mécanisme de délai mis en place au niveau du crochet de pré-négociation revient donc à une forme d'attente active, ce qui n'est pas une solution optimale.

De manière générale, nous pensons que le mécanisme de délai devrait être implémenté directement au niveau de la tâche. Nous avons envisagé d'ajouter un crochet pré-négociation au niveau de la tâche, puis nous nous sommes ravisés afin de minimiser le nombre de fonctions crochets, mais nous ne considérons pas la question des délais comme définitivement réglée. Pour alimenter de futurs développements sur ce point, on note qu'il existe des propositions d'ajout de notions de temps directement au sein du langage de spécification, notamment dans les travaux de M. Sighireanu [Sig99] dans le cadre du langage E-LOTOS. ■

Chapitre 4

Génération automatique d'implémentation distribuée

> Eventually, I decided that thinking was not getting me very far and it was time to try building.
>
> Rob Pike
> *The Text Editor sam*

Ce chapitre présente la manière dont nous procédons en pratique pour générer automatiquement une implémentation distribuée à partir d'une spécification LNT. En partant de la composition parallèle des processus tâches en LNT et des fonctions crochets en C, on génère plusieurs programmes C qui implémentent le comportement des tâches, le protocole de synchronisation et les appels aux fonctions crochets.

La figure 4.1 offre une vue d'ensemble de la génération d'implémentation, qui comprend plusieurs étapes réalisées automatiquement par DLC, en faisant plusieurs fois appel à des outils ou des bibliothèques de CADP. Sur le haut de la figure, on retrouve les entrées de DLC, c'est-à-dire la spécification LNT d'un système distribué formé d'une composition parallèle de tâches, et d'éventuelles fonctions crochets. Au bas de la figure est représentée l'implémentation distribuée, constituée de programmes tâches, de programmes portes, et d'un programme "nœud central" en charge de déployer le système. Au centre, les différentes phases de génération de code sont représentées, ainsi que la section dans laquelle chacune est traitée.

On commence par décrire la bibliothèque de support pour la programmation distribuée. Le flot de compilation d'une tâche est ensuite exposé, suivi par l'implémentation du protocole. On présente ensuite la bibliothèque qui représente les attributs de la spécification. Enfin, une dernière section couvre les limitations actuelles de DLC.

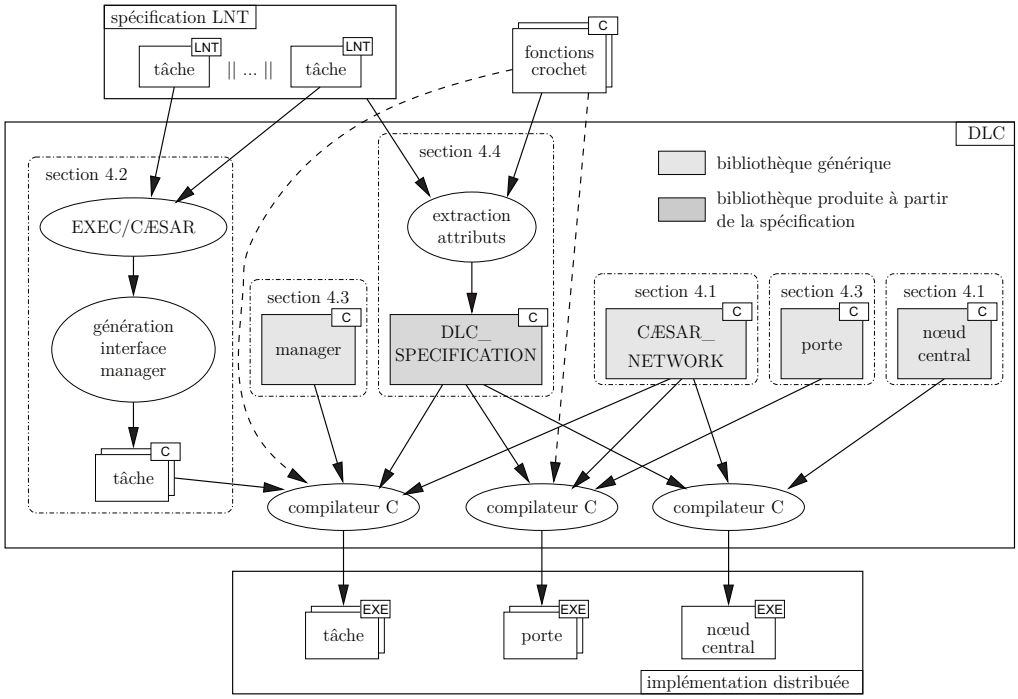


FIGURE 4.1 – Vue d'ensemble de la génération d'implémentation distribuée

4.1 La bibliothèque CÆSAR_NETWORK

DLC génère des implémentations qui utilisent la bibliothèque C "CÆSAR_NETWORK", développée au sein de l'équipe CONVECS. Cette bibliothèque offre notamment des primitives de communication, de déploiement et un mécanisme d'arrêt d'urgence qui sont très utiles pour le développement de programmes distribués. Elle est employée au sein des outils distribués de l'équipe, comme par exemple le générateur d'espace d'états DISTRIBUTOR [GMS13]. On donne ici une vue d'ensemble des primitives de la bibliothèque utilisées dans le cadre de DLC.

Nœuds. Un système distribué est composé de plusieurs programmes, qui sont appelés des *nœuds*. Chaque nœud possède un identifiant unique, qui est un nombre naturel. Le premier programme à être lancé est le nœud *central*, dont l'identifiant est toujours égal à zéro.

Déploiement. La bibliothèque permet d'automatiser le déploiement des nœuds sur une ou plusieurs machines. A son démarrage, le nœud central lit un fichier de configuration qui indique sur quelles machines le système doit être déployé, chaque machine pouvant héberger un ou plusieurs nœuds. Le nœud central se charge alors de copier les différents exécutables du système sur les machines indiquées et de lancer leur exécution. Il suffit donc de démarrer le nœud central pour déployer le système distribué sur une grille de machines. La bibliothèque utilise des utilitaires classiques

tels que `rsh` ou `ssh` pour accéder aux machines distantes, qui sont identifiées par leur nom DNS. Le fichier de configuration est en texte brut avec une syntaxe simple, ce qui permet de l'éditer facilement ou de le générer automatiquement. La configuration de déploiement peut donc être facilement modifiée entre différentes exécutions, sans avoir à recompiler le système.

Communication. La bibliothèque offre des primitives de communication entre nœuds par passage de messages asynchrones. En pratique, ces communications utilisent des sockets POSIX avec le protocole TCP pour garantir que les messages sont délivrés et qu'ils restent dans le même ordre entre un émetteur et un récepteur. La procédure d'initialisation de la bibliothèque établit une connexion entre toutes les paires de nœuds du système, chaque nœud est donc en mesure de communiquer avec n'importe quel autre.

La bibliothèque CÆSAR_NETWORK offre aussi un service de nommage : chaque nœud est désigné par son identifiant, qui correspond à sa place dans la liste des nœuds du fichier de configuration. Il est donc possible de connaître l'identifiant de chaque nœud en contrôlant l'ordre d'apparition des nœuds dans le fichier de configuration. Cela permet de nommer les nœuds indépendamment des machines sur lesquelles ils seront effectivement déployés. Ce nommage facilite la programmation du système distribué, en décorrélant l'identification des processus distants de la topologie de déploiement effectivement utilisée lors d'une exécution.

Arrêt d'urgence. Un mécanisme d'arrêt d'urgence permet à n'importe quel nœud de demander l'arrêt de tous les nœuds du système. En pratique, une requête d'arrêt d'urgence est transmise au nœud central, qui se charge ensuite de stopper tous les nœuds du système. Lorsqu'un nœud reçoit un signal d'arrêt d'urgence, il appelle une fonction définie par l'utilisateur avant de terminer ; cette fonction permet de préparer l'arrêt du nœud. Le mécanisme d'arrêt d'urgence est notamment utilisé pour stopper tous les nœuds quand l'un d'entre eux lève une erreur à l'exécution.

Outre la simplification de la programmation distribuée, la bibliothèque CÆSAR_NETWORK permet aussi d'isoler la couche de communication dans l'implémentation générée. Nous avons fait le choix de nous concentrer sur le protocole de synchronisation plutôt que sur les techniques d'implémentation efficace de passage de messages entre processus, qui sont un sujet de recherche à part entière. Le passage de messages repose sur les sockets POSIX ; si elles ne sont pas forcément les primitives les plus performantes, notamment entre deux processus déployés sur une même machine, elles ont l'avantage d'être disponibles par défaut sur tous les systèmes d'exploitation couverts par CADP, c'est-à-dire Solaris, Linux, OSX et Windows. Si nécessaire, l'isolement au sein de la bibliothèque CÆSAR_NETWORK de l'implémentation du passage de messages devrait permettre d'améliorer cette implémentation dans le futur sans avoir à modifier le code généré par DLC.

4.2 Implémentation d'une tâche

Chaque tâche est convertie en un programme C séquentiel à l'aide du compilateur EXEC/CÆSAR [GVZ01] disponible dans CADP. Ce programme C est ensuite automatiquement complété pour s'interfacer avec le protocole de synchronisation.

4.2.1 Le compilateur EXEC/CÆSAR

Le compilateur EXEC/CÆSAR prend en entrée une spécification et produit en sortie un programme C capable d'explorer un chemin d'exécution au sein de l'espace d'états de la spécification. Plus précisément, le programme C est capable d'énumérer toutes les actions possibles d'une spécification à partir de son état courant, et d'effectuer une de ces actions pour passer dans l'état suivant correspondant.

Cependant, le code C généré n'est pas complet : il manque la procédure de choix de l'action à effectuer, dont l'implémentation est laissée libre. Dans notre cas, la résolution de ce choix est effectué par le biais du protocole de synchronisation. Pour chaque tâche du système, on utilise donc le compilateur EXEC/CÆSAR pour obtenir un programme C, et on complète ce programme pour incorporer la partie manager du protocole de synchronisation.

4.2.2 Interface du programme généré par EXEC/CÆSAR

L'interface offerte par le programme généré par EXEC/CÆSAR varie selon la spécification donnée en entrée. Concrètement, l'interface produite par EXEC/CÆSAR est composée des fonctions suivantes :

- une fonction d'initialisation "`CAESAR_KERNEL_INIT`"
- une fonction de terminaison "`CAESAR_KERNEL_TERM`"
- une fonction de passage dans un état suivant "`CAESAR_KERNEL_NEXT`", qui permet aussi d'énumérer les actions possibles depuis l'état courant
- une *fonction porte* pour chaque porte présente dans la spécification ; ces fonctions portes sont la partie variable de l'interface, et EXEC/CÆSAR ne produit qu'un squelette pour l'implémentation du corps de ces fonctions

L'exécution de la fonction "`CAESAR_KERNEL_NEXT`" déclenche pour chaque action possible depuis l'état courant un appel à la fonction porte correspondante. Une fonction porte retourne un booléen qui indique si l'action doit être effectuée ou non. À la première fonction porte qui retourne vrai, la fonction "`CAESAR_KERNEL_NEXT`" effectue l'action correspondante et n'appelle pas d'autre fonction porte. Si toutes les fonctions portes retournent faux, la fonction "`CAESAR_KERNEL_NEXT`" termine sans effectuer d'action. La fonction "`CAESAR_KERNEL_NEXT`" retourne une valeur parmi les trois suivantes :

— "CAESAR_KERNEL_MOVE" si une fonction porte a répondu vrai, auquel cas le programme a effectué l'action correspondante et la tâche est maintenant dans l'état suivant
— "CAESAR_KERNEL_WAIT" si toutes les fonctions portes ont répondu faux, auquel cas la tâche reste dans le même état
— "CAESAR_KERNEL_STOP" si l'état courant de la tâche est un état puits, sans aucune transition sortante

Le programme généré par EXEC/CÆSAR fournit ainsi des primitives suffisantes pour explorer un chemin d'exécution dans l'espace d'états d'un processus LNT. Ce programme n'est cependant pas complet, il doit être complété par une fonction principale qui se charge d'appeller "CAESAR_KERNEL_INIT", "CAESAR_KERNEL_TERM" et "CAESAR_KERNEL_NEXT", ainsi que par l'implémentation du corps de chaque fonction porte.

4.2.3 Branchement avec la partie manager du protocole

La partie manager du protocole de synchronisation a besoin de connaître la liste des actions possibles depuis l'état courant de sa tâche, et d'indiquer quelle action doit être effectuée. L'interface du programme C généré par EXEC/CÆSAR, variable par ses fonctions portes, n'offre pas une base pratique pour brancher le module générique qui implémente la partie manager du protocole. Nous avons donc mis en place une nouvelle interface indépendante du nombre de portes de la spécification, qui est utilisée par la partie manager pour interagir avec sa tâche.

La mise en place de cette nouvelle interface nécessite notamment de compléter le corps des fonctions portes de chaque tâche. Concrètement, les fonctions portes sont utilisées soit pour collecter les actions possibles depuis l'état courant de la tâche (avant d'envoyer les messages "ready" correspondants), soit pour effectuer une action (après qu'une action ait été décidée via le protocole de synchronisation). On utilise la variable globale "DLC_GATE_FUNCTION_GOAL" pour différencier ces deux buts.

Collecte des actions possibles. La valeur "COLLECT_ACTION" est affectée à la variable globale "DLC_GATE_FUNCTION_GOAL", puis la fonction "CAESAR_KERNEL_NEXT" est exécutée. Chaque fonction porte appelée par "CAESAR_KERNEL_NEXT" va alors suivre l'indication donnée par la variable "DLC_GATE_FUNCTION_GOAL" : elle enregistre son action dans la liste des actions possibles et retourne faux. L'appel à "CAESAR_KERNEL_NEXT" va donc retourner "CAESAR_KERNEL_WAIT", on peut ensuite passer la liste des actions possibles qui vient d'être collectée à la partie manager. La partie manager utilise cette liste pour traiter les négociations du protocole de synchronisation, et décider quelle action doit être effectuée.

Réalisation d'une action. Pour réaliser l'action décidée par le protocole, on stocke cette action dans la variable globale "`CONFIRMED_ACTION`", on affecte la valeur "`EXECUTE_ACTION`" à la variable globale "`DLC_GATE_FUNCTION_GOAL`" et on appelle la fonction "`CAESAR_KERNEL_NEXT`". La variable "`DLC_GATE_FUNCTION_GOAL`" indique cette fois-ci à chaque fonction porte de tester si son action correspond à celle stockée dans la variable "`CONFIRMED_ACTION`", et de répondre vrai si c'est le cas. La fonction "`CAESAR_KERNEL_NEXT`" va alors effectuer l'action, passer dans l'état suivant et enfin retourner "`CAESAR_KERNEL_MOVE`".

Pour résumer, la nouvelle interface est donc composée d'une partie générique qui gère les variables globales et les appels à la fonction "`CAESAR_KERNEL_NEXT`", et de l'implémentation des fonctions portes. Pour chaque tâche, le squelette de l'implémentation des fonctions portes produit par EXEC/CÆSAR doit être complété. Une fonction porte reçoit ses offres en argument, et elle peut être appelée plusieurs fois avec des offres différentes. Le squelette contient du code qui permet d'aiguiller l'exécution selon les différents profils d'offres ; on complète donc ce squelette en plusieurs endroits pour traiter tous les profils possibles sur chaque porte. Cet ajout de code C est réalisé automatiquement lors de la génération de l'implémentation.

4.3 Implémentation du protocole

La logique du protocole est implémentée dans deux modules C qui correspondent au comportement générique d'une porte et d'un manager. Ces deux modules sont la traduction en C des spécifications LNT présentées dans la section 2.7. Ils comportent en plus la gestion des offres et les appels aux différentes fonctions crochets (si elles sont définies par l'utilisateur).

Cette implémentation est réalisée une fois pour toutes, les deux modules génériques étant ensuite repris au sein de l'implémentation générée automatiquement pour une spécification LNT de système distribué. Cette isolation de l'implémentation du cœur de la logique du protocole permet de gagner en confiance en l'absence de bogues, par rapport à une implémentation du protocole qui serait générée à chaque fois.

Étant donné que nous avons spécifié et, comme nous le verrons au chapitre 5, vérifié le comportement d'une porte et d'un manager en LNT, il semble pertinent d'utiliser le compilateur EXEC/CÆSAR pour obtenir un programme C qui implémente les processus porte et manager. Cette approche d'amorce (*bootstrap*) à partir de la spécification LNT est attirante, mais se révèle complexe en pratique : il faut non seulement gérer les offres, mais aussi s'interfacer avec la bibliothèque CÆSAR_NETWORK pour les communications distantes. De plus, la porte et la partie manager doivent appeler les fonctions crochets qui peuvent contenir des effets de bord, or nous avons vu en section 3.1.1 qu'il est difficile de placer des effets de bord au sein d'une spécification LNT autre part que sur une action.

Cependant, l'amorce de l'implémentation du protocole de synchronisation à partir de sa spécification, qui est une étape clé du *bootstrapping* de DLC à partir de LNT, est une piste de travail futur.

Nous avons donc choisi d'écrire directement à la main la logique du protocole en C pour les raisons suivantes :

— cette implémentation manuelle est un effort à réaliser une seule fois
— il est de toute manière nécessaire d'écrire une partie de l'implémentation du protocole en C, ne serait-ce que pour avoir accès aux primitives de communication à distance
— le protocole de synchronisation est un élément crucial de la performance de l'implémentation générée, et l'écrire directement en C offre un meilleur contrôle sur les performances en temps d'exécution mais aussi en gestion de la mémoire

Remarque 4-1

La génération d'implémentation distribuée pour BIP [Qui13] utilise une approche qui met en place le protocole de synchronisation au niveau de la spécification BIP. Une spécification BIP est transformée par l'injection de la logique du protocole de synchronisation, pour obtenir une nouvelle spécification BIP où les interactions entre processus se limitent à du passage de message entre deux processus uniquement. Ensuite, cette spécification est traduite vers du code exécutable, où les échanges de messages entre deux processus peuvent être aiséments transcrits sur des primitives bas niveau de passage de messages.

Cette approche vise à être "correcte par construction" en démontrant formellement l'équivalence entre le modèle BIP avant et après insertion du protocole. Cependant, cette équivalence est vérifiée pour un protocole de synchronisation naïf, mais pas pour le protocole α-core (corrigé) véritablement utilisé dans le générateur de code. ■

4.3.1 Choix non déterministe sur la réception de message

Dans la spécification LNT des processus porte et manager, la réception de messages du protocole se fait par une action sur la porte RECV. Le flot d'exécution du processus est aiguillé selon le type de message reçu grâce à un choix non déterministe entre plusieurs actions sur la porte RECV avec différents types de messages en offre. De plus, un processus peut refuser la réception de certains message selon son état courant. Par exemple, une porte accepte toujours de recevoir un message "ready", mais accepte un message "commit" uniquement si elle est en train de négocier :

```
-- Boucle principale du processus porte
loop
   select
      RECV (?task, ?READY(autolocked));
      ...
   []
```

```
        -- réception gardée par une condition
        only if state == dealing then
           RECV (?task, ?COMMIT (purge) of message);
           ...
     []
        ...
     end select
  end loop
```

Cette approche n'est pas directement transcriptible dans notre implémentation en C. À la place, notre implémentation accepte toujours de réceptionner n'importe quel type de message. Le flot d'exécution est aiguillé selon le type du message, après la réception. Dans la spécification LNT des portes et des managers, les réceptions gardées par des conditions indiquent que certains types de messages ne devraient jamais pouvoir être reçus dans certains états du processus. Dans notre implémentation en C, qui accepte tous les types de messages, la réception d'un type de message incompatible avec l'état courant d'un processus entraîne le déclenchement de la procédure d'arrêt d'urgence qui stoppe tout le système. L'extrait de spécification du processus porte ci-dessus se transcrit ainsi de la manière suivante dans notre implémentation :

```
while (DLC_RECV(message)) {
   switch (message->type) {
   case READY:
      ...
   case COMMIT:
      // arrêt d'urgence si la porte n'est pas en train de négocier
      DLC_ASSERT (GATE_STATE == DEALING);
      ...
   }
}
```

4.3.2 Compatibilité et fusion des offres

Lorsqu'une porte recherche une action réalisable, elle doit tester non seulement l'état des tâches mais aussi leurs offres. De plus, lorsque des offres compatibles sont détectées, la porte doit fusionner ces offres pour produire les offres de négociations.

Contrairement à la spécification du processus porte présenté à la section 2.7.2, notre implémentation du protocole traite bien les offres. Nous avons définis les fonctions de test de compatibilité entre deux offres, et de fusion de deux offres. Nous donnons ici une définition mathématique de ces fonctions. L'implémentation en C de ces fonctions est ensuite relativement directe.

Offre. Une offre est un triplet $o = (m, t, v)$, où :
- m est le mode de l'offre (envoi ou réception)
- t est le type de l'offre

— v est la valeur de l'offre si l'offre est en mode envoi, ou $\perp$ (indéfini) si l'offre est en mode réception

Compatibilité entre deux offres. Deux offres $o_1 = (m_1, t_1, v_1)$ et $o_2 = (m_2, t_2, v_2)$ sont dites *compatibles* si et seulement si :

— les deux offres ont le même type : $t_1 = t_2$

— si les deux offrent sont en mode envoi, alors leur valeur est égale : $(m_1 = m_2 = \text{envoi}) \Rightarrow v_1 = v_2$

Fusion de deux offres. Deux offres $o_1 = (t_1, m_1, v_1)$ et $o_2 = (t_2, m_2, v_2)$ peuvent être fusionnée si et seulement si elles sont compatibles, et leur fusion résulte en une nouvelle offre $o = (t, m, v)$ où :

$$t = t_1 = t_2$$

$$m = \begin{cases} \text{réception} & \text{si } m_1 = m_2 = \text{réception} \\ \text{envoi} & \text{si } m_1 = \text{envoi ou } m_2 = \text{envoi} \end{cases}$$

$$v = \begin{cases} v_1 & \text{si } m_1 = \text{envoi} \\ v_2 & \text{si } m_2 = \text{envoi} \\ \perp & \text{si } m_1 = m_2 = \text{réception} \end{cases}$$

4.3.3 Action interne et option de progrès maximal

Dans la spécification LNT du manager, la réalisation d'une action interne se fait dans une branche du choix non déterministe de la boucle principale. Pour remédier au manque d'opérateur de choix non déterministe en C, dans notre implémentation la réalisation d'une action interne est, dans le cas général, conditionnée par un tirage au hasard.

Lorsque seule une action interne est possible, notre implémentation prend le temps de refuser les éventuelles demandes de verrou qui lui sont parvenues. Ensuite, elle réalise directement l'action interne.

Lorsqu'une action interne et une ou plusieurs actions sur portes sont possibles, notre implémentation laisse la possibilité aux actions sur porte de se réaliser en attendant des demandes de verrou pour ces actions. Quand une demande de verrou est reçue, la tâche tire au hasard pour savoir si elle accepte cette demande de verrou, ou bien si elle la refuse et effectue une action interne. Il se peut que les négociations échouent, et que, comme nous en avons discuté dans la section 2.3.4, l'action interne soit finalement la seule action réalisable. Pour régler ce cas, si aucune négociation n'a abouti après un certain délai, et si la tâche n'est pas couramment verrouillée, alors la tâche effectue à coup sûr l'action interne.

Ce comportement peut être influencé par une option de *progression maximale* (*maximal progress*), qui peut être choisie par l'utilisateur au démarrage de l'implémentation. Cette option a pour effet de rendre les actions internes prioritaires devant les actions sur porte. En pratique, dès lors qu'une action interne est possible, notre implémentation refuse les éventuelles demandes de verrou en attente, puis elle réalise cette action interne. Si d'autres

actions sur porte étaient possibles, la tâche n'aura même pas envoyé de message "ready" pour ces actions.

4.4 La bibliothèque DLC_SPECIFICATION

Plusieurs informations relatives à la spécification passée en entrée à DLC sont regroupées dans la bibliothèque C DLC_SPECIFICATION afin d'être rendues accessibles aux différents programmes de l'implémentation. Parmi ces informations, on retrouve par exemple le nombre total de portes et de tâches, la présence de fonctions crochets sur une certaine porte ou encore les vecteurs de synchronisation de chaque porte.

L'en-tête de la bibliothèque DLC_SPECIFICATION est générique, il définit des fonctions qui sont appelées par le nœud central et les modules de porte et de manager du protocole. Seul le corps de cette bibliothèque est généré pour fournir à chaque fois une implémentation qui reflète les caractéristiques de la spécification. Les attributs de la spécification sont ainsi obtenus à l'exécution par les modules génériques via les primitives de la bibliothèque DLC_SPECIFICATION.

Il est important de regrouper ces informations dans une seule bibliothèque qui sera utilisée par tous les programmes de l'implémentation distribuée, notamment car ceux-ci échangent des références à ces informations par le réseau. Par exemple, les messages du protocole font référence à des identifiants de porte, de tâche, ou encore de type de données pour les offres, et ces identifiants doivent être cohérents entre tous les nœuds de l'implémentation. La bibliothèque DLC_SPECIFICATION regroupe ainsi la définition de tous ces identifiants.

La collecte des caractéristiques de la spécification ne nécessite pas la génération de l'espace d'états des tâches ou de la composition parallèle des tâches. Ainsi, il est possible de générer une implémentation distribuée même pour des spécifications dont l'espace d'états explose. En pratique, il est possible de mener des vérifications formelles sur des spécifications avec quelques tâches en parallèle ou en abstrayant certaines données, puis d'obtenir une implémentation pour une configuration plus grande en ajoutant des tâches à la composition parallèle.

4.5 Limitations actuelles de DLC

Les implémentations générées par DLC sont encore limitées sur quelques aspects. Cette section présente succinctement les trois limitations principales et les perspectives d'améliorations envisagées.

Absence de types “complexes” dans les offres

Les types de données “complexes” tels que les enregistrements, les listes ou les tableaux, sont utilisables dans les spécifications des tâches mais ils ne doivent pas apparaître dans les offres d'actions. En effet, pour le moment DLC accepte dans les offres uniquement des valeurs de types simples comme le type booléen, le type entier, ou encore les types énumérés définis par l'utilisateur. En cas de présence d'un type complexe, une erreur est levée dès la compilation.

Cette limitation est notamment due à la nécessité de transmettre les valeurs des offres sur le réseau. Pour pouvoir gérer les offres d'un type complexe, il faut être capable sérialiser (c'est-à-dire de représenter en une chaîne d'octets) n'importe quelle valeur de ce type, et de dé-sérialiser une chaîne d'octets en une valeur. L'implémentation en C des types et des fonctions LNT est automatiquement générée par les outils de CADP, mais cette implémentation ne contient pas de fonctions de sérialisation. Il est envisageable de générer ce genre de fonctions au niveau de DLC, mais il est délicat de traiter les types complexes dont la taille de la représentation des valeurs est variable. Nous estimons que la production des fonctions de sérialisation doit être assurée au niveau des outils CADP, qui ont déjà une connaissance précise de la manière dont chaque type LNT est représenté en C.

Absence de gestion des gardes de communication sur les actions

En LNT, une action est éventuellement gardée par une condition qui peut dépendre de variables du processus, parmi lesquelles certaines sont des offres en réception. DLC ne prend pas en compte ces gardes car l'interface EXEC/CÆSAR est trop limitée pour pouvoir les traiter correctement.

L'interface EXEC/CÆSAR permet d'indiquer une valeur pour une offre en mode réception. En cas de garde faisant référence à cette valeur, il n'est pas possible de tester si la valeur est valide pour la garde, sans réaliser l'action au passage, si cette valeur est effectivement valide. DLC ne peut donc pas tester une valeur d'offre en réception sans éventuellement déclencher la réalisation de l'action.

Par exemple, l'action suivante sur la porte A peut avoir lieu uniquement si l'entier “x”, reçu lors de l'action, est plus grand que l'entier “y” :

```
var x, y: nat in
   y := ... ;
   A (?x) where x > y;
   ...
end var
```

Le fait de ne pas pouvoir tester différentes valeurs de “x” par rapport à la garde sans éventuellement déclencher l'action rend impossible le test de différentes valeurs proposées pour différentes négociations.

Pour pouvoir traiter correctement les gardes, il faut modifier l'outil EXEC/CÆSAR afin que son interface donne accès aux conditions de gardes.

On note néanmoins qu'une garde qui ne dépend pas d'une valeur reçue lors de l'action peut être transformée en une pré-condition, et autorise ainsi l'utilisation de DLC. Par exemple, la garde suivante peut être tranformée en une condition précédent l'action :

```
-- La garde est indépendante de la valeur de "x" reçue pendant l'action ...
A (?x, y) where y == z

-- ... elle peut donc être transformée en une condition évaluée avant l'action
only if y == z then
   A (?x, y)
end if
```

Absence de création dynamique de tâches

Le nombre de tâches est une constante définie par la composition parallèle donnée en entrée. Une tâche qui contient elle-même une composition parallèle pourrait donner naissance à plusieurs nouvelles tâches de manière dynamique. Cependant, l'utilisation du compilateur EXEC/CÆSAR impose une implémentation séquentielle de cette composition parallèle au sein de la tâche. DLC ne gère donc pas la création dynamique de tâche, ce qui nécessiterait notamment de grands changements au sein du compilateur EXEC/CÆSAR. De plus, les outils de vérification de CADP gèrent uniquement des spécifications dont le nombre de processus est borné et connu statiquement. Néanmoins, il est déjà possible en de traiter de nombreux systèmes en pratique sans être confronté à cette limite.

Chapitre 5

Vérification formelle de protocole de synchronisation

> A program that produces incorrect results twice as fast is infinitely slower.
>
> John Ousterhout

Nous avons élaboré une méthode de vérification formelle de protocole de synchronisation. L'approche générale de notre méthode est de vérifier l'absence de mauvais comportements dans l'espace d'états du modèle d'une implémentation utilisant un protocole de synchronisation. La méthode est structurée en deux phases principales : dans un premier temps, nous produisons, à partir d'une spécification de système distribué, le modèle d'une implémentation de ce système. Dans un deuxième temps, nous conduisons plusieurs vérifications formelles sur ce modèle.

Nous avons commencer à développer cette méthode dès le début de la thèse, afin de pouvoir vérifier des protocoles existants. Nous avons ainsi analysé plusieurs protocoles de la littérature [EL13], en découvrant au passage que le protocole de Parrow et Sjödin peut mener à des interblocages en cas de communications asynchrones. Nous avons découvert en milieu de thèse qu'une méthode de vérification de protocole pour le rendez-vous multiple avaient déjà été proposée [GGvB+95]. Toutefois, cette méthode a été appliquée sur différents protocoles que ceux que nous avons retenu pour notre étude, et elle repose sur des outils qui ne sont plus maintenus aujourd'hui.

Notre méthode de vérification nous a ensuite été d'un grand soutien lors de la mise au point du protocole utilisé dans DLC. Elle n'a donc pas seulement été utilisée pour une étude a posteriori de protocoles existants, mais aussi directement au moment de l'élaboration de notre protocole. Nous avons ainsi pu itérer sur les différentes améliorations présentées au chapitre 2, en vérifiant à chaque étape que nous n'introduisions pas de problèmes au sein du protocole. De plus, nous avons continué d'affiner notre méthode tout au long de la thèse.

Dans ce chapitre, nous décrivons notre méthode de vérification et ses résultats. La première section présente la génération, à partir d'une spécification de système, du modèle de l'implémentation de ce système. La deuxième section couvre les vérifications formelles conduites sur ce modèle. Enfin, la troisième section expose les résultats de l'application de notre méthode.

5.1 Génération de modèle d'une implémentation

À partir de la spécification LNT d'un système distribué, on produit automatiquement une autre spécification LNT qui modélise l'implémentation du système. Cette implémentation utilise le protocole de synchronisation pour gérer les actions du système. Cette génération de code est proche de celle présentée au chapitre 4, à la différence qu'on produit cette fois un modèle LNT plutôt qu'une implémentation en C [1].

Le modèle d'une implémentation est composé des éléments suivants :
- le modèle de chaque tâche associée à un manager
- le modèle de chaque porte
- le modèle des communications par passage de messages asynchrones

Pour le modèle d'un manager et d'une porte, on utilise les processus LNT déjà exposés dans la section 2.7. Le processus manager requiert en argument l'espace d'états de sa tâche, et le manager et la porte nécessitent les vecteurs de synchronisation du système. La suite de cette section présente comment on obtient un modèle LNT de ces arguments, ainsi que la mise en place de communications asynchrones au sein du modèle.

5.1.1 Modèle d'une tâche associée à un manager

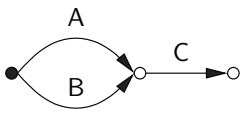
Le comportement d'un manager est spécifié par un processus LNT générique qui prend en argument l'espace d'états de la tâche pour laquelle il doit négocier. On doit donc générer, à partir du processus LNT qui décrit le comportement d'une tâche dans le système distribué, une structure de données LNT qui représente l'espace d'états de cette tâche.

Un espace d'états est un ensemble de transitions, dont chaque transition est un triplet contenant un état d'origine, une action, et un état de destination. Dans le cadre du modèle de l'implémentation, on gère uniquement les actions sans offres : une action se résume donc simplement à un identifiant de porte. Nous avons développé un programme qui s'appuie sur l'interface OPEN/CÆSAR pour produire automatiquement une structure de données modélisant le LTS d'une tâche.

1. Dans le cadre de BIP, une transformation de modèle a systématiquement lieu pour obtenir un modèle comportant un protocole de synchronisation. Ce modèle est ensuite transformé à son tour en une implémentation (voir la remarque dans la section 4.3)

Considérons par exemple la tâche FOO spécifiée de la manière suivante :

```
process FOO [A, B, C: none] is
   select
      A
   [] B
   end select;
   C;
   stop
end process
```



Notre programme génère automatiquement le modèle de l'espace d'états de cette tâche sous la forme d'une structure de données LNT retournée par une fonction :

```
function LTS_FOO : transition_set is
   return {
      transition  (0,  action  (DLC_GATE_A), 1),
      transition  (0,  action  (DLC_GATE_B), 1),
      transition  (1,  action  (DLC_GATE_C), 2)
   }
end function
```

De plus, nous avons aussi défini les fonctions qui permettent d'accéder aux actions possibles à partir de l'état courant de la tâche (“ possible_actions ”), et à la liste des états atteignables après une certaine action depuis l'état courant (“get_next”). La définition de ces fonctions est disponible dans l'Annexe A. Le modèle de la tâche FOO associée à son manager peut ainsi être obtenu en passant la structure de données “LTS_FOO” au processus manager générique.

5.1.2 Modèle des vecteurs de synchronisation

Les processus managers et portes requièrent en argument les vecteurs de synchronisation du système. Ces vecteurs sont représentés par une structure de données en LNT. La liste des vecteurs de synchronisation de chaque porte est stockée dans une constante, qui est passée en argument au processus porte correspondant. De plus, toutes ces listes sont regroupées dans une “carte” (*map*) de synchronisation qui est passée aux processus managers.

Considérons par exemple un système composé de deux tâches au comportement défini par FOO et dont les actions sur la porte C sont synchronisées :

```
par A #2, B in
   C -> FOO [A,B,C]
|| C -> FOO [A,B,C]
||      FOO [A,B,C]
end par
```

Les vecteurs de synchronisation de ce système sont représentés de la manière suivante dans le modèle de l'implémentation (on remarque qu'il est possible d'utiliser plusieurs fois la même tâche, ici FOO, au niveau de la spécification donnée en entrée) :

```
function gate_A_sync_vect : sync_vect_list is
   return {
      { DLC_TASK_0_FOO, DLC_TASK_1_FOO },
      { DLC_TASK_1_FOO, DLC_TASK_2_FOO },
      { DLC_TASK_2_FOO, DLC_TASK_0_FOO }
   }
end function

function gate_B_sync_vect : sync_vect_list is
   return {
      { DLC_TASK_0_FOO, DLC_TASK_1_FOO, DLC_TASK_2_FOO }
   }
end function

function gate_C_sync_vect : sync_vect_list is
   return {
      { DLC_TASK_0_FOO, DLC_TASK_1_FOO }
   }
end function

function global_sync_map : sync_map is
   return {
      sync_map_entry (DLC_GATE_A, gate_A_sync_vect),
      sync_map_entry (DLC_GATE_B, gate_B_sync_vect),
      sync_map_entry (DLC_GATE_C, gate_C_sync_vect)
   }
end function
```

Ces constantes sont ensuite passées aux différents processus portes et managers afin qu'ils puissent accéder aux vecteurs de synchronisation du système.

5.1.3 Modèle des communications asynchrones

Les communications asynchrones entre les portes et les managers sont modélisées à l'aide de processus *buffers* (tampons de communication). Ces processus buffers jouent le rôle des sockets TCP utilisées dans une implémentation réelle ; ils modélisent une file de messages sans perte. Afin de borner l'espace d'états du modèle de l'implémentation, nous limitons la taille de chaque file par une constante globale "bufsize". Nous faisons ensuite en sorte que toute communication entre un manager et une porte, ou entre managers, passe par le biais d'un processus buffer.

5.1.4 Modèle général de l'implémentation

Le modèle général de l'implémentation est constitué de la composition parallèle des processus managers, portes et buffers. Les portes et les managers ne se synchronisent jamais directement entre eux, car leurs communications passent toujours par le biais d'un buffer. De la même manière, deux buffers ne se synchronisent jamais entre eux, car ils servent de pont de communication entre deux processus managers ou portes. La composition parallèle générale est donc formée de deux compositions parallèles imbriquées, avec d'un côté tous les processus buffers, et de l'autre tous les processus portes et managers.

Pour illustrer cette composition, on reprend le système formé de deux tâches FOO déjà utilisé en exemple précédemment. Le modèle général de l'implémentation de ce système a la forme suivante :

```
process MAIN [
   TASK_0_FOO_SEND, TASK_0_FOO_RECV,
   TASK_1_FOO_SEND, TASK_1_FOO_RECV,
   TASK_2_FOO_SEND, TASK_2_FOO_RECV,
   GATE_A_SEND, GATE_A_RECV,
   GATE_B_SEND, GATE_B_RECV,
   GATE_C_SEND, GATE_C_RECV : com,
   ACTION, HOOK_REFUSE : annonce ] is
par
   TASK_0_FOO_SEND, TASK_0_FOO_RECV,
   TASK_1_FOO_SEND, TASK_1_FOO_RECV,
   TASK_2_FOO_SEND, TASK_2_FOO_RECV,
   GATE_A_SEND, GATE_A_RECV,
   GATE_B_SEND, GATE_B_RECV,
   GATE_C_SEND, GATE_C_RECV
in
   par
      buffer [TASK_0_FOO_SEND,TASK_1_FOO_RECV](DLC_TASK_0_FOO,DLC_TASK_1_FOO)
   || buffer [TASK_0_FOO_SEND,TASK_2_FOO_RECV](DLC_TASK_0_FOO,DLC_TASK_2_FOO)
   || buffer [TASK_1_FOO_SEND,TASK_0_FOO_RECV](DLC_TASK_1_FOO,DLC_TASK_0_FOO)
   || ... -- connections entre les managers
   || buffer [TASK_0_FOO_SEND,GATE_A_RECV](DLC_TASK_0_FOO,DLC_GATE_A)
   || buffer [GATE_A_SEND,TASK_0_FOO_RECV](DLC_GATE_A,DLC_TASK_0_FOO)
   || buffer [TASK_0_FOO_SEND,GATE_B_RECV](DLC_TASK_0_FOO,DLC_GATE_B)
   || buffer [GATE_B_SEND,TASK_0_FOO_RECV](DLC_GATE_B,DLC_TASK_0_FOO)
   || ... -- connections entre les managers et les portes
   end par
||
   par
      MANAGER [ACTION, TASK_0_FOO_SEND, TASK_0_FOO_RECV]
         (DLC_TASK_0_FOO, task_FOO_state_space, global_sync_map)
   || MANAGER [ACTION, TASK_1_FOO_SEND, TASK_1_FOO_RECV]
         (DLC_TASK_1_FOO, task_FOO_state_space, global_sync_map)
   || MANAGER [ACTION, TASK_2_FOO_SEND, TASK_2_FOO_RECV]
         (DLC_TASK_2_FOO, task_FOO_state_space, global_sync_map)
   || GATE [GATE_A_SEND, GATE_A_RECV, ACTION, HOOK_REFUSE]
```

```
        (DLC_GATE_A, gate_A_sync_vect)
    || GATE [GATE_B_SEND, GATE_B_RECV, ACTION, HOOK_REFUSE]
        (DLC_GATE_B, gate_B_sync_vect)
    || GATE [GATE_C_SEND, GATE_C_RECV, ACTION, HOOK_REFUSE]
        (DLC_GATE_C, gate_C_sync_vect)
  end par
end par
end process
```

On obtient ainsi un modèle complet d'une implémentation qui utilise le protocole de synchronisation, et où les managers et les portes communiquent par passage de messages asynchrones. La génération de ce modèle à partir de la spécification d'un système distribué est entièrement automatisée.

5.2 Vérifications conduites sur le modèle d'implémentation

Le modèle LNT de l'implémentation permet de vérifier plusieurs propriétés liées au comportement du protocole. Plus précisément, on s'intéresse à trois propriétés :

1. la relation d'équivalence entre les actions réalisables selon la spécification du système et celles réalisables selon le modèle de son implémentation
2. l'absence de *livelock* (boucles infinies) dans les négociations du protocole
3. l'absence de *deadlock* (blocage) dû au protocole

Ces trois propriétés sont vérifiées en utilisant divers outils de CADP. Le flot de vérifications est exprimé à travers un script SVL dont des extraits sont présentés par la suite. Ce script requiert que l'espace d'états de la spécification du système soit déjà généré et stocké dans le fichier "spec.bcg", et que le modèle de l'implémentation soit stocké dans le fichier "implem.lnt".

Dans la suite, le mot "action" devient ambigu, car il peut désigner soit une action au niveau de la spécification, soit une action au niveau du modèle de l'implémentation, comme par exemple la transmission d'un message du protocole entre un manager et un buffer. On réserve désormais et jusqu'à la fin de cette section le mot "action" aux événements de la spécification. Les événements du modèle de l'implémentation sont quant à eux désignés par les mots "transmission" pour un rendez-vous entre un buffer et une porte ou un manager, et "annonce" pour les annonces sur la porte "ACTION" ou sur la porte "HOOK_REFUSE" (qui annonce le refus d'une négociation par une fonction crochet).

5.2.1 Espace d'états du modèle de l'implémentation

On commence par générer deux versions de l'espace d'états du modèle de l'implémentation. La première est une version détaillée qui comporte toutes les transmissions et les annonces du modèle. À partir de la version détaillée, on obtient une deuxième version qui ne conserve que les annonces sur la porte "ACTION". Cette deuxième version est utile pour les propriétés d'équivalence et de deadlock :

```
"detail_implem.bcg" = strong reduction of "implem.lnt";

"implem.bcg" = strong reduction of
   gate hide all but "ACTION" in "detail_implem.bcg";
```

Afin de compresser leur taille autant que possible, chaque LTS est réduit modulo la relation d'équivalence forte (*strong equivalence*) [Par81]. Cette relation conserve toutes les propriétés vérifiables d'un LTS, on peut donc l'appliquer systématiquement.

5.2.2 Équivalence entre la spécification et son implémentation

Le protocole doit permettre de réaliser les actions qui sont possibles selon la spécification de départ. Afin de s'assurer que l'implémentation est cohérente avec la spécification de départ, on compare leurs espaces d'états respectifs selon plusieurs relations d'équivalence.

Nous rappellons brièvement qu'une relation d'équivalence définit une manière de comparer deux LTS pour déterminer s'ils permettent d'effectuer les mêmes séquences d'actions. Il existe plusieurs relations d'équivalence qui diffèrent notamment dans leur manière de traiter les actions internes, nous avons présenté celles disponibles dans CADP à la section 1.3.4.

Pour pouvoir être comparables, les actions de la spécification et les annonces d'action de l'implémentation doivent être désignées par les mêmes étiquettes, et l'implémentation ne doit comporter aucun événement visible mis à part les annonces d'actions. Nous commençons donc par renommer les annonces d'actions au sein de l'espace d'états de l'implémentation, où les communications et les annonces de refus par une fonction crochet ont déjà été dissimulées en étant transformées en actions internes. Les annonces d'actions de la forme "ACTION !DLC_GATE_<nom de porte>" sont renommées en "<nom de porte>" afin de pouvoir être comparable avec les actions de la spécification. Du côté de la spécification, il existe deux types d'actions qui doivent être renommées :

actions "exit" : les actions "exit" sont renommées en "EXIT" afin d'être comparables avec les étiquettes de l'implémentation. LNT n'est pas sensible à la casse et place tous les identifiants en majuscule dans les étiquettes ; dans l'implémentation, une annonce d'action sur la porte exit est ainsi signalée par une étiquette "ACTION !DLC_GATE_EXIT", qui est renommée en "EXIT". Nous homogénéisons la casse des étiquettes avant de

procéder à la comparaison qui, elle, est sensible à la casse [2].

actions internes : les actions internes, désignées par "i", sont renommées en "I", non seulement pour homogénéiser la casse des étiquettes, mais aussi parce que les relations d'équivalences traitent les actions internes d'une manière spécifique. Au niveau d'un LTS, seule l'étiquette "i" (minuscule) désigne une action interne. Nous utilisons donc l'étiquette "I" (majuscule) pour désigner une "action interne" au niveau de la spécification, qui doit se retrouver en tant qu'annonce au niveau de l'implémentation. Le LTS de l'implémentation comporte aussi des actions internes "i" qui résultent de la dissimulation des transmissions.

Nous utilisons l'opérateur de renommage de SVL pour rendre les étiquettes comparables. La capacité de capture d'une partie de l'étiquette par des expressions régulières est utilisée pour le renommage des étiquettes de l'implémentation :

```
"equivalence_implem.bcg" = total rename
   "ACTION !DLC_GATE_\([^ ]*\) .*" -> "\1"
   in "implem.bcg";

"equivalence_spec.bcg" = total rename
   "i"    -> "I",
   "exit" -> "EXIT"
   in "spec.bcg";
```

Les comparaisons des espaces d'états de la spécification et de l'implémentation sont réalisées par une propriété SVL paramétrée par un nom de relation d'équivalence. Ce nom est aussi utilisé comme une variable shell qui récupère le résultat de la comparaison, et comme nom de fichier de diagnostic. Pour s'épargner les problèmes liés à la présence d'un espace ou d'un caractère étoile dans le nom de la relation, nous employons les noms "taustar" et "weaktrace" pour les relations d'équivalence $\tau^*.a$ et *weak trace*, respectivement :

```
property Equivalence (RELATION, RESULT)
   "Specification and implementation are $RELATION equivalent."
is
   % DEFAULT_COMPARISON_RELATION="$RELATION"
   "$RELATION.bcg" = comparison "equivalence_spec.bcg" == "equivalence_implem.bcg";
   result "$RESULT"
   expected TRUE
end property
```

La propriété est ensuite appelée pour les différentes relations d'équivalence, de la plus forte à la plus faible. Nous ne vérifions pas l'équivalence de branchement sensible à la divergence (*divbranching*), car comme nous allons le voir juste après, la présence d'un cycle d'action interne est la manifestation d'un livelock. Avec l'hypothèse qu'il n'y a pas de cycle d'actions internes, la relation de branchement est identique à celle de branchement

2. Les processus portes, managers et buffers de l'implémentation ne terminent pas, il n'existe donc pas d'action "exit" (minuscule) au sein du LTS de l'implémentation.

sensible à la divergence, c'est pourquoi nous n'évaluons pas cette dernière. Enfin, lorsque les deux espaces d'états sont équivalents pour une certaine relation, nous évitons de tester les équivalences plus faibles, en accord avec la classification de la figure 1.1.

5.2.3 Absence de livelock

La présence de livelock dans le protocole se manifeste par une boucle infinie de transmissions qui ne mènent à aucune annonce. Ces boucles représentent un possible échange infini de messages de négociation entre des managers et des portes, sans que ceux-ci ne parviennent à aboutir à une annonce d'action.

Il se peut qu'une négociation parvienne à verrouiller toutes les tâches mais soit ensuite systématiquement refusée par la fonction crochet de post-négociation, pour une raison externe au protocole. Dans ce cas, une boucle infinie d'essais de négociation peut apparaître. Ce livelock n'est pas dû à à la logique du protocole, mais plutôt à la fonction crochet ou à l'environnement qui interdisent la réalisation de l'action. C'est pourquoi le refus d'une négociation par une fonction crochet est identifié par une annonce sur la porte "HOOK_REFUSE". Une boucle infinie comportant une action sur cette porte traduit ainsi un livelock provoqué par la fonction crochet, et non par le protocole.

Nous réduisons l'espace d'états autant que possible avant d'effectuer la recherche de livelock. Pour cela, nous cachons toutes les transmissions, ce qui résulte en de nombreuses actions internes [3] dans le LTS. L'espace d'états est ensuite réduit modulo la relation d'équivalence *divbranching*. Cette relation d'équivalence offre une réduction de taille efficace tout en conservant les cycles d'actions internes. Nous obtenons ainsi un espace d'états plus petit, tout en préservant l'éventuelle présence d'une boucle infinie d'actions internes, qui correspond à un livelock du protocole.

```
"livelock_implem.bcg" = divbranching reduction of
   gate hide all but "ACTION", "HOOK_REFUSE" in "detail_implem.bcg";
```

Un cycle d'actions internes étant la définition canonique d'un livelock, nous utilisons directement l'opérateur SVL "livelock" au sein d'une propriété qui s'attend à ce qu'aucun livelock ne soit détecté :

```
property Absence_Livelock
   "There is no livelock due to the protocol."
is
   livelock of "livelock_implem.bcg";
   result RESULT_LIVELOCK
   expected FALSE
end property
```

3. Nous rappelons que les actions internes de la spécification apparaissent comme des annonces "ACTION !DLC_GATE_I" au niveau du modèle d'implémentation.

Lorsqu'un livelock est détecté, nous effectuons une nouvelle recherche qui est plus lente, mais qui permet en général d'extraire un contre-exemple plus précis. Le contre-exemple qui serait produit par le diagnostic de la propriété précédente contiendrait un chemin d'exécution composé uniquement d'actions internes et d'annonces. Ce contre-exemple ne contient pas beaucoup d'informations pour comprendre quel genre de comportement du protocole mène à un livelock. Il est possible d'obtenir un diagnostic plus précis en recherchant une boucle infinie de transmissions au sein de la version détaillée de l'espace d'états de l'implémentation.

Nous utilisons pour cela la formule MCL suivante, qui est vérifiée par la présence (à partir de n'importe quel état) d'un cycle de transmissions, c'est-à-dire d'événements qui ne sont pas des annonces :

```
<true*> < not ('ACTION .*' or 'HOOK_REFUSE .*') > @;
```

Cette expression est reprise au sein d'une propriété SVL qui indique aussi au *model checker* d'utiliser une recherche en largeur avec l'option "bfs" (*breadth first search*). Cette option est en général plus lente qu'une recherche en profondeur (l'option utilisée par défaut), mais elle peut mener à un contre-exemple plus court :

```
% if [ $RESULT_LIVELOCK != FALSE ]
% then
property Detailed_Livelock_Diagnostic
   "A livelock was detected, produce a detailed counter-example"
is
   "detail_livelock.bcg" = "detail_implem.bcg" |= using bfs
      <true*> < not ('ACTION .*' or "HOOK_REFUSE") > @;
   expected TRUE
end property
% echo ""
% echo "WARNING: protocol livelock detected, abort verification script"
% exit 0
% fi
```

Pour résumer, nous commençons par vérifier rapidement l'absence de livelock dû au protocole sur un espace d'états réduit. Si un livelock est détecté, nous effectuons alors une recherche plus lente afin de produire un contre-exemple précis. Ce contre-exemple contient explicitement les transmissions du protocole qui mènent à une boucle infinie. En cas de présence de livelock, le protocole n'étant pas correct, nous stoppons le script de vérification.

5.2.4 Absence de deadlock

La vérification d'absence de deadlock nécessite de différencier les états puits inhérents à la spécification du système de ceux qui résultent du protocole utilisé dans l'implémentation. En effet, si le système possède un ou plusieurs états puits, ceux-ci se retrouvent dans

l'espace d'états de l'implémentation, indépendamment du protocole utilisé. Les états puits qui nous intéressent sont uniquement ceux dus au protocole.

Un deadlock dû au protocole se manifeste par une certaine négociation qui atteint un état puits *alors qu'une autre négociation aurait pu aboutir à une annonce d'action*. En effet, si un certain échange de messages mène à la réussite d'une négociation, c'est que l'état des tâches permet encore de réaliser au moins une action. Dans ce cas, s'il existe un autre échange de messages qui mène à un état puits sans contenir aucune annonce d'action, alors cet état puits correspond à un blocage de l'implémentation dû à un chemin d'exécution possible du protocole.

Au niveau du LTS de l'implémentation, un deadlock dû au protocole est donc caractérisé par un état depuis lequel il existe à la fois :

— un chemin d'exécution qui mène à une annonce d'action
— un chemin d'exécution qui ne contient aucune annonce d'action et qui mène à un état puits

En d'autres termes, une implémentation utilisant un protocole sans deadlock vérifie la formule MCL suivante, qui signifie "à partir de n'importe quel état, la présence d'un chemin qui ne contient aucune annonce d'action et qui mène à un état puits implique qu'à partir du même état, il n'existe pas de chemin qui contienne une annonce d'action" [4] :

```
[ true* ] (
   ( < (not ('ACTION.*'))* > [true] false )
   implies
   [ true* . 'ACTION.*' ] false
)
```

Cette expression est encapsulée dans une propriété SVL qui s'applique sur l'espace d'états de l'implémentation où les transmissions sont cachées :

```
property Absence_Deadlock
   "There is no deadlock due to the protocol."
is
   "implem.bcg" |=
      [ true* ] (
         ( < not ('ACTION .*') * > [true] false )
         implies
         [ true* . 'ACTION .*' ] false
      );
   result RESULT_DEADLOCK
   expected TRUE
end property
```

4. La propriété est énoncée dans ce sens pour accélérer la vérification, car il existe moins d'états à partir desquels il existe un chemin sans annonce menant à un état puits que d'états à partir desquels il existe un chemin qui contient une annonce d'action.

De plus, de la même manière que pour le livelock, nous enregistrons le résultat de la propriété. Lorsqu'un deadlock est détecté, nous effectons une recherche de contre-exemple précis au sein de l'espace d'états détaillé de l'implémentation.

En résumé, nous obtenons un script qui permet d'automatiser les différentes vérifications conduites sur le modèle de l'implémentation : test d'équivalence avec la spécification, détection de livelock, et détection de deadlock. Le script SVL complet est disponible en Annexe A.

5.3 Application de la méthode de vérification

L'automatisation des deux étapes de notre méthode, à savoir la génération de modèle d'implémentation et la conduite de vérifications formelles sur ce modèle, permet de facilement appliquer cette méthode sur une spécification LNT de système distribué. Une spécification de système peut alors être considérée comme un cas de test du protocole : il suffit d'appliquer notre méthode pour voir si le protocole se comporte de manière correcte pour ce système. Nous avons élaboré une base de tests qui permet de vérifier le comportement d'un protocole dans de nombreuses situations.

Dans cette section, nous commençons par présenter cette base de tests, avant d'analyser les résultats de notre protocole sur cette base. Enfin, nous discutons les résultats de notre méthode pour deux autres protocoles, où des deadlocks sont possibles.

5.3.1 Base de tests

Nous avons développé une base de tests composée de tests écrits à la main d'une part, et de tests générés automatiquement d'autre part. Notre connaissance précise du protocole nous permet d'avoir de l'intuition pour écrire des tests complexes pour le protocole. Cependant, cette intuition est à double tranchant, car elle biaise notre attitude et peut nous faire passer à côté de cas problématiques que nous n'avons pas envisagés. Pour pallier ce biais, nous avons choisis d'obtenir une partie des tests par génération automatique.

Tests écrits manuellement

Afin de structurer la conception des tests écrits à la main, nous nous sommes intéressés aux paramètres qui influencent la complexité d'un test au regard du protocole de synchronisation. Étant donné que le protocole est utilisé par une tâche lorsqu'elle souhaite passer d'un état dans un autre, la taille de l'espace d'états d'une tâche a peu d'impact sur le protocole. C'est plutôt la structure de cet espace d'états, notamment la présence de choix

non déterministes, qui est susceptible de créer des situations complexes pour le protocole. En d'autres termes, il n'est pas très utile d'utiliser des tâches qui ont un comportement long en nombre d'actions ; on préfère les tâches qui terminent ou qui bouclent après une séquence de quelques actions, mais qui possèdent des choix non déterministes.

Si on pousse cette idée à l'extrême, on peut imaginer que des tâches dont le comportement se résume à un seul choix non déterministe sont suffisantes pour tester le protocole. Toutefois, le phénomène de synchronisation en avance de phase rend le protocole sensible aux situations où une tâche est prête sur la même action plusieurs fois à la suite. C'est pourquoi il est important de tester le protocole sur des tâches qui sont capables de réaliser plusieurs actions à la suite.

Nous avons ainsi rédigé à la main une série de 63 tests qui cherchent à déclencher des situations complexes pour le protocole. Par exemple, certains tests comportent des tâches qui bouclent indéfiniment, pour évaluer le protocole sur des systèmes infinis. D'autres tâches sont alternativement prêtes puis auto-verrouillées sur une porte, pour observer la combinaison de l'auto-verrouillage avec les synchronisations en avance de phase. Les exemples qui illustrent la nécessité du système de purge dans la section 2.5 sont issus de cette série de tests.

Remarque 5-1

Si la taille de l'espace d'états d'un système est bornée, la taille de l'espace d'états du modèle de l'implémentation utilisant notre protocole est elle aussi bornée. En particulier, un système dont le comportement boucle indéfiniment dans un espace d'états fini donne lieu à une implémentation qui boucle elle aussi indéfiniment, dans un espace d'états fini lui aussi. Cette propriété est notamment due au fait que notre protocole n'utilise pas de *timestamps*, d'horloges logiques [Lam78] ou d'autres formes de compteurs qui s'incrémentent au cours de l'exécution. Notre protocole permet donc de boucler sur des états similaires, alors que la présence de compteurs différencierait tous les nouveaux états atteints et mènerait à des espaces d'états infinis.

Il est envisageable d'obtenir des espaces d'états finis en présence de compteurs s'il existe une procédure de remise à zéro de ces compteurs. Cependant, la plupart des protocoles utilisant des compteurs que nous avons croisés dans la littérature n'évoquent pas une telle remise à zéro. Il semble donc plus difficile d'évaluer ces protocoles sur des systèmes qui bouclent indéfiniment. ■

Tests générés automatiquement

Nous avons cherché à automatiser la création de tests pour couvrir les compositions parallèles de petites tâches de manière systématique. Nous avons écrit un programme qui énumère une série d'espaces d'états comprenant deux transitions, où chaque transition

peut être étiquetée soit par une action interne “i”, soit par une action sur une porte “A”, soit par une action sur la porte “B”. Nous obtenons ainsi 39 tâches.

Nous avons ensuite énuméré toutes les compositions parallèles de 2 tâches parmi ces 39, avec plusieurs déclinaisons :

— nous produisons systématiquement une composition qui ne requiert de synchronisation sur aucune porte
— nous produisons une composition parallèle qui requiert une synchronisation sur la porte “A” (respectivement “B”) si les deux tâches considérées possèdent au moins une action sur la porte “A” (respectivement “B”).
— nous produisons une composition parallèle qui requiert une synchronisation sur les deux portes “A” et “B” si les deux tâches considérées possèdent chacune une action sur la porte “A” et sur la porte “B”.

Nous obtenons ainsi 1508 compositions parallèles, qui sont autant de tests générés automatiquement. Nous utilisons ces tests par la suite pour évaluer la correction de notre protocole.

5.3.2 Analyse des résultats pour le protocole de DLC

Nous avons appliqué notre méthode de vérification au protocole que nous avons mis au point pour DLC, sur l’ensemble de la base de tests. Au cours du développement du protocole, notre méthode nous a permis de détecter plusieurs fois de possibles livelocks ou deadlocks. Les contre-exemples détaillés ont facilité la compréhension de la source de ces erreurs. Nous avons pu ainsi itérer sur la conception du protocole en s’assurant qu’une nouvelle modification n’introduisait ni livelock, ni deadlock. Aucune possibilité de livelock ou de deadlock n’est détectée sur la version courante de notre protocole, telle que présentée dans la section 2.7.

Les résultats d’équivalence entre la spécification et le modèle de l’implémentation sont discutés plus en détail dans la suite de cette section.

Équivalence entre spécification et modèle d’implémentation

Au même titre que les deadlocks et les livelocks, la vérification d’équivalence entre la spécification du système et le modèle de son implémentation nous a été très utile lors du développement du protocole. En particulier, nous n’avions pas envisagé que la cohabitation de l’auto-verrouillage et des synchronisations en avance de phase puisse mener à des actions invalides, comme le montrent les contre-exemples dans la section 2.5. Encore une fois, ces contre-exemples nous ont aidé dans la compréhension de la source du problème. Lors de la mise au point de la technique des signaux de purge, notre méthode nous a été d’un très grand soutien. La version courante de notre protocole ne relève aucune possibilité d’action invalide, ou manquante.

La table 5.1 montre une classification des tests selon la plus forte relation d'équivalence vérifiée entre la spécification du système et le modèle de son implémentation. On remarque que la relation d'équivalence la plus forte à être vérifiée par l'ensemble de la base de tests est la relation de sûreté (*safety equivalence*) [BFG+91]. Toutefois, une grande partie des tests générés automatiquement vérifient une relation d'équivalence plus forte que celle de sûreté.

Plus forte équivalence	Tests écrits manuellement	Tests générés automatiquement
Forte	0	1
Branchement	22	681
$\tau^*.a$	24	303
Sûreté	17	523

TABLE 5.1 – Nombre de tests pour lesquels une certaine relation d'équivalence est la plus forte à être vérifiée.

Cas de relation d'équivalence forte

Nous avons été surpris de constater qu'il existe un test pour lequel l'équivalence forte est vérifiée. En effet, cette relation d'équivalence ne traite pas les actions internes d'une manière spécifique. Or, n'importe quelle transmission du protocole se traduit par une action interne dans le modèle de l'implémentation.

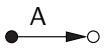
Le test qui vérifie la relation d'équivalence forte correspond à la composition parallèle de deux tâches qui réalisent uniquement des actions internes. Aucune négociation du protocole n'est réalisée, et les tâches n'ont même pas de message "ready" à transférer car elles ne sont jamais prêtes pour une action sur porte. La résolution des actions internes se fait directement au niveau de chaque tâche. Les seules transitions du modèle de l'implémentation sont ainsi des annonces de réalisation d'actions internes. Ce genre de configuration permet de vérifier une équivalence forte entre la spécification et le modèle de son implémentation.

Cas général : relation d'équivalence de sûreté

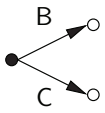
Dans le cas général, la relation d'équivalence la plus forte à être vérifiée est celle de sûreté. Nous avons étudié les tests pour lesquels cette relation est la plus forte à être vérifiée afin de déterminer les causes de ce résultat. Nous avons constaté que les relations d'équivalence plus fortes de celle de sûreté ne sont pas vérifiées lorsque le protocole résout de manière concurrente des choix d'actions qui ne sont pas en conflit.

Nous illustrons ce genre de comportement à l'aide du système suivant :

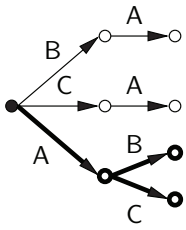
```
process T1 [A: none] is
   A;
   stop
end process
```




```
process T2 [B, C: none] is
   select
      B
   [] C
   end select;
   stop
end process
```




```
par
   T1 [A]
|| T2 [B,C]
end par
```



Dans ce système, les rendez-vous sur la porte A ne sont pas en conflit avec ceux sur les portes B ou C. Au niveau de l'espace d'états de la composition parallèle, on note que lorsque la première action est réalisée sur la porte A, le système atteint alors un état depuis lequel il a le choix entre une action sur la porte B ou une action sur la porte C (chemin indiqué en gras dans la représentation du LTS).

La figure 5.1 donne un aperçu schématique d'une partie du LTS de l'implémentation, où les transitions ont été renommées en vue de la comparaison avec l'espace d'états de la spécification, et toutes les transmissions apparaissent comme des actions internes.

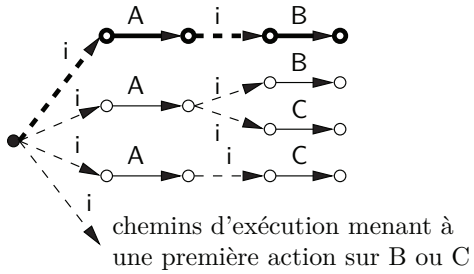


FIGURE 5.1 – Aperçu de l'espace d'états du modèle de l'implémentation après préparation pour la comparaison avec celui de la spécification (une flèche en tirets indique un diagramme de transmissions du protocole).

Dans cet espace d'états, il existe des chemins d'exécutions où, après une première action sur la porte A, le système n'a plus le choix entre une action sur les portes B ou C. Par exemple, dans le chemin marqué en gras sur la figure 5.1, après l'action sur la porte A, seule

l'action sur la porte B est atteignable. Dans ce chemin, l'état qui suit l'annonce de l'action sur la porte A n'a pas d'état bisimilaire dans le LTS de la spécification, où après une action sur la porte A, les deux actions sur les portes B ou C sont atteignables. La présence de cet état empêche les relations d'équivalence $\tau^*.a$ et observationnelle – et, a fortiori, les relations plus forte que ces deux-ci – d'être vérifiées. Au niveau de l'équivalence de trace, la présence d'actions internes uniquement dans le LTS de l'implémentation empêche cette relation d'être vérifiée. La relation la plus forte a être vérifiée est alors celle de sûreté.

La figure 5.2 détaille un chemin d'exécution du système qui est inclus dans le chemin en gras de la figure 5.1. La tâche T1 se déclare auto-verrouillée sur la porte A, et cette porte va annoncer la réalisation d'une action. Entre temps, une négociation pour une action de la tâche T2 sur la porte B a été démarrée, et le verrou a été accepté par la tâche T2. Ainsi, au moment où l'action sur la porte A est annoncée, il est déjà certain que l'action suivante du système sera réalisée par la tâche T2 sur la porte B. Ce déroulement illustre la possibilité de ne plus avoir le choix entre une action sur la porte B ou C après une première action sur la porte A.

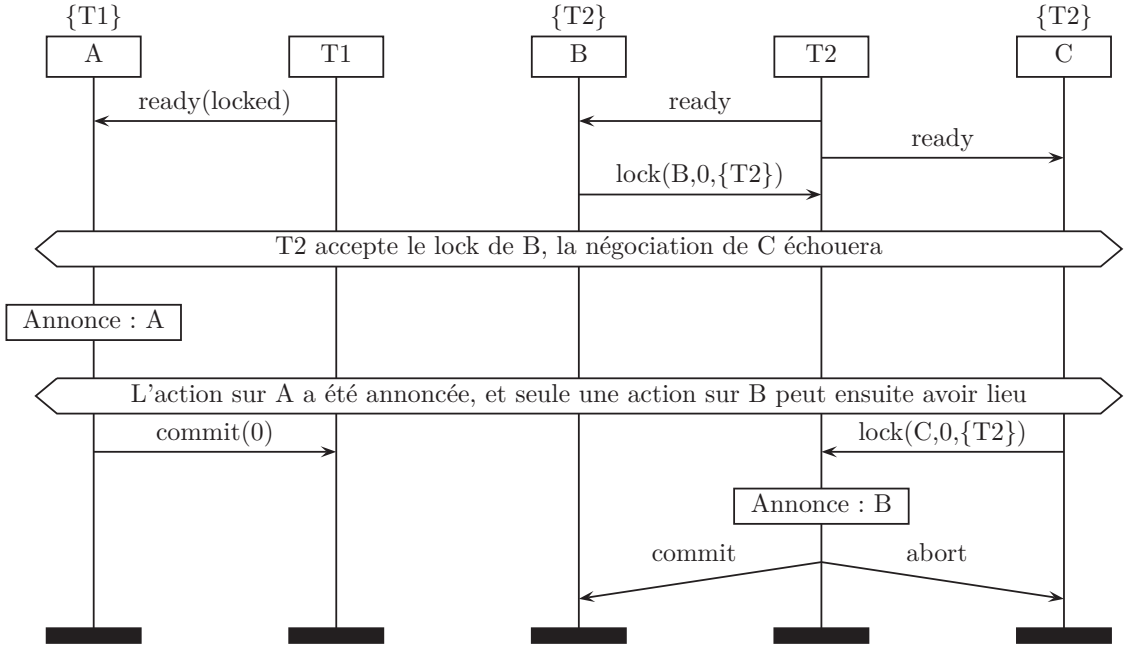


FIGURE 5.2 – Le choix non déterministe d'une action sur la porte B ou sur la porte C pour la tâche T2 peut déjà être entériné au moment où une action sur la porte A est annoncée.

On note que, si nous nous sommes focalisés sur le cas où la deuxième action du système est nécessairement sur la porte B, dans le cas général le protocole ne privilégie pas une action en particulier. En effet, il est possible que ce soit la négociation pour l'action sur

la porte C qui arrive à verrouiller la tâche T2 avant que l'action sur la porte A ne soit annoncée. De même qu'il est possible que la réalisation de l'action de la tâche T2 ne soit pas encore décidée au moment où l'action de la tâche T1 est annoncée, ou encore que la première action a être annoncée soit une action de la tâche T2 sur la porte B ou C.

D'une manière générale, le fait que la résolution de choix d'action par le protocole ne soit pas atomique, car ce choix est résolu via la transmission de plusieurs messages, permet à plusieurs négociation de se dérouler en parallèle. L'annonce du succès d'une négociation peut avoir lieu alors que d'autres négociations sont suffisamment avancées pour restreindre le choix des prochaines actions du système. Ce comportement est donc une manifestation du fait que le protocole résout les synchronisations indépendantes de manière concurrente.

Remarque 5-2

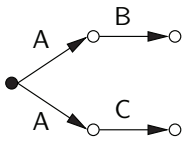
Dans le cadre de leurs travaux sur l'implémentation de LOTOS, Parrow et Sjödin ont mis au point la relation d'équivalence de simulation couplée (*Coupled Simulation*) [PS92]. Cette relation d'équivalence, proche de celle de sûreté, traite spécifiquement les états du LTS de la spécification depuis lesquels un choix non déterministe est possible à travers une notion de *stabilité* d'un état. Cette relation d'équivalence semble plus fine et plus pertinente pour comparer les LTS de spécification et d'implémentation. Nous n'avons cependant pas pu expérimenter la relation d'équivalence de simulation couplée car elle n'est pas disponible au sein de CADP à ce jour. Dans le cadre de travaux futurs, il serait intéressant d'étudier si la relation d'équivalence de simulation couplée peut être ramenée à la vérification de la relation d'équivalence de sûreté complétée par la vérification d'une autre propriété, exprimable en MCL, qui couvrirait la notion de stabilité. ■

Choix interne et relation d'équivalence

Nous justifions ici une particularité de la spécification du comportement du manager, présentée en section 2.7.3.

On dit qu'une tâche a un choix interne lorsqu'à partir d'un état, une même action peut mener à différents états. Par exemple, la tâche T1 suivante a un choix interne lorsqu'elle effectue une action sur la porte A :

```
process T1 [A, B, C: none] is
   select
      A ; B
   [] A ; C
   end select;
   stop
end process
```



Le moment où un choix interne est résolu influe sur la relation d'équivalence entre la spécification et le modèle d'implémentation. Sur le LTS de la tâche T1, on note qu'une

fois une première action sur la porte A effectuée, cette tâche n'a plus le choix pour l'action suivante. En d'autres termes, le choix d'effectuer la deuxième action sur la porte B ou sur la porte C doit être déjà réalisé au moment où l'action sur la porte A a lieu.

Au niveau de l'implémentation de la résolution d'un choix interne, deux approches sont possibles :

Résolution du choix interne après confirmation de l'action. Cette approche, qui semble la plus naturelle au premier abord, consiste à décider quel état la tâche va rejoindre une fois que l'action sur la porte A a été confirmée. Dans le LTS de l'implémentation, après l'annonce de l'action sur la porte A, les deux actions sur les portes B et C restent donc atteignables.

Résolution du choix interne avant toute négociation. Cette approche consiste à choisir quel état la tâche va rejoindre avant même d'envoyer un message "ready" à la porte A. Au moment de l'annonce de l'action sur la porte A, le choix interne est donc déjà résolu, et seule une des deux actions sur les portes B ou C est réalisable par la suite.

Nous avons opté pour la deuxième approche afin que la relation d'équivalence de sûreté soit vérifiée entre la spécification et le modèle de son implémentation. En effet, si après une action sur la porte A, les deux actions actions sur les portes B et C restent accessibles, cette relation d'équivalence n'est pas vérifiée. Ainsi, dans la spécification du comportement du manager, le choix interne est résolu avant même d'envoyer les messages "ready" aux portes afin de renforcer l'équivalence entre la spécification et son implémentation.

5.3.3 Possibles interblocages pour deux protocoles existants

Notre méthode de vérification n'est pas limitée à notre protocole, et nous l'avons appliquée à trois protocoles de la littérature [EL13] : celui proposé par Sjödin dans sa thèse [Sjö91], la version de Parrow et Sjödin [PS96] et le protocole de Sisto, Ciminiera et Valenzano [SCV91]. Notre méthode a révélé de possibles deadlocks dans le protocole de Parrow et Sjödin, que nous avons déjà illustrés dans la section 2.3.2.

Plus tard, nous avons envisagé d'utiliser notre méthode sur le le protocole α-core [PCT04], qui est notamment utilisé pour l'implémentation distribuée de BIP. Cependant, à ce moment nous avons appris qu'une possibilité de deadlock avait déjà été détectée dans ce protocole par Katz et Peled [KP10] [5]. Nous avons appliqué notre méthode sur l'exemple donné dans l'article de Katz et Peled : un deadlock est bien détecté, et un contre-exemple est fourni. Nous n'avons cependant pas eu le temps de mettre au point la génération automatique de modèle d'implémentation pour le protocole α-core.

5. La génération distribuée de BIP utilise une version corrigée du protocole

Ces découvertes de deadlocks possibles montrent qu'il peut subsister des problèmes dans des protocoles dont la correction a été "prouvée" manuellement. Cela souligne l'intérêt d'employer, autant que possible, des vérifications automatiques en renfort des preuves conduites à la main.

Chapitre 6

Performances des implémentations générées

Dans ce chapitre, nous présentons trois expériences qui visent à estimer les performances des implémentations générées par DLC. La première illustre les performances atteintes dans le cadre d'une barrière de synchronisation distribuée, et permet de se comparer directement avec plusieurs langages d'implémentation utilisés dans l'industrie. La seconde expose comment le rendez-vous multiple facilite l'implémentation d'une solution au dîner des philosophes, et détaille les performances de cette solution. Enfin, la troisième est un cas d'étude plus complet, qui couvre l'implémentation de l'algorithme de consensus Raft [OO14].

6.1 Barrière de synchronisation distribuée

Notre première expérience vise à évaluer les performances du rendez-vous multiple en tant qu'opérateur d'interaction entre processus distants. Nous voulons nous comparer à des langages de programmation populaires dans l'industrie, tels que C, Java ou Erlang. Cependant, aucun de ces langages n'offre d'opérateur équivalent au rendez-vous multiple, et nous n'avons pas trouvé de bibliothèques stables offrant cette fonctionnalité.

Par défaut, nous avons donc choisi de comparer les performances de ces langages sur un patron classique souvent présent dans un système distribué, à savoir une barrière de synchronisation distribuée. Pour synchroniser un ensemble de processus, chaque processus s'adresse à une barrière de synchronisation via une interaction bloquante. La barrière de synchronisation autorise chaque processus à continuer uniquement lorsque tous les processus de l'ensemble ont atteint le point où ils bloquent sur la barrière. Une barrière de synchronisation distribuée permet ainsi de synchroniser des processus qui s'exécutent sur différentes machines.

En LNT, le rendez-vous multiple sur une porte permet d'obtenir directement une barrière de synchronisation distribuée, en utilisant DLC comme compilateur. En C, Java et Erlang, un processus est dédié à assurer le rôle de barrière : il attend une requête de chaque processus de l'ensemble à synchroniser avant de répondre à tous. Dans ces trois langages, différentes techniques sont utilisées pour assurer une interaction entre processus distants :

— en C, nous utilisons les sockets POSIX pour envoyer des messages
— en Java, nous nous appuyons sur l'invocation de méthode distante RMI (*Remote Method Invocation*)
— en Erlang, nous employons le passage de message natif au langage

La figure 6.1 illustre le temps requis pour réaliser 1000 synchronisations entre des ensembles de processus distants. Nous constatons que l'implémentation générée par DLC est plus lente que l'implémentation en C, mais plus rapide que celles en Java ou en Erlang. Ce résultat montre tout d'abord que le code C généré par DLC (et EXEC/CÆSAR en sous-main) est suffisamment performant pour atteindre de meilleurs temps d'exécution que les systèmes reposant sur une machine virtuelle, comme Java ou Erlang. De plus, ce résultat illustre que notre protocole est efficace pour implémenter une simple barrière de synchronisation distribuée. En effet, quand tous les processus sont prêts sur une seule porte, ils s'auto-verrouillent, et notre protocole ne nécessite pas plus de messages que l'approche utilisée pour C, Java ou Erlang.

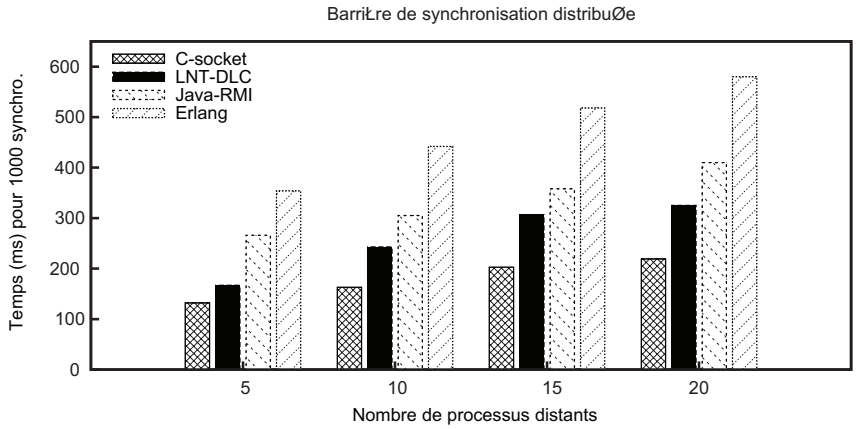


FIGURE 6.1 – Comparaison du temps nécessaire pour réaliser 1000 synchronisations entre plusieurs processus distants, pour des implémentations en C, LNT, Java et Erlang.

En outre, nous avons voulu estimer le gain en performances des modifications que nous avons apportées au protocole de Parrow et Sjödin, notamment la diffusion en parallèle des messages de résultats et l'auto-verrouillage. La figure 6.2 illustre, toujours sur l'exemple d'une barrière de synchronisation, les performances d'une implémentation qui utilise la version de Parrow et Sjödin du protocole, et celles obtenues par une implémentation utilisant notre protocole. Nous remarquons que l'auto-verrouillage et la diffusion de messages de résultat en parallèle permettent un gain conséquent en performance.

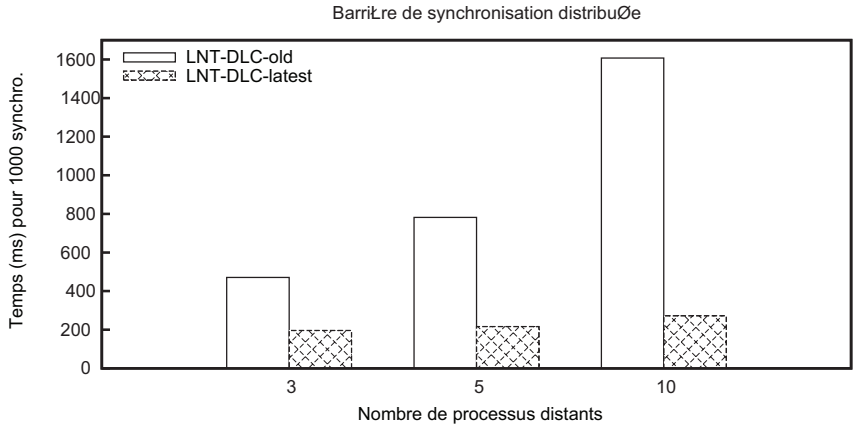


FIGURE 6.2 – Gain en performances de notre protocole par rapport à la version de Parrow et Sjödin.

6.2 Le dîner des philosophes

Notre deuxième expérience consiste à évaluer les performances des implémentations générées pour le problèmes classique du dîner des philosophes [Dij71]. Cette expérience est l'occasion d'illustrer d'une part comment le rendez-vous multiple peut simplifier la programmation concurrente, et d'autre part les performances atteintes pour un système où plusieurs rendez-vous peuvent avoir lieu de manière indépendante.

Nous rappelons brièvement l'énoncé du problème du dîner des philosophes. Plusieurs philosophes mangent autour d'une table ronde. Il existe une fourchette entre chaque paire de philosophes voisins de table. Un philosophe tour à tour pense et mange, et il a besoin de la fourchette à sa gauche et de celle à sa droite pour pouvoir manger. Une fourchette ne peut être utilisée que par un seul philosophe à la fois. Le problème consiste à organiser la prise de fourchette entre philosophes, afin que tous puissent manger.

6.2.1 Ressources partagées et rendez-vous multiple

Le dîner des philosophes est une représentation des problèmes d'accès à des ressources partagées (les fourchettes) par plusieurs processus (les philosophes). C'est un problème classique de la programmation concurrente. Parmi les solutions à ce problème, celle proposée par Dijkstra consiste à définir un ordre sur les fourchettes, et à imposer à chaque philosophe de prendre les fourchettes dans l'ordre. Nous avons déjà évoqué cette solution à la section 1.4.2. Une autre solution possible est de faire intervenir un serveur qui restreint l'accès aux fourchettes à un seul philosophe à la fois. En pratique, cette restriction peut être imposée par une construction qui assure l'exclusion mutuelle entre les processus, telle qu'un “mutex” par exemple.

Toutes ces solutions font l'hypothèse qu'un philosophe n'interagit qu'avec une seule fourchette à la fois. Considérons maintenant que nous avons le rendez-vous multiple à disposition : la prise de fourchettes peut être implémentée en un rendez-vous à trois entre un philosophe et les deux fourchettes à ses côtés. Le rendez-vous multiple garantit que, si l'action a lieu, alors les deux fourchettes ont été prises.

Nous avons ainsi utilisé le rendez-vous multiple pour implémenter un dîner de philosophes en LNT. Le comportement d'un philosophe consiste à penser, à prendre ses fourchettes (action sur la porte TAKE), à manger, puis à reposer ses fourchettes (action sur la porte RELEASE) :

```
process PHILO [TAKE, RELEASE: none] is
   loop
      -- think
      TAKE;
      -- eat
      RELEASE
   end loop
end process
```

Le comportement d'une fourchette consiste à être prise par le philosophe à sa gauche ou bien par celui à sa droite, puis par être reposée par le philosophe qui l'a prise :

```
process FORK [LEFT_TAKE, LEFT_RELEASE, RIGHT_TAKE, RIGHT_RELEASE: none] is
   loop
      select
         LEFT_TAKE;
         LEFT_RELEASE
      []
         RIGHT_TAKE;
         RIGHT_RELEASE
      end select
   end loop
end process
```

Nous pouvons ensuite définir un dîner de philosophes grâce à une composition parallèle de processus philosophes et fourchettes. Par exemple, un dîner à trois philosophes est obtenu avec la composition parallèle suivante :

```
par TAKE_0, RELEASE_0, TAKE_1, RELEASE_1, TAKE_2, RELEASE_2 in
   par
      PHILO [TAKE_0, RELEASE_0]
   || PHILO [TAKE_1, RELEASE_1]
   || PHILO [TAKE_2, RELEASE_2]
   end par
||
   par
      TAKE_0, RELEASE_0, TAKE_1, RELEASE_1 ->
         FORK [TAKE_0, RELEASE_0, TAKE_1, RELEASE_1]
   || TAKE_1, RELEASE_1, TAKE_2, RELEASE_2 ->
         FORK [TAKE_1, RELEASE_1, TAKE_2, RELEASE_2]
```

```
    || TAKE_2, RELEASE_2, TAKE_0, RELEASE_0 ->
          FORK [TAKE_2, RELEASE_2, TAKE_0, RELEASE_0]
  end par
end par
```

Le rendez-vous multiple facilite l'implémentation du dîner des philosophes : nous n'avons pas besoin de construction de mutex, ni d'avoir à préciser un ordre sur les fourchettes. Le rendez-vous multiple permet d'assurer directement l'exclusion mutuelle des philosophes voisins, et le choix non déterministe au niveau d'une fourchette la rend accessible aux deux philosophes l'entourant. Au niveau de l'implémentation générée, le rendez-vous multiple se traduit effectivement par un protocole de synchronisation entre processus. Toutefois, au niveau de la spécification, le rendez-vous multiple offre une abstraction de plus haut niveau que des interactions limitées à deux entités.

6.2.2 Mesures de performances

Nous avons produit un modèle LNT pour plusieurs configurations de dîner de philosophes. Nous avons ensuite utilisé DLC pour obtenir des implémentations distribuées. La figure 6.3 illustre les performances atteintes pour les différentes configurations.

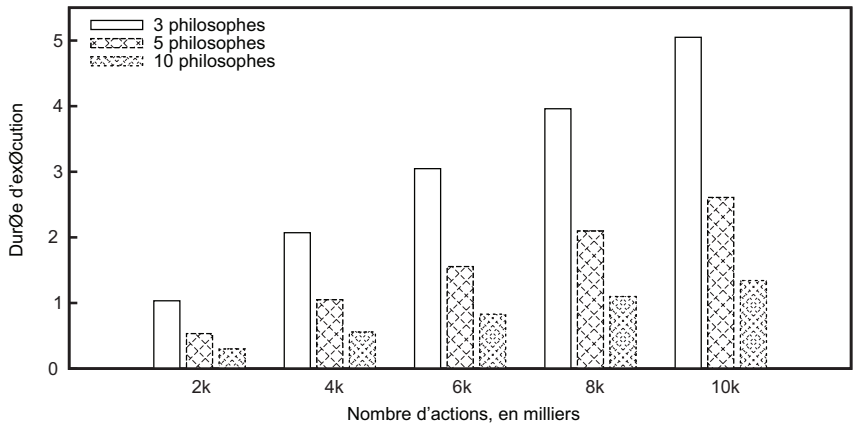


FIGURE 6.3 – Durée nécessaire pour réaliser un certain nombre de rendez-vous, pour plusieurs configuration. Plus il y a de philosophes, plus le nombre d'actions qui peuvent être réalisées en parallèle augmente, et plus la durée d'exécution est courte.

Nous avons mesuré la durée nécessaire pour réaliser un certain nombre d'actions. Toutes les actions sont des rendez-vous à trois entre un philosophe et une paire de fourchettes, qui traduisent une prise ou un relâchement de fourchettes. Nous avons utilisé une option de l'implémentation générée par DLC qui indique au nœud central d'arrêter l'exécution après un certain nombre d'actions réalisées dans le système, et de retourner le temps d'exécution

du système [1]. Ces mesures ont été réalisées sur le cluster “edel” du site de Grenoble de Grid5000. Pour chaque configuration, nous avons utilisé autant de machines qu'il y a de philosophes, et les autres programmes (fourchettes et portes) ont été distribués de manière équitable sur ces machines.

On remarque tout d'abord que lorsque le nombre de philosophes augmente, la durée nécessaire pour réaliser un certain nombre d'actions diminue. Cela illustre que les rendez-vous qui peuvent avoir lieu de manière concurrente sont effectivement réalisés en parallèle. La configuration à 10 philosophes effectue ainsi 10000 actions en à peine plus d'une seconde.

La configuration à trois philosophes est particulièrement intéressante : tous les philosophes sont voisins entre eux, et lorsqu'un philosophe mange, aucun autre philosophe ne peut manger. Toutes les actions de cette configuration sont donc effectuées *en séquence*. L'implémentation générée requiert environ une seconde pour réaliser 2000 actions. Cette performance est moins bonne que celle obtenue pour la synchronisation de trois processus sur la barrière de synchronisation de la section 6.1. Cela s'explique par le choix non déterministe du comportement des fourchettes, qui interdit un auto-verrouillage des fourchettes pour les actions sur les portes TAKE. On note tout de même que l'implémentation réalise plus d'un millier d'actions en séquence par seconde.

6.3 Cas d'étude : l'algorithme de consensus Raft

Afin d'évaluer DLC sur un système réel, nous avons utilisé notre outil pour implémenter un service de stockage répliqué via l'algorithme de consensus Raft [OO14]. Cela nous permet de comparer ensuite les performances de l'implémentation générée par DLC avec celles de Consul [2], un logiciel industriel offrant un service équivalent, et qui repose aussi sur l'algorithme Raft en interne.

Raft est un algorithme de consensus, c'est-à-dire qu'il décrit une manière pour plusieurs processus distants de s'accorder sur une même valeur, bien que ces processus soient susceptibles de tomber en panne. Les algorithmes de consensus sont une brique importante des systèmes distribués car ils permettent, via la technique de réplication de machine à états [Sch90], d'offrir un service (par exemple un *key-value store*, un dictionnaire stockant des couples clé-valeur) qui tolère les pannes de serveurs. À l'heure du *cloud computing*, les algorithmes de consensus sont utilisés par exemple au sein de systèmes critiques de gestion de données à grande échelle, tels que le projet Apache ZooKeeper [HKJR10] ou encore Chubby [Bur06] chez Google.

L'algorithme de référence pour résoudre le consensus est Paxos [Lam98], présenté par Leslie Lamport en 1998, dont la correction est garantie par une preuve formelle. Cependant,

1. Le temps nécessaire au déploiement (c'est-à-dire à la copie des programmes sur différentes machines à l'initialisation) n'est pas compris dans le temps d'exécution.

2. `http://www.consul.io`

Paxos est réputé complexe et non trivial à transcrire en une implémentation [CGR07]. En 2014, Diego Ongaro et John Ousterhout proposent le nouvel algorithme Raft [OO14] en visant à offrir une solution pratique, c'est-à-dire facile à comprendre et à implémenter, au problème du consensus. Raft est formellement spécifié en TLA$^+$ [Ong14], et une preuve de la correction de Raft a été élaborée par les auteurs, bien que manuellement et sans faire appel à un assistant de preuve.

Nous considérons Raft comme un bon exemple pour évaluer DLC car c'est un algorithme distribué, récent, et sa modélisation formelle en TLA$^+$ offre une description non ambiguë de l'algorithme, ce qui facilite sa transcription en LNT. De plus, Raft est déjà utilisé par plusieurs systèmes dans l'industrie[3], ce qui offre des possibilités de comparaison de performances.

Dans cette section, la structure de l'algorithme Raft est d'abord introduite, suivie par son implémentation distribuée en LNT. Ensuite, nous comparons les performances de l'implémentation générée par DLC avec celles de Consul. Enfin, nous discutons des similarités et des différences entre le problème du consensus et celui du rendez-vous multiple.

6.3.1 Structure de l'algorithme Raft

Avant de décrire la structure de l'algorithme Raft, on résume rapidement la technique de réplication de machine à états qui permet de rendre un service tolérant aux pannes [Sch90]. Un service est représenté par une machine à états, qui traite les commandes d'un client. Ces commandes peuvent modifier l'état du service. Par exemple, un client interagit avec un dictionnaire par des commandes de lecture ou d'écriture d'une valeur associée à une clé. L'état d'un service peut être reconstitué à partir d'un état de départ et d'une série de commandes. Un service est rendu tolérant aux pannes de serveurs en répliquant sa machine à états sur plusieurs serveurs : l'état du service reste cohérent tant que tous les serveurs traitent la même série de commandes. Pour cela, les serveurs, qui peuvent tomber en panne à tout moment, doivent s'accorder sur la série des commandes à appliquer, ce qui revient à un problème de consensus.

Dans le cadre de la réplication de service, l'algorithme Raft est donc utilisé pour maintenir la cohérence d'une liste, appelée aussi *log* par la suite, de commandes entre plusieurs serveurs susceptibles de tomber en panne et qui communiquent par des messages asynchrones. Les pannes envisagées sont de type *fail-stop* : un serveur en panne s'arrête et stoppe toute communication, puis redémarre éventuellement au bout d'un certain temps (par opposition aux pannes dites *Byzantines* qui autorisent des comportements erratiques). Raft repose sur la présence d'un *leader* parmi les serveurs. Le leader est responsable de réceptionner les nouvelles commandes de la part d'un client, et de mettre à jour le log des autres serveurs. Plusieurs serveurs peuvent avoir le statut de leader au cours du temps, afin d'assurer la disponibilité du service lorsqu'un serveur leader tombe en panne. Raft est ainsi structuré en

3. Plusieurs implémentations de Raft sont référencées sur `http://raftconsensus.github.io`

deux phases principales : l'élection de leader et la réplication de log. Ces deux phases sont rapidement présentées par la suite, une description complète et argumentée est disponible dans la thèse de D. Ongaro [Ong14].

Phase d'élection de leader

Étant donné qu'un serveur leader peut tomber en panne, plusieurs serveurs peuvent prendre le rôle de leader au cours du temps. Raft décompose le temps en sessions (*term*) [4], qui sont numérotées selon un ordre croissant. Le mécanisme d'élection de leader doit permettre d'élire au plus un leader par session.

Un serveur est notamment défini par son état courant, son index de session et son log de commandes. Il existe trois états possibles pour un serveur : suiveur (*follower*), candidat (*candidate*) ou leader (*leader*). Le log d'un serveur est constitué d'une liste d'entrées (*log entries*), chaque entrée stocke une commande et la session à laquelle cette commande a été traitée. On définit une relation de comparaison entre deux logs : le log l_1 est plus "à jour" (*up-to-date*) que le log l_2 lorsque l'index de session de la dernière entrée de l_1 est plus grand que celui de la dernière entrée de l_2. Si la session est identique pour la dernière entrée de chaque log, alors c'est le log avec le plus d'entrées qui est le plus à jour.

Tous les serveurs démarrent dans l'état suiveur, et un suiveur peut décider de devenir candidat s'il ne reçoit aucun message pendant un certain délai de temps. Lorsqu'il devient candidat, un serveur incrémente son index de session, vote pour lui-même et envoie une requête de vote (*vote request*) à tous les autres serveurs. Un serveur accorde son vote à un candidat si les trois conditions suivantes sont validées :

— son index de session est égal à celui du candidat
— il n'a pas déjà accordé son vote à un autre candidat pour cette session
— le log du candidat est au moins autant à jour que le sien

Quand un candidat reçoit le vote d'une majorité de serveurs, il devient le leader pour cette session. Cette phase permet d'élire au plus un leader par session.

Lorsqu'un leader est élu, il se peut cependant qu'un autre serveur se considère encore leader pour une session antérieure. Pour atténuer cet effet, tous les messages (pour l'élection de leader, mais aussi pour la réplication de log) comportent l'index de session du serveur qui envoie ce message, et lorsqu'un serveur reçoit un message qui indique une session supérieure à la sienne, il met à jour son propre index de session et passe systématiquement dans l'état suiveur. Ce mécanisme ne permet pas d'éviter que différents serveurs se considèrent leader pour différentes sessions au même moment, mais il réduit la durée de ces situations, et plus généralement il permet de maintenir la progression de l'index de session au sein du groupe de serveurs.

4. Pour le vocabulaire spécifique à Raft, on précise les mots anglais utilisés par les auteurs du protocole.

Phase de réplication de log

Un leader est responsable de traiter les nouvelles commandes d'un client et de les répliquer sur les autres serveurs. Cependant, un leader peut tomber en panne ou bien être remplacé par un autre leader alors qu'il est en train de répliquer une commande. En conséquence, lorsqu'un serveur ajoute une entrée dans son log, il doit encore attendre la confirmation qu'une majorité de serveurs ont répliqué cette entrée avant qu'il puisse la considérer comme valide. En pratique, chaque serveur maintient un index de *commit* (*commit index*) qui indique jusqu'à quel index les entrées de son log sont considérées comme valides. Seules les commandes des entrées valides sont appliquées à la machine à états du service.

Un leader transfère les nouvelles entrées dans son log via des requêtes de réplication d'entrée (*append entry request*). Un serveur peut refuser une requête de réplication d'entrée, par exemple car ce serveur a déjà été contacté par un autre leader pour une période ultérieure. Une part importante de l'algorithme Raft consiste à définir sous quelles conditions un serveur accepte ou refuse une requête de réplication d'entrée. Nous indiquons ici le comportement d'un serveur Raft à la réception d'une telle requête, sans toutefois justifier en détail les raisons à l'origine des conditions qui conditionnent l'acceptation de requêtes. Ces justifications sont présentées dans l'article introduisant Raft [OO14].

Lorsqu'un serveur reçoit une requête de réplication d'entrée de la part d'un leader, il réagit de la manière suivante :

1. Si l'index de session du leader est antérieur à celui du serveur, alors le leader est périmé : le serveur répond par un échec de réplication et indique sa session, afin que le leader mette son index de session à jour et retourne à l'état suiveur.
2. Le serveur compare son entrée et celle du leader à l'index précédant celui de l'entrée à répliquer :
   (a) Si ces entrées "précédentes" ne sont pas cohérentes, c'est-à-dire si leur index de session ou leur commande diffèrent, alors le serveur refuse la requête. Dans ce cas, le leader essayera à nouveau une requête de réplication pour une entrée antérieure, jusqu'à retomber sur une entrée qui soit cohérente avec celle du serveur.
   (b) Si ces entrées "précédentes" sont égales, alors le serveur accepte la réplication, met à jour son log en conséquence et répond de manière positive au leader. Cette mise à jour peut nécessiter d'effacer une ou plusieurs entrées en fin de log, qui n'étaient pas valides. Enfin, le serveur met à jour son index de commit, qui vaut le minimum entre l'index de la dernière entrée dans son log et l'index de commit du leader.

Même lorsqu'il n'a pas de commande à répliquer, un leader envoie régulièrement une requête de réplication vide qui fait office de message de pulsation (*heartbeat*) aux autres serveurs.

Ces messages permettent d'empêcher les autres serveurs de passer trop de temps sans nouvelles et de déclencher une nouvelle élection de leader.

Raft nécessite que chaque serveur puisse enregistrer une partie de son état dans une mémoire qui survit aux pannes, c'est-à-dire qui n'est pas modifiée lors de l'arrêt et du redémarrage d'un serveur. Concrètement, un serveur doit pouvoir récupérer son log, son index de période ainsi que l'identifiant du serveur pour lequel il a voté — s'il a déjà voté — lors de cette dernière période.

6.3.2 Implémentation de Raft en LNT

Nous avons modélisé Raft en LNT, dans le but d'obtenir une implémentation distribuée avec DLC. Cette finalité influe sur plusieurs choix de modélisation, comme nous l'expliquons par la suite. Dans cette section, nous présentons uniquement les aspects du modèle qui nous intéressent, sans exposer la totalité de la spécification de l'algorithme. La spécification LNT complète de Raft est disponible en annexe B.

Communications entre serveurs

Le modèle de Raft est constitué de plusieurs processus serveurs qui s'exécutent en parallèle. Dans notre modèle, les serveurs interagissent entre eux en échangeant des messages via des rendez-vous sur la porte COM (pour "communication"). La composition parallèle utilise le rendez-vous à n-parmi-m pour imposer qu'une action sur la porte COM synchronise 2 serveurs parmi le groupe de serveurs :

```
par COM#2 in
   SERVER [COM, ...] (1)
|| SERVER [COM, ...] (2)
||  ...
end par
```

Chaque serveur est identifié par un entier naturel qu'il reçoit en argument et qu'il stocke dans sa variable locale "self". Cet identifiant est utilisé lors des rendez-vous sur la porte COM pour déterminer l'origine et la destination d'un message. Ainsi, les deux premières offres d'une action sur la porte COM sont toujours l'identifiant du serveur qui envoie un message, et l'identifiant du serveur à qui ce message est destiné, dans cet ordre. Pour envoyer ou recevoir un message, un serveur effectue donc des actions dans ce style :

```
-- Envoi d'un message au serveur "destination"
COM (self, destination , message)

-- Réception d'un message dans la variable "message",
-- l'identifiant de l'émetteur est stocké dans la variable "from"
COM (?from, self, ?message)
```

L'algorithme Raft est conçu pour fonctionner entre des processus qui communiquent par passage de messages asynchrones sur un réseau qui peut perdre ou dupliquer des messages. Ce modèle de communications peut être spécifié en LNT à l'aide de processus buffers explicites, d'une manière similaire à notre approche dans le chapitre 5, en introduisant cette fois la possibilité de perte ou de duplication de messages dans le comportement d'un buffer. Toutefois, notre but est de générer une implémentation à partir de ce modèle, et il est clair que les nombreuses interactions engendrées par l'ajout de processus buffers nuiront aux performances globales du système. C'est pourquoi nous avons fait en sorte que les serveurs interagissent par rendez-vous sur la porte COM.

Modélisation d'une panne de serveur

La panne d'un serveur est modélisée par une action sur la porte RESTART. Une panne peut arriver à n'importe quel moment, un serveur doit donc toujours être capable de réaliser une action sur la porte RESTART. En pratique, le processus LNT qui définit le comportement d'un serveur est constitué d'une boucle principale sur un choix non déterministe, dont une branche représente la panne du serveur :

```
process SERVER [COM, RESTART, ...] (self : serverID) is
   ...
   loop -- boucle principale
      select
         -- panne et redemarrage du serveur
         RESTART (self);
         ...
      []
         ...
      end select
   end loop
end process
```

Les autres branches de ce choix non déterministe représentent les comportements liés à l'algorithme Raft. Il faut alors veiller à ne pas créer de région atomique, c'est-à-dire sans possibilité de panne, dans ces autres branches. Considérons par exemple le cas du traitement d'une demande de vote : un serveur reçoit une demande de vote, la traite, puis renvoie la réponse. Si les deux communications de demande et de réponse ont lieu dans la même branche, alors aucune panne n'est possible entre les deux, et on crée une région atomique :

```
loop -- boucle principale
   select
      -- panne et redémarrage du serveur
      RESTART (self);
      ...
   []
      COM (?from, self, (* demande de vote *) );
      -- aucune panne possible avant la transmission de la réponse
```

```
        ...
        COM (self, from, (* réponse *) )
    []
        ...
    end select
end loop
```

Afin de permettre à un serveur de tomber en panne à tout moment, un buffer de message interne au processus est introduit. En pratique, lorsqu'un serveur veut envoyer un message, il place ce message dans une liste "buf". Ensuite, une autre branche du choix non déterministe réalise l'action de transmission de ce message sur la porte COM, ce qui permet d'avoir une panne entre les deux communications. De plus, une panne déclenche la perte des messages en attente de transmission :

```
loop -- boucle principale
    select
        -- panne et redémarrage du serveur
        RESTART (self);
        -- les messages en attente sont perdus
        buf := {};
        ...
    []
        COM (?from, self, (* demande de vote *));
        ...
        buf := append ( (* réponse *), buf)
    []
        ...
    []
        -- Envoi d'un message en attente
        ...
        COM (self, dest, (* message en tête de liste buf *))
    end select
end loop
```

On obtient ainsi une spécification qui permet une action sur la porte RESTART entre n'importe quelles actions sur la porte COM. La présence d'un buffer interne permet de considérer les communications entre serveurs comme asynchrones. De plus, un processus peut perdre des messages suite à une panne. On peut ainsi évaluer l'algorithme de consensus entre des processus susceptibles de tomber en panne et de perdre des messages.

Découverte d'un changement d'état manquant dans la spécification TLA$^+$

Lors de la rédaction du modèle LNT de l'algorithme, nous avons découvert qu'il manquait un changement d'état dans la spécification TLA$^+$ des auteurs de Raft. Plus précisément, lorsqu'un candidat pour la session s reçoit une requête de réplication de la part du leader de la session s, le candidat doit retourner à l'état suiveur. La spécification TLA$^+$ ne modifiait

pas l'état du serveur dans ce cas, le serveur restait donc candidat. Nous avons signalé ce changement d'état manquant aux auteurs, qui ont corrigé la spécification depuis [5].

Par chance, ce changement d'état manquant n'impactait pas le raisonnement de preuve conduit sur la spécification TLA$^+$. Un serveur en mode candidat pouvait en effet redevenir suiveur dans la même période après une panne et un redémarrage, il existait donc un autre moyen de passer de l'état candidat à celui de suiveur.

Néanmoins, cet oubli appuie l'importance d'avoir un langage de spécification dont la syntaxe soit proche d'un langage d'implémentation, afin de pouvoir suivre aisément le déroulement d'un algorithme lors de sa modélisation.

Modélisation d'un service de stockage

Un algorithme de consensus est souvent utilisé pour répliquer un service de stockage, comme un dictionnaire par exemple. Les commandes de lecture ou d'écriture qui s'appliquent sur un dictionnaire possèdent une clé en argument. Cependant, nous souhaitons garder le modèle de l'algorithme le plus simple et général possible, on représente donc les commandes par un simple type énuméré. Cela rend délicat la gestion de commandes avec argument. De plus, nous visons à évaluer l'algorithme de consensus, et non le service en lui-même. C'est pourquoi nous avons préféré mettre en place, pour notre évaluation, un service de stockage sous la forme d'une simple mémoire plutôt que d'un dictionnaire complet.

Notre service stocke une valeur booléenne. Nous définissons trois commandes possibles pour ce service, une commande de lecture et deux commandes de mise à jour de la valeur :

```
type command is
    read,
    writeFalse,
    writeTrue,
    empty_command -- utilisé pour les messages de pulsation
end type
```

Utilisation des fonctions crochets

Notre modèle utilise les fonctions crochets sur plusieurs actions. Les crochets nous permettent de retarder l'exécution d'une action, de contrôler le déclenchement d'une panne et enfin d'interagir avec un client externe à la spécification pour récupérer de nouvelles commandes à traiter.

5. Voir notre message sur la liste de discussion dédiée au développement de Raft, disponible à l'url suivante : `http://groups.google.com/forum/#!topic/raft-dev/yu-wOUx-gnA`

Délai d'attente sur la porte TIMEOUT. L'algorithme Raft nécessite un mécanisme de délai d'attente dans deux cas :

— un serveur dans l'état suiveur passe candidat quand il ne reçoit aucune communication pendant un certain "délai d'élection" (*election timeout*)

— lorsqu'un serveur dans l'état leader n'a pas besoin de répliquer de commandes, il diffuse un message de pulsation régulièrement après un "délai de pulsation" (*heartbeat timeout*)

Ces délais sont réalisés par des actions sur la porte TIMEOUT, retardées à l'aide des crochets de la porte. Nous utilisons pour cela la technique présentée dans la section 3.2.5. Cette fois, la durée d'attente est fixée par la première offre de l'action.

Contrôle des actions sur la porte RESTART. Si les actions sur la porte RESTART sont toujours possibles, un processus serveur de l'implémentation peut passer son temps à effectuer des redémarrages. On souhaite donc limiter la réalisation d'actions sur cette porte. Les crochets de la porte RESTART sont utilisés de manière similaire à celle exposée dans la section 3.2.2. Nous employons les crochets pour mettre en place un serveur TCP qui écoute sur le port 3000, et qui autorise une action sur la porte RESTART uniquement lorsqu'une connexion sur ce port est détectée. Cette approche permet de déclencher à distance une panne au sein d'une implémentation distribuée en cours d'exécution, en connaissant le nom de la machine sur laquelle la porte RESTART s'exécute.

Par exemple, considérons que la porte RESTART s'exécute sur une machine nommée "genepi-8". Il est possible de déclencher une panne depuis une autre machine de la manière suivante (l'outil netcat transfère son entrée standard sur la machine indiquée en paramètre, sur le port TCP passé lui aussi en paramètre) :

```
# Connection sur le port 3000 de la machine "genepi-8"
[ user@genepi-3 ] echo "" | netcat genepi-8 3000
Restart Server 2 OK
# On a déclenché une panne sur le serveur 2
```

Envoi de commandes à répliquer sur la porte CLIENT. Enfin, les fonctions crochets permettent de passer des commandes depuis l'environnement vers l'implémentation par des actions sur la porte CLIENT. Pour réaliser cette interaction, nous reprenons les techniques présentées dans les sections 3.2.3 et 3.2.4. De la même manière que pour la porte RESTART, nous avons mis en place un serveur TCP sur le port 3001 pour permettre de transférer des commandes depuis une machine distante.

Lorsqu'un crochet de pré-négociation sur la porte CLIENT reçoit une nouvelle commande via une connexion depuis un programme externe sur le port 3001 de la machine où s'exécute le programme de la porte, il autorise une action sur cette porte. Cette première action permet au leader actuel du groupe de serveurs Raft de récupérer la nouvelle commande, puis de la répliquer. Une deuxième action sur la porte CLIENT permet de transférer le

résultat de la commande au programme externe, avec lequel les crochets maintiennent la connexion TCP ouverte jusqu'à pouvoir envoyer une réponse. Cependant, la réplication de la commande peut échouer suite à une panne du leader par exemple. Lorsqu'un processus leader en train de répliquer une commande subit une panne, il effectue une action sur la porte CLIENT afin de prévenir que la réplication de la requête a peut-être échoué.

Remarque 6-1

En cas de panne durant la réplication, le leader ne peut pas affirmer si la commande a pu être répliquée sur une majorité de serveurs ou non. C'est au client de s'en assurer par la suite, en questionnant l'état du service. De manière générale, dans un système distribué, un client qui émet une requête et qui ne reçoit aucune réponse après un certain délai ne peut pas tirer de conclusion sur la réussite ou l'échec de sa commande. La gestion de l'interaction avec un client est plus subtile qu'il n'y parait au premier abord, D. Ongaro y consacre par exemple tout un chapitre dans sa thèse [Ong14]. Notre modèle considère qu'un serveur qui tombe en panne peut au moins avertir le client de cette panne. On note au passage que le modèle TLA$^+$ des auteurs de Raft ne traite pas le retour de réponse au client. ■

6.3.3 Performances de l'implémentation générée

Notre implémentation de l'algorithme Raft inclut le service de stockage et la gestion d'un client, et permet d'obtenir une implémentation distribuée via DLC. Elle est formée d'environ 400 lignes de code LNT et 300 lignes de code C pour les crochets, et la plus grande partie des crochets est dédiée à la gestion des connexions TCP avec un programme externe. Pour comparaison, la spécification TLA$^+$ des auteurs de Raft est composée d'environ 300 lignes de code.

Nous comparons les performances atteintes par notre implémentation, désignée "LNT-Raft" dans la suite, avec celles de Consul, un outil industriel qui offre un service de stockage sous forme de dictionnaire et qui utilise aussi l'algorithme Raft. Le code source de Consul est ouvert, et la bibliothèque qui implémente Raft est formée de 4000 lignes de code Go [6]. Notre expérience consiste à déployer un groupe de serveurs, chacun sur une machine distincte, et à mesurer le temps nécessaire pour traiter 1000 commandes en écriture sur le service de stockage.

La figure 6.4 illustre les performances de LNT-Raft et de Consul, pour différentes tailles de groupes de serveurs. Ces résulats sont des moyennes de plusieurs exécutions sur le cluster "chimint" du site de Lille de la grille Grid5000 (les machines de "chimint" sont cadencées à 2,40 Ghz). On constate que Consul est plus rapide que LNT-Raft, mais surtout que Consul est beaucoup moins affecté que LNT-Raft par le nombre de serveurs dans le groupe. Il se trouve que la bibliothèque Raft de Consul utilise une technique de regroupement de

6. La bibliothèque Raft de Consul est disponible à l'url suivante : `http://github.com/hashicorp/raft`

commandes qui permet de limiter le nombre de messages à échanger entre les serveurs. Plus précisément, avec cette technique, lorsque le leader reçoit une nouvelle commande, il attend un certain délai pendant lequel il groupe toutes les nouvelles commandes qu'il reçoit, puis il envoie une seule requête de réplication pour toutes ces nouvelles commandes. La technique de regroupement est simple mais très efficace, car elle permet de réduire le nombre de requêtes de réplication. Cela réduit le nombre de messages à transmettre sur le réseau, et augmente ainsi les performances.

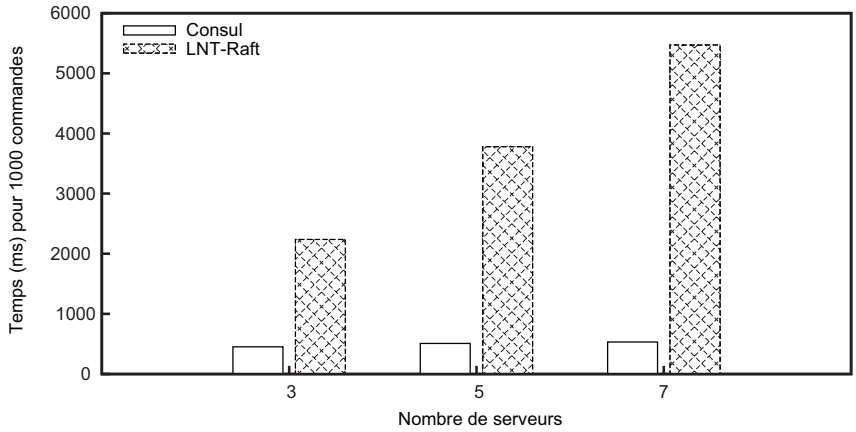


FIGURE 6.4 – Performances de LNT-Raft et de Consul

Nous ne sommes pas en mesure d'utiliser la technique de regroupement au sein de LNT-Raft car elle nécessite de faire passer des listes de commandes en tant qu'offre lors de rendez-vous sur la porte COM, or pour le moment DLC ne supporte pas les types complexes, tels que les listes, dans les offres. Dans LNT-Raft, le serveur leader envoie donc une requête de réplication à chaque serveur suiveur pour chaque commande à traiter. Ceci explique l'allure de la figure 6.4, où le temps de traitement de 1000 requêtes est proportionnellement impacté par le nombre de serveurs dans le groupe.

Il est d'ailleurs intéressant de ramener le temps de traitement aux nombres d'actions sur portes réalisées par LNT-Raft. Le tableau 6.1 résume le nombre d'actions réalisées lors du traitement de 1000 commandes pour différents nombre de serveurs. Étant donné que toutes les actions sur les portes CLIENT et COM font intervenir le serveur leader, elles ont lieu en séquence. On peut donc déduire le temps moyen requis pour exécuter une action au sein de LNT-Raft. Ce temps moyen indique qu'une implémentation générée par DLC peut réaliser plusieurs milliers d'actions sur porte en séquence par seconde. De plus, on constate que cette performance reste stable selon le nombre d'actions réalisées.

Enfin, la technique de regroupement utilisée par Consul est efficace pour traiter plusieurs nouvelles commandes qui arrivent de différents clients. En d'autres termes, cette technique permet d'assurer un bon débit dans le traitement de commandes. Par contre, la latence du système est fortement impactée par cette technique : lorsqu'un unique client soumet plusieurs commandes en séquence, le temps de traitement de chaque commande inclut

Nombre serveurs	3	5	7
Temps de traitement (ms)	2238	3779	5469
Nombre d'actions	6000	11000	14000
Temps moyen par action (ms)	0.373	0.344	0.390

Table 6.1

le délai d'attente du leader, qui patiente pour un éventuel regroupement de commandes. Les performances de Consul présentées sur la figure 6.4 correspondent à des commandes issues par plusieurs clients. Lorsque toutes les commandes émanent d'un unique client, les performances de Consul sont beaucoup moins bonnes, comme le montre la table 6.2 : le traitement de 500 commandes issues en séquence nécessite plus de 20 secondes [7].

Nombre de commandes envoyées en séquence	100	500
Temps (ms) de traitement (réplication sur 3 serveurs)	4974	23464

Table 6.2 – Latence de Consul pour des requêtes envoyées en série

6.3.4 Consensus et rendez-vous distribué

Il existe des similarités entre le consensus et le rendez-vous distribué : plusieurs processus doivent se mettre d'accord sur une certaine valeur.

Utilisation du consensus pour le rendez-vous distribué

Nous aurions pu utiliser un algorithme de consensus pour assurer le rendez-vous distribué. En effet, un rendez-vous peut être considéré comme un consensus entre les tâches qui se synchronisent. Afin d'assurer l'exclusion mutuelle des rendez-vous, toutes les tâches du système sont concernées par le consensus. Cependant, un consensus entre toutes les tâches du système revient à centraliser la décision de réalisation d'actions dans le système, ce qui empêche l'indépendance des rendez-vous qui ne sont pas conflits.

Il paraît envisageable de réaliser un consensus pour chaque groupe de rendez-vous qui sont en conflit, et ainsi de distribuer la réalisation d'actions dans le système. Toutefois, chaque tâche peut être prête sur différentes actions lorsqu'elle atteint un nouvel état. La configuration des groupes de rendez-vous en conflit est donc susceptible d'être modifiée après chaque action dans le système. Il nous semble complexe de redéfinir le découpage des groupes de consensus après chaque action.

7. Il n'est pas possible de désactiver le regroupement dans Consul. Le délai de regroupement est confondu avec celui de la pulsation, et réduire ce delai créerait de nombreux envois de messages de pulsation.

Enfin, les algorithmes de consensus sont conçus pour fonctionner sur des réseaux faillibles, et en présence de pannes de processus. Nos hypothèses sont plus fortes, car nous ne tolérons pas les pannes de tâches, et nous nous appuyons sur des échanges de messages sûrs et ordonnés entre deux programmes. Ces hypothèses nous ont permis de concevoir un protocole pour le rendez-vous multiple qui ne nécessite par exemple pas de compteur de session, ni de mécanisme de pulsation. Un protocole de consensus ne paraît pas vraiment adapté à notre problématique.

Utilisation du rendez-vous multiple pour le consensus

Le problème général du consensus consiste à accorder plusieurs processus sur une même valeur, en tolérant les pannes de processus et les pertes de messages. Ce problème sous-entend déjà que les processus interagissent par échanges de messages asynchrones. Considérons que les processus disposent du rendez-vous multiple pour interagir, mais qu'ils sont toujours susceptibles de tomber en panne.

Le rendez-vous multiple permet d'assurer un consensus entre les processus de manière relativement directe. Pour s'assurer de la participation d'une majorité de processus à l'enregistrement d'une valeur, il suffit en effet de réaliser un rendez-vous multiple entre ces processus. La construction de rendez-vous à n-parmi-m permet de facilement imposer une synchronisation à n'importe quel sous-ensemble de processus serveurs qui constitue une majorité.

Par exemple, considérons un groupe de 3 serveurs, dont n'importe quel sous ensemble de 2 serveurs constitue une majorité. L'enregistrement d'une nouvelle commande du client se fait par une interaction sur la porte NEW_CMD. La composition parallèle suivante impose qu'une interaction sur cette porte concerne le processus client et une majorité de serveurs :

```
par NEW_CMD in
   CLIENT [NEW_CMD, ...]
||
   par NEW_CMD #2 in -- synchronisation d'une majorité
      SERVER [NEW_CMD, ...] (...)
   || SERVER [NEW_CMD, ...] (...)
   || SERVER [NEW_CMD, ...] (...)
   end par
end par
```

Ainsi, lorsqu'une action est réalisée sur la porte NEW_CMD, le client a la garantie qu'une majorité de serveurs a aussi participé à cette action. De plus, si une majorité de serveurs est en panne, cette action ne peut pas être réalisée.

Nous sommes conscient qu'utiliser le rendez-vous multiple pour réaliser un consensus revient à ignorer le problème du réseau faillible. Cela est une manifestation du changement de niveau d'abstraction qu'offre le rendez-vous multiple. Tout comme le protocole TCP offre une abstraction de plus haut niveau que le protocole UDP pour les échanges de messages,

le rendez-vous multiple peut être considéré comme un opérateur de haut niveau pour la conception de systèmes concurrents.

Le rendez-vous multiple a le bénéfice d'être la primitive d'interaction de plusieurs langages formels. Dans le cadre de LNT, CADP offre des méthodes formelles pour vérifier la spécification de systèmes. Enfin, DLC propose maintenant un moyen d'obtenir automatiquement une implémentation distribuée de ces spécifications.

Conclusion

Bilan

La problématique principale abordée dans cette thèse est de concevoir et réaliser une technique de génération automatique d'implémentation distribuée à partir d'une spécification formelle LNT de processus concurrents, asynchrones, et non déterministes, qui interagissent par rendez-vous multiple avec échange de données. Le but est d'obtenir une implémentation qui soit correcte, qui puisse interagir avec son environnement, et dont les performances soient suffisantes pour pouvoir la considérer au moins comme un prototype valide.

Nous avons commencé par étudier différents protocoles existants pour l'implémentation du rendez-vous multiple, et nous avons choisi de baser nos travaux sur le protocole de Parrow et Sjödin [PS96]. Après avoir identifié le phénomène de synchronisation en avance de phase, nous avons apporté plusieurs modifications à ce protocole, tout d'abord pour le simplifier, puis pour le généraliser (communications asynchrones, plusieurs vecteurs de synchronisation par porte, actions internes, offres, confirmations d'action par une porte). Enfin, nous avons incorporé l'optimisation d'auto-verrouillage inspiré du protocole α-core [PCT04], ce qui a nécessité la conception du système de signal de purge.

Nous avons mis au point une méthode de vérification formelle, basée sur les outils de CADP, qui est dédiée aux protocoles de rendez-vous multiple. Cette méthode consiste à produire, à partir d'une spécification de système, le modèle LNT d'une implémentation de ce système utilisant un protocole pour assurer les rendez-vous. Ensuite, l'équivalence entre la spécification et le modèle de son implémentation, ainsi que l'absence de livelock ou de deadlocks dus au protocole sont vérifiées. Nous avons employé cette méthode au cours de la conception de notre protocole. Grâce à elle, nous avons détecté des problèmes que nous ne soupçonnions pas, notamment ceux liés à la cohabitation de l'auto-verrouillage et des synchronisations en avance de phase. L'automatisation de cette méthode nous a permis d'itérer sur la conception du protocole, par exemple lors de la mise en place du système de signal de purge. L'application de la méthode sur notre protocole ne relève ni livelock, ni deadlock, et indique, dans le cas général, une équivalence selon la relation de sûreté entre une spécification et son implémentation.

Notre méthode a aussi permis de découvrir de possibles interblocages dans le protocole

de Parrow et Sjödin [EL13], et de confirmer de possibles interblocages dans le protocole α-core – qui avaient déjà été signalés [KP10]. Chacun de ces protocoles était pourtant accompagné d'une preuve (ou d'une ébauche de preuve) manuelle ; la présence d'interblocages souligne l'importance d'utiliser, autant que possible, des méthodes de vérification automatisées.

Nous avons développé un mécanisme de fonctions crochets afin de rendre possible l'interaction entre l'implémentation et son environnement. Ce mécanisme permet à l'utilisateur, s'il le souhaite, de définir des fonctions crochets en C qui sont ensuite embarquées au sein de l'implémentation, et qui sont appelées à des moments précis au cours de l'exécution. Nous avons illustré comment utiliser ces fonctions crochets pour interagir avec d'autres systèmes de l'environnement, et pour contrôler la réalisation d'actions dans l'implémentation.

Le compilateur EXEC/CÆSAR, le protocole de synchronisation et le mécanisme d'interaction offrent les principaux ingrédients nécessaires pour une implémentation distribuée. Nous avons rassemblé ces ingrédients au sein de l'outil DLC, qui automatise entièrement la génération d'implémentation. Nous avons isolé la logique du protocole au sein de modules génériques, rédigés une fois pour toutes, qui sont ensuite employés dans les implémentations générées. Cette isolation permet de gagner en confiance sur la correction de l'implémentation du protocole.

Nous avons conduit trois expériences pour mesurer la rapidité d'exécution des implémentations produites par DLC. La première montre que, pour une barrière de synchronisation distribuée, le code généré est plus lent qu'une implémentation directe en C, mais plus rapide qu'une solution en Erlang ou en Java. Cette expérience permet d'illustrer l'efficacité de l'optimisation d'auto-verrouillage. La deuxième présente une implémentation du dîner des philosophes, qui repose sur l'utilisation du rendez-vous multiple. Les mesures de performances montrent notamment que des actions indépendantes sont bien réalisées en parallèle. La troisième est un cas d'étude complet sur l'algorithme de consensus Raft. Nous avons implémenté en LNT un service de stockage minimaliste répliqué à l'aide de cet algorithme. Nous avons mis à profit le mécanisme des fonctions crochets pour pouvoir contrôler les pannes et interagir avec le service de stockage depuis l'environnement. Nous avons comparé les performances de l'implémentation produite par DLC avec Consul, un outil industriel qui repose aussi sur Raft. Consul emploie une optimisation au niveau de Raft qui lui permet d'être plus rapide, et nous n'avons pas eu le temps de développer suffisamment DLC pour pouvoir profiter de cette optimisation. Cependant, nous avons constaté que si cette optimisation permet d'améliorer le débit de Consul, elle dégrade fortement sa latence. Enfin, nous avons montré par cette expérience que DLC produit des implémentations qui réalisent plusieurs milliers de rendez-vous à la seconde.

En résumé, nous avons réussi à concevoir une technique de génération automatique d'implémentation distribuée à partir d'une composition parallèle de processus LNT. Les implémentations produites utilisent un protocole validé par notre méthode de vérification, elles permettent de définir une interaction avec l'environnement via les fonctions crochets, et leurs performances nous semblent acceptables pour les employer au moins en tant que

prototypes. Cette technique a été mise en œuvre au sein du nouvel outil DLC, entièrement réalisé au cours de la thèse.

Perspectives

Nous avons amené DLC à un niveau de maturité qui permet de l'utiliser sur des exemples concrets. Nous envisageons plusieurs perspectives pour des travaux futurs.

Couverture du langage LNT

DLC ne couvre pas encore tout le langage LNT. Les limitations actuelles concernent l'utilisation de types complexes dans des offres d'action, et la gestion des gardes de communication. De plus, DLC ne permet pas encore la création dynamique de tâches.

Pour le moment, DLC permet de passer des offres de types "simples", les offres de types "complexes" tels que les listes, les ensembles ou les tableaux ne sont pas autorisées. La gestion de ces types complexes nécessite de pouvoir générer, pour n'importe quel type défini par l'utilisateur, des fonctions de soutien telles que la comparaison de valeurs ou la sérialisation en chaîne d'octets pour le transport de valeur sur le réseau. Nous pensons qu'au prix de quelques efforts techniques, les types complexes peuvent être rendus utilisables au sein des offres.

En ce qui concerne les gardes de communication, la principale limitation est l'interface du code généré par EXEC/CÆSAR. En effet, cette interface ne permet pas d'évaluer si une offre est valide selon une garde sans éventuellement déclencher l'action associée. La gestion des gardes dans DLC requiert donc en premier lieu une extension de l'interface de EXEC/CÆSAR. Nous pensons qu'une bonne interface offrirait chaque garde de communication comme une fonction indépendante qui reçoit toutes ses variables en argument, y compris les variables qui ne font pas partie des offres. Il serait alors possible de regrouper les gardes au niveau des portes. L'interface ferait aussi en sorte que lorsqu'une tâche est prête sur une action gardée, elle puisse indiquer à la porte quelle garde doit être validée, et passer – en plus des offres – toutes les variables nécessaires à l'évaluation de la garde. Ainsi, la porte pourrait tester quelles valeurs d'offres valident les éventuelles gardes de chaque tâche avant même de démarrer une négociation.

Une tâche peut elle-même contenir une composition parallèle de processus, qui se retrouve implémentée au sein du programme séquentiel produit par EXEC/CÆSAR. Lorsque le comportement d'une tâche atteint une composition parallèle, il serait intéressant de pouvoir générer dynamiquement une nouvelle tâche pour chaque processus de la composition parallèle, afin de continuer à distribuer l'exécution autant que possible. Cette génération dynamique de tâches nécessite tout d'abord des modifications au niveau du compilateur EXEC/CÆSAR, qui doit offrir un moyen d'accéder, à l'exécution, à un programme qui

implémente chaque nouvelle tâche à démarrer. Ensuite, le protocole de synchronisation doit être en mesure de modifier, lors de l'exécution, les vecteurs de synchronisation des portes, afin de prendre en compte de nouvelles tâches. Enfin, il faut aussi assurer toute la mécanique nécessaire au niveau de l'implémentation pour amorcer l'exécution des nouvelles tâches, et leur permettre de communiquer avec les portes et les tâches déjà en cours d'exécution.

Affinage du mécanisme d'interaction avec l'environnement

Nous avons mis en place un mécanisme de fonctions crochets qui nous a permis de définir les interactions nécessaires lors de notre cas d'étude sur l'algorithme Raft. Cependant, nous ne sommes pas arrêtés sur la version courante des fonctions crochets, et nous considérons que ce mécanisme est susceptible d'être amélioré. Nous pensons que ce mécanisme peut être affiné grâce à des nouveaux cas d'étude concrets qui utilisent les fonctions crochets, afin de déterminer quels sont les besoins d'un utilisateur de DLC.

En particulier, les actions retardées sont pour l'instant possibles au prix d'une attente active sur un crochet de pré-négociation. Il serait évidemment bénéfique d'éviter cette attente active autant que possible. Nous envisageons de modifier la sémantique de la valeur de retour du crochet de pré-négociation, afin que l'utilisateur puisse indiquer qu'il ne souhaite pas démarrer de négociation pour le moment, mais que la porte pourra réitérer un appel à ce crochet pour la même action après un certain délai de temps. De plus, comme nous en avons déjà discuté en section 3.2.5, il est envisageable d'incorporer des notions de délai directement au sein du langage LNT.

Améliorations pour le protocole

Nous envisageons plusieurs améliorations pour le protocole de synchronisation. Tout d'abord, nous pensons que les actions sur les portes qui n'entraînent pas de synchronisation entre plusieurs tâches pourraient être résolues au niveau de la tâche directement, comme c'est le cas pour les actions internes. Cela éviterait d'avoir à effectuer des négociations pour ces actions. Cette approche n'est pas compatible avec la version courante des fonctions crochets, qui nécessitent d'avoir un unique point de décision pour toutes les actions d'une même porte. Cependant, comme nous l'avons évoqué précédemment, le mécanisme des fonctions crochets peut encore être modifiée, notamment pour permettre cette amélioration.

D'une manière plus générale, nous pensons qu'il est intéressant d'étudier la partition du protocole. Par exemple, dans le cas où une porte possède plusieurs vecteurs de synchronisations qui ne sont pas en conflit, il nous semble possible de séparer cette porte en plusieurs points de synchronisation distincts. L'idéal serait ensuite de pouvoir associer des points de synchronisation à des tâches, afin de limiter le nombre de processus auxiliaires dans l'implémentation.

Nous pensons aussi étudier l'impact du choix de l'ordre de verrouillage des tâches sur les performances du protocole. En effet, il nous paraît pertinent de commencer par verrouiller les tâches qui sont prêtes sur plusieurs portes afin que les négociations en concurrence soient départagées le plus tôt possible : celles qui arrivent à verrouiller les tâches les plus demandées continuent et, surtout, cela permet de bloquer assez tôt les négociations qui échouent.

Notre méthode de vérification formelle de protocole peut continuer à être un soutien pour itérer sur le développement de ces améliorations, en s'assurant que la correction du protocole est maintenue. Nous pensons aussi ajouter à cette méthode des éléments d'évaluation de performances [CGH+10], afin d'estimer l'impact d'une amélioration directement sur le modèle de l'implémentation.

Preuve formelle du protocole

Notre méthode de vérification formelle permet de valider l'absence de problème dans notre protocole sur une série de tests. Cependant, une preuve formelle du protocole serait plus satisfaisante, en démontrant l'existence d'une équivalence entre une spécification et son implémentation.

Notre méthode de vérification permet de conjecturer que la relation d'équivalence de sûreté est toujours vérifiée, mais cela reste à prouver. De plus, il serait intéressant d'étudier, et éventuellement de définir, s'il existe une relation d'équivalence qui soit plus forte que celle de sûreté et qui soit vérifiée dans le cas général, même si l'on sait par l'expérience que cette relation d'équivalence ne fait pas partie de celles présentes dans CADP. Ces travaux pourront s'inspirer de ceux réalisés dans le cadre de l'étude du π-calcul [PSN11, PNG13].

Vérification à l'exécution

Nous savons qu'il est possible de reconstruire, à partir des traces d'exécution de chaque tâche, une trace d'exécution de l'implémentation distribuée. En effectuant cette reconstruction à l'exécution, par exemple au niveau du nœud central qui collecterait les traces de toutes les tâches, il serait envisageable d'utiliser cette trace en entrée d'outils de CADP pour conduire des vérifications à l'exécution (*runtime verification*). De récentes améliorations du *model checker* EVALUATOR4 permettent de vérifier plusieurs types de propriétés sur une trace séquentielle de taille quelconque, avec un coût mémoire borné.

Pour les systèmes dont l'espace d'états est trop grand pour les méthodes de vérification énumérative, la vérification à l'exécution couplée à DLC offre un moyen d'obtenir une implémentation distribuée dont chaque exécution peut être validée. Nous envisageons aussi de coupler la détection d'une violation de propriété avec le déclenchement d'une nouvelle fonction crochet, qui pourrait entraîner l'arrêt d'urgence de l'implémentation.

Applications pratiques

Nous considérons que la mise à disposition du rendez-vous multiple comme primitive d'interaction facilite la programmation concurrente. En effet, plusieurs problèmes classiques, tels qu'une barrière de synchronisation, l'élection d'un leader, ou une diffusion ordonnée de messages, se résolvent de manière directe à l'aide du rendez-vous multiple. Il nous semble intéressant d'explorer plus généralement comment cette primitive peut être mise à profit dans la conception de systèmes concurrents.

En particulier, un des domaines d'application pratique est la réalisation de méthodes de vérification énumérative qui soient distribuées. Ces méthodes permettent de traiter de grands espaces d'états en exploitant les ressources de plusieurs machines. Elles sont cependant complexes, et difficiles à implémenter correctement. DLC offre un moyen d'utiliser LNT lors du développement de méthodes de vérification distribuées, ce qui permet de bénéficier d'un langage formel et d'une abstraction de haut niveau, à travers le rendez-vous multiple, pour spécifier les interactions entre processus.

Enfin, nous pensons que DLC permet de favoriser l'utilisation des méthodes formelles. Les ingénieurs de l'industrie font régulièrement la remarque que lorsqu'une méthode formelle est appliquée à un modèle, elle n'offre alors pas de garantie sur l'implémentation réelle. Un ingénieur peut maintenant profiter de LNT et de sa syntaxe proche des langages d'implémentation pour modéliser un système concurrent, puis exploiter les outils de CADP pour conduire des vérifications formelles sur ce système, et enfin obtenir automatiquement une implémentation distribuée grâce à DLC.

Annexe A

Vérification de protocole de synchronisation

Dans cette annexe, nous présentons les sources liées à notre méthode de vérification. La première section comporte la spécification LNT du protocole de synchronisation. La seconde section contient le script SVL qui décrit le flot de vérification de notre méthode.

A.1 Spécification LNT du protocole utilisé dans DLC

Nous présentons ici toute la partie générique de la spécification LNT du protocole de synchronisation. Les parties qui ne sont pas présentées ici sont celles qui dépendent du système sur lequel notre méthode est appliquée, notamment l'espace d'états des tâches et la composition parallèle générale, comme nous l'avons vu à la section 5.1.

A.1.1 Types de données

Les types de données s'appuyent sur l'existence d'un type "DLC_ID" qui contient les identifiants des tâches et des portes. Ce type est défini automatiquement à partir du système, et il contient toujours le constructeur "DLC_NULL_ID".

Les autres types sont définis de la manière suivante :

```
1 type nat_set is
2    set of nat
3    with "length", "access", "member"
4 end type
5
6 type id_set is
```

```
7     sorted set of DLC_ID
8     with "head", "length", "access", "member", "diff", "union", "remove", "empty", "inter"
9  end type
10
11 type sync_vect_list is
12     list of id_set
13    with "head", "access", "length"
14 end type
15
16 type sync_map_entry is
17    sync_map_entry (gate : DLC_ID, vect_list : sync_vect_list)
18    with "get"
19 end type
20
21 type sync_map is
22     list of sync_map_entry
23    with "access", "length"
24 end type
25
26 type dlc_action is
27    action (gate : DLC_ID)
28    with "get", "=="
29 end type
30
31 type action_set is
32    set of dlc_action
33    with "length", "access", "member"
34 end type
35
36 type transition is
37     nil_transition ,
38     transition (action : dlc_action, next_states : nat_set)
39    with "get", "=="
40 end type
41
42 type transition_list is
43     list of transition
44 end type
45
46 type state is
47     nil_state ,
48     state ( id : nat, transitions : transition_list )
49    with "get"
50 end type
51
52 type state_list is
53     list of state
54 end type
55
56 type lock is
57    lock (action : dlc_action, index : nat, path : id_set, confirm : bool, purge : id_set)
```

```
58      with "get", "set"
59  end type
60
61  type lock_list is
62      list of lock
63      with "empty", "append", "head", "length", "access", "tail"
64  end type
65
66  type message is
67      READY (autolocked : bool),
68      LOCK (lock : lock),
69      COMMIT,
70      COMMIT (purge : id_set),
71      ABORT,
72      ABORT (purge : id_set)
73  end type
74
75  type message_list is
76      list of message
77      with "append", "head", "tail", "length", "empty"
78  end type
79
80  type arrival is
81      arrival (action : dlc_action, arrival : nat)
82      with "get"
83  end type
84
85  type arrival_list is
86      list of arrival
87      with "access", "length"
88  end type
89
90  type gate_state is
91      idle,
92      dealing
93      with "=="
94  end type
95
96  type manager_state is
97      free,
98      locked,
99      autolock_free,
100     autolock_locked
101     with "==", "!="
102 end type
```

A.1.2 Fonctions

Les fonctions sont définies de la manière suivante :

```
1  function find_state (space : state_list , id : nat) : state is
2     case space in
3     var i : nat, tra : transition_list , tail : state_list in
4        {} -> return nil_state
5     | cons(state (i , tra ), any state_list ) where i == id ->
6           return state (i , tra )
7     | cons (any state , tail ) ->
8           return find_state ( tail , id )
9     end case
10 end function
11
12 function find_transition (tl : transition_list , act : dlc_action) : transition is
13    case tl in
14    var a : dlc_action , nl : nat_set, tail : transition_list in
15       {} -> return nil_transition
16    | cons ( transition ( a, nl ), any transition_list ) where a == act ->
17          return transition (a, nl)
18    | cons (any transition , tail ) ->
19          return find_transition ( tail , act)
20    end case
21 end function
22
23 function get_next(space : state_list , id : nat, action : dlc_action) : nat_set is
24    var t : transition in
25       t := find_transition ( get_transitions (find_state (space, id )), action) ;
26       if t == nil_transition then
27          return {}
28       else
29          return get_next_states (t)
30       end if
31    end var
32 end function
33
34 function collect_action ( tl : transition_list , al : action_set ) : action_set is
35    case tl in
36    var act : dlc_action , tail : transition_list in
37       {} -> return al
38    | cons ( transition (act , any nat_set) , tail ) ->
39          return collect_action ( tail , insert (act, al))
40    | cons ( nil_transition , tail ) ->
41          -- should never happen, remove compiler warning
42          return collect_action ( tail , al)
43    end case
44 end function
45
46 function possible_actions (space : state_list , id : nat) : action_set is
47    return collect_action ( get_transitions (find_state (space, id )), {})
48 end function
49
50 function extract_gate (al : action_set , gl : id_set) : id_set is
51    case al in
```

```
52      var g : DLC_ID, tail : action_set in
53          {} -> return gl
54      |   cons (action (g), tail ) ->
55              return extract_gate ( tail , insert (g, gl))
56      end case
57  end function
58
59  function arrival_state (dl : arrival_list , act : dlc_action) : nat
60      raises action_not_found : none
61  is
62      var n : nat in
63          for n := 1 while n <= length (dl) by n := n+1 loop
64              if get_action (access (dl, n)) == act then
65                  return get_arrival (access (dl, n))
66              end if
67          end loop;
68          raise action_not_found
69      end var
70  end function
71
72  function isin (vect, rdytask : id_set) : bool is
73      var n : nat in
74          for n := 1 while n <= length (vect) by n := n+1 loop
75              if not (member (access (vect, n), rdytask)) then
76                  return false
77              end if
78          end loop ;
79          return true
80      end var
81  end function
82
83  function possible_rdv (rdytask : id_set, vectors : sync_vect_list) : bool is
84      var vect : id_set, n : nat in
85          for n := 1 while n <= length (vectors) by n := n+1 loop
86              vect := (access (vectors, n));
87              if isin (vect, rdytask) then
88                  return true
89              end if
90          end loop ;
91          return false
92      end var
93  end function
94
95  function list_rdv_index (rdytask : id_set, vectors : sync_vect_list) : nat_set is
96      var vect : id_set, n : nat, result : nat_set in
97          result := {};
98          for n := 1 while n <= length (vectors) by n := n+1 loop
99              vect := (access (vectors, n));
100             if isin (vect, rdytask) then
101                 result := insert (n, result)
102             end if
```

```
103         end loop ;
104         return result
105     end var
106 end function
107
108 function lock_state (inout manager : manager_state) raises invalid_state : none is
109     case manager in
110         free          -> manager := locked
111     |   autolock_free -> manager := autolock_locked
112     |   any           -> raise invalid_state
113     end case
114 end function
115
116 function get_sync_vect (lock : lock, gsm : sync_map) : id_set is
117     var g : DLC_ID, n, index : nat in
118         g := get_gate (get_action (lock));
119         index := get_index (lock);
120         for n := 1 while n <= length (gsm) by n := n+1 loop
121             if get_gate (access (gsm, n)) == g then
122                 return access (get_vect_list ( access (gsm, n)), index)
123             end if
124         end loop ;
125         return {} of id_set
126     end var
127 end function
128
129 function next_task ( task : DLC_ID, vect : id_set) : DLC_ID is
130     var n : nat in
131         for n := 1 while n < length (vect) by n := n+1 loop
132             if task == access (vect, n) then
133                 return access (vect, n+1)
134             end if
135         end loop ;
136         return DLC_NULL_ID
137     end var
138 end function
139
140 function update_purge (in out purgel : id_list , purge : id_list , in out autolock : id_set) is
141     var id : DLC_ID, newpurge : id_list in
142         purgel := union (purgel, purge);
143         newpurge := {};
144         while not (empty (purgel)) loop
145             id := head (purgel);
146             if member (id, autolock) then
147                 autolock := remove (id, autolock)
148             else
149                 newpurge := cons (id, newpurge)
150             end if;
151             purgel := tail (purgel)
152         end loop;
153         purgel := newpurge
```

```
154     end var
155 end function
```

A.1.3 Canaux de communications (channels)

Les canaux de communications sont définis de la manière suivante :

```
1  channel com is
2      (DLC_ID, message)
3  end channel
4
5  channel annonce is
6      (DLC_ID, id_set)
7  end channel
```

A.1.4 Spécification des processus

Processus buffer

Un processus buffer est spécifié de la manière suivante :

```
1  function BUFSIZE : nat is
2      return 1
3  end function
4
5  process BUFFER [GETFROM, SENDTO : com] (from, to : DLC_ID) is
6      var
7          msg : message,
8          mq  : message_list
9      in
10         mq := {};
11         loop
12             select
13                 only if length (mq) < BUFSIZE then
14                     GETFROM (to, ?msg);
15                     mq := append (msg, mq)
16                 end if
17             []
18                 only if not (empty (mq)) then
19                     SENDTO (from, head (mq));
20                     mq := tail (mq)
21                 end if
22             end select
23         end loop
24     end var
```

```
25 end process
```

Processus porte

Le comportement d'une porte est spécifié de la manière suivante :

```
1  process GATE [SEND, RECV : com, ACTION, HOOK_REFUSE : annonce]
2              (gate : DLC_ID, vectors : sync_vect_list)
3  is
4     var
5        state          : gate_state, -- etat de la porte
6        readyset       : id_set,     -- taches pretes
7        autolock       : id_set,     -- taches autoverrouillees
8        dealreadyset   : id_set,     -- taches pretes pdt une negociation
9        dealautolock   : id_set,     -- taches autoverrouillees pdt une negociation
10       dealvect       : id_set,     -- vecteur de synchro de la negociation
11       dealindex      : nat,        -- index du vecteur de negociation
12       dealpath       : id_set,     -- chemin de verrouillage
13       purgelist      : id_list,    -- le prochain rdy(lck) devient rdy
14       -- variables de stockage temporaire
15       n              : nat,
16       task           : DLC_ID,
17       lock           : lock,
18       confirm        : bool,
19       purge          : id_list,
20       autolocked     : bool,
21       vectindexes    : nat_set
22    in
23       -- initialisation
24       state          := idle;
25       readyset       := {};
26       autolock       := {};
27       dealreadyset   := {};
28       dealautolock   := {};
29       dealvect       := {};
30       purgelist      := {};
31       dealpath       := {}; -- evite une erreur de compilation 'non init'
32
33       -- boucle principale
34       loop
35          select
36             -- Recoit un message READY
37             RECV (?task, ?READY (autolocked));
38
39             if member (task, purgelist) and (autolocked) then
40                purgelist := delete (task, purgelist);
41                -- ignore l'auto-verrouillage
42                autolocked := false
43             end if;
```

```
44
45              if state == dealing then
46                 dealreadyset := insert (task, dealreadyset);
47                 if autolocked then
48                    dealautolock := insert (task, dealautolock)
49                 end if
50              else
51                 readyset := insert (task, readyset);
52                 if autolocked then
53                    autolock := insert (task, autolock)
54                 end if
55              end if
56
57           []
58              -- Commence une negociation
59              only if (state == idle) and (possible_rdv (readyset, vectors)) then
60                 vectindexes := list_rdv_index (readyset, vectors);
61                 -- Choisi au hasard parmi les synchros possibles
62                 dealindex := any nat where member (dealindex, vectindexes);
63                 dealvect  := access (vectors, dealindex);
64                 dealpath  := diff (dealvect, autolock);
65                 if empty (dealpath) then
66                    -- Toutes les taches sont autoverrouillees
67                    select
68                       -- Refuse l'action pour des raisons externes
69                       HOOK_REFUSE (gate, dealvect)
70                    []
71                       -- Accepte l'action et diffuse le resultat
72                       ACTION (gate, dealvect);
73                       for n := 1 while n <= length (dealvect) by n := n+1 loop
74                          SEND (access (dealvect, n), COMMIT)
75                       end loop;
76                       readyset := diff (readyset, dealvect);
77                       autolock := diff (autolock, dealvect)
78                    end select
79                 else
80                    task := head (dealpath);
81                    -- La necessite d'une confirmation est choisie au hasard
82                    confirm := any bool;
83                    SEND (task, LOCK (lock (action(gate), dealindex, dealpath,
84                                            confirm, {})));
85                    -- Passe dans l'etat negociation
86                    dealreadyset := {};
87                    dealautolock := {};
88                    state := dealing
89                 end if
90              end if
91           []
92              -- Recoit un message COMMIT
93              only if state == dealing then
94                 RECV (?task, ?COMMIT (purge) of message);
```

```
95
96              readyset := diff (readyset, dealvect);
97              readyset := union (readyset, dealreadyset);
98              readyset := remove (task, readyset);
99              -- idem pour autolock
100             autolock := diff (autolock, dealvect);
101             autolock := union (autolock, dealautolock);
102             autolock := remove (task, autolock);
103
104             eval update_purge (!?purgelist, purge, !?autolock);
105             state := idle
106         end if
107     []
108         -- Recoit un message ABORT
109         only if state == dealing then
110             RECV (?task, ?ABORT (purge) of message);
111
112             readyset := remove (task, readyset);
113             readyset := union (readyset, dealreadyset);
114             -- idem pour autolock
115             autolock := remove (task, autolock);
116             autolock := union (autolock, dealautolock);
117
118             eval update_purge (!?purgelist, purge, !?autolock);
119             state := idle
120         end if
121     []
122         -- Recoit un message LOCK
123         only if state == dealing then
124             RECV (?task, ? LOCK (lock) of message);
125             select
126                 HOOK_REFUSE (gate, dealvect);
127                 for n := 1 while n <= length (dealpath) by n := n+1 loop
128                     SEND (access (dealpath, n), ABORT)
129                 end loop;
130                 -- update idem qu'a la reception d'un ABORT
131                 readyset := union (readyset, dealreadyset);
132                 autolock := union (autolock, dealautolock)
133
134             []
135                 ACTION (gate, dealvect);
136                 for n := 1 while n <= length (dealvect) by n := n+1 loop
137                     SEND (access (dealvect, n), COMMIT)
138                 end loop;
139
140                 readyset := diff (readyset, dealvect);
141                 readyset := union (readyset, dealreadyset);
142                 readyset := remove (task, readyset);
143                 -- idem pour autolock
144                 autolock := diff (autolock, dealvect);
145                 autolock := union (autolock, dealautolock);
```

```
146                     autolock := remove (task, autolock)
147
148                 end select;
149
150                 eval update_purge (!?purgelist, lock.purge, !?autolock);
151                 state := idle
152             end if
153         end select
154     end loop
155   end var
156 end process
```

Processus manager

Le comportement d'un manager est spécifié de la manière suivante :

```
1  process MANAGER [SEND, RECV : com, ACTION : annonce]
2                  (task : DLC_ID, statespace : state_list, map : sync_map)
3  is
4    var
5      manager     : manager_state, -- etat du manager
6      actions     : action_set,    -- actions possibles pour la tache
7      arriv_list  : arrival_list,  -- liste de paires (action, destination)
8      taskstate   : nat,           -- etat courant de la tache
9      waitlock    : lock_list,     -- verrous en attente
10     lock        : lock,          -- verrou actif
11     action      : dlc_action,    -- prochaine action a realiser
12     internal    : bool,          -- une action interne est possible
13     sigpurge    : bool,          -- on doit se signaler a la purge
14     -- variables de stockage temporaire
15     n           : nat,
16     l           : lock,
17     to, gate    : DLC_ID,
18     vect        : id_set
19   in
20     -- initialisation
21     taskstate := 0;
22     waitlock  := {};
23
24     -- boucle principale
25     loop
26       -- Reinitialisation du manager
27       manager   := free;
28       internal  := false;
29       -- Evite les warnings du compilateur
30       action    := action (DLC_NULL_ID);
31       sigpurge  := false;
32
33       -- Recupere les actions possibles pour la tache
```

```
34          actions := possible_actions (statespace, taskstate);
35
36          -- Choix externe : quand une meme action peut mener a des etats
37          -- differents, il faut choisir *avant* la negociation quel etat
38          -- sera atteint si cette action a lieu.
39           arriv_list    := {};
40          for n := 1 while n <= length(actions) by n := n+1 loop
41             var dest_set : nat_set, dest : nat, act : dlc_action in
42                act := access (actions, n);
43                dest_set := get_next (statespace, taskstate, act);
44                -- on choisi un etat d'arrivee au hasard
45                dest := any nat where member (dest, dest_set);
46                 arriv_list := cons ( arrival (act, dest), arriv_list )
47             end var
48          end loop;
49
50          if (length (actions) == 1)
51             and ((get_gate (access (actions, 1))) != DLC_GATE_I)
52          then
53             -- Autoverrouillage
54             action := access (actions, 1);
55             SEND (action.gate, READY (true));
56             manager := autolock_free;
57             sigpurge := true
58          else
59             for n := 1 while n <= length (actions) by n := n+1 loop
60                gate := get_gate (access (actions, n));
61                if (gate == DLC_GATE_I) then
62                   internal := true
63                else
64                   SEND (gate, READY (false))
65                end if
66             end loop
67          end if;
68
69          -- Boucle de negociation
70          loop NEGOCIATION in
71             select
72                -- Recoit un message LOCK
73                RECV (? any DLC_ID, ?LOCK (l) of message);
74                waitlock := append (l, waitlock)
75             []
76                -- Traitement du plus vieux verrou en attente
77                only if not (empty (waitlock))
78                   and ((manager == free) or (manager == autolock_free))
79                then
80                   lock     := head (waitlock);
81                   waitlock := tail (waitlock);
82
83                   if member (lock.action, actions) then
84
```

```
85                  -- on se signale au plus une fois à purger
86                  if (manager == autolock_free) and (sigpurge) then
87                     lock := lock.{purge => cons (task, lock.purge)};
88                     sigpurge := false
89                  end if;
90
91                  action := lock.action;
92                  if task == access (lock.path, length (lock.path)) then
93                     -- Fin du chemin de verrouillage
94                     if lock.confirm then
95                        -- Demande confirmation a la porte
96                        SEND (lock.action.gate, LOCK (lock));
97                        eval lock_state (!?manager)
98                     else
99                        -- Conclut la negociation
100                       vect := get_sync_vect (lock, map);
101                       ACTION (lock.action.gate, vect);
102                       SEND (lock.action.gate, COMMIT (lock.purge));
103                       for n := 1 while n <= length(vect) by n := n+1 loop
104                          to := access(vect, n);
105                          if to != task then
106                             SEND (to, COMMIT)
107                          end if
108                       end loop;
109                       break NEGOCIATION
110                    end if
111                 else
112                    -- Transmission du verrou
113                    to := next_task (task, lock.path);
114                    SEND (to, LOCK (lock));
115                    eval lock_state (!?manager)
116                 end if
117              else
118                 -- Refus du verrou
119                 SEND (lock.action.gate, ABORT (lock.purge));
120                 for n := 1 while n <= length (lock.path) by n := n+1 loop
121                    to := access (lock.path, n);
122                    if to < task then
123                       SEND (to, ABORT)
124                    end if
125                 end loop
126              end if
127           end if
128        []
129           -- Reception d'un message COMMIT
130           only if manager != free then
131              RECV (? any DLC_ID, COMMIT);
132              break NEGOCIATION
133           end if
134
135        []
```

```
136                 -- Reception d'un message ABORT
137                 only if (manager == locked) or (manager == autolock_locked) then
138                     RECV (? any DLC_ID, ABORT);
139                     if manager == locked then
140                         manager := free
141                     elsif manager == autolock_locked then
142                         manager := autolock_free
143                     end if
144                 end if
145             []
146                 -- Decide d'effectuer une action interne
147                 only if (manager == free) and (internal) then
148                     ACTION (DLC_GATE_I, {task} of id_set);
149                     action := action (DLC_GATE_I);
150                     break NEGOCIATION
151                 end if
152             end select
153         end loop; -- Fin de boucle NEGOCIATION
154
155         -- Refus des verrous en attente
156         while not (empty (waitlock)) loop
157             l := head (waitlock);
158             waitlock := tail (waitlock);
159             SEND (l.action.gate, ABORT (l.purge));
160             for n := 1 while n < length (l.path) by n := n+1 loop
161                 to := access (l.path, n);
162                 if to < task then
163                     SEND (to, ABORT)
164                 end if
165             end loop
166         end loop;
167
168         -- Recupere l'etat d'arrivee choisi pour l'action effectuee
169         taskstate := arrival_state (arriv_list, action)
170
171     end loop -- Fin de boucle principale
172   end var
173 end process
```

A.2 Script de vérification SVL

Le script SVL qui définit le scénario de notre méthode de vérification est le suivant :

```
1  -- Génération d'espaces d'états
2
3  -- La version "detail" est utilisée pour obtenir des diagnostiques détaillés
4  "detail_implem.bcg" = strong reduction of "implem.lnt";
5
6  "implem.bcg" = strong reduction of
7     gate hide all but "ACTION" in "detail_implem.bcg";
8
9  (*************************************************************************)
10 -- Vérification par équivalence
11
12 -- Prépare les espaces d'états pour la comparaison
13 "equivalence_implem.bcg" = total rename "ACTION !DLC_GATE_\([^ ]*\) .*" -> "\1"
14    in "implem.bcg";
15 "equivalence_spec.bcg" = total rename "i" -> "I", "exit" -> "EXIT"
16    in "spec.bcg";
17
18 property Equivalence (RELATION, RESULT)
19    "Specification and implementation are $RELATION equivalent."
20 is
21    % DEFAULT_COMPARISON_RELATION="$RELATION"
22    "$RELATION.bcg" = comparison "equivalence_spec.bcg" == "equivalence_implem.bcg";
23    result "$RESULT"
24    expected TRUE
25 end property
26
27 check Equivalence ("strong", "result_strong");
28 % if [ $result_strong = FALSE ] ; then
29      check Equivalence ("trace", "result_trace");
30      check Equivalence ("branching", "result_branching");
31 %    if [ $result_branching = FALSE ] ; then
32           check Equivalence ("tau*.a", "result_taustar");
33           check Equivalence ("observational", "result_observational");
34 %         if [ $result_taustar = FALSE -a $result_observational = FALSE ] ; then
35               check Equivalence ("safety", "result_safety");
36 %         fi
37 %    fi
38 %    if [ $result_branching = FALSE -a $result_trace = FALSE ] ; then
39           check Equivalence ("weak trace", "result_weaktrace");
40 %    fi
41 % fi
42
43 (*************************************************************************)
44 -- Livelock
45
46 "livelock_implem.bcg" = divbranching reduction of
```

```
47    gate hide all but "ACTION", "HOOK_REFUSE" in "detail_implem.bcg";
48
49  property Absence_Livelock
50    "There is no livelock due to the protocol."
51  is
52    "livelock.bcg" = livelock of "livelock_implem.bcg";
53    result RESULT_LIVELOCK
54    expected FALSE
55  end property
56
57  -- En cas de livelock, produire un diagnostic détaillé et terminer
58  % if [ $RESULT_LIVELOCK != FALSE ]
59  % then
60  property Detailed_Livelock_Diagnostic
61    "A livelock was detected, produce a detailed counter-example"
62  is
63    "detail_livelock.bcg" = "detail_implem.bcg"
64      |= using bfs with evaluator4
65      <true*> < not ('ACTION .*' or 'HOOK_REFUSE .*') > @;
66    expected TRUE
67  end property
68  % echo ""
69  % echo "WARNING: protocol livelock detected, abort verification script"
70  % exit 0
71  % fi
72
73  (*************************************************************************)
74  -- Deadlock
75
76  property Absence_Deadlock
77    "There is no deadlock due to the protocol."
78  is
79    "implem.bcg" |=
80      [ true* ] (
81        ( < not ('ACTION .*') * > [true] false )
82        implies
83        [ true* . 'ACTION .*' ] false
84      );
85    result RESULT_DEADLOCK
86    expected TRUE
87  end property
88
89  -- En cas de deadlock, produire un diagnostic détaillé et terminer
90  % if [ $RESULT_DEADLOCK != TRUE ]
91  % then
92  property Detailed_Deadlock_Diagnostic
93    "A deadlock was detected, produce a detailed counter-example"
94  is
95    "detail_deadlock.bcg" = "detail_implem.bcg" |= using bfs
96      [ true* ] (
97        ( < not ('ACTION .*') * > [true] false )
```

```
98              implies
99              [ true* . 'ACTION .*' ] false
100         );
101     expected FALSE
102 end property
103 % echo ""
104 % echo "WARNING: protocol deadlock detected, abort verification script"
105 % exit 0
106 % fi
```

Annexe B

Spécification de l'algorithme de consensus Raft

Dans cette annexe, nous présentons la spécification LNT de l'algorithme Raft.

Types de données

Les types de données sont définis de la manière suivante :

```
1  -- Commandes du service
2  type command is
3      read,
4      writeFalse,
5      writeTrue,
6      empty_command -- pour les pulsations heartbeat
7  end type
8
9  type entry is
10     entry(term : nat, cmd : command)
11     with "get"
12 end type
13
14 type entry_list is
15     list of entry
16     with "access", "append", "head", "tail"
17 end type
18
19 -- 3 servers configuration
20 type serverID is
21     range 1 .. 3 of nat
22     with "==","!=","<=","val","ord"
23 end type
```

```
24
25 type bool_array is
26     array [0 .. 2] of bool
27 end type
28
29 type nat_array is
30     array [0 .. 2] of nat
31 end type
32
33 type serverState is
34     follower, candidate, leader
35     with "=="
36 end type
37
38 type voteID is
39     nil,
40     voteID (s : serverID)
41     with "=="
42 end type
43
44 type message is
45     RequestVoteRequest (to : serverID, term, lastLogIndex, lastLogTerm : nat),
46     RequestVoteResponse (to : serverID, term : nat, voteGranted : bool),
47     AppendEntryResponse (to : serverID, term : nat, success : bool, matchIndex : nat),
48     AppendEntryRequest (to : serverID, term, prevLogIndex, prevLogTerm, entryTerm : nat,
49         entryCommand : command, leaderCommit : nat)
50 end type
51
52 type message_list is
53     list of message
54     with "append", "empty", "head", " tail "
55 end type
56
57 type client_reply is
58     ok, fail
59 end type
```

Canaux de communication (channels)

Les channels sont définis de la manière suivante :

```
1 channel RaftRPC is
2     (serverID, serverID, nat, nat, nat),                    -- Vote Request
3     (serverID, serverID, nat, bool),                        -- Vote Response
4     (serverID, serverID, nat, nat, nat, nat, command, nat), -- AppendEntry Request
5     (serverID, serverID, nat, bool, nat)                    -- AppendEntry Response
6 end channel
7
8 channel Client is
```

```
9     (command), (client_reply, bool)
10  end channel
11
12  channel Leader is
13    (serverID, nat)
14  end channel
```

Fonctions

Les fonctions sont définis de la manière suivante :

```
1   function LastTerm(log : entry_list, logLength : nat) : nat is
2     if logLength == 0 then
3       return 0
4     else
5       return get_term (access (log, logLength))
6     end if
7   end function
8
9   function UpdateTerm (mTerm : nat, inout currentTerm : nat,
10    inout state : serverState, inout votedFor : voteID)
11  is
12    if mTerm > currentTerm then
13      currentTerm := mTerm ;
14      state := follower ;
15      votedFor := nil
16    else
17      -- no change, just remove compiler warning
18      state := state ;
19      votedFor := votedFor
20    end if
21  end function
22
23  function majorityAgree(index : nat, matchIndex : nat_array) : bool is
24    var n, nbagree : nat in
25      -- start nbagree at one since the caller always agree, but its own
26      -- matchIndex is zero
27      nbagree := 1;
28      for n := 0 while n < maxServer by n := n + 1 loop
29        if index <= matchIndex[n] then
30          nbagree := nbagree + 1
31        end if
32      end loop;
33      return nbagree >= majority
34    end var
35  end function
36
37  function maxAgreeIndex(lengthLog : nat, matchIndex : nat_array) : nat is
38    var index : nat in
```

```
39          index := lengthLog ;
40          while not(majorityAgree(index, matchIndex)) loop
41              index := index - 1
42          end loop;
43          return index
44      end var
45  end function
46
47  function AdvanceCommitIndex (state : serverState, currentTerm : nat,
48      log : entry_list, logLength : nat, matchIndex : nat_array,
49      inout commitIndex : nat)
50  is
51      var index : nat in
52          if state == leader then
53              index := maxAgreeIndex(logLength, matchIndex);
54              if (index > 0) and (get_term (access (log, index)) == currentTerm) then
55                  commitIndex := index
56              else
57                  -- remove compiler warning
58                  commitIndex := commitIndex
59              end if
60          end if
61      end var
62  end function
63
64  function BroadcastAppendEntryRequests (self : serverID, log : entry_list,
65      logLength : nat, nextIndex : nat_array, currentTerm, commitIndex : nat,
66      inout buf : message_list)
67  is
68      var
69          n, prevLogIndex, prevLogTerm, entryTerm, lastEntry : nat,
70          entryCommand : command
71      in
72          for n := 1 while n < maxServer by n := n + 1 loop
73              if n != ord (self) then
74                  prevLogIndex := nextIndex[n] - 1;
75                  if prevLogIndex > 0 then
76                      prevLogTerm := get_term (access (log, prevLogIndex))
77                  else
78                      prevLogTerm := 0
79                  end if;
80                  lastEntry := min (logLength, nextIndex[n]+1);
81                  if (nextIndex[n] > logLength) -- follower is up-to-date
82                      or (logLength == 0)       -- log is empty
83                  then
84                      -- just send empty entry as a heartbeat
85                      entryTerm := 0;
86                      entryCommand := empty_command
87                  else
88                      entryTerm := get_term (access (log, nextIndex[n]));
89                      entryCommand := get_cmd (access (log, nextIndex[n]))
```

```
90              end if;
91              buf := append (AppendEntryRequest (val (n), currentTerm, prevLogIndex,
92                  prevLogTerm, entryTerm, entryCommand, min (commitIndex, lastEntry)), buf)
93          end if
94      end loop
95   end var
96 end function
```

Processus serveur

Enfin, le comportement d'un serveur est spécifié de la manière suivante :

```
1  process RAFT_SERVER [
2     COM     : RaftRPC,
3     LEADER : Leader,
4     RESTART : any,
5     TIMEOUT : any,
6     CLIENT : Client ]
7     ( self : serverID )
8  is
9     var
10       -- persistent (survive a restart)
11       currentTerm : nat,
12       votedFor : voteID,
13       log : entry_list ,
14       logLength : nat,
15
16       -- volatile (reset at restart)
17       state : serverState ,
18       commitIndex : nat,
19       votesGranted : nat, -- is an array in the TLA+ spec
20       votesResponded : bool_array,
21       nextIndex : nat_array,
22       matchIndex : nat_array,
23
24       -- helpers
25       lastApplied : nat,
26       n : nat,
27       logOK, grant : bool,
28       index, currentMatchIndex : nat,
29
30       -- client
31       clientbusy : bool,
32       clientindex : nat,
33
34       -- store
35       store : bool,
36       cmd : command,
37
```

```
38        -- to decompose messages
39        mFrom : serverID,
40        mTerm, mLastLogIndex, mLastLogTerm, mMatchIndex : nat,
41        mPrevLogIndex, mPrevLogTerm, mEntryTerm, mLeaderCommit : nat,
42        mEntryCommand : command,
43        mVoteGranted, mSuccess : bool,
44        buf : message_list
45    in
46
47        -- Initialisation
48        currentTerm := 0;
49        votedFor := nil ;
50        state := follower ;
51        log := {};
52        logLength := 0;
53        commitIndex := 0;
54        votesGranted := 0;
55        votesResponded := bool_array ( false );
56        nextIndex := nat_array (1);
57        matchIndex := nat_array (0);
58        lastApplied := 0;
59        buf := {};
60        clientbusy := false ;
61        clientindex := 0;
62        store := false ; -- default value
63
64        -- Main loop
65        loop
66            select
67                -- Restart
68                RESTART (self);
69                if clientbusy then
70                    CLIENT (fail, false );
71                    clientbusy := false
72                end if;
73                state := follower ;
74                votesResponded := bool_array ( false );
75                votesGranted := 0;
76                nextIndex := nat_array (1);
77                matchIndex := nat_array (0);
78                commitIndex := 0;
79                lastApplied := 0;
80                store := false ;
81                buf := {}
82            []
83                -- Become candidate
84                only if (state == follower) or (state == candidate) then
85                    TIMEOUT (4 of int, self);
86                    state := candidate;
87                    currentTerm := currentTerm + 1;
88                    -- DIFF TLA: vote directly for myself
```

```
89              votedFor := voteID ( self );
90              votesResponded := bool_array ( false );
91              votesGranted := 1;
92
93              for n := 1 while n < maxServer by n := n+1 loop
94                  if n != ord ( self ) then
95                      buf := append (RequestVoteRequest (val (n), currentTerm,
96                          logLength, LastTerm (log, logLength)), buf)
97                  end if
98              end loop
99          end if
100     []
101         -- Handle RequestVote request
102         COM (?mFrom, self, ?mTerm, ?mLastLogIndex, ?mLastLogTerm);
103
104         if mTerm > currentTerm and clientbusy then
105             CLIENT (fail, false );
106             clientbusy := false
107         end if;
108         eval UpdateTerm (mTerm, !?currentTerm, !?state, !?votedFor);
109
110         logOK := (mLastLogTerm > LastTerm (log, logLength)) or
111             ((mLastLogTerm == LastTerm (log, logLength)) and
112             (mLastLogIndex >= logLength));
113
114         grant :=      (mTerm == currentTerm)
115         and (logOK)
116         and (     (votedFor == nil)
117         or (votedFor == voteID (mFrom)) );
118
119         if grant then
120             votedFor := voteID (mFrom)
121         end if;
122         buf := append (RequestVoteResponse (mFrom, currentTerm, grant), buf)
123     []
124         -- Handle RequestVote response
125         COM (?mFrom, self, ?mTerm, ?mVoteGranted);
126
127         if mTerm >= currentTerm then
128             if mTerm > currentTerm and clientbusy then
129                 CLIENT (fail, false );
130                 clientbusy := false
131             end if;
132             eval UpdateTerm (mTerm, !?currentTerm, !?state, !?votedFor);
133             -- only a candidate care about a vote response
134             if (state == candidate) and mVoteGranted then
135                 votesGranted := votesGranted + 1;
136                 votesResponded[ord (mFrom)] := true;
137
138                 -- Become leader
139                 if (votesGranted >= majority) then
```

```
140                     LEADER (self, currentTerm);
141                     state := leader;
142                     nextIndex := nat_array (logLength+1);
143                     matchIndex := nat_array (0)
144                 end if
145
146             end if
147         end if
148     []
149         -- Append entry
150         only if (state == leader) then
151             TIMEOUT (1 of int);
152             eval BroadcastAppendEntryRequests (self, log, logLength, nextIndex,
153                 currentTerm, commitIndex, !?buf)
154         end if
155     []
156         -- Client request
157         only if (state == leader) and not (clientbusy) then
158             CLIENT (?cmd);
159             logLength := logLength+1;
160             log := append (entry (currentTerm, cmd), log);
161             clientbusy := true;
162             clientindex := logLength;
163
164             eval BroadcastAppendEntryRequests (self, log, logLength, nextIndex,
165                 currentTerm, commitIndex, !?buf)
166         end if
167     []
168         -- Handle AppendEntry request
169         COM (?mFrom, self, ?mTerm, ?mPrevLogIndex, ?mPrevLogTerm,
170         ?mEntryTerm, ?mEntryCommand, ?mLeaderCommit);
171
172         if mTerm > currentTerm and clientbusy then
173             CLIENT (fail, false );
174             clientbusy := false
175         end if;
176         eval UpdateTerm (mTerm, !?currentTerm, !?state, !?votedFor);
177
178         logOK :=    (mPrevLogIndex == 0)
179         or (     (mPrevLogIndex > 0)
180         and (mPrevLogIndex <= logLength)
181         and (mPrevLogTerm ==
182         get_term (access (log, mPrevLogIndex))) );
183
184         if (not (logOK)) or (mTerm < currentTerm) then
185             -- reject request
186             buf := append (AppendEntryResponse(mFrom, currentTerm, False, 0 of nat), buf)
187         else
188             -- accept request
189             index := mPrevLogIndex + 1;
190             if (mEntryTerm == 0) then
```

```
191                 -- heartbeat
192                 currentMatchIndex := mPrevLogIndex
193               else
194                 -- real entry, remove the log tail if conflict
195                 if       (logLength >= index)
196                   and (get_term (access (log, index)) != mEntryTerm)
197                 then
198                   var newlog : entry_list in
199                     logLength := min (logLength, mPrevLogIndex);
200                     newlog := {};
201                     for n := 1 while n <= logLength by n := n+1 loop
202                       newlog := append (head (log), newlog);
203                       log := tail (log)
204                     end loop;
205                     log := newlog
206                   end var
207                 end if;
208                 -- append entry
209                 if logLength == mPrevLogIndex then
210                   logLength := logLength+1;
211                   log := append (entry (mEntryTerm, mEntryCommand), log)
212                 end if;
213                 -- always one entry
214                 currentMatchIndex := mPrevLogIndex + 1
215               end if;
216               commitIndex := mLeaderCommit;
217               buf := append (AppendEntryResponse(mFrom, currentTerm, True,
218                 currentmatchIndex), buf)
219             end if
220           []
221             -- apply commands
222             only if (lastApplied < commitIndex) and (lastApplied < logLength) then
223               lastApplied := lastApplied + 1;
224               case get_cmd (access (log, lastApplied)) of command in
225                 writeFalse -> store := false
226               | writeTrue  -> store := true
227               | read       -> null
228               end case;
229               if clientbusy and (clientindex == lastApplied) then
230                 CLIENT (ok, store);
231                 clientbusy := false
232               end if
233             end if
234           []
235             -- Handle AppendEntry response
236             COM (?mFrom, self, ?mTerm, ?mSuccess, ?mMatchIndex);
237
238             if mTerm >= currentTerm then
239               if mTerm > currentTerm and clientbusy then
240                 CLIENT (fail, false );
241                 clientbusy := false
```

```
242                 end if;
243                 eval UpdateTerm (mTerm, !?currentTerm, !?state, !?votedFor);
244                 if state == leader then
245                     if mSuccess then
246                         nextIndex[ord (mFrom)] := mMatchIndex + 1;
247                         matchIndex[ord (mFrom)] := mMatchIndex
248                     else
249                         nextIndex[ord (mFrom)] := max (1 of nat,
250                         nextIndex[ord (mFrom)] - 1)
251                     end if;
252
253                     -- DIFF TLA: advance commit index after append entry
254                     eval AdvanceCommitIndex (state, currentTerm, log,
255                     logLength, matchIndex, !?commitIndex)
256
257                 end if
258             end if
259
260         []
261             -- send message
262             only if not (empty (buf)) then
263                 case head (buf) of message in
264                     RequestVoteRequest (mFrom, mTerm, mLastLogIndex, mLastLogTerm) ->
265                         COM (self, mFrom, mTerm, mLastLogIndex, mLastLogTerm)
266                 | RequestVoteResponse (mFrom, mTerm, mVoteGranted) ->
267                         COM (self, mFrom, mTerm, mVoteGranted)
268                 | AppendEntryRequest (mFrom, mTerm, mPrevLogIndex, mPrevLogTerm,
269                         mEntryTerm, mEntryCommand, mLeaderCommit) ->
270                         COM (self, mFrom, mTerm, mPrevLogIndex, mPrevLogTerm, mEntryTerm,
271                             mEntryCommand, mLeaderCommit)
272                 | AppendEntryResponse (mFrom, mTerm, mSuccess, mMatchIndex) ->
273                         COM (self, mFrom, mTerm, mSuccess, mMatchIndex)
274                 end case;
275                 buf := tail (buf)
276             end if
277
278         end select
279     end loop
280 end var
281 end process
```

Bibliographie

[AFM+04] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES : a tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2004.

[Arb04] Farhad Arbab. Reo : a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3) :329–366, 2004.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, November 2003.

[Bag85] R. Bagrodia. A distributed algorithm for n-party interactions. *MCC Technical Report*, (STP-053-85), August 1985.

[Bag89] R. Bagrodia. Process synchronization : Design and performance evaluation of distributed algorithms. *Software Engineering, IEEE Transactions on*, 15(9) :1053–1065, 1989.

[BBJ+10a] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 209–218. ACM, 2010.

[BBJ+10b] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. Automated conflict-free distributed implementation of component-based models. In *IEEE Fifth International Symposium on Industrial Embedded Systems - SIES 2010, University of Trento, Italy, July 7-9, 2010*, pages 108–117. IEEE, 2010.

[BBQ11] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. Automated distributed implementation of component-based models with priorities. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 59–68. ACM, 2011.

[BBQ13] Borzoo Bonakdarpour, Marius Bozga, and Jean Quilbeuf. Model-based implementation of distributed systems with priorities. *Design Autom. for Emb. Sys.*, 17(2) :251–276, 2013.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 3–12. IEEE Computer Society, 2006.

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for branching time semantics. In *Proceedings of 18th ICALP*. Springer, July 1991.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BGL+11] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2 : Towards efficient correctness of incremental design. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 453–458, 2011.

[BGW89] G. Bochmann, Q. Gao, and C. Wu. On the distributed implementation of LOTOS. In *Proc. 2nd Intl. Conf. on Formal Description Techniques*, pages 133–146, 1989.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3) :560–599, July 1984.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker : A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[BLM+01] Gerd Behrmann, Kim G Larsen, Oliver Moller, Alexandre David, Paul Pettersson, and Wang Yi. Uppaal-present and future. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2881–2886. IEEE, 2001.

[Bur06] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh,*

UK, April 4-8, 2005, Proceedings, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCG+14] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS, 131 pages, August 2014.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. *CoRR*, abs/1011.2560, 2010.

[CGH+10] Nicolas Coste, Hubert Garavel, Holger Hermanns, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Ten years of performance evaluation for concurrent systems using CADP. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA 2010 (Amirandes, Heraclion, Crete), Part II*, volume 6416 of *Lecture Notes in Computer Science*, pages 128–142. Springer, October 2010.

[CGR07] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live : an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407, 2007.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[CM88] K Mani Chandy and Jayadev Misra. Parallel program design : a foundation. *Addison-Wesley*, 1988.

[CM13] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design : multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013.

[dBNR09] Niels H. M. Aan de Brugh, Viet Yen Nguyen, and Theo C. Ruys. Moonwalker : Verification of .net programs. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 170–173. Springer, 2009.

[DDG89] Michel Diaz, Jean Dufau, and Roland Groz. Experiences using estelle within SEDOS estelle demonstrator. In *Proceedings of the IFIP TC/WG6. 1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 455–470. North-Holland Publishing Co., 1989.

[Dij68] Edsger W. Dijkstra. The structure of "the"-multiprogramming system. *Commun. ACM*, 11(5) :341–346, 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1 :115–138, 1971.

[EL13] Hugues Evrard and Frédéric Lang. Formal verification of distributed branching multiway synchronization protocols. In Dirk Beyer and Michele Boreale, editors, *Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems FORTE/FMOODS'2013 (Florence, Italy)*, volume 7892 of *Lecture Notes in Computer Science*, pages 146–160. IFIP, Springer, June 2013.

[EL15] Hugues Evrard and Frédéric Lang. Automatic distributed code generation from formal models of asynchronous concurrent processes. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing PDP'2015 (Turku, Finland)*. IEEE, 2015.

[FM91] Jean-Claude Fernandez and Laurent Mounier. "on the fly" verification of behavioural equivalences and preorders. In Kim G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, July 1991.

[Gar98] Hubert Garavel. Open/cæsar : An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer. Full version available as INRIA Research Report RR-3352.

[GGvB+95] Qiang Gao, Roland Groz, Gregor von Bochmann, Joumana Dargham, and E. Houssain Htite. Validation of distributed algorithms and protocols. In *1995 International Conference on Network Protocols, ICNP 1995, November 7-10, 1995, Tokyo, Japan*, pages 110–117. IEEE Computer Society, 1995.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011 : A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2) :89–107, April 2013.

[GMS13] Hubert Garavel, Radu Mateescu, and Wendelin Serwe. Large-scale Distributed Verification using CADP : Beyond Clusters to Grids. *Electronic Notes in Theoretical Computer Science*, 296 :145–161, August 2013.

[God05] Patrice Godefroid. Software model checking : The verisoft approach. *Formal Methods in System Design*, 26(2) :77–101, 2005.

[GVZ01] Hubert Garavel, César Viho, and Massimo Zendri. System design of a cc-numa multiprocessor architecture using formal specification, model-checking,

co-simulation, and test generation. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3) :314–331, July 2001. Also available as INRIA Research Report RR-4041.

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4) :238–241, 1970.

[Hav68] J.W. Havender. Avoiding deadlock in multitasking systems. *IBM systems journal*, 7(2) :74–84, 1968.

[HC11] Sean Halle and Albert Cohen. A mutable hardware abstraction to replace threads. In Sanjay V. Rajopadhye and Michelle Mills Strout, editors, *Languages and Compilers for Parallel Computing, 24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011. Revised Selected Papers*, volume 7146 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2011.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper : Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoa88] C Antony R Hoare. Occam 2 reference manual. *INMOS Limited*, page 27, 1988.

[Hol04] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[ISO88] ISO/IEC. Estelle — a formal description technique based on an extended state transition model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, September 1988.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, September 1989.

[ISO01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437 :2001, International Organization for Standardization — Information Technology, Geneva, September 2001.

[JJ92] Claude Jard and Jean-Marc Jézéquel. Echidna, an estelle compiler to prototype protocols on distributed computers. *Concurrency - Practice and Experience*, 4(5) :377–397, 1992.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.

[JS94] Yuh-Jzer Joung and Scott A. Smolka. Coordinating first-order multiparty interactions. *ACM Trans. Program. Lang. Syst.*, 16(3) :954–985, 1994.

[JSA14] Sung-Shik T. Q. Jongmans, Francesco Santini, and Farhad Arbab. Partially-distributed coordination with reo. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 697–706, 2014.

[Knu84] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2) :97–111, 1984.

[KP10] Gal Katz and Doron Peled. Code mutation in verification and automatic code correction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 435–450. Springer, 2010.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[Kum90] D. Kumar. An implementation of n-party synchronization using tokens. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 320–327, 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2) :133–169, 1998.

[Lan02] Frédéric Lang. Compositional verification using svl scripts. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer, April 2002.

[Lan05] Frédéric Lang. Exp.open 2.0 : A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer, November 2005. Full version available as INRIA Research Report RR-5673.

[Löf96] Siegfried Löffler. From specification to implementation : A PROMELA to C compiler. *Project Report Ecole Nationale Supérieure des Télécommunications*, 1996.

[MdMSA93] José A. Mañas, Tomás de Miguel, Joaquín Salvachúa, and Arturo Azcorra. Tool Support to Implement LOTOS Formal Specifications. *Computer Networks and ISDN Systems*, 25(7) :815–839, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems : the Pi-Calculus*. Cambridge University Press, 1999.

[MO08] Radu Mateescu and Emilie Oudot. Bisimulator 2.0 : An on-the-fly equivalence checker based on boolean equation systems. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008 (Anaheim, CA, USA)*, pages 73–74. IEEE Computer Society Press, June 2008.

[MQ06] Madan Musuvathi and Shaz Qadeer. CHESS : systematic stress testing of concurrent software. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, volume 4407 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2006.

[MQR11] Stephan Merz, Martin Quinson, and Cristian Daniel Rosa. SimGrid MC : Verification Support for a Multi-API Simulation Platform. In Roberto Bruni and Jürgen Dingel, editors, *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6722 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2011.

[MS13] Radu Mateescu and Wendelin Serwe. Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. *Science of Computer Programming*, 78(7) :843–861, July 2013.

[MT08] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, May 2008.

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 673–676. Springer, 1992.

[Ode09] Martin Odersky. Essentials of scala. In Bernard Carré and Olivier Zendra, editors, *Langages et Modèles à Objets, LMO 2009, Nancy, France, 25-27 mars 2009*, volume L-3 of *RNTI*, page 2. Cépaduès-Éditions, 2009.

[Ong14] Diego Ongaro. *Consensus : Bridging Theory and Practice*. PhD thesis, Stanford University, USA, August 2014.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, March 1981.

[PCdVA12] Jose Proenca, Dave Clarke, Erik de Vink, and Farhad Arbab. Dreams : a framework for distributed synchronous coordination. In *SAC*. ACM, 2012.

[PCT04] José Antonio Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency - Practice and Experience*, 16(12) :1173–1206, 2004.

[PNG13] Kirstin Peters, Uwe Nestmann, and Ursula Goltz. On distributability in process calculi. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2013.

[PS92] Joachim Parrow and Peter Sjödin. Multiway synchronization verified with coupled simulation. In *CONCUR'92*, pages 518–533. Springer, 1992.

[PS96] J. Parrow and P. Sjödin. Designing a multiway synchronization protocol. *Computer communications*, 19(14) :1151–1160, 1996.

[PSN11] Kirstin Peters, Jens-Wolfhard Schicke, and Uwe Nestmann. Synchrony vs causality in the asynchronous pi-calculus. In Bas Luttik and Frank Valencia, editors, *Proceedings 18th International Workshop on Expressiveness in Concurrency, EXPRESS 2011, Aachen, Germany, 5th September 2011.*, volume 64 of *EPTCS*, pages 89–103, 2011.

[Qui13] Jean Quilbeuf. *Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions*. PhD thesis, Université de Grenoble, September 2013.

[Sch90] Fred B Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys (CSUR)*, 22(4) :299–319, 1990.

[SCV91] R. Sisto, L. Ciminiera, and A. Valenzano. A protocol for multirendezvous of LOTOS processes. *Computers, IEEE Transactions on*, 40(4) :437–447, 1991.

[Sha13] Asankhaya Sharma. A refinement calculus for promela. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 75–84. IEEE, 2013.

[Sig99] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1999.

[Sjö91] Peter Sjödin. *From LOTOS Specifications to Distributed Implementations*. PhD thesis, Department of Computer Science, University of Uppsala (Sweden), 1991.

[Taf00] Tucker S Taft. *Consolidated Ada Reference Manual : Language and Standard Libraries : International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1*, volume 2219. Springer Science & Business Media, 2000.

[Tap98] Josef Tapken. MOBY/PLC - A design tool for hierarchical real-time automata. In *FASE*, pages 326–329, 1998.

[Tau87] Dirk Taubner. On the implementation of petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1988, covers the 8th European Workshop on Applications and Theory of Petri Nets, held in Zaragoza, Spain in June 1987, selected papers*, volume 340 of *Lecture Notes in Computer Science*, pages 418–434. Springer, 1987.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[vGW96] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3) :555–600, 1996.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In Yves Ledru, editor, *Proceedings of the 15th IEEE International Conference on Automated Software Engineering ASE'2000 (Grenoble, France)*, pages 3–12, September 2000.

[Win83] Józef Winkowski. A distributed implementation of petri nets. Technical Report 518 (1983), Polish Academy of Science, Institute of Computer Science, Warsaw, 1983.

[YHT01] K. Yasumoto, T. Higashino, and K. Taniguchi. A compiler to implement LOTOS specifications in distributed environments. *Computer Networks*, 36(2) :291–310, 2001.